A short introduction to F2PY

Pierre Schnizer

November 10, 2002

Abstract

For long times and even nowadays FORTRAN is the language of choice to implement CPU intensive tasks. F2PY offers capabilities to automatically wrap arbitrary FORTRAN routines. In this tutorial this process is illustrated by means of a simple example. Further points are described where f2py could be mislead.

1 Introduction

Since the introduction of FORTRAN many solvers have been implemented dealing with various numerical problems (See e.g. http://www.netlib.org). Even nowadays many projects are based on FORTRAN due to its excellent numerical capabilities. Analysing a typical problem one finds that $\approx 10\%$ of the Code account for $\approx 90\%$ of the time. Developing programs in interpreted languages was found to be faster than in compiled languages. Since *Guido van Russum* implemented Python many packages were added due to its extensibility. In 1999 *Pearu Peterson* started to develop a tool which allows to wrap FORTRAN Routines similar as SWIG does for C. f2py uses a mixed language approach, writing an C interface to convert the python data to C structures, and FORTRAN wrappers, when the C structures can not be passed to the FORTRAN routine directly.

In this article I illustrate the necessary steps to wrap a FORTRAN routine by means of a simple routine followed by an callback illustration.

2 First steps

As firs example a sum implemented in FORTRAN is shown. So all this routine does is

$$b = \sum_{i=1}^{n} a_i \,. \tag{1}$$

The corresponding code is found in the file "simple/dsum.f".

```
1       subroutine dsum (b, a, n)
2       double precision a(n), b
3       b = 0.0D0
```

```
4       do 100 i =1,n
5          b = b + a(i)
6   100 continue
7       end
```

f2py wraps this code so it can be accessed from python. > denotes the shell prompt in this document. The first step is to ask f2py to write the signature of the routine to a description file. So in the directory "simple" I type at the shell prompt:

```
>f2py -m dsum -h dsum.pyf dsum.f
```

The "-m" flag gives the name the python module should have. The "-h" flag tells f2py in which file it should write the signature. After that all the FORTRAN files are listed f2py should parse. In the case here it is only the file "dsum.f". Now the file "dsum.pyf" should look like the following lines. It uses the interface specifications of FORTRAN 90 with some extensions.

```
1  !%f90 -*- f90 -*-
2  python module dsum ! in
3      interface  ! in :dsum
4          subroutine dsum(b,a,n) ! in :dsum:dsum.f
5              double precision :: b
6              double precision dimension(n) :: a
7              integer optional,check(len(a)>=n),depend(a) :: n=len(a)
8          end subroutine dsum
9      end interface
10 end python module dsum
11
12 ! This file was auto-generated with f2py (version:2.23.190-1372).
13 ! See http://cens.ioc.ee/projects/f2py2e/
```

So everything after the exlamation mark is a comment. The first line tricks editors in FORTAN 90 mode. The python module line defines the name of the module. Here the interface contains only one subroutine "dsum". This subroutine takes an array of type double precision with length "n" and returns the sum of its elements. So one line 4 one can see the signature of the FORTRAN routine. Then "b" is declared. On line 6 the array and its dimension is declared. The declaration of "n" is unusual. Here one can see four non FORTRAN keywords: depend, check, len and "=". Depend tells f2py which other variables are needed to fully resolve the value of the variable in question. So here also the variable "a" is needed. And it is used to calculate "n" using the size of the python array a. After the double colon you find the cooresponding statement n=len(a). The f2py wrapper still allows one more trick. If "n" is given, it checks if "a" has at least equal elements as n says. The optional keyword is also known from FORTRAN. Here it means that the python wrapper can make n an optional argument.

Now intent statements are added to define "b" as output and "a" and "n" as input:

```
1  !%f90 -*- f90 -*-
2  python module dsum ! in
3      interface  ! in :dsum
```

```
4           subroutine dsum(b,a,n) ! in :dsum:dsum.f
5               double precision, intent(out) :: b
6               double precision dimension(n), intent(in) :: a
7               integer optional,check(len(a)>=n),depend(a),intent(in) :: n=
                    len(a)
8           end subroutine dsum
9       end interface
10  end python module dsum
```

Now everything is ready and the module can be compiled. F2py will try to find a compiler in your path and use it. So all to be typed is:

```
>f2py -c dsum.pyf dsum.f
```

f2py will write the wrapper files, compile dsum.f and the wrapper files, and link them in a shared object. After this step one can start python and load the extension:

```
> python
>>> import dsum
>>> print dsum.__doc__
This module 'dsum' is auto-generated with f2py (version:2.23.190-1372).
Functions:
    b = dsum(a,n=len(a))
.
>>> print dsum.dsum.__doc__
dsum - Function signature:
    b = dsum(a,[n])
Required arguments:
    a : input rank-1 array('d') with bounds (n)
Optional arguments:
    n := len(a) input int
Return objects:
    b : float

>>> print dsum.dsum((1,2,3,4,5))
15.0
>>> print dsum.dsum((1,2,3,4,5), 2)
3.0
```

One can see that f2py also added some description to the wrapped functions. The last but one line shows the first example and the last line illustrates the use of the optional argument "n". Voillà, the first fortran extension is ready.

3 Wrapping a simple routine

Todays power supply's are often based on thyristors. When these devices cut the current, they can generate electromagnetic noise disturbing faint sensor signals (See also Figure 1).

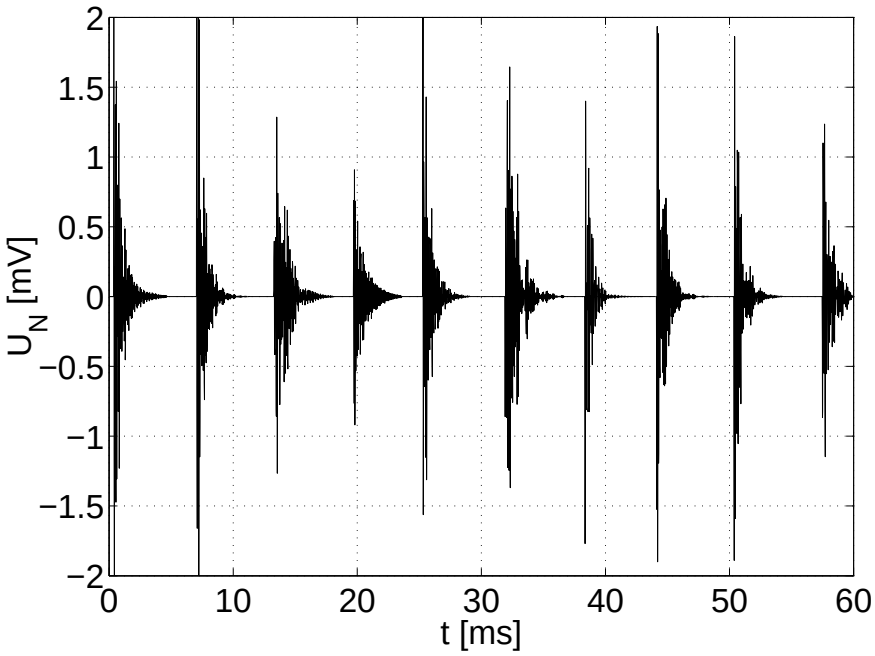

Figure 1: Electromagnetic noise generated by an power supply. The induced voltage U_N in (Milli volts) is given versus time t in (Milliseconds). The main periodicity has a frequency of $600\,Hz$.

One Peak can be described as

$$U_N(t) = \sum_{i=1}^{N} q_i e^{-r_i(t-t_{0i})} \sin\left[s_i\,(t - t_{0i}) + \phi_i\right] \qquad t_0 < t < t_0 + max(q) * 5, \tag{2}$$

with U_N the voltage of the noise and q, r, s, t_0, ϕ the coefficients of one vibration mode. The coefficients are generated randomly for each mode, and each peak is made up of N modes; N is typically selected to 10. As a deterministic evaluation of U_N was needed for one run, 600 peaks were generated during setup. Further it was used inside a loop, so evaluation time became critical. Therefore I decided to implement the coefficient generation part in Python and the code for Equation (2) in FORTRAN (See also Section C).

The FORTRAN function evaluates an vector containing *m* points in time:

```
1   subroutine expnta(volts, m, time, freq, n, ampl, phase, decay, t0, tend
        )
```

volts correspond to U_N, *freq* to s, *ampl* to q, *phase* to ϕ, decay to r. *t0* and *tend* describe the time interval in which the formula shall be evaluated cutting away of many irrelevant components.

Using the above routine I will show which steps are necessary to generate the wrapper: First start f2py to generate a signature file (assuming that the Fortran code is in file *"expnoise.f"*):

```
> f2py  -h ExponentNoise.pyf  -m  ExponentNoise expnoise.f
```

The option -h tells f2py to generate a signature file with name *"ExponentNoise.pyf"* for the routines found in "expnoise.f". '-m' tells f2py the name of the target. The following file *"ExponentNoise.pyf"* is generated:

```
1  !%f90 -*- f90 -*-
2  python module ExponentNoise ! in
3      interface  ! in :ExponentNoise
4          function expnt(time,freq,n,ampl,phase,decay,t0,tend) ! in :
               ExponentNoise:expnoise.f
5              double precision :: time
6              double precision dimension(n) :: freq
7              integer optional,check(len(freq)>=n),depend(freq) :: n=len(
                   freq)
8              double precision dimension(n),depend(n) :: ampl
9              double precision dimension(n),depend(n) :: phase
10             double precision dimension(n),depend(n) :: decay
11             double precision dimension(n),depend(n) :: t0
12             double precision dimension(n),depend(n) :: tend
13             double precision :: expnt
14         end function expnt
15         subroutine expnta(volts,m,time,freq,n,ampl,phase,decay,t0,tend) !
               in :ExponentNoise:expnoise.f
16             double precision dimension(m) :: volts
17             integer optional,check(len(volts)>=m),depend(volts) :: m=len(
                   volts)
18             double precision dimension(m),depend(m) :: time
19             double precision dimension(n) :: freq
20             integer optional,check(len(freq)>=n),depend(freq) :: n=len(
                   freq)
21             double precision dimension(n),depend(n) :: ampl
22             double precision dimension(n),depend(n) :: phase
23             double precision dimension(n),depend(n) :: decay
24             double precision dimension(n),depend(n) :: t0
25             double precision dimension(n),depend(n) :: tend
26         end subroutine expnta
27     end interface
28 end python module ExponentNoise
29
30 ! This file was auto-generated with f2py (version:2.17.177-1265).
31 ! See http://cens.ioc.ee/projects/f2py2e/
```

The syntax is borrowed from the FORTRAN 90 interfaces. The first line says

```
python module ExponentNoise
```

stating the name of the python module. So in the compile run of f2py one will see, that f2py will generate a Python module with the name "ExponentNoise". The next line starts the interface definition for that module:

```
interface ! in :ExponentNoise
```

On lines 4 – 14 the definition of the FORTRAN function *"expnt"* is given followed by the definition of the subroutine *"expnta"* on lines 15-26. Then the interface and the

module is closed. Now lets focus on the subroutine definition. First the parameters of the Fortran Routine are listed (Line 15) in the order of the Fortran function. In the Lines $16-26$ the input data types are listed as in FORTRAN 90 interface blocks. Here some other additional tasks have to be fulfilled. Python array store their size, so the user does not need to pass their size explicit. The key tokens *depend*, = and *check* help the compile process to deal with this issue:

- = on line 18 tells f2py to calculate m using the size of the *volts* array. f2py will add m as an optional argument, allowing to specify a subset of the array.
- *check* tells f2py to check the variable properties. Here it states, that the volts array needs more or equal items than the value of m.
- *depend* tells f2py, that this variable needs more information. On line 20 it tells f2py, that n needs the variable *freq*. (f2py will check if the freq vector has the correct length).

