Driving and Extending Legacy Codes using Python

Neilen Marais
Department of Electrical and Electronic Engineering
University of Stellenbosch, South Africa
email: nmarais@gmail.com

Abstract: Software Engineering aspects in Computational Electromagnetics (CEM) are becoming more important as the complexity of CEM codes continue to increase. Object orientated programming (OOP) methods promise to alleviate the challenges posed by more complex software systems, but offers little help for legacy codes. Python, an object-oriented very high level language (VHLL), can be used to extend legacy codes. It provides the dual benefit of a very productive programming environment and of enabling legacy codes to be migrated to object orientated designs with low risk. The application of this method is described in the context of eMAGUS, a microwave Finite Element Method code.

Keywords: Object Orientated Design, Python, Fortran, Finite Element Method

I. Introduction

The purpose of Computational Electromagnetics (CEM) is the numerical solution of EM problems, but the end product of CEM development itself is software. As such, good software engineering principles should be an important tool in the CEM worker's toolbox. However, the mathematical and electromagnetic insight required for CEM has traditionally overshadowed the importance of the software aspects.

On the other hand, the software complexity of CEM codes continues to increase thanks to several factors. The solution of progressively more general problems require complex software interactions. The full-wave solution of ever larger problems require more sophisticated field modelling and more effective solution strategies. The unsustainability of historic single CPU performance growth (Fig. 1)[1] is forcing the adoption of parallel methods, along with the increased complexity they bring. At the same time, users are demanding better solution accuracy while geometries are simultaneously becoming more complex, necessitating more flexible geometry handling. These trends have lead to software engineering aspects becoming quite important to CEM workers.

In this paper, the use of Python to extend the microwave Finite Element Method (FEM) code, eMAGUS, is discussed. eMAGUS and its predecessors has been developed by the Computational Electromagnetics Group (CEMAGG) [1] at the University of Stellenbosch over almost a decade, and is coded in Fortran 90. It is shown how existing code and data structures can be utilised and extended with Python, and what the advantages are.

[1]Last data-point added by the authors.

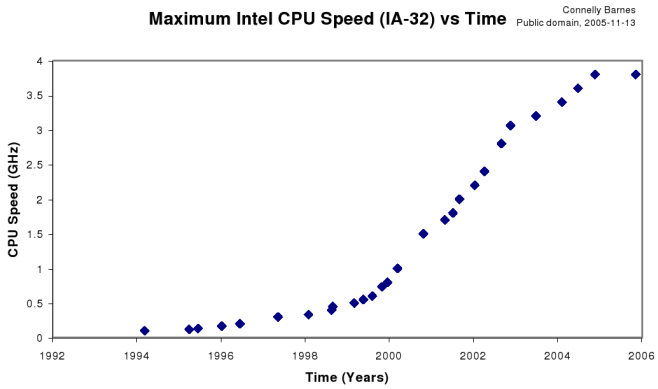

Fig. 1. Maximum Intel CPU speed vs. Time

II. CEM Development Practice

Most numerical computation has traditionally been done in Fortran, particularly Fortran 77 (F77), which is arguably archaic as far as language features are concerned. More recently, Fortran 90 (F90), C and C++ have become more popular. While F90 and C both have their respective advantages, neither are well suited to the use of more modern software paradigms. C++ is certainly capable in this regard, but presents a rather steep learning curve.

Regardless of a particular language's merits, it is of no use if existing code is written in another language. Investments in old code are often considerable. Even if a new language were guaranteed to boost productivity significantly, the cost, and perhaps more significantly, risk involved in either porting or re-implementing a code can be very unattractive.

The use of very high-level languages (VHLL) such as Python [2], Ruby [3] and Perl [4] have shown promise as a means of moving existing codes forward without taking the risk of a full re-implementation. At first glance, these languages may seem unattractive for numerical computation, since VHLL execution speed may be too slow. However, most VHLLs can interface with compiled libraries for speed critical loops, while some provide additional means of speeding up critical code sections.

III. Modern Development Methods

Software development is challenging at best, but using appropriate techniques can ease the process. Compared to older procedural techniques, the appropriate use of Object Oriented Programming (OOP) [5] can make software easier to

maintain and extend; this becomes more apparent as software complexity increases.

