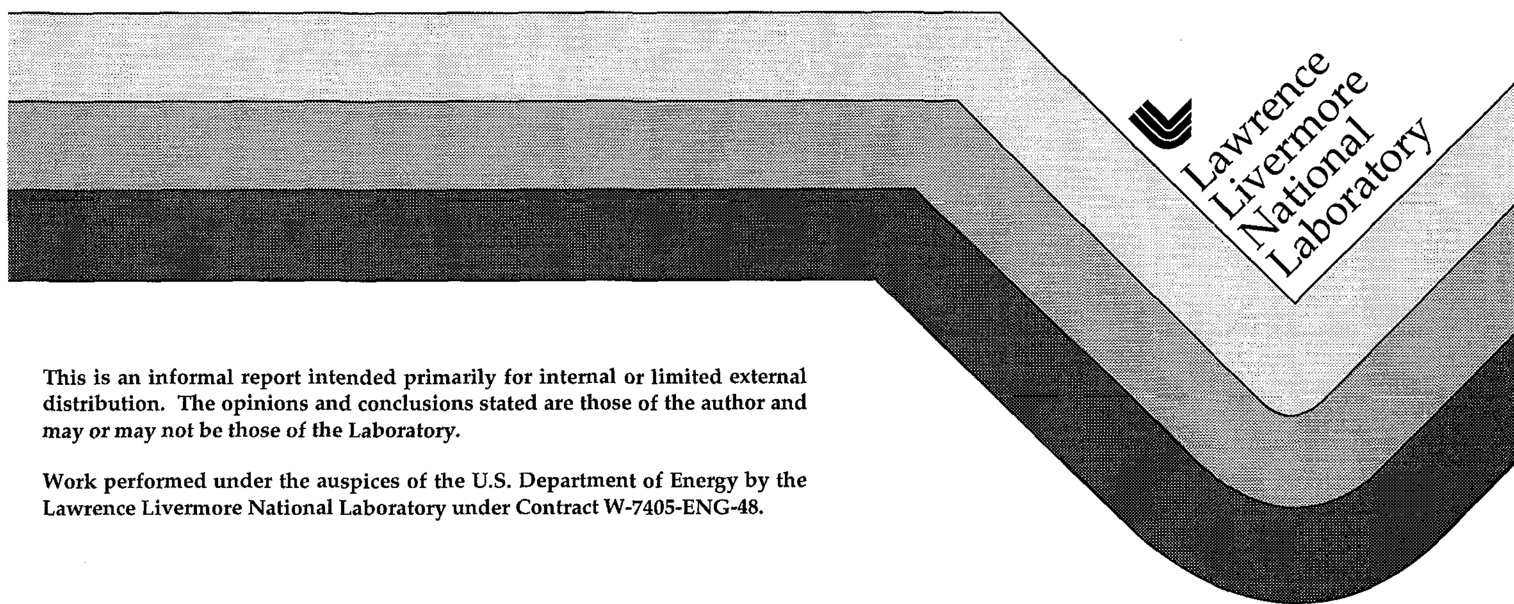
UCRL-ID-133262

An Object-Oriented Framework for Magnetic-Fusion Modeling and Analysis Codes

R. H. Cohen
T.-Y. Brian Yang

March 4, 1999



This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

An Object-Oriented Framework for Magnetic-Fusion Modeling and Analysis Codes

Ronald H. Cohen and T.-Y. Brian Yang

Magnetic Fusion Energy Program

Lawrence Livermore National Laboratory

1 Introduction

The magnetic-fusion energy (MFE) program, like many other scientific and engineering activities, has a need to efficiently develop complex modeling codes which combine detailed models of components to make an integrated model of a device, as well as a rich supply of legacy code that could provide the component models. There is also growing recognition in many technical fields of the desirability of steerable software: computer programs whose functionality can be changed by the user as it is run. This project had as its goals the development of two key pieces of infrastructure that are needed to combine existing code modules, written mainly in Fortran, into flexible, steerable, object-oriented integrated modeling codes for magnetic-fusion applications. These two pieces are (1) a set of tools to facilitate the interfacing of Fortran code with a steerable object-oriented framework (which we have chosen to be based on Python[1-3], an object-oriented interpreted language), and (2) a skeleton for the integrated modeling code which defines the relationships between the modules. The first of these activities obviously has immediate applicability to a spectrum of projects; the second is more focussed on the MFE application, but may be of value as an example for other applications.

2 Tool Development

For codes developed based on the Basis[4] system, the code developers provide interface description files which contain information about the variables and routines in the code. A tool in the Basis system called MAC can then be

used to generate the required glue functions that enable the Basis interpreter to access the variables in the code, and to invoke Fortran functions and subroutines. Although Basis system has been ported to many computer platforms, there are still some platforms, especially parallel computers, where Basis is not supported. In order to run their code on platforms where Basis is not available, LLNL Heavy Ion Fusion (HIF) program is the process of converting their Basis based code to a Python based code. As part of the conversion process, David Grote has developed a tool called PYMAC, which serves the same purposes as MAC, i.e., parsing the interface description files written for the Basis base code and generating the required glue functions, only in this case for the Python interpreter instead of the Basis interpreter.

