PYFEMAX: A PYTHON FINITE ELEMENT MAXWELL SOLVER

R. Geus (ETHZ), P. Arbenz (ETHZ), L. Stingelin

The advanced Maxwell eigenvalue solver PyFemax is evolving from a research project into a mature simulation tool for RF-cavity design. Its object oriented design, based on the scripting language Python, gives flexibility and portability. For high performance, the time-critical parts are written in C. Presently, PyFemax relies on the Jacobi-Davidson algorithm, a recently proposed method for solving matrix eigenvalue problems. The matrices originate from a Nédélec finite element discretisation of Maxwell's equations. Current research is focusing on the integration of an algebraic multigrid preconditioner and the LOBPCG eigenvalue solver, which are expected to improve both memory consumption and solution time.

INTRODUCTION

The PyFemax eigenvalue solver [3], developed at the Institute for Scientific Computing at ETH Zurich, is a tool for calculating eigenmodes of large RF-structures. It uses unstructured tetrahedral grids to represent the geometry and special algorithms in order to avoid spurious modes. PyFemax has been successfully used to calculate resonance frequencies, electric and magnetic field distributions of large and complex structures such as the new COMET cyclotron (Figs. 1, 2) and the large TRIUMF 520 MeV cyclotron.

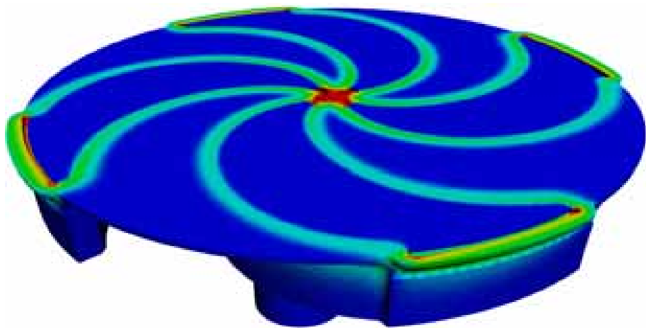

Fig. 1: View on the beam plane of the COMET RF-structure with the visualisation program MayaVi. The contour-plot of the electric field strength is shown. Only the lower part of the structure is simulated taking a magnetic surface as symmetry condition.

PYTHON IMPLEMENTATION

For the Python implementation, 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 the Matlab code.

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

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

In brief: We wanted to achieve 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 not well suited for high-performance numerical applications, in its basic form. However, Python can easily be extended using modules written in C for performance-critical tasks. 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.

FEATURES

The time-harmonic Maxwell equations describing the electromagnetic field are discretised using *second order Nédélec vector elements*. The resulting generalised symmetric matrix eigenvalue problem is solved using the *JDSYM algorithm*, which is an implementation of the *Jacobi Davidson method* optimised for the symmetric eigenvalue problem.

Several methods for preventing the convergence to the nullspace are implemented: *projection methods*, the *AD method* [2] and a *modification of the eigensolver*. The projection methods keep all relevant vectors in the positive-definite subspace to avoid convergence to the nullspace. The AD method transforms the original matrix eigenvalue problem to a matrix eigenvalue problem of lower dimension with the nullspace removed. Both methods exploit the knowledge of a sparse basis of the nullspace and require the solution of a large sparse linear system in each iteration. The performance of these two methods is comparable.

The convergence to the nullspace can be prevented in the JDSYM eigensolver itself by introducing a dynamic shift selection strategy and altering the ordering scheme of the Ritz values. This method has the advantage that no additional linear systems have to be solved and no memory for additional matrices is consumed. Unfortunately, this method requires more iteration steps since the influence of the nullspace degrades the convergence. The projection and AD methods lead to faster convergence, if the linear systems can be solved efficiently and enough memory is available.

To improve the speed of the eigensolver, a set of preconditioners were implemented: the *Jacobi* and *SSOR* preconditioners, the *ILUS incomplete factorisation preconditioner*, a *two-level hierarchical basis preconditioner* and a preconditioner based on algebraic multigrid (AMG). The Jacobi and SSOR methods are both inexpensive preconditioners. SSOR is the best method for preconditioning the Poisson systems arising in the projection and AD methods. The two-level hierarchical basis preconditioner exploits the hierarchical organisation of the finite-element basis functions. It represents a 2×2-block Gauss-Seidel method for which the second order diagonal block is only solved approximately. For the first-order diagonal block, we use an iterative method accelerated by an AMG preconditioner [11]. For small problems, a direct method like SuperLU can be used instead. The two-level hierarchical basis preconditioner has the advantage that the number of iterations is independent of the problem size. This fact makes it attractive for very large problems.

BENEFITS OF THE MIXED-LANGUAGE APPROACH

Improved usability and flexibility With PyFemax, the computation is steered by a small script instead of a conventional large parameter file. 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 specific needs. The scripting facilities give the user more flexibility for controlling the computation.

Extensibility Thanks to its object-oriented design, PyFemax can be easily extended. 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 concise. 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 implementation details.

DRAWBACKS OF MIXED-LANGUAGE APPROACH

Intricate 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 reduced. Our experiments indicate that the overall performance loss is at most 20%.