Now I start to change the file to reflect my needs. f2py could not derive the intent of the variables, as there was nothing stated in the code. So I add a *intent(in)* statement to all variables except for the one describing the *volts* variable which gets an *intent(out)* statement, being the result of my calculation. (See also Listing 1). But in this constellation the intent(out) has a serious side effect. Normally f2py will generate an array of appropriate size for all intent(out) statements. But here the volts variable is used to calculate m. As the caller will not pass the array to the wrapper it can not calculate m. So the allocation of the volts array will fail. Therefore the $=$ equal statement for m (Line 17) has to be changed to:

```
integer, optional, check(len(time)>=m),depend(time), intent(in) ::
m=len(time)
```

Also the depend(m) statement must be removed from line 18. (The time variable does not depend on m any more, as m is calculated form the shape of the *time* array). Now the wrapper will use the size of the *time* array to calculate m and to allocate the appropriate *volts* array. As I only want to wrap the subroutine, (the function is a helper function) I delete the lines 4 to 14 describing the function "expnt". So the final interface file is given in (Listing 1).

Listing 1: Interface file "ExponentNoise.pyf"

```
1 !%f90 -*- f90 -*-
2 python module ExponentNoise ! in
3     interface  ! in :ExponentNoise
4         subroutine expnta(volts,m,time,freq,n,ampl,phase,decay,t0,tend) !
              in :ExponentNoise:expnoise.f
5             double precision dimension(m), intent(in) :: time
6             integer optional,check(len(time)>=m),depend(time) :: m=len(
                  time)
7             double precision dimension(m),depend(m), intent(out) :: volts
8             double precision dimension(n) :: freq
9             integer optional,check(len(freq)>=n),depend(freq) :: n=len(
                  freq)
```

```
10              double precision dimension(n),depend(n), intent(in) :: ampl
11              double precision dimension(n),depend(n), intent(in) :: phase
12              double precision dimension(n),depend(n), intent(in) :: decay
13              double precision dimension(n),depend(n), intent(in) :: t0
14              double precision dimension(n),depend(n), intent(in) :: tend
15          end subroutine expnta
16      end interface
17  end python module ExponentNoise
```

To compile the extension module "ExponentNoise" (".so" or. "dll" or ...depending on your OS) one types:

```
> f2py -c ExponentNoise.pyf expnoise.f
```

in the shell. The -c option tells f2py

- to write the wrapper based on the information found in the file *"ExponentNoise.pyf"*,
- to generate the appropriate interface code
- and to generate the extension.

Following this ".pyf"-file all libraries and FORTRAN files must be listed, which contain the code of the extension. In this case its just one file *"expnoise.f"*. Now I start python, import the module[1] and read its doc strings:

```
> python
>>> import ExponentNoise
>>> print ExponentNoise.__doc__
This module 'ExponentNoise' is auto-generated with f2py (version:2.17.177-1265).
Functions:
    volts = expnta(time,freq,ampl,phase,decay,t0,tend,m=len(time),n=len(freq))
.
>>> print ExponentNoise.expnta.__doc__
expnta - Function signature:
    volts = expnta(time,freq,ampl,phase,decay,t0,tend,[m,n])
Required arguments:
    time : input rank-1 array('d') with bounds (m)
    freq : input rank-1 array('d') with bounds (n)
    ampl : input rank-1 array('d') with bounds (n)
    phase : input rank-1 array('d') with bounds (n)
```

[1] If import fails with the following or a similar message:

```
>>> import ExponentNoise
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError: ./ExponentNoise.so: undefined symbol: expnta_
```

you most likely forgot to add "expnoise.f" as an argument when compiling. For shared objects (or dll's) all routines can not be resolved at link time , so unresolved routines are only identified during import. If you add "expnoise.f", you will see that f2py compiles this routine and links this routine to the ExponentNoise dynamic object.

```
    decay : input rank-1 array('d') with bounds (n)
    t0 : input rank-1 array('d') with bounds (n)
    tend : input rank-1 array('d') with bounds (n)
Optional arguments:
    m := len(time) input int
    n := len(freq) input int
Return objects:
    volts : rank-1 array('d') with bounds (m)
```

The module doc string gives a short summary of the wrapped routines. Here its only the routine *"expnta"*. It requests the array *time*, whose subrange can be given optionally using *m*, and the arrays *freq,ampl,phase, decay, t0, tend*, whose subrange can be given by *n*. It is recommended to wrap the Fortran function in python routine, as it is simpler to add meaningful error messages in python. (f2py already gives nice error messages, but still one often needs more.) Further changes to the Fortran code, or the signature file can result in a reordering of arguments, so a python wrapper allows adaption only changing code in one place. The routine is wrapped by:

```
class _ExponentNoiseDirectCall(_ExponentNoiseWrapper): """ The whole
    Formula
is calculated in Fortran """ def eval(self, time, freq, amplitude,
    phase,
decay, starttime, endtime): return expnta(time, freq, amplitude, phase,
decay, starttime, endtime)
```

All the data handling code is given in Listing 4. (Warning it handles also the following examples. So up to now only the 'direct' test has been implemented .) The interface code is given in Listing 5 (Again, up to now, only the class _ExponentNoiseDirectCall) is used.

With this information given in this section you are already prepared to wrap standard Fortran routines. The following steps need to be done:

1. Use f2py to generate a signature file.
2. Edit this signature file to your needs
3. Compile the extension module.
4. Write a python wrapper for your comfort.

4 Pitfalls when passing arrays with more than one dimension

In the above example all arrays had only one dimension. The numerical extensions follow the C conventions, where the last index is the fastest, while the first index is the fastest running for FORTRAN arrays. To familiarise lets develop the following simple routine, which takes an array, prints its elements, and multiplies it with 2.

```
      subroutine ind2d(a, m, n)
      IMPLICIT NONE
      integer n,m
      double precision a(m,n)

      integer i,j


      do j = 1,n
         do i = 1,m
            write(*,*) "a(",i,",",j,")=",a(i,j)
           a(i,j) = a(i,j) * 2.0
         enddo
      enddo
      end
```

The signature file is given by,after adding the intent statements:

```
!%f90 -*- f90 -*-
python module IndexTest ! in
    interface  ! in :IndexTest
        subroutine ind2d(a,m,n) ! in :IndexTest:index_test.f
            double precision dimension(m,n),intent(in,out) :: a
            integer optional,check(shape(a,0)==m),depend(a),intent(in) :: m=shape(a,0)
            integer optional,check(shape(a,1)==n),depend(a),intent(in) :: n=shape(a,1)
        end subroutine ind2d
    end interface
end python module IndexTest
```

Here you can see the macro for checking the size of arrays with more than one dimension: *shape*. It works like the Numpy *shape* function, except that instead of writing *shape(a)[0]* one writes *shape(a,0)*. Further notice, that f2py has changed the check to == as the return array has the same name as the one passed in, which means that the input array itself is used as return value.

To test I add a simple test routine (see also Listing 2):

```
def test():
    a = arange(6)
    a.shape = (2,3)
    b = a
    print "-------------IndexTest Results-------------"
    print "-                   a                     -"
    print_array(a)
    print "-            Fortran Output               -"
    b = IndexTest.ind2d(a)
    print "-              Results                    -"
    print "-                   a                     -"
    print_array(a)
    print "-                   b                     -"
    print_array(b)
    print "-------------------------------------------"
```

When you run the file you can see from the output[2] that the same value is accessed with the same indices i and j. Only the numbers on the right are not ascending in order due to the different indexing of arrays in Python and Fortran.

If you examined the above signature file carefully, you will have noticed, that I wrote

[2]Some Fortran Compilers (e.g. Nag f95) send the output of write(*,*) to a file with a default name. (For Nag f95 it is fort.6 .)

```
double precision dimension(m,n),intent(in,out) :: a
```

This is a important difference to *intent(inout)*. As you know this intent statement is not valid for FORTRAN 90. There it would be *intent(inout)*. f2py supports *intent(inout)*, it is, however, better practice to change it to *intent(in,out)* because:

- the intent(in,out) statement tells f2py to copy the data of *a* into a new array, which is then changed by FORTRAN. Python arrays can possess many references. So a "in place" change of values can yield unexpected results.
- The f2py wrappers tries to address any conversion issues (e.g. casting the array to a proper type). This can result in the creation of some temporary array. To get the *intent(inout)* behaviour the data would need to be copied back again. As these double copying does not make sense f2py is not supporting it.

Only in one case this statement makes sense:

- If the array is so big, that making a copy results in a serious performance impact on the target machine.

The above mentioned oddities are shown by means of the file "index_test/IndexTestWRONG.pyf" and a second test routine to "index_test/test.py". There you can see, that you allocate the array with proper type it have to transpose it to satisfy the (inout) requirements, resulting that one always has to take the fact into account that the indices in Python are reversed with respect to C. I.e `a[i,j] = a(j,i)`. Sub arrays of *a* would unexpectedly. (Any reference to the array becomes a FORTRAN EQUIVALENCE statement).

So if the FORTRAN routine changes the values of some array, declare it *intent(in,out)* unless the array needs very large amounts of memory. If you worry about performance compile the appropriate module with the option `-DF2PY_REPORT_ATEXIT`. For the "ExponentNoise" module the command would be:

```
> f2py  -c -DF2PY_REPORT_ATEXIT ExponentNoise.pyf expnoise.f
```

When the program exits, the time spent in the wrappers and the Fortran routine will be displayed.

Listing 2: The python test routine

```
#!/usr/bin/env python
import IndexTest
import IndexTestWRONG

from Numeric import *

def print_array(a):
    for j in range(a.shape[1]):
            for i in  range(a.shape[0]):
                print "[",i,",",j,"] =  ", a[i,j]

def test():
    a = arange(6)
```

```
    a = reshape(a, (3,2))
    b = a
    print "-------------IndexTest Results-------------"
    print "-                    a                    -"
    print_array(a)
    print "-             Fortran Output              -"
    b = IndexTest.ind2d(a)
    print "-                 Results                 -"
    print "-                    a                    -"
    print_array(a)
    print "-                    b                    -"
    print_array(b)
    print "-----------------------------------------"


def testWRONG():
    a = arange(6).astype(Float)
    a = reshape(a, (3,2))
    b = a
    print "-------------IndexTest WRONG Results------"
    print "-                    a                    -"
    print_array(a)
    print "-             Fortran Output              -"
    # A needs to be transposed to be a proper array
    # for fortran
    a = transpose(a)
    if not IndexTestWRONG.has_column_major_storage(a):
        print "A is not in proper storage!"
        print "Can not pass such an array!"
    IndexTestWRONG.ind2d(a)
    print "-                 Results                 -"
    print "-                    a                    -"
    print_array(a)
    print "-                    b                    -"
    print_array(b)
    print "-----------------------------------------"
if __name__ == '__main__':
    test()
    testWRONG()
```

5 Wrapping a call back

Up to now only FORTRAN routines were discussed, which only call FORTRAN routines. Many solvers however evaluate a user defined routine (e.g. ordinary differential equation solvers need a routine describing the differential equation system), which can be implemented in python. f2py provides this capabilities.