Test Driven Development (TDD) [6] is a young technique that is rapidly gaining popularity. While testing has always been part of software development, it is usually seen an additional step. In TDD, testing is an integral part of the design and coding process.

The basis of TDD is to develop fine grained automated unit tests [7] before writing the actual implementation. Test development thus forms part of the design process, often preventing design mistakes at the earliest possible moment. Implementation code is written and the tests continually re-run until the tests are satisfied. Higher level functional tests can also be added as the need arises.

Tests and new code are usually added in an incremental fashion, interspersed with code and design refactoring. Refactoring is critical to keep code manageable as its complexity grows, but carries the risk of introducing new bugs. Unit tests are invaluable when refactoring since they show up new coding errors almost as soon as they occur.

TDD may seem inefficient since it introduces the extra work of developing tests, but the efficiencies gained from the reduction in subtle bugs, improved design and ease of refactoring handsomely repays the time invested in writing tests. Since test development is part of the design process, the amount of time spent doing traditional design is also reduced.

To be effective, both OOP and TDD depend critically on language or tool-chain support. While it is possible use OOP concepts in a language with no specific support, it would be error prone and involve much extra work. Similarly TDD is infeasible without tools that automate the tests and make it easy to locate test failures.

IV. Python and Numeric Computing

Python is a modern object orientated VHLL featuring exceptionally clear syntax and powerful language features. It is both easy to learn, and a very productive environment for expert users. Features like automatic memory management, high level data types and extensive, freely available software libraries allow very rapid development, and much shorter code than that of an equivalent program written in e.g. C++ or Fortran. This combined with its free and open source licencing makes Python attractive for most any task.

Python has always had a C language API, allowing the extension of Python modules such that it is transparent (i.e. appears to be part of the Python environment) to the Python user, at the expense of extra work for the C programmer. More recently, automatic Python wrapper generators such as SWIG [8] and F2PY [9] have made it quite painless to extend Python using new or existing code written in compiled languages such as C, C++ and Fortran.

Python's ability to wrap existing code led to standard computational libraries such as LAPACK [10] and FFTPACK [11] along with fast array and matrix handling becoming available through the Numerical Python [12] package for Python. The Numerical Python work has since been merged and extended by the SciPy [13] project. An active developer and user community has built around SciPy.

```
MODULE DATA
  IMPLICIT NONE

  REAL, DIMENSION(5) :: test_arr

END MODULE DATA

MODULE prog
  USE data
  IMPLICIT NONE
CONTAINS

  SUBROUTINE init_data()
    ! Initialise the array
    test_arr = (/1,2,3,4,5/)
  END SUBROUTINE init_data

  SUBROUTINE process_data(factors, n)
    REAL, DIMENSION(n), INTENT(in) :: factors
    INTEGER, INTENT(in) :: n
    INTEGER :: i
    DO i = 1,n
       test_arr = test_arr - (-1)**i &
                  * test_arr**factors(i)
    END DO
  END SUBROUTINE process_data

END MODULE prog
```

Fig. 2. Example F90 code

Building on SciPy, the iPython [14] interactive shell and the Matplotlib [15] Pylab mode make for comfortable interactive use in the style of MATLAB (R), while keeping the full capabilities of a general language and its libraries available. The recent addition of iterative and direct sparse matrix solvers to SciPy makes it well suited for FEM codes. Worked examples of several real numerical problems and links to other useful Python software is on the website of the Python4Science Workshop [16] that was recently held at Stellenbosch.

V. Wrapping Fortran 90

F2PY is an automated Python wrapper generator for F77 and F90. It has comprehensive support for F77 language features. It can automate many of the low level details F77 subroutine calls usually require such as determining the size of arrays. It also supports F90 module data and subroutines, but does not have direct support for derived types and assumed size dummy arguments at this time. Both limitations can be overcome by the addition of some simple wrappers to the Fortran code.

