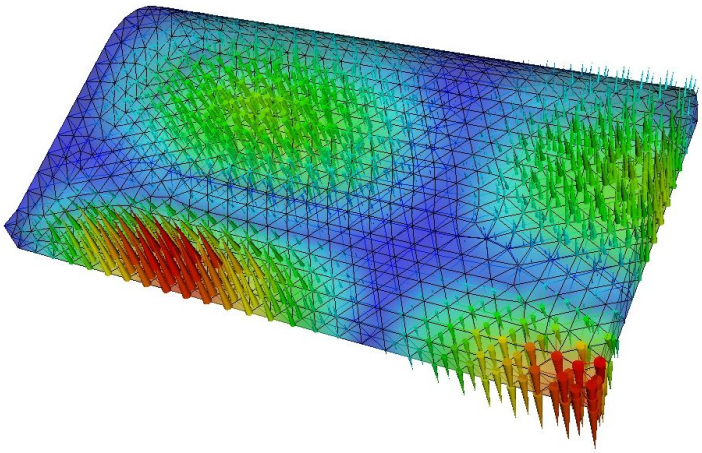
PySparse and PyFemax: A Python framework for large scale sparse linear algebra

Roman Geus and Peter Arbenz
Swiss Federal Institute of Technology Zurich
Institute of Scientific Computing
mailto:geus@inf.ethz.ch
mailto:arbenz@inf.ethz.ch

March 13, 2003



In scientific computing, software is traditionally developed using compiled languages like Fortran or C for maximal performance. However, for most applications, the time-critical portion of the code that requires the efficiency of a compiled language, is confined to a small set of well-defined functions. Implementing the remaining part of the application using an interactive and interpreted high-level language has many advantages without a big performance degradation.

This paper describes our experience with the redesign and reimplementation of a large-scale application from accelerator physics using this mixed-

language programming approach with Python and C.

The application code which was originally written in Fortran 90, with smaller parts written in C and Fortran 77, primarily solves eigenvalue problems and linear systems involving large sparse matrices [1]. We solved problems with matrix orders up to 7.5 million. The memory requirements of a single of these matrices is beyond 2 GBytes.

The paper also presents the two Python software packages: PySparse, for manipulating sparse matrices, and PyFemax, a finite element code for computing three-dimensional electro-magnetic fields in accelerator cavities.

1 Motivation

In its final state, the original Fortran/C code was approximately 33'000 lines long. During its development time of approximately four years, the code became cluttered and was no longer well modularised, since many features were added that were not forseen in the beginning. The code was difficult to understand, maintain and extend.

The many features of the program were controlled by means of a parameter file containing over 100 different values to adjust. This approach that was handy initially turned out to be inflexible for the large amount of parameters.

2 Redesign using a mixed-language approach with Python

The above reasons motivated us to redesign and rewrite the code. We had the following goals in mind:

Modularity The code should be organised in reusable modules. These modules should not have side effects.

Brevity The code should be short and concise, e.g. as Matlab code.

Object based Sparse and dense matrices and vectors as well as preconditioners and solvers should be implemented as objects, having attributes and methods for improved readability and extensibility.

High performance The code's performance should be comparable to the Fortran/C implementation.

In short: We wanted to have the *simplicity of Matlab* together with the *performance of Fortran and C*. In fact we wanted to have more than that,

since Matlab lacks many language features desirable for building large applications.

Based on our previous experience with scripting languages, we found Python to be the language of choice for the redesign.

As an interpreted programming language, Python is "out of the box" not well suited for high-performance numerical applications. However, Python can easily be extended using modules written in C for performance-critical tasks.

We designed and implemented the *PySparse* package, which extends Python with new sparse matrix object types and some operations on them. The *PyFemax* package contains the application-specific code.

PySparse and PyFemax rely on the *Numerical Python* [2] package for storing and operating on dense matrices and vectors. PySparse is implemented using a highly optimised C code, while PyFemax is written almost completely in Python.

What we are calling the *Python implementation* is actually a set of modules which are implemented using a mixed-language programming approach: The application logic, the input/output routines and the finite element code are implemented in Python. On the other hand, the time-critical parts, like the sparse and dense linear algebra routines, including iterative solvers, preconditioners, sparse matrix factorisations and the eigensolver are implemented in C and are tightly integrated into the Python framework.