5.1 A simple python function

The first example is very simple. Instead of calling the Fortran function *sin* in Equation (2) a call back is used. (I know that it does not make much sense form a computing point of view.) The changes to the Fortran files are given in Section D. The first step:

```
> f2py -h ExponentNoiseCallback.pyf expnoise_callback.f
```

writes a signature file. The file is listed here:

```
1  !%f90 -*- f90 -*-
2  python module expntf__user__routines
3      interface expntf_user_interface
4          subroutine df(tmp1,tmp2) ! in expnoise_callback.f:expntf:unknown_interface
5              double precision :: tmp1
6              double precision :: tmp2
7          end subroutine df
8      end interface expntf_user_interface
9  end python module expntf__user__routines
10 python module expntaf__user__routines
11     interface expntaf_user_interface
12         external df
13     end interface expntaf_user_interface
14 end python module expntaf__user__routines
15 function expntf(time,freq,n,ampl,phase,decay,t0,tend,df) ! in expnoise_callback.f
16     use expntf__user__routines
17     double precision :: time
18     double precision dimension(n) :: freq
19     integer optional,check(len(freq)>=n),depend(freq) :: n=len(freq)
20     double precision dimension(n),depend(n) :: ampl
21     double precision dimension(n),depend(n) :: phase
22     double precision dimension(n),depend(n) :: decay
23     double precision dimension(n),depend(n) :: t0
24     double precision dimension(n),depend(n) :: tend
25     external df
26     double precision :: expntf
27 end function expntf
28 subroutine expntaf(volts,m,time,freq,n,ampl,phase,decay,t0,tend,df) ! in expnoise_callback.
       f
29     use expntaf__user__routines
30     double precision dimension(m) :: volts
31     integer optional,check(len(volts)>=m),depend(volts) :: m=len(volts)
32     double precision dimension(m),depend(m) :: time
33     double precision dimension(n) :: freq
34     integer optional,check(len(freq)>=n),depend(freq) :: n=len(freq)
35     double precision dimension(n),depend(n) :: ampl
36     double precision dimension(n),depend(n) :: phase
37     double precision dimension(n),depend(n) :: decay
38     double precision dimension(n),depend(n) :: t0
39     double precision dimension(n),depend(n) :: tend
40     external df
41 end subroutine expntaf
42
43 ! This file was auto-generated with f2py (version:2.17.177-1265).
44 ! See http://cens.ioc.ee/projects/f2py2e/
```

Here the most notable part are the "__user__" modules. When f2py finds a routine in the argument list, it will try to guess its signature and generates an appropriate module wrapping the callback. Here you can see, that it found the subroutine *df* called by the expntf function (Line 4). For subroutine expntaf this mechanism failed, as the callback is not evaluated there, but passed to expntf. Further notice, that f2py added a use statement to the function and subroutine definitions (Line 16 and 29) to import the appropriate modules. Now I add the intent statements, and copy the definition of the subroutine *df* from the "expnt__user__routines" to the "expnta__user__routines". I delete the statements concerning the expntf function, as my entry point is the expnta routine. So I finally get the file:

```
1  !%f90 -*- f90 -*-
2  python module expntf__user__routines
3      interface expntf_user_interface
4          subroutine df(tmp1, tmp2)! in :ExponentNoiseCallback:expnoise_callback.f:expntf:
               unknown_interface
5              double precision, intent(in)  :: tmp1
```

```
6               double precision, intent(out) :: tmp2
7             end subroutine df
8         end interface expntf_user_interface
9     end python module expntf__user__routines
10    python module ExponentNoiseCallback ! in
11        interface  ! in :ExponentNoiseCallback
12            subroutine expntaf(volts,m,time,freq,n,ampl,phase,decay,t0,tend,df) ! in :
                  ExponentNoiseCallback:expnoise_callback.f
13                use expntf__user__routines
14                double precision dimension(m) :: volts
15                integer optional,check(len(volts)>=m),depend(volts) :: m=len(volts)
16                double precision dimension(m),depend(m) :: time
17                double precision dimension(n) :: freq
18                integer optional,check(len(freq)>=n),depend(freq) :: n=len(freq)
19                double precision dimension(n),depend(n) :: ampl
20                double precision dimension(n),depend(n) :: phase
21                double precision dimension(n),depend(n) :: decay
22                double precision dimension(n),depend(n) :: t0
23                double precision dimension(n),depend(n) :: tend
24                external df
25            end subroutine expntaf
26        end interface
27    end python module ExponentNoiseCallback
```

For testing I subclass the *_ExponentNoiseWrapper*.

```
1   class _ExponentNoiseCallBackSimple(_ExponentNoiseWrapper):
2     """
3     Callback Illustration. Use the sin from Numeric instead of the Fortran one.
4     """
5     def _sin(self, angle):
6         return  sin(angle)
7
8     def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
9         volts = expntaf(time, freq, amplitude, phase, decay, starttime, endtime,
10                lambda x, y=self: y._sin(x))
11        return volts
```

The lambda function is necessary, as f2py accepts only pure functions or lambda's as a call back routine.

5.2 Calling the whole formula

So up to now you are ready to generate a Call back for nearly any Fortran 77 code. Here I want to show you a second issue, advanced creation of help arrays. Complex Fortran 77 routines often need work arrays. Here I will show how an array of varying size can be easily specified to f2py. In Section D the full code is listed for the function findf. Here I add all value of time for one pair of coefficients i. Python is called to evaluate Equation (2). As the data is stored in arrays the python evaluation is very fast.

The signature of the routine findf is given by:

```
      subroutine findf(volts, m, time, freq, n, ampl, phase,
     1                 decay, t0, tend, pyfunc, help1, help2, l)
```

Here two help arrays have to be passed with size l. f2py writes the following wrapper:

```
subroutine findf(volts,m,time,freq,n,ampl,phase,decay,t0,tend,pyfunc,help1,help2,l) ! in
    expnoise_callback.f
    use findf__user__routines
    double precision dimension(m) :: volts
    integer optional,check(len(volts)>=m),depend(volts) :: m=len(volts)
    double precision dimension(m),depend(m) :: time
    double precision dimension(n) :: freq
    integer optional,check(len(freq)>=n),depend(freq) :: n=len(freq)
    double precision dimension(n),depend(n) :: ampl
    double precision dimension(n),depend(n) :: phase
    double precision dimension(n),depend(n) :: decay
    double precision dimension(n),depend(n) :: t0
    double precision dimension(n),depend(n) :: tend
    external pyfunc
    double precision dimension(l) :: help1
    double precision dimension(l),depend(l) :: help2
    integer optional,check(len(help1)>=l),depend(help1) :: l=len(help1)
end subroutine findf
```

In the last lines the help arrays are listed again. The function can deal with arbitrary size arrays, so I changed the last lines to:

```
  integer, intent(in) :: l
  double precision dimension(l),depend(l), optional :: help1
  double precision dimension(l),depend(l), optional :: help2
```

Now the help arrays got optional arguments, and *l* will yield the size. In the wrapper, you can see that l has been set to 1000.

```
class _ExponentNoiseCallBack1d(_ExponentNoiseWrapper):
    """
    Handles one data set of freq, ampl, phase, decay for a long data sample
    of time.
    """
    def __call__(self, help1, freq, ampl, phase, decay):
        return  exp(help1 * decay) * ampl * sin(help1 * freq + phase)

    def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
        l = 1000
        votls = findf(time, freq, amplitude, phase,  decay, starttime, endtime,
              lambda x,y,z,a,b,c,s=self : s.__call__(x,y,z,a,b),
              l)
        return volts
```

f2py will now allocate the memory under the hood, and passes the arrays to the called function.

6 A real world example

All examples above were centered around a simple example. Here a full wrapper for the lsodare routine from the ODEPACK (see is shown. As a first step a simpler wrapper using hidden work arrays is shown. As a second step an advanced wrapper providing a class hiding all internal details will be descirbed.

In this section the following features of f2py will be described:

- Callbacks
- intent(hide, copy, cache)

- MAX

6.1 A simple wrapper

As described in section 3 the raw f2py file is generated. A full listing of this file is given in Listing 6. Due to its size only the most important parts are shown here: The call back wrapper. When f2py finds a function in the argument list of the routine it tries to write such a description file. The "__user__routines" defines a module which describes call back functions. Here one can see that it found three such items. But only the function is called in the routine lsodar and not the others. Thus, it can only generate the interface for only for the function "f" and not for the others.

```
1  !%f90 -*- f90 -*-
2  python module lsodar__user__routines
3      interface lsodar_user_interface
4          external jac
5          external g
6          subroutine f(neq,t,y,e_rwork_lf0_e) ! in :lsodar:lsodar.f:lsodar:unknown_interface
7              integer dimension(1) :: neq
8              double precision :: t
9              double precision dimension(1) :: y
10             double precision :: e_rwork_lf0_e
11         end subroutine f
12     end interface lsodar_user_interface
13 end python module lsodar__user__routines
```

In the lsodar module the use statement includes the description of the call backs.

```
14 python module lsodar ! in
15     interface  ! in :lsodar
16         subroutine lsodar(f,neq,y,t,tout,itol,rtol,atol,itask,istate,iopt,rwork,lrw,iwork,
               liw,jac,jt,g,ng,jroot) ! in :lsodar:lsodar.f
17             use lsodar__user__routines
18             external f
```

The last variable in the routine signatur is jroot. The following variables are not part of the routine signature but belong to the common block of the routine. Here only a part is shown.

```
37             integer dimension(ng) :: jroot
38             double precision dimension(209) :: rowns
39             double precision :: ccmax
40             double precision :: el0
```

And here the declaration of the block itself follows.

```
104            integer :: mxords
105            common /ls0001/ rowns,ccmax,el0,h,hmin,hmxi,hu,rc,tn,uround,illin,init,lyh,lewt
                   ,lacor,lsavf,lwm,liwm,mxstep,mxhnil,nhnil,ntrep,nslast,nyh,iowns,icf,ierpj,
                   iersl,jcur,jstart,kflag,l,meth,miter,maxord,maxcor,msbp,mxncf,n,nq,nst,nfe,
                   nje,nqu
106            common /lsr001/ rownr3,t0,tlast,toutc,lg0,lg1,lgx,iownr3,irfnd,itaskc,ngc,nge
107            common /lsa001/ tsw,rowns2,pdnorm,insufr,insufi,ixpr,iowns2,jtyp,mused,mxordn,
                   mxords
108        end subroutine lsodar
109    end interface
110 end python module lsodar
111
112 ! This file was auto-generated with f2py (version:2.13.175-1250).
113 ! See http://cens.ioc.ee/projects/f2py2e/
```

To generate the additional callback routines (the "jacobi" and the "border") one writes a help file with the approbriate functions and let f2py wrap these functions:

```
      subroutine fex (neq, t, y, ydot)
      double precision t, y, ydot
      dimension y(neq), ydot(neq)
      end

      subroutine gex (neq, t, y, ng, gout)
      double precision t, y, gout
      dimension y(neq), gout(ng)
      end

      subroutine jac (neq, t, y, ml, mu, pd, nrowpd)
      integer neq, ml, mu
      double precision t, y(neq), pd(nrowpd,neq)
      end
```

Now this function is wrapped and the descriptions are copied into the "lsodar__user__routines". The result is shown in Listing 7.

Now the file is ready, but does not reflect the needs. Here I will show two final interfaces to lsodar. One is kept as simple as possible, and the other will show how to wrap a FORTRAN function in a class.

6.2 A simple interface

The following interface should be as straight forward as possible and only exhibit the necessary details. So again intent(in) and intent(out) were added to all lines. Two work arrays "rwork" and "iwork" have to be provided to lsodar. These are generated using the intent(hide, cache) statement. "hide" tells f2py not to add them as optional arguments to the list. And "cache" tells f2py to generate them once during loading of the module and keep them in memory.

```
44            double precision dimension(22 + neq * MAX(16, neq+9) + 3 * ng), intent(hide,
                  cache) :: rwork
45            integer dimension(20+neq), intent(hide,cache) :: iwork
```

A new statement is "MAX". It allows here to calculate the necessary size of the work arrays.

It is always good practise to introduce a python layer between the wrapped module and the rest of the code. The following wrapper allows to specify a border if desired and calculates the size of the array returned by the border automatically.

```
#!/usr/bin/env python
import Numeric
import _lsodarsimple

def lsodar(f, y, t, tout, *args, **keywords):
    """
    Wrapper around lsodar from ODEPACK
    """

    itol = 1; rtol = 1e-6;    atol = 1e-6
    task = 1; state = 1;
    jac = None;    jt = 2
    if keywords.has_key('border'):
        border = keywords['border']
        tmp = border(t,y)
```

```
16          if type(tmp) != type(Numeric.array((0,0))):
17              raise TypeError, "The border function must return an array!"
18
19          ng = len(tmp)
20          assert(ng != None)
21          assert(ng > 0)
22      else:
23          border = lambda x:x
24          ng = 0
25      if keywords.has_key('jacobi'):
26          jacobi = keywords['jacobi']
27      else:
28          jacobi = lambda x:x
29
30      assert(itol == 1 or itol == 2)
31      y,t,state,jroot = _lsodarsimple.lsodar(f, y, t, tout, itol, rtol, atol,
32                                             task, state, jacobi, jt,
33                                             border, ng)
34      return y, t, state, jroot
```

6.3 Data handling in the class

The above wrapper does not use the full functionality of lsodar. lsodar can save the status of the calculation internally and is faster if one continoues with the calculation. This state is stored in COMMON blocks. lsodar provides functions so that the data of the COMMON blocks can be set from arrays and stored in arrays. The class illustrated here will handle these data internally. Further lsodar provides functions to calculate the derivatives at each calculational step. The class can offer a method offering this capabillity transparent to the user. Further optional parameters are stored in the work arrays. Methods with meaningful names can be probided.

