Calling the NAG Fortran Library from Python using F2PY

Mat Cross, NAG *Ltd*

1 Introduction

The Python tool F2PY [9] is an automatic generator of Python wrappers for C and FORTRAN subprograms. Guides on the general use of F2PY exist: one written by Langtangen [1] and one within the documentation for the SAGE package [11].
F2PY is not the perfect solution for wrapping a NAG Library. Its primary advantage is that it is quite smart: a lot of the complications inherent in mixed-language programming are handled automatically, and the generated interfaces are Pythonic; for example, array dimensions can be hidden as optional arguments, arguments are added for communicating arbitrary data to callback functions, and workspaces can be removed and handled internally. The standard headaches (such as with calling conventions for strings, differences in array storage, and interfacing to callback functions) that are encountered when using a more-direct method of producing wrappers (e.g., by using the Python `ctypes` module) are (usually) taken care of.
A disadvantage of F2PY is that sometimes it can try to be **too** smart. If its idea of how your interface should look is inappropriate you might need to explicitly tell it exactly what to do. It's also nonintuitive—all the details of what it actually does can get hidden away—and it can be difficult to fine tune (see Section 5 for an example of an array dimension it can't handle, and Section 6 for a tale of when more-involved fine tuning is necessary).
Further, only a subset of FORTRAN 90 is currently parseable by F2PY. Since the NAG interfaces are strictly FORTRAN 77 and are designed to be amenable to mixed-language use, this is unlikely to cause problems.
In Section 6 we'll see how in general it can be a little more difficult to use F2PY on WINDOWS than on LINUX.

2 Set up

To run the specific examples in this article you'll need access to a NAG FORTRAN Library. We're going to work with shared libraries, so you'll also need to make sure that the linker has in its search path the locations of the (shared) FORTRAN Library and any dependent vendor libraries. If not, you'll need to know those locations explicitly (contact your system guru). Let's assume that these materials reside in the directory named in the environment variable `NAG_FL_DIR`. To use F2PY to build your own wrappers to the Library in addition to the examples given in this paper, you'll find the FORTRAN interface-block modules that are distributed with the Library useful (see your Library's Installer's Note, `in.html`, for their location).
Let's use a LINUX `gfortran` `libnag_acml.so` FORTRAN Library for the examples below. That means we need to have `${NAG_FL_DIR}` somewhere in the `LD_LIBRARY_PATH` environment variable at runtime. I'm using F2PY Version `2_4422` with Python 2.5.2, and the NAG `a00aafe` example program for the FORTRAN Library I'm using outputs

```
*** Start of NAG Library implementation details ***

Implementation title: AMD64, Linux64, GFORTRAN
           Precision: FORTRAN double precision
        Product Code: FLL6A21DFL
                Mark: 21 E

*** End of NAG Library implementation details ***
```

3 S21BAF

A very simple subprogram to wrap is S21BAF [8], which computes the degenerate symmetrised elliptic integral of the first kind. Here is its specification:

```
  FUNCTION S21BAF(X, Y, IFAIL)
  DOUBLE PRECISION S21BAF, X, Y
  INTEGER          IFAIL
```

Moreover, the subprogram's documentation tells us that X and Y are input variables, while IFAIL is input/output. In conjunction with the interface for S21BAF in nag_f77_s_chapter.f90, this allows us to form the following signature file, nag_py_s21baf.pyf, for F2PY:

```
python MODULE nag_py_s21baf
  PUBLIC
  INTERFACE
     FUNCTION S21BAF(X,Y,IFAIL) RESULT(F)
        DOUBLE PRECISION :: F
        DOUBLE PRECISION, INTENT (IN) :: X
        DOUBLE PRECISION, INTENT (IN) :: Y
        INTEGER, INTENT (IN,OUT) :: IFAIL
     END FUNCTION S21BAF
  END INTERFACE
END python MODULE nag_py_s21baf
```

Note that by using the specification INTENT(IN,OUT) for IFAIL we tell F2PY that this argument is input **and** output. The use of INTENT(INOUT) is discouraged because it instructs F2PY to generate an in-place-modified argument.
Now invoke F2PY using

```
shell-prompt> f2py -c --lower --fcompiler='gnu95' nag_py_s21baf.pyf \
                  -L${NAG_FL_DIR}/lib -lnag_acml \
                  -L${NAG_FL_DIR}/acml -lacml
```