Assuming iPython and F2PY are set up, and that Intel's Fortran compiler is being used in a UNIX environment, the code in Fig. 2 can be compiled into a Python module by issuing:

```
$ f2py --fcompiler=intel -m testmod \
             -c test_data.f90 test_prog.f90
```

The `-m test` flag specifies the Python module name as `testmod`. The `-c test_data.f90 test_prog.f90` flag specifies the Fortran source files to be wrapped and compiled. Now we can manipulate the data and call the Fortran routines from the iPython shell.

```
$ ipython -pylab
...
  Welcome to pylab, a matplotlib-based Python
  environment.
  For more information, type 'help(pylab)'.
In [1]:import testmod
In [2]:testmod.data.test_arr
Out[2]:array([ 0., 0., 0., 0., 0.])
In [3]:testmod.prog.init_data()
In [4]:testmod.data.test_arr
Out[4]:array([ 1., 2., 3., 4., 5.])
In [5]:testmod.prog.process_data([1,2])
In [6]:testmod.data.test_arr
Out[6]:array([ -2.,-12.,-30.,-56.,-90.])
```

`In [x]` represents the x'th user input and `Out [x]` the x'th output from the iPython shell. `In [1]` loads the Python module created by F2PY. `In [2]` causes the value to be printed. Note how a new namespace is assigned for each module, in contrast to the F90 norm of importing all symbols into the current namespace. `In [5]` shows how the length parameter is automatically passed to the Fortran routine.

A. Numpy Arrays vs. Python Lists

Python has a built in list type that is optimised to store dynamic lists of heterogeneous types. Python lists are incredibly useful, but are not well suited for numerical computation; they use memory and CPU inefficiently when they are used to store homogeneous collections and support multi-dimensional arrays only as lists of lists.

The `numpy.ndarray` type is optimised for numerical computation involving homogeneous multi dimensional arrays such as matrices. F2PY uses `ndarrays` to encapsulate the Fortran data structures. Python lists and `ndarrays` behave very similarly with some notable exceptions described below, where `ndarray` behaviour is designed to benefit numerical computing above general computing.

```
$ ipython
...
In [1]:import numpy
```

Here we have imported the `numpy` module. By prepending "`numpy.`", all `numpy`'s functions, variables and classes can be accessed.

```
In [2]:nd_array=numpy.array([1,2,3])
In [3]:p_list = [1,2,3]
```

Multiplying an `ndarray` by a scalar `n` does element-wise multiplication, whereas it replicates a `list` `n` times:

```
In [4]:nd_array*3
Out[4]:array([3, 6, 9])
In [5]:p_list*3
Out[5]:[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Adding two `ndarrays` result in element-wise addition, while adding two `lists` concatenates them:

```
In [6]:nd_array+nd_array
Out[6]:array([2, 4, 6])
In [7]:p_list+p_list
Out[7]:[1, 2, 3, 1, 2, 3]
```

`ndarray` subtraction, multiplication and division works like addition, while `lists` do not support those operations:

```
In [8]:nd_array-nd_array
Out[8]:array([0, 0, 0])
In [9]:p_list-p_list
TypeError: unsupported operand type(s) for -:
                                  'list' and 'list'
```

`ndarrays` support array addressing, `lists` not:

```
In [10]:nd_array[[0,2]]
Out[10]:array([1, 3])
In [11]:p_list[[0,2]]
TypeError: list indices must be integers
```

However, `lists` support arbitrary data types:

```
In [12]:p_list=[1, 3.14159, "hello"]
```

while `ndarrays` must be homogeneous. We specified 8-bit integers, but strings and floating point numbers cannot be converted to integers without loss of precision:

```
In [13]:nd_array=numpy.array([1, 3.14159, "hello"],
                             numpy.int8)
TypeError: an integer is required
```

There is a fundamental difference between how Python lists and `ndarrays` treat multi-dimensional arrays. Such arrays are trivially handled with an `ndarray`:

```
In [28]:nd_matrix=numpy.array([[11, 12],
                               [21, 22]])
In [29]:nd_matrix[0,0]
Out[29]:11
In [30]:nd_matrix[1,1]
Out[30]:22
# Second Column
In [31]:nd_matrix[:,1]
Out[31]:array([12, 22])
# Second Row
In [32]:nd_matrix[1,:]
Out[32]:array([21, 22])
```

Python lists are always one-dimensional, but can store lists as elements. A matrix can be stored as:

```
In [32]:p_list_matrix = [[11, 12],
                         [21, 22]]
```

Repeated application of the `[]` operator can be used to obtain any element:

```
In [24]:p_list_matrix[0][0]
Out[24]:11
In [26]:p_list_matrix[1][1]
Out[26]:22
```

However, one runs into trouble trying to obtain a column:

```
In [27]:p_list_matrix[:][1]
Out[27]:[21, 22]
```

Instead of returning the second column as expected, it returns the second row. This is because `p_list_matrix[:]` refers to the whole outer list, and its second element is a list containing the second row.

VI. Extending eMAGUS

eMAGUS (formerly known as FEMFEKO [17, pp. 350]) is a research microwave FEM code developed by the CEMAGG group at Stellenbosch University over the last decade. It is chiefly used for research in FEM methods, and is coded in F90. eMAGUS contains many well tested routines that can be re-used when trying new techniques, though its code structure is somewhat specialised to frequency domain work.

Applying OOP and TDD to eMAGUS is becoming increasingly attractive as its complexity increases, but F90 lacks the language features to do so cleanly and easily. One option would be a rewrite from scratch, but such a move is always risky. F90 wrappers for Python provide a way of moving forward without the associated risk.

Python language features can be used to encapsulate the existing program structure in an OO framework piece by piece. New code can be written in Python using the OO structure, or in F90 and wrapped. New code can even be written in another compiled language (e.g. C++) using SWIG to integrate it with the main Python program. If replacing existing F90 routines with Python equivalents become desirable, it is a boon to have the known-good F90 implementations to compare results with.

Performance critical code sections can be kept in F90, or written in another compiled language at only a small cost in flexibility. Since the overall program structure would be defined in Python, the effect of specialised compiled modules on the overall design will be localised. Another possibility is to treat the Python code of performance critical sections using tools like Weave [18] and Pyrex [19].

Following this method, the code is gradually moved to a flexible and easily extensible OO design. Useful code built up over the years can be re-used, and at no time will the code be unusable. This means new development and research could happen in parallel to the modernising of the code structure, avoiding the case of two divergent code trees. Using a distributed version control system such as Bazaar-ng [20] or GNU Arch [21] to keep track of simultaneous development may also be helpful.

Wrapping eMAGUS with F2PY provides several immediate benefits:

- Python's extensive use of namespaces allows the re-use of eMAGUS routines without dictating the structure of the new code.
- New methods are developed much faster owing to the use of a VHLL.
- Interactive simulation and plotting with scripting capabilities become a possibility.
- Readily available Python libraries can be used for many purposes, like plotting, model visualisation and data IO in various formats.
- The availability of several matrix solution modules with minimal extra coding requirements.

The primary author is undertaking research into Finite Element Time Domain (FETD) methods [22]. F2PY wrappers are used to access the mesh processing and curl-conforming basis function and field-reconstruction routines from eMAGUS, while new code is written using the core SciPy array data-types and numerical libraries.

VII. Example of Wrapping Existing F90 FEM Codes

The central data-structure in a FEM code tends to be the mesh. In Fortran codes the mesh is often represented by a series of arrays representing the node coordinates, the node indices that define the elements and similar arrays for other mesh entities such as faces and edges. Using Python proxy objects, an object oriented face can be put on the mesh without modifying the underlying Fortran code.