For writing the C extension modules, we used the *Python modulator* to generate skeletons which were then extended manually. This approach allowed us not only to have full control over the interface, but also to keep the calling overhead minimal and to use features not offered by automatic interface generators like SWIG.

3 PySparse and PyFemax

PySparse and PyFemax have been designed as independent modules with reuse and extensibility in mind. Sparse matrices, preconditioners and solvers are Python objects. For performance reasons, most objects in PySparse and PyFemax are implemented as extension types. Some of them are implemented as Python classes. Interoperability between these objects is ensured by imposing certain standards on their attributes and methods:

Every Python object that has a *shape* attribute for returning the dimensions of a matrix and a *matvec* method for performing the matrix-vector multiplication can be passed as a matrix to PySparse routines.

Preconditioners have a *shape* attribute and also a *precon* method, that

applies the preconditioner on one vector and stores the result in another vector. In analogy, a solver has a *shape* attribute and a *solve* method.

In this way, it possible to introduce e.g. a new preconditioner type without changing any of the existing library code in PySparse and PyFemax. Only the top level script, which creates objects of the new type needs to be adjusted.

A more elegant way to guarantee interoperability would be to design an appropriate class hierarchy, deriving all objects of a certain kind from a common abstract base class. With Python 2.2 this is not possible, since extension types cannot be used as base classes. However, there is an ongoing effort to do away with the differences between extension types and Python classes.

The modules that can be potentially used in a wide range of scientific applications have been placed in PySparse, while the more application-specific modules are part of PyFemax.

From the application user's point of view, other improvements are more important: The computation can now be steered by scripts, which usually consist of less than 30 lines of Python code. This offers exciting new flexibilities. An additional benefit is the fact that the computational structure is directly reflected in these scripts and thus becomes much more transparent.

3.1 PySparse

PySparse extends the Python interpreter by a set of sparse matrix types holding double precision values. The entries of such a sparse matrix can be accessed conveniently from Python using two-dimensional array indices. Submatrices can be accessed similarly using slices.

One sparse matrix type (*ll_mat*) is designed for efficiently creating or modifying matrices. Another sparse matrix (*csr_mat*) type is designed for memory efficiency and fast row-by-row access to its elements, which is desirable for matrix-vector multiplication. PySparse can store symmetric matrices efficiently and provides conversion routines for the different formats.

PySparse also includes modules that implement

- iterative methods for solving linear systems of equations
- a set of standard preconditioners
- an interface to a direct solver for sparse linear systems of equations (SuperLU)
- a Jacobi-Davidson eigenvalue solver for the symmetric, generalised matrix eigenvalue problem (JDSYM)

The code example below illustrates the use of the new sparse matrix types. The function `poisson2d_sym_blk` builds an *ll_mat* object, representing a sparse 2D-Poisson matrix of order n^2. Notice that only the elements on and below the diagonal are stored. With two-dimensional indices, matrix elements and submatrices can accessed conveniently. The syntax is the same as the one used for NumPy's dense arrays types.

```
import spmatrix

def poisson2d_sym_blk(n):
    L = spmatrix.ll_mat_sym(n*n)
    I = spmatrix.ll_mat_sym(n)
    for i in range(n):
        I[i,i] = -1
    P = spmatrix.ll_mat_sym(n)
    for i in range(n):
        P[i,i] = 4
        if i > 0:
            P[i,i-1] = -1
    for i in range(0, n*n, n):
        L[i:i+n,i:i+n] = P
        if i > 0: L[i:i+n,i-n:i] = I
    return L
```

The experimental results in section 5 indicate, that PySparse is able to handle sparse matrices more efficiently than e.g. Matlab.

More information and further code examples for PySparse can be found in [1, Section 9.4] and [3].

3.2 PyFemax

PyFemax is a set of Python modules containing the application-specific code:

- a finite element code for discretising Maxwell's equations using second order Nédélec finite elements
- eigenvalue solvers for the Maxwell problem, building on top of the JDSYM solver in PySparse
- various file import routines for reading tetrahedral mesh data
- postprocessing routines for visualising the electromagnetic fields using VTK [4] or MayaVi [5]

More information and code examples for PyFemax can be found in [1, Section 9.5 and 9.6] and [3].