The option -c triggers the building of the Python extension module containing the Python wrapper to S21BAF, --fcompiler='gnu95' specifies that gfortran should be used as compiler and linker (which is important since we're using the gfortran NAG Library), and --lower downcases the subprogram names in the signature file (which makes the naming more Pythonic). We pass the shared ACML libnag and ACML itself to the linker. Static libraries can be used instead of shared if required, but they'll need to be explicitly added to the end of the F2PY command.
Note that the arguments you need to give to F2PY will depend on the FORTRAN compiler used to create the libnag against which you link. See the F2PY documentation [10] for accepted arguments.
Now we can check that the extension module has been created successfully (it should be in the current directory):

```
shell-prompt> python -c "import nag_py_s21baf, sys; \
                         sys.stdout.writelines(nag_py_s21baf.__doc__ + '\n')"
This module 'nag_py_s21baf' is auto-generated with f2py (version:2_4422).
Functions:
  f,ifail = s21baf(x,y,ifail)
.
```

(F2PY automatically adds a __doc__ method to the wrappers it generates).
Now we're ready to invoke the function in Python, including one illegal call with argument x negative:

```
py-prompt>>> from nag_py_s21baf import s21baf; from sys import stdout
py-prompt>>> stdout.writelines(s21baf.__doc__)
s21baf - Function signature:
  f,ifail = s21baf(x,y,ifail)
Required arguments:
  x : input float
  y : input float
  ifail : input int
Return objects:
  f : float
  ifail : int
```

```
py-prompt>>> s21baf(x=0.5, y=1.0, ifail=-1)
(1.1107207345395913, 0)
py-prompt>>> s21baf(x=-0.5, y=1.0, ifail=-1)
 ** On entry, X is less than zero: X = -5.00000E-01.
 ** ABNORMAL EXIT from NAG Library routine S21BAF: IFAIL =     1
 ** NAG soft failure - control returned
(0.0, 1)
py-prompt>>> exit()
```

4 C05ADF

F2PY can also wrap FORTRAN subprograms that take procedure arguments, as long as you provide interfaces for such arguments in the signature file. Consider the following signature file for the root-finder `C05ADF` [4]:

```
python MODULE __user__routines
  INTERFACE
     FUNCTION C05ADF_F(X) RESULT (F)
       DOUBLE PRECISION :: X
       DOUBLE PRECISION :: F
     END FUNCTION C05ADF_F
  END INTERFACE
END python MODULE  __user__routines

python MODULE nag_py_c05adf
  PUBLIC
  INTERFACE
    SUBROUTINE C05ADF(A,B,EPS,ETA,F,X,IFAIL)
       USE __user__routines, F => C05ADF_F
       DOUBLE PRECISION, INTENT (IN) :: A
       DOUBLE PRECISION, INTENT (IN) :: B
       DOUBLE PRECISION, INTENT (IN) :: EPS
       DOUBLE PRECISION, INTENT (IN) :: ETA
       EXTERNAL :: F
       DOUBLE PRECISION, INTENT (OUT) :: X
       INTEGER, INTENT (IN,OUT) :: IFAIL
    END SUBROUTINE C05ADF
  END INTERFACE
END python MODULE nag_py_c05adf
```

Save that to `nag_py_c05adf.pyf` and then run

```
shell-prompt> f2py -c --lower --fcompiler='gnu95' nag_py_c05adf.pyf \
                  -L${NAG_FL_DIR}/lib -lnag_acml \
                  -L${NAG_FL_DIR}/acml -lacml
```

and check

```
shell-prompt> python -c "import nag_py_c05adf, sys; \
                         sys.stdout.writelines(nag_py_c05adf.__doc__ + '\n')"
This module 'nag_py_c05adf' is auto-generated with f2py (version:2_4422).
Functions:
  x,ifail = c05adf(a,b,eps,eta,f,ifail,f_extra_args=())
.
```

This time F2PY has added an optional argument through which data can be communicated to the objective function (we'll hear more about using this feature in Section 5). Let's find some zeros:

```
py-prompt>>> from nag_py_c05adf import c05adf; from sys import stdout
py-prompt>>> from math import exp
py-prompt>>> stdout.writelines(c05adf.__doc__)
c05adf - Function signature:
  x,ifail = c05adf(a,b,eps,eta,f,ifail,[f_extra_args])
```

```
Required arguments:
  a : input float
  b : input float
  eps : input float
  eta : input float
  f : call-back function => c05adf_f
  ifail : input int
Optional arguments:
  f_extra_args := () input tuple
Return objects:
  x : float
  ifail : int
Call-back functions:
  def c05adf_f(x): return f
  Required arguments:
    x : input float
  Return objects:
    f : float
py-prompt>>> c05adf(a=0.0, b=1.0, eps=1.0e-5, eta=0.0, f=(lambda x: exp(-x) - x), ifail=-1)
(0.56714329029094324, 0)
py-prompt>>> exit()
```