```
MODULE mesh
  ! The x,y,z coordinates of each node in the mesh
  REAL(8), DIMENSION(:,:), ALLOCATABLE &
                           :: node_coordinates
  ! The 4 node indices per element that define all
  ! the mesh elements
  INTEGER, DIMENSION(:,:), ALLOCATABLE &
                           :: element_nodes
CONTAINS
  SUBROUTINE init_mesh()
    ALLOCATE(node_coordinates(3,5))
    node_coordinates(:,1) = (/-0.5,  0.5, -0.5/)
    node_coordinates(:,2) = (/ 0.5,  0.5,  0.5/)
    node_coordinates(:,3) = (/ 0.5, -0.5, -0.5/)
    node_coordinates(:,4) = (/-0.5, -0.5,  0.5/)
    node_coordinates(:,5) = (/-0.5, -0.5, -0.5/)
    ALLOCATE(element_nodes(4,2))
    element_nodes(:,1) = (/1, 2, 3, 4/)
    element_nodes(:,2) = (/1, 3, 4, 5/)
  END SUBROUTINE init_mesh
END MODULE mesh
```

Fig. 3. Simple Fortran Mesh Datastructure: mesh.f90

A. Fortran Data Representation

A simple example of wrapping a basic tetrahedral mesh element will be developed. A mesh element is defined by its four vertex node coordinates. A Fortran representation of this data is shown in Fig. 3

The code snippet `element_nodes(:,1)` refers to the x, y and z coordinates of mesh node 1, and `element_nodes(:,2) = (/1, 3, 4, 5/)` to the four global node indices that define element 2. In a real code, the mesh would be read from a file, but for this example it is initialised to a simple two-element mesh using the `init_mesh()` subroutine.

B. Python Proxy Object

Fig. 3 is wrapped into a Python module called `f90modules` by the method described in Section V. Within Python, the Fortran data arrays are now accessible as `f90modules.mesh.node_coordinates` and `f90modules.mesh.element_nodes`.

The desired design is show in in Fig. 4 All the information

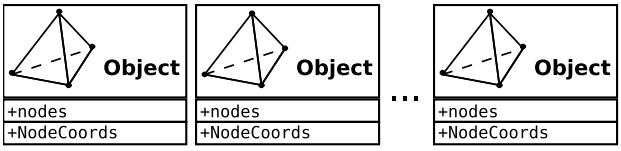

Fig. 4. Schematic Representation of OO Mesh Design

pertaining to an individual element is collected together in an element object. The mesh is represented as an ordered list of element objects. The `MeshElement` class in Fig. 5 pulls the separate data arrays together and allows the elements to be addressed one by one.

```
class MeshElement(object):
    def __init__(self, nodeCoords, elementNodes):
            self.index = 0
            self.numberof = len(elementNodes)
            self._elementNodes = elementNodes
            self._nodeCoords = nodeCoords

    @property
    def nodes(self):
        return self._elementNodes[self.index] - 1

    @property
    def nodeCoords(self):
        return self._nodeCoords[self.nodes]
```

Fig. 5. MeshElement Proxy Class

The statement `class MeshElement(object):` derives `MeshElement` from the built in Python `object` type that defines standard object behaviour. Note the indentation of `def __init__(...):`. Python groups code blocks by indentation rather than braces or begin/end clauses. A function or method definition is started with `def`. The `__init__()` method initialises a new object instance to a known state.

The first parameter of any method, called `self` by convention, is always passed a reference to the current object. `self` is analogous to the `this` keyword in C++ and Java. In C++ and Java a method has implicit access to its object's whole namespace, making the use of `this` optional in most cases. In Python an explicit `self` reference is always required. This may seem onerous but in practice is effective at reducing the occurance of subtle bugs.

Instantiating a `MeshElement` object as

```
meshElement = MeshElement(
    node_coordinates.transpose(),
    element_nodes.transpose())
```

results in the `__init__(self, nodeCoords, elementNodes)` method being called with the `self` parameter automatically bound to the new instance. The `transpose()` method is used due to Fortran's column major storage convention; in Fortran programs it is more natural to loop over multi-dimensional arrays by varying the first array index the fastest, therefore the Fortran program stores one node coordinate per column in `node_coordinates` and one element's node indices per column in `element_nodes`.

Python uses the C convention of row major storage which means that multi-dimensional arrays are usually looped over by varying the last index fastest. The `transpose()` method does not copy the array, but returns a transposed "view" of the array. This allows more natural (in Python) row-based data access to be used without any overhead.

Also note that the `meshElement` instance does not copy the `node_coordinates` and `element_nodes` arrays, since Python, like Fortran, passes values by reference. If a copy is desired, the `copy()` methods of `node_coordinates` and `element_nodes` can be called.

By setting `meshElement.index`, each mesh element can be accessed in turn. Python always uses 0 based indexing, so setting `meshElement.index` to 0 accesses the first element, 1 the second and so forth.

We import the `MeshElement` class from the `MeshWrap.py` file, and the Fortan `mesh` module from the Fortran wrappers. The `import` statements are explained in Section VII-F.