4 Benefits and Drawbacks

4.1 Benefits

Improved usability and flexibility With PyFemax, the computation is steered by a small script instead of a large parameter file, as it is conventionally done. Since most parameters have default values, usually only a small set of parameters has to be explicitly specified in the script. The user writing such a script can tailor the simulation to his/her specific needs. The scripting facilities give the user more flexibility for controlling the computation.

Extensibility Thanks to its object-oriented design, PyFemax can be extended easily. Matrices, solvers, preconditioners, etc., are designed as objects that implement certain interfaces in order to be interoperable with the PyFemax framework. New methods can be added to PyFemax without changing the existing code.

Since all objects of a given type (e.g. preconditioners) conform to the same standards, they are in fact interchangeable. Many algorithmic variants can be easily tested by combining existing objects.

Brevity The object-oriented design promotes the reusability of the code and thus leads to shorter programs, that are easier to read.

Improved maintainability Modularisation was an important design guideline for the development of PyFemax. The module interfaces are kept lean. No global variables are used. Modules can be tested and debugged independently. All this contributes to improved clarity and maintainability of the code.

Explorative computation Since Python is an interpreted and interactive scripting language, the user can undertake computations in an explorative manner: Intermediate results can be examined and taken into account before undertaking the next computational step.

Rapid development Both Python's high level data types and its large collection of standard modules assist the programmer in focusing on the problem, instead of on the implementation details.

4.2 Drawbacks

Tricky installation on some systems Some features of the current PySparse and PyFemax releases require the installation of version 2.2 of the Python interpreter together with some additional software packages. On some (more exotic) platforms the installation process can become quite involved and is best carried out by a person familiar with the operating system.

Performance penalty Since some portions of the code are interpreted and because there is some calling overhead for Python functions, the performance is deteriorated. Our experiments indicate that the overall performance loss is at most 20%.

Lack of compile-time checks In Python all type checks must be performed at run-time. Trivial program errors may manifest themselves only after hours of computation. With a compiled programming language such errors can be detected at compile-time. With additional tools like *PyChecker* [6, 7] it is possible to find some of these errors in Python code. The trouble with the existing tools is that the module to be checked for errors has to be imported. This means that the steering scripts have to be executed until completion in order to be checked, which is clearly not what we desire. *PyChecker2*, which is currently in development, operates on the source code alone and will address this problem.

Success of future parallelisation unclear Research has been conducted in the direction of parallelising Python applications: *MPI Python* is a framework for developing parallel Python applications using MPI [8]. *PyPAR* [9] is a more light-weight wrapper of the MPI library for Python. Another alternative is the *Python BSP* package [10, 11], that supports the more high-level Bulk Synchronous Parallel approach.

In all these approaches, additional overhead is introduced by the Python interface. Since our algorithms only support a relatively fine-grained parallelism, it is not clear whether one of these approaches could lead to a successfully parallelised of the application.

5 Experimental results

Let us evaluate the performance of PySparse's sparse matrix functionality by building a large 2D-Poisson matrix. For the following experiments a Sun Enterprise E3500 machine has been used.

The table below shows the execution times (in seconds) for several Python and Matlab functions generating 2D-Poisson matrices of different sizes. The Python and Matlab source codes and further details regarding this experiment can be found in [1, Section 9.4.1] and in Section 3.1.

Function	$n = 100^2$	$n = 300^2$	$n = 500^2$	$n = 1000^2$
Python `poisson2d`	0.44	4.11	11.34	45.50
Python `poisson2d_sym`	0.26	2.34	6.55	26.33
Python `poisson2d_sym_blk`	0.03	0.21	0.62	2.22
Matlab `poisson2d`	28.19	3464.9	38859	almost ∞
Matlab `poisson2d_blk`	6.85	309.20	1912.1	almost ∞
Matlab `poisson2d_kron`	0.21	2.05	6.23	29.96

The results clearly show that the PySparse implementation is about ten times faster than the Matlab implementation, when comparing the best codes. This huge difference in speed can be attributed mainly to the choice of data-structures representing a sparse matrix. Matlab's storage format is inefficient for inserting new matrix entries. PySparse benefits from the *ll_mat* sparse matrix type which is based on a linked-list data structure and is thus well suited for building sparse matrices.