6.4 The interface file

The full file is given in Listing 9. Two more functions were added. srcar copies the common block data and intdy allows to calculate the derivative.

```
28  python module _lsodar ! in
29      interface  ! in :lsodar
30          subroutine srcar(rsav,isav,job)
31              double precision, dimension(245), check(len(rsav)>=245) :: rsav
32              integer, dimension(59), check(len(isav)>=59) :: isav
33              integer :: job
34          end subroutine srcar
35          subroutine intdy(t, deriv, rwork, nyh, dky, iflag)
36              double precision, intent(in) :: t
37              integer, intent(in) :: deriv
38              double precision dimension(:), intent(inout) :: rwork
39              integer, intent(in), required :: nyh
40              double precision, dimension(nyh), intent(out) :: dky
41              integer intent(out) :: iflag
42          end subroutine intdy
```

Lsodar allows to specify the calculational tolerances for each part of the differential equation system or one for all. iopt=1 is one parameter for all and iopt=2 is the second case. Using MAX() f2py can check that.

```
78              double precision dimension(MAX(neq * (itol-1), 1)), intent(in), depend(itol)
                    :: rtol
```

```
79              double precision dimension(MAX(neq * (itol-1), 1)), intent(in), depend(itol)
                    :: atol
```

Above the work arrays were hidden. Here it is desired to store them inside the instance so that each instance keeps track about its status.

```
83              double precision dimension(:), check(len(rwork)>=(22 + neq * MAX(16, neq+9)
                    + 3 * ng)), depend(neq), depend(ng), intent(inout) :: rwork
84              integer dimension(:), check(len(iwork)>=(20+neq)), depend(neq), intent(inout)
                    :: iwork
```

The whole class is not listed here, only the main call to the wrapper.

```
1  y = self.y
2  t0 = self.t
3  self.__prae() # Write Common Blocks
4  tmp = _lsodar.lsodar(integrand, y, t0, t,
5                       # Type of tolerances and tolerances
6                       self.itol, self.rtol, self.atol,
7                       # Task and status of calculation
8                       task, self.state,
9                       # Optional paramters and work arrays
10                      iopt, self.rwork, self.iwork,
11                      # Jacobi matrix, external not supported yet
12                      jacobi, jt,
13                      # Border conditions
14                      border, self.borderdimension)
15 yout, t, state, jroot = tmp
16 self.__post() # Save Common Block Status
17 self.y = yout
18 self.t = t
```

6.5 What could be added to the Interface

- Jacobi Matrix Support
- varying the number of equations ...
- Rewrite part of the __call__ method in FORTRAN.

7 Using Advanced Features of FORTRAN 90

The FORTRAN 90 standard brought a new set of features. Here I will show how to handle two new features: the kind operator and arrays with implicit size. Dynamically allocated arrays are not discussed here.

7.1 Calling FORTRAN functions with sized arrays

FORTRAN 90 allows to pass arrays without specifying their size using separate parameters. All arrays can be "SIZE"ed. So the size of array "a" defined as

```
  double precision dimension(:,:) :: a
```

can be determined by calling:

```
d1 = SIZE(a, 1)
d2 = SIZE(a, 2)
```

with *d1,d2* making the dimension available in the routine. However the storage of the length dimension is implementation dependent. But one can write a FORTRAN wrapper:

```
1  subroutine wrap_foo(a, d1, d2, b, d3)
2  IMPLICIT NONE
3  integer, intent(in)                                  :: d1, d2
4  integer, intent(out)                                 :: d3
5  double precision, dimension(d1,d2), intent(in)       :: a
6  double precision, dimension(d3),    intent(out)      :: b
7
8  INTERFACE
9    SUBROUTINE FOO (a,b)
10     IMPLICIT NONE
11       double precision, dimension(:,:),  intent(in) :: a
12       double precision, dimension(:),    intent(out) :: b
13   END SUBROUTINE FOO
14 END INTERFACE
15 CALL foo(a,b)
16 d3 = SIZE(b)
17 end subroutine wrap_foo
```

Here the arrays are first declared with a fixed dimension (Lines 5 and 6). During run time these dimensions are then fixed to the array. Then in the interface for the subroutine "foo" the arrays are declared as FORTRAN 90 style arrays. In the subroutine "foo" the array "a" can be sized.

7.2 The kind operator

The kind operator allows to inform the compiler what accuracy is needed for the variables. The compiler then chooses the appropriate native type. When writing this document f2py did not support this feature. Still routines using such type declarations can be wrapped. Find out the type the kind operator evaluates to e.g. by making a small FORTRAN stand alone program and printing the return value of the kind call. Use the documentation of your FORTRAN compiler, to find out to which C-type it maps. Generate the f2py file as described before. Now edit the file, remove the kind entry and replace the integer or float with the "integer*" or "float*" with the appropriate byte count.

A Adding a new compiler

Even if f2py supports an impressive list of compilers, still one will find compilers not supported yet. A full description would exceed the limit of this tutorial, so I

focus here on the main steps to add a new compiler. As an example I use the Lahey - Compiler.

As a first step I write a small FORTRAN program:

```
1   a = 2 write(*,*) sin(a) end
```

After compilation the executable is used to search for the shared libraries, which this object needs. Tools like *ldd* can list the necessary libraries. On my machine running linux ldd displays:

```
libfj9i6.so.1 => /opt/lf9560/lib/libfj9i6.so.1 (0x40023000)
libfj9f6.so.1 => /opt/lf9560/lib/libfj9f6.so.1 (0x400a7000)
libc.so.6 => /lib/libc.so.6 (0x40191000)
libm.so.6 => /lib/libm.so.6 (0x40286000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Here the interesting two lines are the first two, which point to the libraries libfj9i6.so.1 and libfj9f6.so.1 . These libraries are needed during the linking of the python extension. All the Fortran compiler issues are encapsulated by the class *fortran_compiler_base* in the file scipy_distutils/command/build_flib.py. For each supported compiler a subclass is derived. Most of the time it is simpler to use a full implemented compiler subclass and not to start from the beginning. So I chose the class nag_fortran_compiler as a starting point. In Listing 3 the implementation of the class is shown.

Listing 3: "The class describing the lahey compiler."

```
1  class lahey_fortran_compiler(fortran_compiler_base):
2      vendor = 'Lahey/Fujitsu'
3      ver_match = r'Lahey/Fujitsu Fortran 95 Compiler Release(?P<
           version>[^\s*]*)'
4
5      def __init__(self, fc = None, f90c = None):
6          fortran_compiler_base.__init__(self)
7
8          if fc is None:
9              fc = 'lf95'
10         if f90c is None:
11             f90c = fc
12
13         self.f77_compiler = fc
14         self.f90_compiler = f90c
15
16         switches = ''
17         debug = ' -g --chk --chkglobal '
18
19         self.f77_switches = self.f90_switches = switches
20         self.f77_switches = self.f77_switches + ' --fix '
21         self.f77_debug = self.f90_debug = debug
```

```
22          self.f77_opt = self.f90_opt = self.get_opt()
23
24          self.ver_cmd = self.f77_compiler+' --version'
25          try:
26              dir = os.environ['LAHEY']
27              self.library_dirs = [os.path.join(dir,'lib')]
28          except KeyError:
29              self.library_dirs = []
30
31          self.libraries = ['fj9f6', 'fj9i6', 'fj9ipp', 'fj9e6']
32
33
34      def get_opt(self):
35          opt = ' -O'
36          return opt
37
38      def get_linker_so(self):
39          return [self.f77_compiler,'--shared']
```

. To make the new class available to the build process one has to add the compiler to the file "build_flib.py" and add it to the list of available compilers at the bottom of this file.

B The data handling Code

Listing 4: The data handling "ExponentNoise.py"

```
#!/usr/bin/env python

import Numeric
import MLab
import f77


def checkargumentshape(args, names):
    testshape = args[0].shape
    passed = 1
    for i in args[1:]:
        if testshape != i.shape:
            passed = 0
            break
    if passed != 1:
        errm = "The shape of all " + str(len(args)) +\
               " arguments must be the same! \n"
        for i in range(len(args)):
            errm = errm + "Argument " + str(i) + " '" + names[i] +\
                   "' had a shape of " + str(args[i].shape) + " \n"
        raise TypeError, errm



class _FortranExponentNoise:
    def __init__(self, freq, ampl, phase, decay, t0, type):
        checkargumentshape((freq, ampl, phase, decay, t0),
                           ('Frequency', 'Amplitude', 'Phase', 'Decay', 't0'))
        self.starttime = t0

        tmp = Numeric.absolute(1./decay) * 50.0
        self.maxdecay = MLab.max(tmp)
        self.endtime  = t0 + tmp
        self.freq     = freq
        self.amplitude= ampl
        self.phase    = phase
        self.decay    = decay
        self.SetFortranFunction(type)

    def SetFortranFunction(self, type):
        self.expfortran = f77.ExponentNoise(type)

    def eval(self, time):
        volts = self.expfortran.eval(time, self.freq, self.amplitude,
                                     self.phase, self.decay, self.starttime,
                                     self.endtime)
        return volts


def testExponentNoise():
    from RandomArray import uniform
    import Gnuplot

    len = 1000
    n   = 10
    maxvolts = 1

    g = Gnuplot.Gnuplot()

    myfreq  = uniform(len*50,    len*100,  (n*len,))
    myampl  = uniform(-maxvolts, maxvolts, (n*len,))
    mydecay = uniform(-3.*len,   -1.*len,  (n*len,))
    myt0    = uniform(0.0,       1e-3,     (n*len,))
```

```
64      tmp      = Numeric.arange(len) * 2 * Numeric.pi / len
65      tmp      = Numeric.ravel(Numeric.multiply.outer(Numeric.ones(n),tmp))
66      myt0     = tmp + myt0
67      phase    = Numeric.zeros((len * n))
68      tmp      = MLab.max(Numeric.absolute(1./mydecay)) * 5.0
69
70      data = []
71      step = 0
72      for type in ('direct', 'simplecb', 'cb1d','cb2d'):
73      #bfor type in ('cb2d',):
74              print "Constructing Objects"
75              t = _FortranExponentNoise(myfreq, myampl, phase, mydecay, myt0,
76                                        type)
77              print "Evaluation"
78              hlp = Numeric.arange(3*len) * 2 * Numeric.pi / len / 500.
79              res = t.eval(hlp)
80              data.append(Gnuplot.Data(hlp,res+step, with='line', title=type))
81              step = step + 2
82      apply(g.plot, data)
83      raw_input()
84
85  def test():
86      testExponentNoise()
87
88  if __name__ == '__main__':
89      test()
```

Listing 5: The data handling "f77/__init__.py"

```
1   #!/usr/bin/env python
2   """
3   Encapsulates the Fortran 77 Versions of the tests
4   """
5   from Numeric import sin, exp, ones, multiply, sum, Float, ravel, where, less
6   from Numeric import transpose
7   from MLab import min, max
8   from ExponentNoise import expnta
9
10  from ExponentNoiseCallback import expntaf, findf, findf2d
11
12
13  class _ExponentNoiseWrapper:
14      instance_counter = 0
15      def __init__(self):
16          self.instance_counter = self.instance_counter + 1
17
18      def eval(volts, time, freq, amplitude,
19                  phase,  decay, starttime, endtime):
20          """
21          Pure Virtual class, use a derived one for the
22          implementation!
23          """
24
25  class _ExponentNoiseDirectCall(_ExponentNoiseWrapper):
26      """
27      The whole Forumla is calculated in Fortran
28      """
29      def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
30          return expnta(time, freq, amplitude, phase,  decay, starttime, endtime)
31
32  class _ExponentNoiseCallBackSimple(_ExponentNoiseWrapper):
33      """
34      Callback Illustration. Use the sin from Numeric instead of the fortran one.
35      """
36      def _sin(self, angle):
37          return  sin(angle)
38
39      def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
```