Most of the Fortran legacy codes in the MFE community were not written for use with Basis; therefore, we could not use PYMAC directly in our project. Instead, we borrowed some techniques in PYMAC and developed our own tools to automate the Python-Fortran interface. Our tools consists of two sets of Python codes, one (PYCOMMON) for making Fortran common-block variables accessible to the Python interpreter, and the other (PYCFORTRAN) which enables the Python interpreter to invoke Fortran functions and subroutines.

2.1 PYCOMMON

The input to PYCOMMON are files containing standard Fortran-77 common-block declarations and variable-type declarations. The following is an example of input files to PYCOMMON:

```
      integer kion, kk, kj, curtype, njcur,diffeq_methd
      parameter (kj = 51, kion = 5, kk = kion + 4)
      real*8 u, usave, en, te, ti, rbp, ene, enesav, curden, etor,
     .       uav, uav0, eneav0, eneav1, curpar_soln
      common /soln/ u(kk,kj), usave(kk,kj), en(kj,kion), te(kj),
     .       ti(kj), rbp(kj), ene(kj), enesav(kj), curden(kj),
     .       etor(kj), uav(kk,kj), uav0(kk,kj), eneav0(kj),
     .       eneav1(kj), curpar_soln(kj), njcur, curtype,
     .       diffeq_methd
```

The output are files containing wrapper functions written in C. These output files can then be compiled and linked with the Fortran-77 code to

form a dynamically loadable Python module. When imported by Python, this Python module contain Python objects with attributes corresponding to the items in the Fortran common blocks. If the file above is named 'soln.i', the Python module will contain a Python object named 'soln' which is a proxy to all the parameters and variables in 'soln.i'. Parameters $kion, kk$, and kj are mapped into read-only attributes of the proxy object 'soln'. Attributes corresponding the the variables in the common blocks allow both read and write access. For example, the following Python statement:

```
>>> soln.njcur = 1
```

sets the variable 'njcur' in the common block to 1.

Attributes corresponding to array variables are references to Python objects of the 'PyArray' type, a built-in type defined in the Python Numeric module. The elements of the arrays can be manipulated through the built-in functionality of the 'PyArray' type. For example, the following Python statements:

```
>>> soln.te[:] = 0.2
>>> soln.u[0] = soln.te
```

first set all the elements of the one-dimensional array 'te' to 0.2, and then element-wise copy 'te' to the first row of the two-dimensional array 'u'. Notice that the indices of 'PyArray' objects start from 0; therefore, $soln.u[i, j]$ corresponds to $u(i+1, j+1)$ in Fortran.

2.2 PYCFORTRAN

The input to PYCFORTRAN are files containing standard declarations of Fortran-77 subroutines, functions, and variable-type, with an extension to specify arguments as input or output variables. The following is an example of such input files:

```
% ModuleName = an_example
% CommonBlocks =
% Fortran
    subroutine diff(a:in, diff_a:out, n:in)
```

```
    integer n
    real*8 a(n), diff_a(n-1)
% Fortran
    subroutine diff_x(a:in, diff_a:out, m:in, n:in)
    integer m, n
    real*8 a(m, n), diff_a(m-1, n)
% Fortran
    subroutine diff_y(a:in, diff_a:out, m:in, n:in)
    integer m, n
    real*8 a(m, n), diff_a(m, n-1)
```

The output of PYCFORTRAN are also files containing wrapper functions written in C. Again, such files can be compiled and linked with the Fortran-77 code to form dynamically loadable Python modules. In the input file for PYCFORTRAN, users can also specify names of PYCOMMON input files to be included as parts of the Python Module, so that the wrapper functions for the corresponding common blocks are included in the output of PYCFORTRAN. For example, if the second line in the above file were to be changed to:

```
% CommonBlocks = soln
```

The Python proxy object 'soln' in the example in the previous section would be in the Python module 'an_example'.

In addition to the files for wrapper functions, PYCFORTRAN also generates Python interface description files which are only documentations of the signatures of the Python functions and are neither compiled nor linked with the Python extension modules. If the above file is named 'an_example.pack', the corresponding Python interface description file generated by PYCFORTRAN will be named 'an_example.py.signature' containing the following information:

```
double diff_a[n-1] <- diff (double a[n], int n)
double diff_a[m-1, n] <- diff_x (double a[m, n], int m, int n)
double diff_a[m, n-1] <- diff_y (double a[m, n], int m, int n)
```

Notice that the arguments declared with the ':out' attribute do not appear in

the argument lists of the Python functions. Instead these arguments become the returned values of the routines. For array arguments, this also means that new 'PyArray' objects are created. Since the Fortran routine can be a function with a returned value and there can be more than one argument tagged with the attribute ':out', the Python functions needs to return a Python 'tuple' object. The order of the returned objects in the 'tuple' object is the same as the order they are declared in the Fortran routine. If the Fortran routine is a function, its returned value is the first element of the returned 'tuple' object. For a Fortran subroutine which has no argument with an ':out' tag the corresponding Python function returns 'None', a special Python object representing 'nothing'.