In the next experiment a linear system with the 2D-Poisson matrix is solved by the preconditioned conjugate gradient method using the Python code, a Matlab implementation and a native C implementation. The problems of equal size were identical such that the iterations proceeded equally in all three implementations.

Size n	*Python time*	*Native C time*	*Matlab time*
100^2	1.12	0.96	8.85
300^2	49.65	48.38	387.26
500^2	299.39	288.67	1905.67

The results show that the Python code using PySparse is almost as fast as the native C implementation. The codes and further details regarding this experiment can be found in [1, Section 9.4.3].

Regarding the calculation of electromagnetic fields, we have compared the performance of the old natively compiled code with the performance of the new mixed-language programming approach in [1, Section 9.6.1]. In our experiments, the natively compiled code was at most 20% faster. The overhead in the new implementation can be mainly attributed to Python function argument handling, slow Python file I/O and some interpreted mesh handling code that should be moved to a C extension module.

Python imposes no restriction on the size of the problems we can solve. In our biggest experiment a total of 18 GBytes of memory was used. The

resulting sparse matrices had an order of 7.5 million and had up to 160 million non-zero elements, requiring up to 2.4 GBytes of memory. The Python code was running for about 40 hours to successfully compute the requested results.

6 Conclusions

We successfully applied the mixed-language programming approach based on Python and C to a large-scale scientific application computing electromagnetic fields in accelerator cavities. The new implementation based on PySparse and PyFemax is shorter, better maintainable, extensible and offers improved flexibility through scripting. The performance penalty introduced by Python is relatively small.

PySparse adds support for sparse matrices to Python and has been designed to be easy to use, applicable to a wide range of computational problems and also efficient in terms of memory and speed.

PySparse is already used in several other scientific projects: e.g. Oliver Bröker's WolfAMG [12], an object-oriented framework for building algebraic multigrid solvers, is based on PySparse.

PySparse and PyFemax are open source software packages and can be downloaded from `http://www.inf.ethz.ch/personal/geus/pyfemax`.

References

[1] Roman Geus. *The Jacobi-Davidson algorithm for solving large sparse symmetric eigenvalue problems with application to the design of accelerator cavities.* PhD thesis, Swiss Federal Institute of Technology Zurich, December 2002. Diss. ETH No. 14734, `http://e-collection.ethbib.ethz.ch/cgi-bin/show.pl?type=diss&nr=14734`.

[2] David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, and Travis Oliphant. *Numerical Python.* Lawrence Livermore National Laboratory, September 2001.

[3] Roman Geus. Calculating electro-magnetic waves in accelerator cavities using Python. Website at `http://www.inf.ethz.ch/personal/geus/pyfemax/`.

[4] William J. Schroeder, editor. *The VTK User's Guide.* Kitware, Inc., 2001.

[5] Prabhu Ramachandran. Mayavi: A free tool for CFD data visualization. In *4th Annual CFD Symposium*. Aeronautical Society of India, August 2001.

[6] Neal Norwitz. PyChecker. SourceForge project `http://pychecker.sourceforge.net/`.

[7] Cameron Laird and Kathryn Soraiz. Regular expressions: Syntax checking the scripting way. UNIX Review, April 2002. `http://www.unixreview.com/`.

[8] Pat Miller. MPI Python. SourceForge project `http://sourceforge.net/projects/pympi/`.

[9] Ole Nielsen. PyPAR - Parallel Python. `http://datamining.anu.edu.au/~ole/pypar/`.

[10] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 1998.

[11] Konrad Hinsen. High level scientific programming with Python. In Peter M. A. Sloot, C. J. Kenneth Tan, Jack J. Dongarra, and Alfons G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2331 of *LNCS*, pages 691–700. Springer, 2002.

[12] Oliver Bröker. WolfAMG. Website at `http://wolfamg.ethz.ch/`.