5 E04UCA

FORTRAN subprograms in the E04 Chapter [5] have some of the most complicated interfaces in the Library. Let's try wrapping E04UCA [6]. Its FORTRAN interface (with INTENTs) is

```
SUBROUTINE E04UCA(N,NCLIN,NCNLN,LDA,LDCJ,LDR,A,BL,BU,CONFUN,OBJFUN, &
      ITER,ISTATE,C,CJAC,CLAMDA,OBJF,OBJGRD,R,X,IWORK,LIWORK,WORK, &
      LWORK,IUSER,RUSER,LWSAV,IWSAV,RWSAV,IFAIL)
   INTEGER, INTENT (IN) :: N
   INTEGER, INTENT (IN) :: NCLIN
   INTEGER, INTENT (IN) :: NCNLN
   INTEGER, INTENT (IN) :: LDA
   INTEGER, INTENT (IN) :: LDCJ
   INTEGER, INTENT (IN) :: LDR
   DOUBLE PRECISION, INTENT (IN) :: A(LDA,*)
   DOUBLE PRECISION, INTENT (IN) :: BL(N+NCLIN+NCNLN)
   DOUBLE PRECISION, INTENT (IN) :: BU(N+NCLIN+NCNLN)
   INTEGER, INTENT (OUT) :: ITER
   INTEGER, INTENT (INOUT) :: ISTATE(N+NCLIN+NCNLN)
   DOUBLE PRECISION, INTENT (OUT) :: C(MAX(1,NCNLN))
   DOUBLE PRECISION, INTENT (INOUT) :: CJAC(LDCJ,*)
   DOUBLE PRECISION, INTENT (INOUT) :: CLAMDA(N+NCLIN+NCNLN)
   DOUBLE PRECISION, INTENT (OUT) :: OBJF
   DOUBLE PRECISION, INTENT (OUT) :: OBJGRD(N)
   DOUBLE PRECISION, INTENT (INOUT) :: R(LDR,N)
   DOUBLE PRECISION, INTENT (INOUT) :: X(N)
   INTEGER, INTENT (IN) :: LIWORK
   INTEGER, INTENT (IN) :: LWORK
   INTEGER, INTENT (INOUT) :: IUSER(*)
   DOUBLE PRECISION, INTENT (INOUT) :: RUSER(*)
   LOGICAL, INTENT (INOUT) :: LWSAV(120)
   INTEGER, INTENT (INOUT) :: IWSAV(610)
   DOUBLE PRECISION, INTENT (INOUT) :: RWSAV(475)
   INTEGER, INTENT (INOUT) :: IFAIL
   INTEGER, INTENT (OUT) :: IWORK(LIWORK)
   DOUBLE PRECISION, INTENT (OUT) :: WORK(LWORK)
```

```
   EXTERNAL CONFUN
   EXTERNAL OBJFUN
END SUBROUTINE E04UCA
```

So there are two callback arguments (`CONFUN` and `OBJFUN`), several array arguments (which F2PY will have to transpose in going from C array storage to FORTRAN array storage), four assumed-size arrays (`A`, `CJAC` and the user-communication arrays `IUSER` and `RUSER`), three arrays with leading dimensions (`A`, `CJAC` and `R`), two workspace arrays (`IWORK` and `WORK`) and several array dimensions that could be made optional (`N`, `NCLIN` and `NCNLN`). Note that the subprogram's communication arrays `LWSAV`, `IWSAV` and `RWSAV` must always be passed in and out and not modified by us: they're used to communicate information (such as the values of optional parameters) between calls in a thread-safe manner.

Leading dimensions are a legacy consequence of FORTRAN 77 not allowing information about the dimensions of array arguments to be passed automatically to a subprogram. We can hide the three leading-dimension integers `LDA`, `LDCJ` and `LDR` by setting them `INTENT(IN,HIDE)` and providing default values for them (otherwise F2PY will default them to `0` and `E04UCA` will fail). We also need to remember to redimension the associated arrays accordingly: for example, `R(N,N)`, because `N` is really the intended row length of `R`, as we see from the description of `LDR` in the `E04UCA` document.

A shortcoming of F2PY is that it cannot handle the (legal) dimension declaration for the array `C`[1]. We'll have to re-declare `C` to have dimension `NCNLN`. Understandably, F2PY also needs to be handed explicit lengths for all assumed-size arrays that will be present in the Python interface. In our case we'll have to hide `IUSER` and `RUSER` from `e04uca` in the signature file and declare them of dimension 1. Fortunately the optional communication arguments added automatically by F2PY will do part of their job for us instead. It's important to note though that some functionality is lost here: with the F2PY interface it is not possible to pass data back **out** of callback functions, as you can using the Library communication arrays.