```
40          volts = expntaf(time, freq, amplitude, phase, decay, starttime,
41                          endtime, lambda x, y=self: y._sin(x))
42          return volts
43
44  class _ExponentNoiseCallBack1d(_ExponentNoiseWrapper):
45      """
46      Handles one data set of freq, ampl, phase, decay for a long data sample
47      of time.
48      """
49      def __call__(self, help1, freq, ampl, phase, decay):
50          return  exp(help1 * decay) * ampl * sin(help1 * freq + phase)
51
52      def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
53          volts = ones(time.shape, Float)
54          findf(volts, time, freq, amplitude, phase,  decay, starttime, endtime,
55                lambda x,y,z,a,b,c,s=self : s.__call__(x,y,z,a,b),
56                1000)
57          return volts
58
59  def _exponentnoiseclb2d(help1, freq, ampl, phase, decay, n1, j,l):
60      help1[:n1,:] = exp(help1[:n1,:] * decay[:n1,:]) * ampl[:n1,:] * sin(help1[:n1,:] * freq
            [:n1,:] + phase[:n1,:])
61      return help1
62
63
64  class _ExponentNoiseCallBack2d(_ExponentNoiseWrapper):
65      """
66      Gathers datasets of freq, ampl, phase, decay for some time values. This
67      allows to evaluate the sparse values as a matrix in python.
68      """
69      def __call__(self, help1, freq, ampl, phase, decay, n1):
70          tmp = multiply.outer(ones(n1), help1)
71          tmp1 = exp(tmp * decay[:n1,:]) * ampl[:n1,:]
72          tmp2 = sin(tmp[:n1,:] * freq[:n1,:] + phase[:n1,:])
73          return sum(tmp1*tmp2, 0)
74
75      def eval(self, time, freq, amplitude, phase,  decay, starttime, endtime):
76          check, volts = findf2d(time, freq, amplitude, phase,  decay, starttime,
77                                 endtime, _exponentnoiseclb2d,
78                                 100, 100)
79          if check < 0:
80              print "Got to column :", check
81              raise ValueError, "Not enough data forseen for each column"
82          print "Used ", check, "number of columns!"
83          return volts
84
85
86  _expevaltypedict = {'direct'    : _ExponentNoiseDirectCall,
87                      'simplecb' : _ExponentNoiseCallBackSimple,
88                      'cb1d'      : _ExponentNoiseCallBack1d,
89                      'cb2d'      : _ExponentNoiseCallBack2d
90                      }
91  def ExponentNoise(type):
92      try:
93          tmp = _expevaltypedict[type]
94      except KeyError, des:
95          errm = "The specified type " +  str(type) + "is unknown." +\
96                 "I only know about the following keys : " +\
97                 string.join(_expevaltypedict.keys(), "  ")
98          raise TypeError, errm
99      return tmp()
100
101
102 if __name__ == '__main__':
103     test()
```

C The FORTRAN source “expnoise.f”

```
1  C-----------------------------------------------------------------------
2  C Calculates the exponential voltage for a given time t
3  C-----------------------------------------------------------------------
4        function expnt(time, freq, n, ampl, phase, decay, t0, tend)
5        IMPLICIT NONE
6        Integer n
7        double precision expnt, time, freq(n), ampl(n), phase(n),
8       1                 decay(n),  t0(n), tend(n)
9
10
11       Integer i, testc
12       double precision alpha, mydec, calct, tmp1
13
14       expnt = 0.0D0
15       testc = 0
16       DO 400, i = 1,n
17          if (time .gt. t0(i) .and. time .lt. tend(i)) then
18             calct = time - t0(i)
19             tmp1  = calct * freq(i)
20             alpha = ampl(i) * dsin(tmp1  + phase(i))
21             mydec = alpha   * dexp(calct * decay(i))
22             expnt = expnt + mydec
23             testc = testc + 1
24          endif
25   400 CONTINUE
26       END
27
28 C-----------------------------------------------------------------------
29 C Calculates expnt for each time(i)
30 C-----------------------------------------------------------------------
31       subroutine expnta(volts, m, time, freq, n, ampl, phase,
32      1                 decay, t0, tend)
33       IMPLICIT NONE
34       Integer m, n
35       double precision expnt, time(m), freq(n), ampl(n), phase(n),
36      1                 decay(n),  t0(n), tend(n), volts(m)
37
38
39       Integer i
40
41       DO 500, i = 1,m
42          volts(i) = expnt(time(i), freq, n, ampl, phase, decay,
43      1                    t0, tend)
44   500 CONTINUE
45       END
```

D The FORTRAN source “expnoise_callback.f”

```
1  C-----------------------------------------------------------------------
2  C Author: Pierre Schnizer <Pierre.Schnizer@cern.ch>
3  C Date  : 22. May 2002.
4  C Purpose: Tutorial for f2py
5  C License: LGPL. You should have recieved a copy of the license together with
6  C this software. No Warrenty at all.
7  C-----------------------------------------------------------------------
8  C-----------------------------------------------------------------------
9  C Calculates the exponential voltage for a given time t according to the
10 C formula:
11 C volt = exp(decay (t-t0)) ampl df((t-to) * freq * phase) for
12 C all  t0 < time < tend
13 C-----------------------------------------------------------------------
```

```
14        function expntf(time, freq, n, ampl, phase, decay, t0, tend, df)
15        IMPLICIT NONE
16        Integer n
17        external df
18        double precision expntf, time, freq(n), ampl(n), phase(n),
19       1                 decay(n),  t0(n), tend(n)
20
21
22        Integer i
23        double precision alpha, mydec, calct, tmp1, tmp2
24
25        expntf = 0.0D0
26
27        DO 400, i = 1,n
28           if (time .gt. t0(i) .and. time .lt. tend(i)) then
29              calct = time - t0(i)
30              tmp1  = calct * freq(i) + phase(i)
31              tmp2  = -10
32              CALL df(tmp1, tmp2)
33              !tmp2  = dsin(tmp1)
34              alpha = ampl(i) * tmp2
35              mydec = alpha   * dexp(calct * decay(i))
36              expntf = expntf + mydec
37
38           endif
39   400  CONTINUE
40        END
41
42  C-------------------------------------------------------------------------
43  C Calculates expnt for each time(i)
44  C-------------------------------------------------------------------------
45        subroutine expntaf(volts, m, time, freq, n, ampl, phase,
46       1                   decay, t0, tend, df)
47        IMPLICIT NONE
48        Integer m, n
49        external df
50        double precision expntf, time(m), freq(n), ampl(n), phase(n),
51       1                 decay(n),  t0(n), tend(n), volts(m)
52
53
54        Integer i
55  C      write(*,*) "n,m = ", n, m
56        DO 500, i = 1,m
57  C         write(*,*) "i = ", i
58           volts(i) = expntf(time(i), freq, n, ampl, phase, decay,
59       1                     t0, tend, df)
60   500  CONTINUE
61        END
62
63  C-------------------------------------------------------------------------
64  C Only Search for the intervals, where exp(decay * t) * sin (a * t + phi)
65  C has to be evaluated. Call Python to evaluate the formula. In this case it is
66  C done dealing with all as many data from the time vector as possible using one
67  C (freq, ampl, phase, decay) data set. The calling routine must allocate the
68  C helper arrays to some size (l) (Only values <= m make sense.) This allows for
69  C memory tuning of the callback.
70  C-------------------------------------------------------------------------
71        subroutine findf(volts, m, time, freq, n, ampl, phase,
72       1                 decay, t0, tend, pyfunc, help1, help2, l)
73        IMPLICIT NONE
74        Integer m, n, l
75        external pyfunc
76        double precision time(m), freq(n), ampl(n), phase(n),
77       1                 decay(n),  t0(n), tend(n), volts(m),
78       2                 help1(l), help2(l)
79
80
81        Integer i, j, k, l1, ind
```

```
82 C       write(*,*) "n,m = ", n, m, l
83
84 C     Initalise volts to zero:
85       DO 500, k = 1,m
86          volts(k) = 0.0D0
87   500 CONTINUE
88
89       DO 700, k = 1, n
90          j = 0
91          DO 600, i = 1, m
92 C             write(*,*) "Testing :", i, time(i), t0(k), tend(k)
93             if (time(i) .gt. t0(k) .and. time(i) .lt. tend(k)) then
94                j = j + 1
95 C               write(*,*) " Adding time : ", time(i), "at index", j
96                help1(j) = time(i) - t0(k)
97                help2(j) = i
98             endif
99             if (j .ge. l) then
100 C              Help array is filled
101 C              Call python function for all relevant data
102 C               write(*,*) "in do calling pyf with ", l, " arguments"
103              CALL pyfunc(help1, l, freq(k), ampl(k), phase(k),
104       2                  decay(k))
105              DO 550, l1 = 1, l
106                 ind = help2(l1)
107 C                write(*,*) "adding volts", help1(l1), "at index", ind
108                volts(ind)=volts(ind) + help1(l1)
109  550         CONTINUE
110              j = 0
111           endif
112  600    CONTINUE
113 C        Call the pyton function for the rest of the help array
114          if (j .gt. 0) then
115 C           write(*,*) "finishing calling pyf with ", j, " arguments"
116            CALL pyfunc(help1, j, freq(k), ampl(k), phase(k),
117       2          decay(k))
118            DO 650, l1 = 1, j
119               ind = help2(l1)
120 C              write(*,*) "adding volts", help1(l1), "at index", ind
121               volts(ind)=volts(ind) + help1(l1)
122  650       CONTINUE
123          endif
124  700 CONTINUE
125       END
```

E The lsodar wrapper code

Listing 6: lsodar/lsodar_raw.pyf

```
!%f90 -*- f90 -*-
python module lsodar__user__routines
    interface lsodar_user_interface
        external jac
        external g
        subroutine f(neq,t,y,e_rwork_lf0_e) ! in :lsodar:lsodar.f:lsodar:unknown_interface
            integer dimension(1) :: neq
            double precision :: t
            double precision dimension(1) :: y
            double precision :: e_rwork_lf0_e
        end subroutine f
    end interface lsodar_user_interface
end python module lsodar__user__routines
python module lsodar ! in
    interface  ! in :lsodar
        subroutine lsodar(f,neq,y,t,tout,itol,rtol,atol,itask,istate,iopt,rwork,lrw,iwork,
            liw,jac,jt,g,ng,jroot) ! in :lsodar:lsodar.f
            use lsodar__user__routines
            external f
            integer dimension(1) :: neq
            double precision dimension(1) :: y
            double precision :: t
            double precision :: tout
            integer :: itol
            double precision dimension(1) :: rtol
            double precision dimension(1) :: atol
            integer :: itask
            integer :: istate
            integer :: iopt
            double precision dimension(lrw) :: rwork
            integer optional,check(len(rwork)>=lrw),depend(rwork) :: lrw=len(rwork)
            integer dimension(liw) :: iwork
            integer optional,check(len(iwork)>=liw),depend(iwork) :: liw=len(iwork)
            external jac
            integer :: jt
            external g
            integer optional,check(len(jroot)>=ng),depend(jroot) :: ng=len(jroot)
            integer dimension(ng) :: jroot
            double precision dimension(209) :: rowns
            double precision :: ccmax
            double precision :: el0
            double precision :: h
            double precision :: hmin
            double precision :: hmxi
            double precision :: hu
            double precision :: rc
            double precision :: tn
            double precision :: uround
            integer :: illin
            integer :: init
            integer :: lyh
            integer :: lewt
            integer :: lacor
            integer :: lsavf
            integer :: lwm
            integer :: liwm
            integer :: mxstep
            integer :: mxhnil
            integer :: nhnil
            integer :: ntrep
            integer :: nslast
            integer :: nyh
            integer dimension(6) :: iowns
```