Lack of compile-time checks In Python all type checks are 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* [9, 6] it is possible to find some of these errors in Python code. The trouble with the tools presently available 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 undesirable. *PyChecker2*, which is currently under development, operates on the source code alone and will solve 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 [7]. *PyPAR* [8] is a more light-weight wrapper of the MPI library for Python. Another alternative is the *Python BSP* package [4, 5] 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 any of these approaches will lead to a successfully parallelised version of PyFemax.

APPLICATION TO THE COMET STRUCTURE

The unstructured grid enables a good approximation of complex geometries, such as the RF-structure of the COMET-cyclotron (Figs. 1, 2) for proton therapy. An accurate simulation of the electromagnetic fields and the coupling of the Dees is a challenging aspect of this structure. PyFemax was used for a much more detailed simulation than that described in [12] and enables the use of elements of quadratic instead of linear degree.

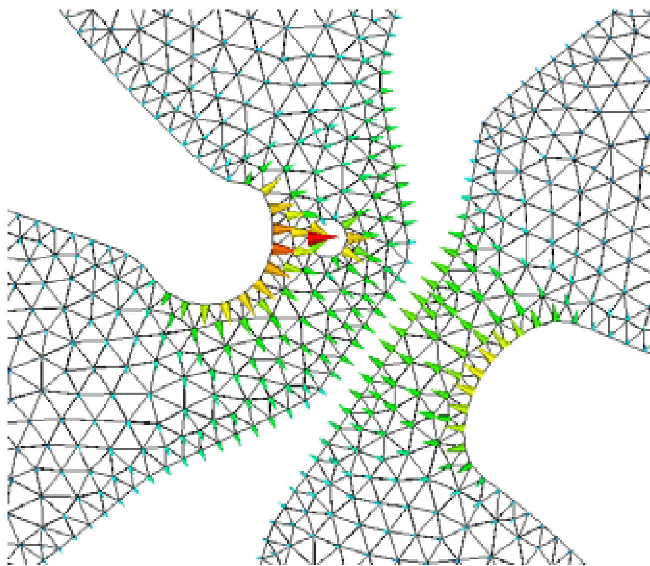

Fig. 2: Mesh and electric field of the fundamental mode in the beam plane in the central region of the COMET structure. The chimney of the ion source (circle in the centre) as well as the capacitive coupling and the galvanic coupling of the Dees are shown.

About 1.2 million tetrahedra are used leading to 10 million degrees of freedom. The simulation required 12 GBytes of memory and 31 hours of CPU-time on the HP Superdome at ETHZ for the calculation of the first five eigenmodes. The resonance frequencies were found at 73.18 MHz, 73.44 MHz, 73.69 MHz, 92.46 MHz and 129.97 MHz with a relative residual below 8.2×10^{-5}.

CONCLUSIONS

Numerical experiments already confirm the good performance of PyFemax. This motivates further extension of a powerful simulation tool. Postprocessing operations like path and surface integrals should be implemented for the calculation of RF-parameters such as the shunt impedance and the quality factor. Field interpolation could be used for exporting the field values into beam dynamics programs.

Improvement of the interface to pre- and post-processing tools such as NETGEN [1] for meshing and MayaVi [10] for visualisation should simplify the usage of PyFemax and increase the potential user community. Implementation of first-order Nédélec finite elements for fast and less accurate simulations with much less memory usage will make calculations of complex structures on small desktop computers possible.

Additional solver types for efficient eigenmode calculation in a given interval of a dense spectrum should be studied. This will enable the calculation of a set of eigenmodes as basis functions for the cavity field distribution.

An intelligent restart algorithm would help to save computation time, in cases where the mesh is deformed slightly in each iteration for geometry optimisation.

REFERENCES

[1] http://www.sfb013.uni-linz.ac.at/˜joachim/netgen/.

[2] P. Arbenz and Z. Drmač. On positive semidefinite matrices with known null space. *SIAM J. Matrix Anal. Appl.*, 24(1):132–149, 2002.

[3] R. 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.

[4] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 1998.

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

[6] C. Laird and K. Soraiz. Regular expressions: Syntax checking the scripting way. UNIX Review, April 2002. http://www.unixreview.com/.

[7] P. Miller. MPI Python. SourceForge project http://sourceforge.net/projects/pympi/.

[8] O. Nielsen. PyPAR - Parallel Python. http://datamining.anu.edu.au/˜ole/pypar/.

[9] N. Norwitz. PyChecker. SourceForge project http://pychecker.sourceforge.net/.

[10] P. Ramachandran. Mayavi: A free tool for CFD data visualization. In *4th Annual CFD Symposium*. Aeronautical Society of India, August 2001. http://mayavi.sourceforge.net/.

[11] S. Reitzinger and J. Schöberl. An algebraic multigrid method for finite element discretizations with edge elements. *Numer. Linear Algebra Appl.*, 9(3):223–238, 2002.

[12] L. Stingelin. Computational electrodynamics on the LINUX-cluster. PSI Scientific Report, 2001. VI.