We'll need to cheat slightly with the assumed-size rank-2 arrays. Taking a look at the documentation for `E04UCA` we see, for example, that the second dimension of `A` is `1` or `N` depending on `NCLIN`. This allows memory to be saved if there are no linear constraints. We're not fussed about that potential saving right now, so let's just declare `A(NCLIN,N)` in the signature file. Mutatis mutandis for `CJAC`.

F2PY can be told to hide workspace. In `E04UCA` the integer workspace has a length that is a FORTRAN *scalar-int-expr*, so we can easily hide it from Python, but the real workspace does not. Were we creating a wrapper for `E04UCA` by hand we could encode a logical block for computing `LWORK` in our wrapper and allocate `WORK` internally. Instead we'll have to retain this workspace in the Python interface, and it's declared `INTENT(IN)` because the `E04UCA` document doesn't say there's anything of interest in it on exit.

`E04UCA` must be initialized using `E04WBF` [7], so we'll need to tell F2PY about that subprogram too. Here's the full signature file (`nag_py_e_chapter.pyf`):

```
python MODULE __user__routines
  INTERFACE
     SUBROUTINE E04UCA_CONFUN(MODE,NCNLN,N,LDCJ,NEEDC,X,C,CJAC,NSTATE, &
          IUSER,RUSER)
       INTEGER, INTENT(IN,OUT) :: MODE
       INTEGER, INTENT(IN) :: NCNLN
       INTEGER, INTENT(IN) :: N
       INTEGER, INTENT(IN) :: LDCJ
       INTEGER, INTENT(IN) :: NEEDC(NCNLN)
       DOUBLE PRECISION, INTENT(IN) :: X(N)
       DOUBLE PRECISION, INTENT(OUT) :: C(NCNLN)
       DOUBLE PRECISION, INTENT(IN,OUT) :: CJAC(LDCJ,N)
       INTEGER, INTENT(IN) :: NSTATE
       INTEGER, INTENT(HIDE) :: IUSER(1)
       DOUBLE PRECISION, INTENT(HIDE) :: RUSER(1)
     END SUBROUTINE E04UCA_CONFUN
     SUBROUTINE E04UCA_OBJFUN(MODE,N,X,OBJF,OBJGRD,NSTATE,IUSER,RUSER)
       INTEGER, INTENT(IN,OUT) :: MODE
       INTEGER, INTENT(IN) :: N
       DOUBLE PRECISION, INTENT(IN) :: X(N)
       DOUBLE PRECISION, INTENT(OUT) :: OBJF
       DOUBLE PRECISION, INTENT(IN,OUT) :: OBJGRD(N)
```