```
$ ipython
...
In [1]:import MeshWrap
In [2]:from f90modules import mesh
```

The Fortan mesh is initialised by calling `init_mesh()`, and a copy of `MeshElement` instantiated using the mesh data from the Fortran mesh.

```
In [3]:mesh.init_mesh()
In [4]:meshElement=MeshWrap.MeshElement(
   ...:mesh.node_coordinates.transpose(),
   ...:mesh.element_nodes.transpose())
```

By setting `meshElement.index` to 0, we can access the first mesh element,

```
In [5]:meshElement.index=0
In [6]:meshElement.nodes
Out[6]:array([0, 1, 2, 3])
In [7]:meshElement.nodeCoords
Out[7]:array([[-0.5,  1. , -0.5],
              [ 0.5,  0.5,  1. ],
              [ 1. , -0.5, -0.5],
              [-1.5, -0.5,  0.5]])
```

and setting `meshElement.index` to 1, we access the second.

```
In [8]:meshElement.index=1
In [9]:meshElement.nodes
Out[9]:array([0, 2, 3, 4])
In [10]:meshElement.nodeCoords
Out[10]:array([[-0.5,  1. , -0.5],
               [ 1. , -0.5, -0.5],
               [-1.5, -0.5,  0.5],
               [-0.5, -1.5, -0.5]])
```

C. Proxy Object Explained

Fortran array indices are 1-based by default, i.e. `fortran_array(1)` refers to the first element of the array, whereas Python indices are always zero-based, meaning that when wrapping Fortran codes, indices such as the element node numbers cannot be shared directly. Since we do not want to modify the Fortran data, the `-1` is added in the `nodes()` method of `class MeshElement`.

In the example `meshElement.nodes()` and `meshElement.nodeCoords()` methods were used without `()` as if they were data members rather than method calls. This is thanks to the `@property` statement preceding the method definition:

```
    @property
    def nodes(self):
        return self._elementNodes[self.index] - 1
```

The `@` notation forms what is called a decorator in Python. The `@property` decorator turns the `nodes()` method into a getter function for the data attribute `nodes`. This also has the side effect of making `nodes` read only, since no setter function is specified. While not treated here, it is also possible to define a setter function that could be used to convert between zero and one based indices.

```
class MeshElementList(object):
    def __init__(self, element):
        self.element = element
        self.numberof = element.numberof

    def __getitem__(self, index):
        self.element.index = index
        return self.element

    def __len__(self):
        return self.numberof

    def __iter__(self):
        for i in xrange(self.numberof):
            self.element.index = i
            yield self.element
```

Fig. 6. List Proxy Code

D. Python List Proxy

While the proxy object described in Section VII-B is directly usable, manually updating the index is tedious and error-prone. Fortunately it is very easy to overload the Python list syntax `list[index]`. The `MeshElementList` class show in Fig. 6 achieves this.

The `__init__()` method should self explanatory. Defining a `__getitem__()` method allows any object to handle the Python subscript operator `[]`. Python simply passes whatever was inside the square brackets into the `__getitem__()` method. The `__len__()` method is called whenever the length of an object is requested as `len(object)`. This allows:

```
In [11]:meshList = MeshWrap.MeshElementList(
                   meshElement)
In [12]:meshList[0].nodes
Out[12]:array([0, 1, 2, 3])
In [13]:meshList[1].nodes
Out[13]:array([0, 2, 3, 4])
In [14]:len(meshList)
Out[14]:2
```

The `__iter__()` method allows Python's looping constructs to iterate over the elements without explicitly indexing them. With `__iter__()` defined to return an appropriate iterator, we can write:

```
In [15]:for el in meshList:
   ....:    print el.nodes
   ....:
[0 1 2 3]
[0 2 3 4]
```

The `MeshElement` class would also need an `__iter__()` method to handle looping over a subset of the elements. Iterators are a very attractive Python language feature; the reader is referred to the Python documentation [23], [24] for more details.

E. Extending the Proxy Object

The basic mesh proxy class can easily be extended using standard Python class inheritance. The `MyMeshElement` class shown in Fig. 7 is derived from the `MeshElement` class. It inherits all the attributes and methods from `MeshElement` class, and is extended by adding the `volume()` method. We can still use the original `MeshElementList` to hold `MyMeshElement`, since the behaviour inherited from `MeshElement` satisfies `MeshElementList`'s requirements:

```
In [20]:import MyMeshWrap
In [24]:myMeshElement = MyMeshWrap.MyMeshElement(
....:mesh.node_coordinates.transpose(),
....:mesh.element_nodes.transpose())
In [28]:myMeshList=MeshWrap.MeshElementList(
                    myMeshElement)
In [29]:myMeshList[0].volume()
Out[29]:1.0625
In [30]:myMeshList[1].volume()
Out[30]:0.625
```

F. Code Organisation

The code as presented is organised in three files: `mesh.f90` (Fig. 3), `MeshWrap.py` (Fig. 8) and `MyMeshWrap.py` (Fig. 7). The Python `import` statement loads code from other files as modules. Each module has its

```
import MeshWrap
from numpy import linalg
class MyMeshElement(MeshWrap.MeshElement):
    def volume(self):
        el_coords = self.nodeCoords
        matr = [el_coords[0]-el_coords[1],
                el_coords[1]-el_coords[2],
                el_coords[2]-el_coords[3]]
        return abs(linalg.det(matr))/6
```

Fig. 7. MyMeshElement source, extending MeshElement through inheritance: MyMeshElement.py

```
class MeshElement(object):
    def __init__(self, nodeCoords, elementNodes):
            self.index = 0
            self.numberof = len(elementNodes)
            self._elementNodes = elementNodes
            self._nodeCoords = nodeCoords

    @property
    def nodes(self):
        return self._elementNodes[self.index] - 1

    @property
    def nodeCoords(self):
        return self._nodeCoords[self.nodes]

class MeshElementList(object):
    def __init__(self, element):
        self.element = element
        self.numberof = element.numberof

    def __getitem__(self, index):
        self.element.index = index
        return self.element