A Fortran scalar argument with a ':in' tag appears in the argument list but not in the returned tuple. Furthermore, the argument will not be modified at the Python level even when the Fortran function inadvertently modifies that argument. This is because the scalar argument being passed to thc Fortran function is a copy of the Python scalar used as the argument. A Fortran scalar argument without an input/output tag appears in both the argument list and the returned tuple, because the argument will not be modified and the updated value has to be accessed as an element in the returned tuple. A Fortran array argument with an ':in' tag, or without a tag, appears in the argument list but not in the returned tuple. Beware that the ':in' tag has no effect on array arguments, i.e., they can still be modified by the function. This is because Fortran 77 has no way to prevent an argument from being modified, and we choose not to make a copy of the array before passing it to the Fortran routine.

The Fortran types supported by PYCFORTRAN so far are 'integer' and 'real*8', both scalars and arrays with dimensions no more than 5. Fortran integer scalars are mapped to Python 'int'-type objects, and Fortran 'real*8' scalars are mapped to Python 'float'-type objects. Fortran arrays are mapped to Python 'PyArray'-type objects. PyArray objects have another level of type system among themselves. Fortran integer arrays are mapped to 'PyArray_INT', and Fortran real*8 array are mapped to 'PyArray_DOUBLE'. By default, PyArray objects have row-major strides. However, the Python functions generated by PYCFORTRAN accept only PyArray objects with column-major strides because Fortran arrays have column-major strides. The Python function 'transpose' in the 'Numeric' module can be used to turn a PyArray object with row-major strides into one with column-major strides.

For example, if a Python function 'f(x)' expects the variable 'x' to be a 2x3 array with column-major strides, and if 'y' is a 3x2 row-major Python array, the following will be a legal statement:

```
>>> from Numeric import transpose
>>> f(transpose(y))
```

When Python functions generated by PYCFORTRAN return PyArray objects, the returned PyArray objects always have column-major strides.

3 An Object-Oriented Transport Code

The second part of the project is to dcsign the skeleton for an objcct-oriented transport code – the basic prototype for a tokamak modeling code. A key element in the design process is the development of a class hierarchy. A class, in a crude way, can be regarded a data structure plus functions that manipulate the data structure. In an object-oriented code, the logical components of the code are objects each of which is an instance of one class or another. Because the class hierarchy, which reflects the relationships between classes, determines how objects in the code interact with one another, it affects the stability of the code structure as the code continues to evolve to include more capability. A well designed class hierarchy can reduce the dependency among the components so that new features can be added to the code with minimum changes.

Recognizing that design of the class hierarchy is an important but difficult task that requires many iterations of trial and error, we first developed a first-generation class hierarchy for a prototype Python transport code. The reason for building the prototype in Python is because it is a interpreted language providing many flexible features that a compiled object-oriented language like C++ lacks. Moreover, Python can be used as a scripting language which allows code steering, one of the goals for this project.

In May 1998 the national Magnetic Fusion Energy Program launched the National Transport Code (NTC) Demonstration Project. We brought to the table our design concepts developed as part of our LDRD activity earlier in the year, and re-oriented them to be compatible with objectives of the NTC project: web-invocable, distributed web client, physics application, and

database server, and a distributed development team. Our efforts in this area, since May, have been supported by NTC and base-program funds, but have drawn heavily from our earlier experience. Currently the class structure takes advantage of the encapsulation methodology of object-oriented design such that any changes in transport models, methods of updating fields (solving PDE or reading from a data base), independent variables (toroidal flux or poloidal flux), and so on, can be achieved by changing a very small isolated part of the code. This makes the code stable and easy to maintain.

The Python transport code has been one of the keys elements for the NTC Project to successfully demonstrate that modern computer science techniques can be applied to the development of a transport code that performs non-trivial physics simulations. Although the NTC Project plans to migrate the physics code to C++, it is intended that the Python transport code will continue to play a significant role in testing new capabilities before they are implemented in the C++ code.

References

[1] Guido van Rossum has published four manuals for the Python language, available at http://www.python.org/doc. The www.python.org Web-site has a plethora of other references helpful in learning Python, as well as sources and some pre-compiled binaries.

[2] Mark Lutz, **Programming Python**, O'Reilly Press, Sebastopol, CA (1996).

[3] Aaron Watters, Guido van Rossum, James C. Ahlstrom, **Internet Programming with Python**, M&T Books, New York, NY (1996).

[4] P.F. Dubois, "Basis: Setting The Scientist Free", **16th IEEE Conference on Numerical Simulation of Plasmas**, Buffalo, NY (1989).