```
63              integer :: icf
64              integer :: ierpj
65              integer :: iersl
66              integer :: jcur
67              integer :: jstart
68              integer :: kflag
69              integer :: l
70              integer :: meth
71              integer :: miter
72              integer :: maxord
73              integer :: maxcor
74              integer :: msbp
75              integer :: mxncf
76              integer :: n
77              integer :: nq
78              integer :: nst
79              integer :: nfe
80              integer :: nje
81              integer :: nqu
82              double precision dimension(2) :: rownr3
83              double precision :: t0
84              double precision :: tlast
85              double precision :: toutc
86              integer :: lg0
87              integer :: lg1
88              integer :: lgx
89              integer dimension(2) :: iownr3
90              integer :: irfnd
91              integer :: itaskc
92              integer :: ngc
93              integer :: nge
94              double precision :: tsw
95              double precision dimension(20) :: rowns2
96              double precision :: pdnorm
97              integer :: insufr
98              integer :: insufi
99              integer :: ixpr
100             integer dimension(2) :: iowns2
101             integer :: jtyp
102             integer :: mused
103             integer :: mxordn
104             integer :: mxords
105             common /ls0001/ rowns,ccmax,el0,h,hmin,hmxi,hu,rc,tn,uround,illin,init,lyh,lewt
                    ,lacor,lsavf,lwm,liwm,mxstep,mxhnil,nhnil,ntrep,nslast,nyh,iowns,icf,ierpj,
                    iersl,jcur,jstart,kflag,l,meth,miter,maxord,maxcor,msbp,mxncf,n,nq,nst,nfe,
                    nje,nqu
106             common /lsr001/ rownr3,t0,tlast,toutc,lg0,lg1,lgx,iownr3,irfnd,itaskc,ngc,nge
107             common /lsa001/ tsw,rowns2,pdnorm,insufr,insufi,ixpr,iowns2,jtyp,mused,mxordn,
                    mxords
108         end subroutine lsodar
109     end interface
110 end python module lsodar
111
112 ! This file was auto-generated with f2py (version:2.13.175-1250).
113 ! See http://cens.ioc.ee/projects/f2py2e/
```

Listing 7: lsodar/lsodarsimple.pyf

```
1  !%f90 -*- f90 -*-
2  python module _lsodar__user__routines
3      interface _lsodar_user_interface
4          subroutine integrand(neq, t, y, ydot)
5              integer optional,check(len(y)>=neq),depend(y), intent(in) :: neq=len(y)
6              double precision, intent(in) :: t
7              double precision dimension(neq), intent(in) :: y
8              double precision dimension(neq),depend(neq), intent(out), check(len(ydot)>=neq)
                   :: ydot
9          end subroutine integrand
```

```
10          subroutine border(neq, t, y, ng, gout)
11              integer optional,check(len(y)>=neq),depend(y), intent(in) :: neq=len(y)
12              double precision, intent(in) :: t
13              double precision dimension(neq), intent(in) :: y
14              integer optional,check(len(gout)==ng),depend(gout), intent(hide) :: ng=len(gout
                    )
15              double precision dimension(ng), intent(out) :: gout
16          end subroutine border
17          subroutine jacobi(neq,t,y,ml,mu,pd,nrowpd)
18              integer optional,intent(in), check(len(y)>=neq),depend(y) :: neq=len(y)
19              double precision,intent(in) :: t
20              double precision dimension(neq) :: y
21              integer, intent(in) :: ml
22              integer, intent(in) :: mu
23              double precision dimension(nrowpd,neq), depend(neq) :: pd
24              integer intent(in), optional,check(shape(pd,0)==nrowpd),depend(pd) :: nrowpd=
                    shape(pd,0)
25          end subroutine jacobi
26      end interface _lsodar_user_interface
27  end python module _lsodar__user__routines
28  python module _lsodarsimple ! in
29      interface  ! in :lsodar
30          subroutine lsodar(integrand,neq,y,t,tout,itol,rtol,atol,itask,istate,iopt,rwork,lrw
                ,iwork,liw,jacobi,jt,border,ng,jroot)
31              use _lsodar__user__routines
32              external integrand
33              integer :: neq
34              double precision dimension(neq), intent(in,out) :: y
35              double precision, intent(in,out) :: t
36              double precision, intent(in) :: tout
37              integer, intent(in) :: itol
38              ! If itol is one scalars are passed not arrays. Using MAX to inform f2py
39              double precision dimension(MAX(neq * (itol-1), 1)), intent(in) :: rtol
40              double precision dimension(MAX(neq * (itol-1), 1)), intent(in) :: atol
41              integer, intent(in) :: itask
42              integer, intent(in,out) :: istate
43              integer, intent(in), parameter :: iopt=1
44              double precision dimension(22 + neq * MAX(16, neq+9) + 3 * ng), intent(hide,
                    cache) :: rwork
45              integer dimension(20+neq), intent(hide,cache) :: iwork
46              integer depend(rwork), intent(hide) :: lrw=len(rwork)
47              integer depend(iwork), intent(hide) :: liw=len(iwork)
48              external jacobi
49              integer, intent(in) :: jt
50              integer, required, intent(in) :: ng
51              integer dimension(MAX(ng,1)), intent(out) :: jroot
52              external  border
53          end subroutine lsodar
54       end interface
55  end python module _lsodarsimple
```

Listing 8: lsodar/lsodarsimple.py

```
1   #!/usr/bin/env python
2   import Numeric
3   import _lsodarsimple
4
5   def lsodar(f, y, t, tout,  *args, **keywords):
6       """
7       Wrapper around lsodar from ODEPACK
8       """
9
10      itol = 1;  rtol = 1e-6;    atol = 1e-6
11      task = 1;  state = 1;
12      jac = None;     jt = 2
13      if keywords.has_key('border'):
14          border = keywords['border']
15          tmp = border(t,y)
```

```
16          if type(tmp) != type(Numeric.array((0,0))):
17              raise TypeError, "The border function must return an array!"
18
19          ng = len(tmp)
20          assert(ng != None)
21          assert(ng > 0)
22      else:
23          border = lambda x:x
24          ng = 0
25      if keywords.has_key('jacobi'):
26          jacobi = keywords['jacobi']
27      else:
28          jacobi = lambda x:x
29
30      assert(itol == 1 or itol == 2)
31      y,t,state,jroot = _lsodarsimple.lsodar(f, y, t, tout, itol, rtol, atol,
32                                             task, state, jacobi, jt,
33                                             border, ng)
34      return y, t, state, jroot
```

E.1 Wrapping it in a class

Listing 9: lsodar/lsodar.pyf

```
1  !%f90 -*- f90 -*-
2  python module _lsodar__user__routines
3      interface _lsodar_user_interface
4          subroutine integrand(neq, t, y, ydot)
5              integer optional,check(len(y)>=neq),depend(y), intent(in) :: neq=len(y)
6              double precision, intent(in) :: t
7              double precision dimension(neq), intent(in) :: y
8              double precision dimension(neq),depend(neq), intent(out), check(len(ydot)>=neq)
                   :: ydot
9          end subroutine integrand
10         subroutine border(neq, t, y, ng, gout)
11             integer optional,check(len(y)>=neq),depend(y), intent(in) :: neq=len(y)
12             double precision, intent(in) :: t
13             double precision dimension(neq), intent(in) :: y
14             integer optional,check(len(gout)==ng),depend(gout), intent(hide) :: ng=len(gout
                   )
15             double precision dimension(ng), intent(out) :: gout
16         end subroutine border
17         subroutine jacobi(neq,t,y,ml,mu,pd,nrowpd)
18             integer optional,intent(in), check(len(y)>=neq),depend(y) :: neq=len(y)
19             double precision,intent(in) :: t
20             double precision dimension(neq) :: y
21             integer, intent(in) :: ml
22             integer, intent(in) :: mu
23             double precision dimension(nrowpd,neq), depend(neq) :: pd
24             integer intent(in), optional,check(shape(pd,0)==nrowpd),depend(pd) :: nrowpd=
                   shape(pd,0)
25         end subroutine jacobi
26     end interface _lsodar_user_interface
27 end python module _lsodar__user__routines
28 python module _lsodar ! in
29     interface  ! in :lsodar
30         subroutine srcar(rsav,isav,job)
31             double precision, dimension(245), check(len(rsav)>=245) :: rsav
32             integer, dimension(59), check(len(isav)>=59) :: isav
33             integer :: job
34         end subroutine srcar
35         subroutine intdy(t, deriv, rwork, nyh, dky, iflag)
36             double precision, intent(in) :: t
37             integer, intent(in) :: deriv
38             double precision dimension(:), intent(inout) :: rwork
```

```
39              integer, intent(in), required :: nyh
40              double precision, dimension(nyh), intent(out) :: dky
41              integer intent(out) :: iflag
42          end subroutine intdy
43          subroutine lsodar_wrap(integrand,neq,y0,yout,t,n2,itol,rtol,atol,itask,istate,iopt,
                rwork,lrw,iwork,liw,jacobi,jt,border,ng,jroot,last)
44              use _lsodar__user__routines
45              external integrand
46              integer optional,check(len(y0)>=neq),depend(y0), intent(in) :: neq=len(y0)
47              double precision dimension(neq), intent(in) :: y0
48              double precision dimension(neq,n2 - 1),depend(neq),depend(n2), intent(out) ::
                    yout
49              double precision dimension(n2) :: t
50              integer optional,check(len(t)>=n2),depend(t) :: n2=len(t)
51              integer, intent(in) :: itol
52              ! If itol is one scalars are passed not arrays. Using MAX to inform f2py
53              double precision dimension(MAX(neq * (itol-1), 1)), intent(in) :: rtol
54              double precision dimension(MAX(neq * (itol-1), 1)), intent(in) :: atol
55              integer, intent(in) :: itask
56              integer, intent(in,out) :: istate
57              integer, intent(in) :: iopt
58              double precision dimension(22 + neq * MAX(16, neq+9) + 3 * ng), intent(inout)
                    :: rwork
59              integer optional,check(len(rwork)>=lrw),depend(rwork) :: lrw=len(rwork)
60              integer dimension(liw) :: iwork
61              integer optional,check(len(iwork)>=liw),depend(iwork),intent(inout) :: liw=len(
                    iwork)
62              external jacobi
63              integer, intent(in) :: jt
64              external border
65              integer, required, intent(in) :: ng
66              integer dimension(MAX(ng,1)), intent(out) :: jroot
67              integer, intent(out) :: last
68          end subroutine lsodar_wrap
69          subroutine lsodar(integrand,neq,y,t,tout,itol,rtol,atol,itask,istate,iopt,rwork,lrw
                ,iwork,liw,jacobi,jt,border,ng,jroot)
70              use _lsodar__user__routines
71              external integrand
72              integer, depend(y), intent(hide) :: neq=len(y)
73              double precision dimension(:), intent(in,copy,out) :: y
74              double precision, intent(in,copy,out) :: t
75              double precision, intent(in) :: tout
76              integer, intent(in) :: itol
77              ! If itol is one scalars are passed not arrays. Using MAX to inform f2py
78              double precision dimension(MAX(neq * (itol-1), 1)), intent(in), depend(itol)
                    :: rtol
79              double precision dimension(MAX(neq * (itol-1), 1)), intent(in), depend(itol)
                    :: atol
80              integer, intent(in) :: itask
81              integer, intent(in, out) :: istate
82              integer, intent(in) :: iopt
83              double precision dimension(:), check(len(rwork)>=(22 + neq * MAX(16, neq+9)
                    + 3 * ng)), depend(neq), depend(ng), intent(inout) :: rwork
84              integer dimension(:), check(len(iwork)>=(20+neq)), depend(neq), intent(inout)
                    :: iwork
85              integer depend(rwork), intent(hide) :: lrw=len(rwork)
86              integer depend(iwork), intent(hide) :: liw=len(iwork)
87              external jacobi
88              integer, intent(in) :: jt
89              integer, required, intent(in) :: ng
90              integer dimension(MAX(ng,1)), intent(out) :: jroot
91              external  border
92          end subroutine lsodar
93      end interface
94  end python module _lsodar
95
96  ! This file was auto-generated with f2py (version:2.13.175-1250).
97  ! See http://cens.ioc.ee/projects/f2py2e/
```

Listing 10: lsodar/ClassIntegrate.py

```
#!/usr/bin/env python
#-----------------------------------------------------------------------
# Author:    Pierre Schnizer <Pierre.Schnizer@cern.ch>
# Date:      05. 11. 2002
# Purpose:   To illustrate the usage and capability of f2py
# Version:   Not even alpha. This is illustration code, so do not use it in
#            production environement
# License:   LGPL
#-----------------------------------------------------------------------
import Numeric
import MLab
import _lsodar

class _LastJob:
    jobid = -2
    joblist = {}

    def GetLastJobId(self):
        return self.jobid

    def GetJob(self, id):
        return self.joblist[id]

    def Register(self, instance):
        tmp = id(instance)
        self.joblist[tmp] = instance
        return tmp

    def DeRegister(self, instance):
        tmp = id(instance)
        del self.joblist[tmp]

    def SetActiveJobId(self, id):
        self.jobid = id

_lastjob = _LastJob()

_msgs = {2: "Integration successful.",
         3: "Found Border",
         -1: "Excess work done on this call (perhaps wrong Dfun type).",
         -2: "Excess accuracy requested (tolerances too small).",
         -3: "Illegal input detected (internal error).",
         -4: "Repeated error test failures (internal error).",
         -5: "Repeated convergence failures (perhaps bad Jacobian or"+\
         " tolerances).",
         -6: "Error weight became zero during problem.",
         -7: "Internal workspace insufficient to finish (internal error)."
         }


#----------------------- Class Integrate -------------------------------
class Integrate:
    """
    Wraps lsodar. Allows multiple instances.

