Developing Statistical Software in Fortran 95 and R

Joe Schafer and David Lemmon
The Methodology Center
The Pennsylvania State University

Presented at the fall seminar series Introduction to Computational Science Tools, November 10, 2008.
Outline

1. Motivation
2. Statistical Programming in Fortran
3. Interfacing Fortran with R
4. Creating R Packages


1. Motivation

- Many areas of statistics (especially applied areas) require non-trivial amounts of computing
- Publishing new methods without code is a recipe for non-use
- Methodological grants (e.g., from NIH) are easier to land if you promise to develop software
- Few statisticians are good programmers
- “Statistical computing” is basically numerical methods
WHAT OUR BOOK COVERS

• Review of Fortran 95

• Emphasis on programming style

• Suggested architecture for statistical programming
  (pseudo-object oriented) in Fortran

• Strategies for interfacing computational routines written in
  Fortran with other apps, especially in Windows (Excel, VB, R,
  Splus, SAS, SPSS, . . . )

WHAT THE TECH REPORT COVERS

• How to bundle Fortran routines and R code as an R package
2. Statistical Programming in Fortran

Principles that we espouse

• stay within the 95 standard; avoid compiler-specific extensions
• comment everything
• use modules
• strive for computational efficiency at lower levels
• use fancy data structures (derived types) liberally, but keep their contents private
• use fancy interfaces (e.g., function overloading) for private routines to make development easy
• make interface for public routines very conventional (no assumed-shape arrays or optional arguments) so that they can easily be called from other languages

• do things that may not always be required (e.g., explicit initialization and deallocation of locally allocated arrays) to avoid problems later

• use well thought-out strategy for error handling from the start of any project. For example, our error handler module always returns a character string with full traceback to indicate exactly where the error occurred

• for computational routines: test, test and test again
EXAMPLE: ERROR HANDLER MODULE

!********************************************************************************
module error_handler
  ! Generic error message handler for both console and non-console
  ! applications. The routines in this module do not halt program
  ! execution; they merely store error messages for subsequent
  ! retrieval.
  ! Written by J.L. Schafer, 4/19/02
implicit none
private ! by default
! declare public types
public :: error_type
! declare public subroutines and functions
public :: err_reset, err_handle, err_msg_present, err_get_msgs
! Parameters private to this module
integer, parameter, public :: &
  ! max width of any single error message line
  err_msg_width=70
!********************************************************************************
type :: error_type
  ! Public type for holding a linked list of messages
sequence
  private ! contents of this type are private to this module
  logical :: msg_present=.false.
  type(err_msg_line), pointer :: head=>null(), tail=>null()
end type error_type
!********************************************************************************
subroutine err_handle(err, err_code, called_from, file_name, &
   line_no, object_name, custom_1, custom_2, custom_3)
!
subroutine err_get_msgs(err, msg_string, platform)

Example: Fitting a mixture (chapter 4)

Suppose we have a sample $y_1, \ldots, y_n$ from a population with density

$$f(y_i) = \pi \lambda_1 \exp(-\lambda_1 y_i) + (1 - \pi) \lambda_2 \exp(-\lambda_2 y_i)$$

for $y_i > 0$

- mixture of two exponential distributions
- parameters to be estimated are $\pi \in [0, 1]$, $\lambda_1 > 0$, and $\lambda_2 > 0$
- We will compute maximum-likelihood (ML) estimates using an EM algorithm
Iterations of EM

Given the current estimates of $\pi$, $\lambda_1$ and $\lambda_2$, compute the posterior probability that $y_i$ came from the first component,

$$\hat{\delta}_i = \frac{\hat{\pi}\hat{\lambda}_1 \exp(-\hat{\lambda}_1 y_i)}{\hat{\pi}\hat{\lambda}_1 \exp(-\hat{\lambda}_1 y_i) + (1 - \hat{\pi})\hat{\lambda}_2 \exp(-\hat{\lambda}_2 y_i)},$$

$i = 1, \ldots, n$.

Then update the estimates by

$$\hat{\pi} = \frac{\sum_i \hat{\delta}_i}{n}, \quad \hat{\lambda}_1 = \frac{\sum_i \hat{\delta}_i y_i}{\sum_i \hat{\delta}_i y_i}, \quad \hat{\lambda}_2 = \frac{\sum_i (1 - \hat{\delta}_i) y_i}{\sum_i (1 - \hat{\delta}_i) y_i}.$$