1. P. Peterson informs me this has been fixed in the development tree.

```
      INTEGER, INTENT(IN) :: NSTATE
      INTEGER, INTENT(HIDE) :: IUSER(1)
      DOUBLE PRECISION, INTENT(HIDE) :: RUSER(1)
    END SUBROUTINE E04UCA_OBJFUN
  END INTERFACE
END python MODULE  __user__routines

python MODULE nag_py_e_chapter
  PUBLIC
  INTERFACE
    SUBROUTINE E04UCA(N,NCLIN,NCNLN,LDA,LDCJ,LDR,A,BL,BU,CONFUN,OBJFUN, &
               ITER,ISTATE,C,CJAC,CLAMDA,OBJF,OBJGRD,R,X,IWORK,LIWORK,WORK, &
               LWORK,IUSER,RUSER,LWSAV,IWSAV,RWSAV,IFAIL)
      USE __user__routines, CONFUN => E04UCA_CONFUN, &
                            OBJFUN => E04UCA_OBJFUN
      INTEGER, INTENT(IN) :: N
      INTEGER, INTENT(IN) :: NCLIN
      INTEGER, INTENT(IN) :: NCNLN
      INTEGER, INTENT(IN,HIDE) :: LDA=shape(A,0)
      INTEGER, INTENT(IN,HIDE) :: LDCJ=shape(CJAC,0)
      INTEGER, INTENT(IN,HIDE) :: LDR=shape(R,0)
      DOUBLE PRECISION, INTENT (IN) :: A(NCLIN,N)
      DOUBLE PRECISION, INTENT (IN) :: BL(N+NCLIN+NCNLN)
      DOUBLE PRECISION, INTENT (IN) :: BU(N+NCLIN+NCNLN)
      INTEGER, INTENT (OUT) :: ITER
      INTEGER, INTENT (IN,OUT) :: ISTATE(N+NCLIN+NCNLN)
      DOUBLE PRECISION, INTENT (OUT) :: C(NCNLN)
      DOUBLE PRECISION, INTENT (IN,OUT) :: CJAC(NCNLN,N)
      DOUBLE PRECISION, INTENT (IN,OUT) :: CLAMDA(N+NCLIN+NCNLN)
      DOUBLE PRECISION, INTENT (OUT) :: OBJF
      DOUBLE PRECISION, INTENT (OUT) :: OBJGRD(N)
      DOUBLE PRECISION, INTENT (IN,OUT) :: R(N,N)
      DOUBLE PRECISION, INTENT (IN,OUT) :: X(N)
      INTEGER, INTENT (IN,HIDE) :: LIWORK=len(IWORK)
      INTEGER, INTENT (IN) :: LWORK
      INTEGER, INTENT (HIDE) :: IUSER(1)
      DOUBLE PRECISION, INTENT (HIDE) :: RUSER(1)
      LOGICAL, INTENT (IN,OUT) :: LWSAV(120)
      INTEGER, INTENT (IN,OUT) :: IWSAV(610)
      DOUBLE PRECISION, INTENT (IN,OUT) :: RWSAV(475)
      INTEGER, INTENT (IN,OUT) :: IFAIL
      INTEGER, INTENT (IN,HIDE) :: IWORK(3*N+NCLIN+2*NCNLN)
      DOUBLE PRECISION, INTENT (IN) :: WORK(LWORK)
      EXTERNAL CONFUN
      EXTERNAL OBJFUN
    END SUBROUTINE E04UCA
    SUBROUTINE E04WBF(RNAME,CWSAV,LCWSAV,LWSAV,LLWSAV,IWSAV,LIWSAV, &
                      RWSAV,LRWSAV,IFAIL)
      CHARACTER*6, INTENT (IN) :: RNAME
      INTEGER, INTENT (IN) :: LCWSAV
      INTEGER, INTENT (IN) :: LLWSAV
      INTEGER, INTENT (IN) :: LIWSAV
      INTEGER, INTENT (IN) :: LRWSAV
      INTEGER, INTENT (IN,OUT) :: IFAIL
      CHARACTER*80, INTENT (OUT) :: CWSAV(LCWSAV)
      LOGICAL, INTENT (OUT) :: LWSAV(LLWSAV)
      INTEGER, INTENT (OUT) :: IWSAV(LIWSAV)
      DOUBLE PRECISION, INTENT (OUT) :: RWSAV(LRWSAV)
```

```
    END SUBROUTINE E04WBF
  END INTERFACE
END python MODULE nag_py_e_chapter
```

Then we have

```
shell-prompt> f2py -c --lower --fcompiler='gnu95' nag_py_e_chapter.pyf \
                  -L${NAG_FL_DIR}/lib -lnag_acml \
                  -L${NAG_FL_DIR}/acml -lacml
shell-prompt> python -c "import nag_py_e_chapter, sys; \
                         sys.stdout.writelines(nag_py_e_chapter.__doc__ + '\n')"
This module 'nag_py_e_chapter' is auto-generated with f2py (version:2_4422).
Functions:
  iter,istate,c,cjac,clamda,objf,objgrd,r,x, lwsav,iwsav,rwsav,ifail = \
      e04uca(a,bl,bu,confun,objfun,istate,cjac,clamda,r,x,\
             work,lwsav,iwsav,rwsav,ifail,\
             n=shape(a,1),nclin=shape(a,0),ncnln=shape(cjac,0),\
             lwork=len(work),confun_extra_args=(),objfun_extra_args=())
  cwsav,lwsav,iwsav,rwsav,ifail = \
      e04wbf(rname,lcwsav,llwsav,liwsav,lrwsav,ifail)
.
```

Notice the optional array lengths and hidden user-communication arrays. For demonstration purposes, let's use the `confun_extra_args` facility to pass a print flag through to the constraint function. The F2PY documentation [10] describes how the tool deals with potential mismatches in callback argument lists when using this facility.