    LSODAR from the ODEPACK package provides a flexible ODE solver. It solves a
    system of first order differential equations. Further it searches for the
    roots in a function specified as border.

    For each problem you should set up a seperate object. Between calls the
    internal state is preserved.

    Currently jacobian matrices can not be supplied.
    """
    def __del__(self):
        self._lastjob.Deregister(self)

```

```
68      def __init__(self, integrand, y0, t0, **dic):
69          """
70          Input : integrand ... the function system describing the problem
71                  y0        ... the start vector for the dependent variables
72                  t0        ... the start value for the independen variable
73
74          optional: border  ... a funtion returning an array. When one of its
75                                values is zero,  the evaluation will terminate.
76          """
77          self._lastjob = _lastjob
78          self.jobid = self._lastjob.Register(self)
79          assert(id(self) == id(self._lastjob.GetJob(self.jobid)))
80
81          border = None
82          args = None
83          borderargs = None
84          y0 = y0
85          t0 = t0
86          if dic.has_key('border'):
87              border = dic['border']
88          if dic.has_key('args'):
89              args = tuple(dic['args'])
90          if dic.has_key('borderargs'):
91              borderargs = tuple(dic['borderargs'])
92
93          self.args = args
94          self.borderargs = borderargs
95          tmp = (t0, y0)
96          if args != None:
97              tmp += args
98          dimension = len(Numeric.array(apply(integrand, tmp)))
99          if border != None:
100             tmp = (t0, y0)
101             if borderargs != None:
102                 tmp += borderargs
103             self.borderdimension = len(Numeric.array(apply(border, tmp)))
104             self.border = border
105         else:
106             self.borderdimension = 0
107             self.border = lambda x: x
108         self.integrand = integrand
109         self.rsav   = Numeric.zeros((245), Numeric.Float)
110         self.isav   = Numeric.zeros((59), Numeric.Int)
111         rlength = 22 + dimension * MLab.max((16, dimension+9))
112         rlength += 3 * self.borderdimension
113         ilength = 20 + dimension
114         self.rwork = Numeric.zeros((rlength), Numeric.Float)
115         self.iwork = Numeric.zeros((ilength), Numeric.Int)
116
117         self.dimension = dimension
118         self.state = 1
119         self.itol = 1
120         self.rtol = Numeric.array(1e-6)
121         self.atol = Numeric.array(1e-6)
122         self.iopt = 0
123         self.y = y0
124         self.t = t0
125
126         # Enable Common Block handling of lsodar.f
127         self.needpostcleanup = -1
128
129         #print "id of _lastjob : ", id(self._lastjob),
130         #print "Printed from init :",id(self)
131
132     def _SetPrecision(self, which, precision):
133         tmp = Numeric.array(precision)
134         length = len(tmp)
135         assert(length == 1 or length == self.dimension)
```

```
136             if which == 'absolute':
137                 self.atol = tmp
138             if which == 'relative':
139                 self.rtol = tmp
140             al = len(self.atol)
141             rl = len(self.rtol)
142             if length > 1 or al > 1 or rl > 1:
143                 self.itol = 2
144             else:
145                 self.itol = 1
146             if self.itol == 2:
147                 #Multidimensional ...
148                 if al == 1:
149                     self.atol = self.atol * Numeric.ones(self.dimension)
150                 if il == 1:
151                     self.itol = self.itol * Numeric.ones(self.dimension)
152 
153 
154         def SetAbsolutePrecision(self, precision):
155             self._SetPrecision('absolute', precision)
156 
157 
158         def SetRelativePrecision(self, precision):
159             self._SetPrecision('relative', precision)
160 
161 
162 
163 
164         def Reinit(self, y0, t0):
165             self.state = 1
166             assert(len(y0) == self.dimension)
167             self.y = y0
168             self.t = t0
169 
170         def __call__(self, t, **dic):
171             """
172             The calculation routine.
173 
174             Input:  t     ...  values of the independent variable.
175                                The dependent variable will be caluclate
176                                for all values of t[1:]
177                                So the length of t must be 2 at minimum
178 
179             Keywords:  task: an index specifying the task to be performed.
180                                task has the following values and meanings.
181                                1  means normal computation of output values of y(t)
182                                   at t = tout (by overshooting and interpolating).
183                                2  means take one step only and return.
184                                3  means stop at the first internal mesh point at or
185                                   beyond t = tout and return.
186                                If you specifiy a critical point in evaluation only
187                                value 1 and 2 are allowed.
188 
189                        tcrit: Defines a critical point, where the problem should
190                                not be evaluted. tcrit may be equal to or beyond the
191                                last value of t, but not behind it in the direction
192                                of integration.  this option is useful if the problem
193                                has a singularity at or beyond t = tcrit.
194 
195 
196             Output: yout, t, jroot
197                     yout    ...   the values corresponding to t
198                     t       ...   passed to see what value t had if a border was
199                                   found
200                     jroot   ...   indicates which of the border conditions
201                                   has been fulfilled. The last values
202                                   of t an
203 
```

```
204         !!!!!!!!!!!!!!!!!!!
205         !!!   Warning   !!!!
206         !!!!!!!!!!!!!!!!!!!
207
208         Save's the system state internally. So use one object for each problem!
209         """
210         task = 1
211         if dic.has_key('task'):
212             task = dic['task']
213         if dic.has_key('tcrit'):
214             if task !=  1 and task != 2 :
215                 des = "Wrong Value of task when calculating with a critcal "+\
216                 "time. Task was " + str(task) + " but only 1 or 2 are allowed!"
217                 raise ValueError, des
218             task = task + 3
219             self.rwork[1 - 1] = dic['tcrit']
220         else:
221             if task !=  1 and task != 2 and task != 3:
222                 des = "Wrong Value of task when calculating with a critcal"+\
223                       " time. Task was " + str(task) + " but only 1, 2  or"+\
224                       " 3 are allowed!"
225                 raise ValueError, des
226
227         iopt = self.iopt
228
229         # Callback to jacobi not supported yet!
230         jacobi = lambda x:x
231         jt = 2
232
233         # Handle additional arguments to the functions
234         integrand = self.integrand
235         border    = self.border
236         if self.args != None:
237             def integrand(t,y, tmp, myint=self.integrand, myargs=self.args):
238                 tmp1 =  apply(myint, (t,y) + myargs)
239                 return tmp1
240
241         if self.borderargs != None:
242             def border(t,y, tmp, myb = self.border, myargs=self.borderargs):
243                 return apply(myb, (t,y) + myargs)
244
245         # Handle a vector input of y
246         result = None
247         if type(t) == Numeric.arraytype:
248             result = Numeric.zeros((len(self.y),
249                                     t.shape[0]), Numeric.Float)
250
251         # Call the FORTRAN routine
252         self.__prae() # Write Common Blocks
253         if result == None:
254             # Only one datum for t
255             y = self.y
256             t0 = self.t
257             tmp = _lsodar.lsodar(integrand, y, t0, t,
258                                  # Type of tolerances and tolerances
259                                  self.itol, self.rtol, self.atol,
260                                  # Task and status of calculation
261                                  task, self.state,
262                                  # Optional paramters and work arrays
263                                  iopt, self.rwork, self.iwork,
264                                  # Jacobi matrix, external not supported yet
265                                  jacobi, jt,
266                                  # Border conditions
267                                  border, self.borderdimension)
268             yout, t, state, jroot = tmp
269         else:
270             # handle a vector
271             y = self.y
```

```
272             # Do vector operations ...
273             for i in Numeric.arange(t.shape[0]):
274                 if i == 0:
275                     t0 = self.t
276                 else:
277                     t0 = t[i-1]
278                 tmp=_lsodar.lsodar(integrand, y, t0, t[i],
279                                    # Type of tolerances and tolerances
280                                    self.itol, self.rtol, self.atol,
281                                    # Task and status of calculation
282                                    task, self.state,
283                                    # Optional paramters and work arrays
284                                    iopt, self.rwork, self.iwork,
285                                    # Jacobi matrix, external not supported yet
286                                    jacobi, jt,
287                                    # Border conditions
288                                    border, self.borderdimension)
289                 y, tend, state, jroot = tmp
290                 result[:,i] = y
291                 if state < 1:
292                     # Error -> raise Error
293                     break
294                 if state == 3:
295                     break
296 
297                 # Important: Must be inside for loop
298                 self.state = state
299                 # End Of for loop -- to big loop ..
300             if state == 2 or state == 1:
301                 yout = result
302             elif state == 3:
303                 # Save the time when a border was encountered
304                 t[i] = tend
305                 t = t[:i+1]
306                 yout = result[:,:i+1]
307             # End of if  -- toooo big conditional
308 
309 
310         self.__post() # Save Common Block Status
311 
312 
313         # Handle errors from the routine
314         if state < 1:
315             raise ValueError, _msgs[state]
316         if result == None:
317             self.y = yout
318             self.t = t
319         else:
320             self.y = yout[:,-1]
321             self.t = t[-1]
322 
323         if state == 3:
324             # Border condition ...
325             self.state = 2
326             return yout, jroot, t
327 
328         self.state = state
329         return yout, None, None
330 
331 
332 
333     def Derivative(self, t, order):
334         """
335         Calculates the order derivative at the value t.
336 
337         !!!!!!!!!!!!!!!!!!!
338         !!!   Warning   !!!!
339         !!!!!!!!!!!!!!!!!!!
```

```
340         Can only be used after the calculation!
341         """
342         assert (self.state == 2 or self.state == 3)
343         self.__prae()
344         tmp = _lsodar.intdy(t, order, self.rwork, self.dimension)
345         self.__post()
346         return tmp
347
348     def _SaveCommonBlockData(self):
349         # Load common's information to arrays
350         _lsodar.srcar(self.rsav, self.isav, 1)
351
352     def _StoreCommonBlockData(self):
353         # Restore common's from arrays
354         _lsodar.srcar(self.rsav, self.isav, 2)
355
356     def __prae(self):
357         #print "id of _lastjob : ", id(self._lastjob),
358         #print "Printed from prae : ",id(self)
359         tmp = self._lastjob.GetLastJobId()
360
361         # Jobids lower than 0 mean nothing needs to be done
362         if  tmp != self.jobid and tmp > 0:
363             # Tell the other job to save its data
364             self._lastjob.GetJob(tmp)._SaveCommonBlockData()
365             self._StoreCommonBlockData()
366             self.needpostcleanup = tmp
367         else:
368             self.needpostcleanup = 0
369         #print "self.needpostcleanup = ", self.needpostcleanup
370
371         self._lastjob.SetActiveJobId(self.jobid)
372
373     def __post(self):
374         #print "id of _lastjob : ", id(self._lastjob),
375         #print "Printed from post : ",id(self)
376         if self.needpostcleanup > 0:
377             self._SaveCommonBlockData()
378             tmp = self._lastjob.GetJob(self.needpostcleanup)
379             tmp._StoreCommonBlockData()
380             self._lastjob.SetActiveJobId(self.needpostcleanup)
381             self.needpostcleanup = -1
382         elif self.needpostcleanup == 0:
383             pass
384         else:
385             raise TypeError, "self.needpostcleanup in instance " + `id(self)`+\
386                   " should be 0 or 1 \n but was found to: " +\
387                   `self.needpostcleanup`
```

Listing 11: lsodar/ClassIntegrateExpert.py