EM converges reliably to a (local) mode, but it does not automatically produce standard errors.
Differenting the loglikelihood \( l = \sum_i \log f(y_i) \) once gives the score functions

\[
\frac{\partial l}{\partial \pi} = \frac{\sum_i (\delta_i - \pi)}{\pi (1 - \pi)},
\]

\[
\frac{\partial l}{\partial \lambda_1} = - \sum_i \delta_i (y_i - \lambda_1^{-1}),
\]

\[
\frac{\partial l}{\partial \lambda_2} = - \sum_i (1 - \delta_i) (y_i - \lambda_2^{-1}),
\]

and twice gives the Hessian, with diagonal elements

\[
\frac{\partial^2 l}{\partial \pi^2} = - \frac{\sum_i (\delta_i - \pi)^2}{\pi^2 (1 - \pi)^2},
\]

\[
\frac{\partial^2 l}{\partial \lambda_1^2} = \sum_i \left\{ -\delta_i \lambda_1^{-2} + \delta_i (1 - \delta_i) (y_i - \lambda_1^{-1})^2 \right\},
\]

\[
\frac{\partial^2 l}{\partial \lambda_2^2} = \sum_i \left\{ -(1 - \delta_i) \lambda_2^{-2} + \delta_i (1 - \delta_i) (y_i - \lambda_2^{-1})^2 \right\},
\]
and off-diagonal elements

\[
\frac{\partial^2 l}{\partial \pi \partial \lambda_1} = -\sum_i \delta_i (1 - \delta_i) \frac{(y_i - \lambda_1^{-1})}{\pi (1 - \pi)},
\]

\[
\frac{\partial^2 l}{\partial \pi \partial \lambda_2} = \sum_i \delta_i (1 - \delta_i) \frac{(y_i - \lambda_2^{-1})}{\pi (1 - \pi)},
\]

and

\[
\frac{\partial^2 l}{\partial \lambda_1 \partial \lambda_2} = -\sum_i \delta_i (1 - \delta_i) (y_i - \lambda_1^{-1})(y_i - \lambda_2^{-1}).
\]

At the ML solution, the score functions should be zero, and the inverse of (minus one times the Hessian) gives an estimated covariance matrix for the parameters.
We put a module like this in every program to define constants.
SKELETON FOR THE COMPUTATIONAL MODULE

module em_exponential_engine
  use error_handler
  use program_constants
  implicit none
  private ! by default
  public :: run_em_exponential
  ! parameters private to this module
  character(len=*), parameter :: modname = "em_exponential_engine"
contains
  ! all public and private procedures will be placed here
end module em_exponential_engine
Local procedure for computing score and Hessian

```fortran
!*****************************************************************************
integer(kind=our_int) function eval_score_and_hessian( y, pi, &
lambda_1, lambda_2, score, hessian, err) result(answer)
implicit none
! declare arguments
real(kind=our_dble), intent(in) :: y(:)
real(kind=our_dble), intent(in) :: pi, lambda_1, lambda_2
real(kind=our_dble), intent(out) :: score(3), hessian(3,3)
type(error_type), intent(inout) :: err
! locals
integer(kind=our_int) :: i, n
real(kind=our_dble) :: d, e1, e2, f, sumd, sumy, sumdy
character(len=*) , parameter :: subname = 
"eval_score_and_hessian"
! begin
answer = RETURN_FAIL
n = size(y)
score(:) = 0.
hessian(:,:) = 0.
sumd = 0.
sumdy = 0.
sumy = sum(y)
if( ( pi == 0. ) .or. ( pi == 1. ) ) goto 700
do i = 1, n
f = pi * lambda_1 * exp( -lambda_1 * y(i) ) &
+ (1 - pi) * lambda_2 * exp( -lambda_2 * y(i) )
d = pi * lambda_1 * exp( -lambda_1 * y(i) ) / f
sumd = sumd + d
sumdy = sumdy + d * y(i)
```

\[ e_1 = y(i) - 1. / \lambda_1 \]
\[ e_2 = y(i) - 1. / \lambda_2 \]

\[
\begin{align*}
\text{hessian}(1,1) &= \text{hessian}(1,1) + (d - \pi)^2 \\
\text{hessian}(2,2) &= \text{hessian}(2,2) + d \times (1 - d) \times e_1^2 \\
\text{hessian}(3,3) &= \text{hessian}(3,3) + d \times (1 - d) \times e_2^2 \\
\text{hessian}(1,2) &= \text{hessian}(1,2) + d \times (1 - d) \times e_1 \\
\text{hessian}(1,3) &= \text{hessian}(1,3) + d \times (1 - d) \times e_2 \\
\text{hessian}(2,3) &= \text{hessian}(2,3) + d \times (1 - d) \times e_1 \times e_2
\end{align*}
\]