The following Python script duplicates the simple example from the FORTRAN Library example program `e04ucae.f`.

```
#!/usr/bin/env python
import sys
from nag_py_e_chapter import e04uca, e04wbf
from numpy import *

def e04uca_confun(mode,needc,x,cjac,nstate,quiet_con):
    c = zeros(len(needc), 'd')

    if nstate == 1:

        # First call. Set all Jacobian elements to zero.
        # Note that this will only work when 'Derivative Level = 3'
        # (the default).

        if (not quiet_con):
            sys.stdout.write(' (First call to confun)\n')

        cjac = zeros(cjac.shape, 'd')

    if needc[0] > 0:

        if mode == 0 or mode == 2:
            c[0] = x[0]**2 + x[1]**2 + x[2]**2 + x[3]**2

        if mode == 1 or mode == 2:
            cjac[0,0] = 2.0*x[0]
            cjac[0,1] = 2.0*x[1]
            cjac[0,2] = 2.0*x[2]
            cjac[0,3] = 2.0*x[3]

    if needc[1] > 0:
```

```
        if mode == 0 or mode == 2:
            c[1] = x[0]*x[1]*x[2]*x[3]

        if mode == 1 or mode == 2:
            cjac[1,0] = x[1]*x[2]*x[3]
            cjac[1,1] = x[0]*x[2]*x[3]
            cjac[1,2] = x[0]*x[1]*x[3]
            cjac[1,3] = x[0]*x[1]*x[2]

    return mode, c, cjac

def e04uca_objfun(mode,x,objgrd,nstate):

    if mode == 0 or mode == 2:
        objf = x[0]*x[3]*(x[0] + x[1] + x[2]) + x[2]
    else:
        objf = None

    if mode == 1 or mode == 2:
        objgrd[0] = x[3]*(2.0*x[0] + x[1] + x[2])
        objgrd[1] = x[0]*x[3]
        objgrd[2] = x[0]*x[3] + 1.0
        objgrd[3] = x[0]*(x[0] + x[1] + x[2])

    return mode, objf, objgrd

# Failure modes
noisy, hard, quiet = -1, 0, 1
infail = noisy

nclin = 1
ncnln = 2
n = 4

a = array([[1.0, 1.0, 1.0, 1.0]])
bl = array([1.0, 1.0, 1.0, 1.0, -1.0e25, -1.0e25, 25.0])
bu = array([5.0, 5.0, 5.0, 5.0, 20.0, 40.0, 1.0e25])
x = array([1.0, 5.0, 5.0, 1.0])

# Initialize e04uca. Note that e04wbf only accepts upper-case
# rnames.
cwsav, lwsav, iwsav, rwsav, ifail = e04wbf(rname='E04UCA',
                                           lcwsav=1,
                                           llwsav=120,
                                           liwsav=610,
                                           lrwsav=475,
                                           ifail=infail)

if ifail != 0:
    sys.stderr.write('\n ** e04wbf returned with ifail = ' + \
                     str(ifail) + '\n')
    sys.exit(ifail)

# Define some arrays
istate = zeros(n + nclin + ncnln)
cjac = matrix(zeros((ncnln, n), 'd'))
clamda = zeros(n + nclin + ncnln, 'd')
r = matrix(zeros((n, n), 'd'))
```

```
# Set workspace length. The expression here is too complex for f2py to
# handle internally.
if ncnln == 0:

    if nclin == 0:
        lwork = 20*n
    else:
        lwork = 2*n**2 + 20*n + 11*nclin

else:
    lwork = 2*n**2 + n*nclin + 2*n*ncnln + 20*n + 11*nclin + 21*ncnln

work = zeros(lwork, 'd')

ifail = infail
quiet_con = False

iter, istate, c, cjac, clamda, objf, objgrd, \
      r, x, lwsav, iwsav, rwsav, ifail = e04uca(a, bl, bu,
                                                e04uca_confun,
                                                e04uca_objfun,
                                                istate, cjac,
                                                clamda, r, x,
                                                work,
                                                lwsav, iwsav, rwsav,
                                                ifail=infail,
                                                confun_extra_args=(quiet_con,))

# Check for error exits
if ifail >= 9:
    sys.stderr.write('  ** An input parameter is invalid\n')
elif ifail == 7:
    sys.stderr.write('  ** User-supplied derivatives are incorrect\n')
elif ifail == -399:

    if infail == quiet:
        sys.stderr.write('  ** e04uca returned with ifail = ' +
                         str(ifail) + '\n')

elif ifail < 0:
    sys.stderr.write('  ** mode < 0 on exit from objfun or confun.\n\n' +
                     'Problem abandoned\n')
else:
    sys.stdout.write(' e04uca returned with ifail = ' +
                     str(ifail) + '\n\n')

    common_header = '  Istate       Value       Lagr Mult\n\n'
    common_format = ' %3d  %3d    %14.6g  %12.4g\n'
    sys.stdout.write(' Varbl ' + common_header)

    for i in range(n):
        sys.stdout.write(' V' + common_format %
                         (i, istate[i], x[i], clamda[i]))

    if nclin > 0:

        # Put the matrix-vector multiplication a*x (the
```

```
        # linear-constraint values) in the first nclin locations of work.
        work[:nclin] = dot(a[:nclin,:n],x[:n])

        sys.stdout.write('\n\n L  Con' + common_header)

        for i in range(n, n + nclin):
            j = i - n
            sys.stdout.write(' L' + common_format %
                             (j, istate[i], work[j], clamda[i]))

    if ncnln > 0:
        sys.stdout.write('\n\n N  Con' + common_header)

        for i in range(n + nclin, n + nclin + ncnln):
            j = i - n - nclin
            sys.stdout.write(' N' + common_format %
                             (j, istate[i], c[j], clamda[i]))

    sys.stdout.write('\n\n Final objective value is ' + str(objf) + '\n')

sys.exit(ifail)
```