```
1  #!/usr/bin/env python
2  #-----------------------------------------------------------------------
3  # Author:    Pierre Schnizer <Pierre.Schnizer@cern.ch>
4  # Date:      05. 11. 2002
5  # Purpose:   To illustrate the usage and capability of f2py
6  # Version:   Not even alpha. This is illustration code, so do not use it in
7  #            production environement
8  # License:   LGPL
9  #-----------------------------------------------------------------------
10 import Numeric
11
12 import ClassIntegrate
13 Integrate = ClassIntegrate.Integrate
14
15 class IntegrateExpert(Integrate):
16     """
17     This class uses the same internal alghorithm as the class integrate,
```

```
18      but it offers many methods for fine tuning.
19
20      """
21      def _UseOptionalInput(self, precision):
22          """
23          Internal Method.
24          """
25          # lsodar uses a variable to find out if it should investigate the
26          # additional input paramters. These are honoured here.
27          self.iopt = 1
28          # Make _lsodar notice the change ... state 2 means, that init was
29          # allready done.
30          if self.state == 2:
31              self.state = 3
32
33      def SetFirstStepSize(self, size):
34          """
35          The step size to be attempted on the first step.
36          the default value is determined by the solver.
37
38          This method can only be called before the first
39          calculation!
40          """
41          if self.state != 1:
42              raise TypeError, """This method can only be called before
43              the problem is evaluated!"""
44          self.rwork[5 - 1] = size
45          self._UseOptionalInput()
46
47      def SetMaximumStepSize(self, size):
48          """
49          the maximum absolute step size allowed.
50          the default value is infinite.
51          """
52          self.rwork[6 - 1] = size
53          self._UseOptionalInput()
54
55      def SetMinimumStepSize(self, size):
56          """
57          the minimum absolute step size allowed.
58          the default value is 0.  (this lower bound is not
59          enforced on the final step before reaching tcrit
60          when task = 4 or 5.)
61          """
62          self.rwork[7 - 1] = size
63          self._UseOptionalInput()
64
65      def PrintAtMethodSwitch(self, bool):
66          """
67           flag to generate extra printing at method switches.
68           ixpr = 0 means no extra printing (the default).
69           ixpr = 1 means print data on each switch.
70           t, h, and nst will be printed on the same logical
71           unit as used for error messages.
72
73           Allowed Input 0 and 1
74
75           Warning : This output will go to the device/file with logical number
76           6 and thus depends which FORTRAN compiler translated the _lsodar
77           extension.
78          """
79          assert(bool  == 0 or bool == 1)
80          self.iwork[5 - 1] = bool
81          self._UseOptionalInput()
82
83      def SetMaximumNumberOfSteps(self, step):
84          """
85          Maximum number of (internally defined) steps
```

```
86          allowed during one call to the solver.
87          the default value is 500.
88          """
89          assert(step > 0)
90          self.iwork[6 - 1] = step
91          self._UseOptionalInput()
92
93      def SetMaximumNumberOfMessages(self, step):
94          """
95           Maximum number of messages printed (per problem)
96           warning that t + h = t on a step (h = step size).
97           this must be positive to result in a non-default
98           value.  the default value is 10.
99          """
100         #assert(step > 0)
101         self.iwork[7 - 1] = step
102         self._UseOptionalInput()
103
104     def SetMaximumOrderForNonStiffSolver(self, order):
105         """
106         The maximum order to be allowed for the nonstiff
107         (adams) method.  the default value is 12.
108         Higher values are not accepted.
109         This value is held constant during the problem.
110         """
111         assert(order <= 12  or order >= 0)
112         self.iwork[8 - 1] = order
113         self._UseOptionalInput()
114
115     def SetMaximumOrderForStiffSolver(self, order):
116         """
117         the maximum order to be allowed for the stiff
118         (bdf) method.  the default value is 5.
119         Higher values are not accepted.
120         This value is held constant during the problem.
121         """
122         assert(order <= 5  or order >= 0)
123         self.iwork[9 - 1] = order
124         self._UseOptionalInput()
125
126     def GetLastStepSize(self):
127         """
128         the step size in t last used (successfully)
129         """
130         return self.rwork[11 - 1]
131
132     def GetNextStepSize(self):
133         """
134         the step size to be attempted on the next step
135         """
136         return self.rwork[12 - 1]
137
138     def GetCurrentValueOfIndependentVariable(self):
139         """
140          the current value of the independent variable
141          which the solver has actually reached, i.e. the
142          current internal mesh point in t.  on output, tcur
143          will always be at least as far as the argument
144          t, but may be farther (if interpolation was done).
145         """
146         return self.rwork[13 - 1]
147
148     def GetToleranceFactor(self):
149         """
150          a tolerance scale factor, greater than 1.0,
151          computed when a request for too much accuracy was
152          detected (state = -3 if detected at the start of
153          the problem, state = -2 otherwise).  if the relative
```

```
154             and the absolute accuracy are uniformly
155             scaled up by a factor of tolsf for the next call,
156             then the solver is deemed likely to succeed.
157             (the user may also ignore tolsf and alter the
158             tolerance parameters in any other way appropriate.)
159             """
160             return self.rwork[14 - 1]
161
162         def GetValueAtLastMethodSwitch(self):
163             """
164             the value of t at the time of the last method
165             switch, if any
166             """
167             return self.rwork[15 - 1]
168
169         def GetNumberOfBorderEvaluations(self):
170             return self.iwork[10 - 1]
171
172         def GetNumberOfSteps(self):
173             """
174             the number of steps taken for the problem so far.
175             """
176             return self.iwork[11 - 1]
177
178         def GetNumberOfIntegrandEvaluations(self):
179             """
180             the number of integrand evaluations for the problem so far.
181             """
182             return self.iwork[12 - 1]
183         def GetNumberOfJacobiEvaluations(self):
184             """
185             the number of jacobian evaluations (and of matrix
186             lu decompositions) for the problem so far.
187             """
188             return self.iwork[13 - 1]
189
190         def GetMethodOrder(self):
191             """
192             the method order last used (successfully)
193             """
194             return self.iwork[14 - 1]
195
196         def GetNextMethodOrder(self):
197             """
198             the method order attempted on the next step
199             """
200             return self.iwork[15 - 1]
201
202         def GetIndexOfLargestErrorMagnitude(self):
203             """
204             the index of the component of largest magnitude in
205             the weighted local error vector ( e(i)/ewt(i) ),
206             on an error return with state = -4 or -5.
207             """
208             return self.iwork[16 - 1]
209
210         def GetMethodOfLastStep(self):
211             """
212             the method indicator for the last successful step..
213             1 means adams (nonstiff), 2 means bdf (stiff).
214             """
215             tmp = self.iwork[19 -1]
216             if tmp == 1:
217                 string = 'non stiff'
218             else:
219                 string = 'stiff'
220             return tmp, string
221
```

```
222     def GetMethodOfNextStep(self):
223         """
224         the method indicator for the next step..
225         1 means adams (nonstiff), 2 means bdf (stiff).
226         """
227         tmp = self.iwork[19-1]
228         if tmp == 1:
229             string = 'non stiff'
230         else:
231             string = 'stiff'
232         return tmp, string
233 
234 #----------------------End Class Integrate Expert-------------------------------
```

Listing 12: An example using the objects

```
1  #!/usr/bin/env python
2  #-----------------------------------------------------------------------
3  # Author:    Pierre Schnizer <Pierre.Schnizer@cern.ch>
4  # Date:      05. 11. 2002
5  # Purpose:   To illustrate the usage and capability of f2py
6  # Version:   Not even alpha. This is illustration code, so do not use it in
7  #            production environement
8  #-----------------------------------------------------------------------
9  import time
10 import copy
11 import sys
12 
13 import Numeric
14 import Gnuplot
15 
16 import lsodar
17 Integrate = lsodar.Integrate
18 IntegrateExpert = lsodar.IntegrateExpert
19 
20 gravity = 9.81
21 
22 def BulletMovement(t, vec, weight, resistance):
23     """
24     Use the equation descibing a bullet
25 
26     x  ... vec[0]
27     vx ... vec[1]
28     y  ... vec[2]
29     vy ... vec[3]
30 
31     """
32     #sys.stderr.write("BM1 " + str(t) + "\n")
33     vec = copy.copy(vec)
34 
35     x  = vec[0]
36     vx = vec[1]
37     y  = vec[2]
38     vy = vec[3]
39 
40     tmp = Numeric.ones(4, Numeric.Float)
41     tmp[0] =  vx
42     tmp[1] =  resistance * vx
43     tmp[2] =  vy
44     tmp[3] =  -weight * gravity + resistance * vy
45 
46     return tmp
47 
48 
49 def Border(t, vec, bullet, t2, result, index = [1,],
50            last=Numeric.zeros((5,5), Numeric.Float), lastindex = [0,]):
51     """
52     bullet describes the movement of the second bullet
```

```
53      """
54
55      t = copy.copy(t)
56      vec = copy.copy(vec)
57
58
59      # Calculate the postion for all t2's which are not reached yet
60      tindex = index[0]
61      if t > t2[tindex] and t <= t2[-1]:
62          indices = Numeric.less_equal(t2[tindex:], t)
63          ttmp = Numeric.compress(indices, t2[tindex:])
64          index[0] = int(index[0]) + len(ttmp)
65          rt = list(bullet(ttmp)[0])
66          rt = Numeric.transpose(Numeric.array(rt))
67          map(result.append, rt)
68
69      #Get the position of the second bullet
70      r2 = bullet.GetCurrentValueOfIndependentVariable()
71      r1 = r2 - bullet.GetLastStepSize()
72
73      if t<r1:
74          # T out of scope we have to recalculate
75          test = t - last[0,:]
76          i = Numeric.argmin(Numeric.clip(test, 0.0, float('Inf')))
77          ttmp =  last[0, i]
78          tvec =  last[1:, i]
79          bullet.Reinit(tvec,  ttmp)
80          tmp = bullet(t)
81      else:
82          tmp = bullet(t)
83          last[0, lastindex[0]] = t
84
85          last[1:, lastindex[0]] = tmp[0]
86          lastindex[0] = (lastindex[0] + 1) % 5
87
88
89      assert(tmp[1] == None)
90      assert(tmp[2] == None)
91      pos = copy.copy(tmp[0])
92      tmp = pos[::2] - vec[::2]
93      tmp1 = Numeric.sum(tmp ** 2) + tmp[0]
94      return tmp1
95
96  def run():
97      t = Numeric.arange(1000) / 1000.0 * 2
98      result2 = []
99      bullet2 = IntegrateExpert(BulletMovement, Numeric.array((25., -10, 0, 10)),
100                                0, args=(1.0, .1))
101     bullet1 = IntegrateExpert(BulletMovement, Numeric.array((0., 10, 0.0, 10)),
102                               0, args=(1.0, 0.1), border=Border,
103                               borderargs=(bullet2, t, result2))
104
105
106     tmp, jroot, t1 = bullet1(t)
107     if t1 != None:
108         t = t1
109
110     result = Numeric.transpose(tmp)
111     result2 = Numeric.array(result2)
112
113
114     tmp = t[:len(result2[:,0])]
115     x = result2[:,0]
116
117
118     g = Gnuplot.Gnuplot()
119     command = """
120     set grid
```

```
121     set xlabel 'x [m]'
122     set ylabel 'y [m]'
123     """
124     g(command)
125
126
127     g.plot(Gnuplot.Data(result[:,0], result[:,2], title='b1', with='line'),
128             Gnuplot.Data(result2[:,0], result2[:,2], title='b2', with='line'),
129             )
130     print result.shape
131     print result2.shape
132
133     raw_input()
134
135
136 if __name__ == '__main__':
137     run()
```