    def __len__(self):
        return self.numberof

    def __iter__(self):
        for i in xrange(self.numberof):
            self.element.index = i
            yield self.element
```

Fig. 8. MeshElement and MeshElementList in MeshWrap.py

own namespace. When using `import SomeFile`, the file `SomeFile.py` is read. The definitions of `SomeFile.py` is accessed by prepending "`SomeFile.`" to their names. Using the form `from SomeFile import name` imports `name` into the current namespace. This allows direct access to `name` without "`SomeFile.`" being prepended.

VIII. OO Wrapping advantages

The advantages claimed for OOP often seem nebulous without concrete examples. The wrappers discussed here are fairly simple, but can serve to demonstrate some OO features.

A. Abstraction

Abstraction attempts to separate the abstract properties of a piece of data or function from its concrete implementation details. The rationale is twofold: by concentrating on only the necessary abstract details when using an object, the programmer is freed from considering incidental details and can work at a higher level; code using only the abstract properties of a object is insulated from future changes in the concrete implementation of this object.

In the `MeshElement` class the element's node coordinates is an abstract interface. In this case the node coordinates are stored in a separate list and referenced by node indices, however users of the `nodeCoords` property are insulated from this detail, leaving them free to concentrate one more important matters.

B. Encapsulation

Encapsulation attempts to hide the design decisions of a computer program that are most likely to change behind a stable interface. Users of the stable interface are thereby insulated when the encapsulated design decisions change. Deciding how to encapsulate often goes hand in hand with choosing abstractions.

In Section VII-C we saw that it was necessary to turn the one-based Fortran indices into zero-based indices for use in Python. The use of zero- or one-based indices is an arbitrary choice controlled by external factors. By fixing the Fortran data to appear zero-based in the `MeshElement.nodes` property, this choice is encapsulated.

The `MeshElement.nodeCoords` property used as an example of abstraction in Section VIII-A is also an example of encapsulation. If an element class chooses to store coordinates directly, or even to calculate them based on e.g. some mesh deformation, users of `nodeCoords` are not affected at all.

C. Inheritance

Inheritance is the process of constructing a new class by *inheriting* behaviour and attributes from a pre-existing class. The new class *derives* from the original *base* class, and is therefore called a *derived* class. This is an effective way to reuse code by factoring it into a base class that more specialised classes derive from.

The `MyMeshElement` class derives from the original MeshElement class, extending it with a `volume()` method. Other classes could in turn inherit from the `MyMeshElement` class to add more functionality. By using the encapsulated `nodeCoords` interface, the `volume()` method is also shielded from the concrete implementation of the node coordinate storage.

D. Other Advantages

Wrapping existing code allows it to benefit from a number of Python's general strengths, such as its extensive libraries and the ease of developing new features in Python. Python includes a testing framework in its standard libraries, making Test Driven Development(TDD) methods mentioned in Section III practical. The use of TDD can be extended even to Fortran code development by combining a Python test framework with automatic F2PY wrapping of the Fortran code.

IX. Conclusion

With the help of automatic wrapper generators, it is easy to access compiled code from within Python. Using F2PY to wrap an existing Fortran 90 microwave FEM code and the language features of Python, an object oriented interface is layered on the original code. In addition to allowing code extensions to benefit from object orientation and the syntactic efficiency of Python, other advantages accrued from the environment include: Interactive use facilitating experimentation; easy employment of test driven development; the availability of rich numerical, plotting and visualisation libraries; Python's strength as a general purpose language which is very useful when interacting with the outside world. Additionally, the risk of throwing away old code and rewriting from scratch is avoided by wrapping it.

References

[1] "University of Stellenbosch Computational Electromagnetics Group home page," Available: http://research.ee.sun.ac.za/cem/index.html.

[2] G. van Rossum *et al.*, "Python," Available: http://www.python.org.

[3] Y. Matsumoto *et al.*, "Ruby," Available: http://www.ruby-lang.org/en/.

[4] L. Wall *et al.*, "Perl," Available: http://www.perl.org/.

[5] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. Addison-Wesley Professional, 1993.

[6] "Test-driven development," Available: http://en.wikipedia.org/wiki/Test_driven_development.

[7] "Unit test," Available: http://en.wikipedia.org/wiki/Unit_test.

[8] D. Beazley *et al.*, "Simplified Wrapper and Interface Generator (SWIG)," Available: http://www.swig.org/index.html.

[9] P. Peterson, "F2PY: Fortran to Python interface generator," Available: http://cens.ioc.ee/projects/f2py2e/.

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA, or http://www.netlib.org/lapack/lug/lapack_lug.html: Society for Industrial and Applied Mathematics, 1999.

[11] P. N. Swarztrauber, "FFTPACK," Available: http://www.netlib.org/fftpack/.

[12] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, *et al.*, *Numerical Python*, http://numeric.scipy.org/numpydoc/numdoc.htm.

[13] "SciPy - Scientific tools for Python," Available: http://www.scipy.org/.

[14] F. Pérez, "iPython: An Enhanced Interactive Python shell," Available: http://ipython.scipy.org/.

[15] J. Hunter *et al.*, "Matplotlib," Available: http://matplotlib.sourceforge.net/.

[16] F. Prez and S. van der Walt, "Python4Science Workshop, University of Stellenbosch," Apr. 2006, Available: http://mentat.za.net/py4science/.

[17] D. B. Davidson, *Computational Electromagnetics for RF and Microwave Engineering*. Cambridge, UK: Cambridge University Press, 2005.
[18] E. Jones, "Weave," Available: http://www.scipy.org/documentation/weave/.
[19] G. Ewing, "Pyrex - a Language for Writing Python Extension Modules," Available: http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.
[20] "Bazaar-NG, next-generation distributed revision control," Available: http://bazaar-vcs.org/.
[21] T. Lord, "GNU Arch," Available: http://www.gnuarch.org/.
[22] J.-F. Lee, R. Lee, and A. Cangellaris, "Time-domain finite-element methods," *IEEE Trans. Antennas Propagat.*, vol. 45, no. 3, pp. 430–442, Mar. 1997.
[23] G. van Rossum *et al.*, "Python Library Reference: Iterator Types," Available: http://www.python.org/doc/2.4.3/lib/typeiter.html.
[24] ——, "What's New in Python 2.2: Pep 255: Simple Generators," Available: http://www.python.org/doc/2.2.2/whatsnew/node5.html.