Notice the additional `quiet_con` argument to the constraint function.
The script should output

```
 (First call to confun)
 e04uca returned with ifail = 0

 Varbl   Istate       Value       Lagr Mult

 V   0    1                1         1.088
 V   1    0            4.743             0
 V   2    0          3.82115             0
 V   3    0          1.37941             0


 L  Con  Istate       Value       Lagr Mult

 L   0    0          10.9436             0


 N  Con  Istate       Value       Lagr Mult

 N   0    2               40       -0.1615
 N   1    1               25        0.5523


 Final objective value is 17.0140172891
```

6 Experiences on Windows

To test the above on 32-bit Windows I used `FLDLL214AL` on Windows XP. I had to give F2PY the flags `-DNO_APPEND_FORTRAN` and `-DUPPERCASE_FORTRAN` to enable it to pick up the correctly-decorated symbols from the import library. There are other preprocessor flags available too: see the F2PY documentation [10]. Also, I was not able to get F2PY to work with a static library: I had to use a DLL.
With the correct symbol names the Python module can be linked. However, `FLDLL214AL` uses the default calling convention used by the Compaq Visual Fortran (CVF) compiler. This requires all functions in and out of the DLL to be declared `__stdcall`, and for each character argument to have its length passed immediately following the argument itself. You can verify how the argument lists look from C for a particular Fortran Library DLL

by referring to the C header files for the distribution (see your Library's Installer's Note, `in.html`, for their location). The usual convention on UNIX is to append all character lengths after the 'visible' arguments.
There appears to be no way to tell F2PY to use the `__stdcall` calling convention[2] without hand-editing the `*module.c` wrapper files it generates. To do this you need to specify the `--build-dir <dir>` command-line option (otherwise the C files are deleted after compilation), and it's probably worthwhile to set `--debug-capi` for the initial run, to check everything's OK. If you tell F2PY just to generate the C API file, then edit it by hand, then compile, everything should work. For example, with the `C05ADF` signature file

```
DOS-prompt>f2py.py --no-wrap-functions --lower --debug-capi \
                   --build-dir f2py_tmp\src.win32-2.5 nag_py_c05adf.pyf
```

Note the omission of the link options and of `-c`. I found that I had to use `--no-wrap-functions` because my installation of NUMPY could not pick up `ifort` correctly: this option disables F2PY's use of a FORTRAN compiler (it'll just use the native C compiler instead). Then, edit the generated C file found at

```
f2py_tmp\src.win32-2.5\nag_py_c05adfmodule.c
```

so that the callback wrapper (in my case this was called `cb_c05adf_f_in___user__routines`) and the Library call (via `f2py_func`) are declared `__stdcall` in addition to their existing signatures. I had to change the lines

```
typedef double(*cb_c05adf_f_in___user__routines_typedef)(double *);
...
static double cb_c05adf_f_in___user__routines (double *x_cb_capi) {
...
void (*f2py_func)(double*,double*,double*,double*,
                  cb_c05adf_f_in___user__routines_typedef,
                  double*,int*)) {
```

to

```
typedef double(__stdcall *cb_c05adf_f_in___user__routines_typedef)(double *);
...
static double __stdcall cb_c05adf_f_in___user__routines (double *x_cb_capi) {
...
void (__stdcall *f2py_func)(double*,double*,double*,double*,
                            cb_c05adf_f_in___user__routines_typedef,
                            double*,int*)) {
```

The source materials [2] that accompany this article include the full C file for this `C05ADF` example and for the `E04UCA` example. Then compile:

```
DOS-prompt>f2py.py -c --no-wrap-functions --lower --debug-capi \
                   --build-dir f2py_tmp nag_py_c05adf.pyf \
                   -DUPPERCASE_FORTRAN -DNO_APPEND_FORTRAN \
                   -L%NAG_FL_DIR% -lFLDLL214AL_mkl
```

Be careful when F2PY is compiling that the C source doesn't go to a directory **different** from what you specified for `--build-dir`. I noticed that I needed to give `--build-dir` in the first command as `f2py_tmp\src.win32-2.5` because the `-c` option in the second command output to that directory when accompanied by the option `--build-dir f2py_tmp`.
Following all this, the Python invokation as in Section 4 was successful.
To pass character arguments correctly, two steps may be required. Let's consider the `E04WBF` wrapper generated using the `nag_py_e_chapter` signature file. First, each 'hidden' length argument in the call to the `e04wbf f2py_func` in the wrapper must be moved from the end of the argument list to immediately after the argument for which it is a length. For example, from the Library header file `nagmk21.h` we see that

```
extern void __stdcall E04WBF(
  CONST char rname[], int length_rname,
  CONST char cwsav[], int length_cwsav,
  CONST int *lcwsav,
  int lwsav[],
  CONST int *llwsav,
  CONST int iwsav[],
  CONST int *liwsav,
  CONST double rwsav[],
```

2. P. Peterson has let me know that this capability will be available in future releases of F2PY.

```
  CONST int *lrwsav,
  int *ifail
);
```

is the expected signature for `E04WBF`. This means in the Python wrapper, following the addition of `__stdcall`, we first need to move the length of `rname` from

```
/*callfortranroutine*/
(__stdcall *f2py_func)(rname,
                       cwsav,&lcwsav,
                       lwsav,&llwsav,
                       iwsav,&liwsav,
                       rwsav,&lrwsav,
                       &ifail,slen(rname));
```

to

```
/*callfortranroutine*/
(__stdcall *f2py_func)(rname,slen(rname),
                       cwsav,&lcwsav,
                       lwsav,&llwsav,
                       iwsav,&liwsav,
                       rwsav,&lrwsav,
                       &ifail);
```

The second step, which may not always need to be carried out but does in this case, is to add a length argument for `cwsav`. See the guide to calling the FORTRAN Library from C (the file `techdoc.html` in the Library installation area) for a simple example of this. We need only to pass the common character length of each element of `cwsav` (which is `80`), **not** the total number of characters in the array (which is `80*lcwsav`). The former is stored in the variable `cwsav_Dims[1]`, so we must further update the line to

```
/*callfortranroutine*/
(__stdcall *f2py_func)(rname,slen(rname),
                       cwsav,cwsav_Dims[1],&lcwsav,
                       lwsav,&llwsav,
                       iwsav,&liwsav,
                       rwsav,&lrwsav,
                       &ifail);
```

Note now that the call to `f2py_func` matches the interface of `E04WBF` given in the header file. Remember too to update the interface of `f2py_func` in the call to `f2py_rout_nag_py_e_chapter_e04wbf` to reflect the repositioning and adding of arguments. The `cwsav_Dims[1]` length argument should be declared there to be of type `size_t`. On 64-bit WINDOWS (Server 2003 Enterprise Edition with FORTRAN Library `FLW6IDCL`) the F2PY process is as smooth as on LINUX, although I did need to set the environment variable `MSSdk=1` to tell `distutils` to use the same compiler with which Python had been compiled.

7 Further Reading and Additional Materials

- The syntax of F2PY's signature files, and how to invoke the tool [10].
- Calling a CVF-compatible NAG FORTRAN Library from C [3] (this document is found under the Library's installation directory).
- Source code for examples in this article [2].

8 Acknowledgements

Thanks to Mike Dewar and Pearu Peterson for their helpful comments.

References

[1] H. P. Langtangen, *Python Scripting for Computational Science*, Springer–Verlag (2004)

[2] NAG, *Accompanying Materials for TR1/08*,
http://www.nag.co.uk/doc/TechRep/pdf/tr0108/tr0108.zip
[3] NAG, *NAG C Header Files (Windows Implementation)*
[4] NAG, *NAG Fortran Library Routine Document, C05ADF*,
http://www.nag.co.uk/numeric/fl/manual/pdf/C05/c05adf.pdf
[5] NAG, *NAG Fortran Library Chapter Introduction, E04: Minimizing or Maximizing a Function*,
http://www.nag.co.uk/numeric/FL/manual/pdf/E04/e04_intro.pdf
[6] NAG, *NAG Fortran Library Routine Document, E04UCA*,
http://www.nag.co.uk/numeric/fl/manual/pdf/E04/e04ucf.pdf
[7] NAG, *NAG Fortran Library Routine Document, E04WBF*,
http://www.nag.co.uk/numeric/fl/manual/pdf/E04/e04wbf.pdf
[8] NAG, *NAG Fortran Library Routine Document, S21BAF*,
http://www.nag.co.uk/numeric/fl/manual/pdf/S/s21baf.pdf
[9] P. Peterson, *f2py - Main - The F2PY Project*,
http://www.f2py.org
[10] P. Peterson, *F2PY Users Guide and Reference Manual*,
http://cens.ioc.ee/projects/f2py2e/usersguide/index.html
[11] *SAGE: Open Source Mathematics Software*,
http://www.sagemath.org