\[
\begin{align*}
\text{score}(1) &= (\text{sumd} - \text{real}(n) \times \pi) / (\pi \times (1 - \pi)) \\
\text{score}(2) &= -\text{sumdy} + \text{sumd} / \lambda_1 \\
\text{score}(3) &= -\text{sumy} + \text{real}(n) / \lambda_2 + \text{sumdy} \\
&\quad - \text{sumd} / \lambda_2 \\
\text{hessian}(1,1) &= -\text{hessian}(1,1) / (\pi^2 \times (1 - \pi)^2) \\
\text{hessian}(2,2) &= \text{hessian}(2,2) - \text{sumd} / \lambda_1^2 \\
\text{hessian}(3,3) &= \text{hessian}(3,3) - (\text{real}(n) - \text{sumd}) / \lambda_2^2 \\
\text{hessian}(1,2) &= -\text{hessian}(1,2) / (\pi \times (1 - \pi)) \\
\text{hessian}(1,3) &= \text{hessian}(1,3) / (\pi \times (1 - \pi)) \\
\text{hessian}(2,3) &= -\text{hessian}(2,3) \\
\text{hessian}(2,1) &= \text{hessian}(1,2) \\
\text{hessian}(3,1) &= \text{hessian}(1,3) \\
\text{hessian}(3,2) &= \text{hessian}(2,3)
\end{align*}
\]

! normal exit

\[
\text{answer} = \text{return SUCCESS}
\]

return

! error traps

700 call err_handle(err, 102, 
&
called_from = subname//" in MOD "/modname)

return

end function eval_score_and_hessian
PUBLIC FUNCTION FOR FITTING THE MIXTURE

integer(kind=our_int) function run_em_exponential( y, &
   pi, lambda_1, lambda_2, iter, converged, loglik, &
   score, hessian, err, maxits, eps) result(answer)

! EM algorithm for computing ML estimates for the mixture of 
! two exponentials,
!   pi * exponential with mean 1/lambda_1
! + (1 - pi) * exponential with mean 1/lambda_2
implicit none
! Input data containing the sample:
real(kind=our_dble), intent(in) :: y(:)
! Starting values for parameters; these will also return 
! the estimates:
real(kind=our_dble), intent(inout) :: pi, lambda_1, lambda_2
! Number of EM iterations performed:
integer(kind=our_int), intent(out) :: iter
! T if EM converged, F otherwise:
logical, intent(out) :: converged
! Loglikelihood function and its first two derivatives at 
! the parameter estimates:
real(kind=our_dble), intent(out) :: loglik, score(3), &
   hessian(3,3)
! Error messages:
type(error_type), intent(inout) :: err
! Optional: maximum number of iterations and 
! criterion for judging convergence.
integer(kind=our_int), intent(in), optional :: maxits
real(kind=our_dble), intent(in), optional :: eps
! locals
integer(kind=our_int) :: max_iter, i, n
real(kind=our_dble) :: epsilon, oldpi, oldlambda_1, &
   oldlambda_2, d, f, sumd, sumy, sumdy
character(len=12) :: sInt
character(len=*), parameter :: subname = "run_em_exponential"
! begin
answer = RETURN_FAIL
score(:) = 0.
hessian(:,:) = 0.
! check input arguments and set defaults
if( (pi < 0.) .or. (pi > 1.) ) goto 300
if(( lambda_1 <= 0. ) .or. ( lambda_2 <= 0. ) ) goto 400
if( present(maxits) ) then
  if(maxits < 0) goto 500
  max_iter = maxits
else
  max_iter = 1000
end if
if( present(eps) ) then
  if( eps < 0. ) goto 600
  epsilon = eps
else
  epsilon = .00001
end if
! run EM
! n = size(y)
if( n == 0 ) goto 700
sumy = sum(y)
converged = .false.
iter = 0
do
  iter = iter + 1
  write( sInt, "(I12)" ) iter
  sInt = adjustl( sInt )
oldpi = pi
oldlambda_1 = lambda_1
oldlambda_2 = lambda_2

loglik = 0.
sumd = 0.
sumdy = 0.
do i = 1, n
    f = pi * lambda_1 * exp( -lambda_1 * y(i) ) &
        + (1 - pi) * lambda_2 * exp( -lambda_2 * y(i) )
    loglik = loglik + log(f)
    d = pi * lambda_1 * exp( -lambda_1 * y(i) ) / f
    sumd = sumd + d
    sumdy = sumdy + d * y(i)
end do

pi = sumd / real(n)
if( sumdy == 0. ) goto 700
lambda_1 = sumd / sumdy
if( sumy == sumdy ) goto 700
lambda_2 = ( real(n) - sumd ) / ( sumy - sumdy )
converged = ( abs(pi - oldpi) < epsilon ) .and. &
            ( abs(lambda_1 - oldlambda_1 ) < epsilon ) .and. &
            ( abs(lambda_2 - oldlambda_2 ) < epsilon )
if( ( iter > max_iter ) .or. converged ) exit
end do

if( .not. converged ) &
    call err_handle(err, 1000, &
        called_from = subname" in MOD "modname, &
        custom_1 = "Algorithm failed to converge by " &
        // "iteration " // trim(sInt) )
if( eval_score_and_hessian( y, pi, lambda_1, lambda_2, &
    score, hessian, err ) == RETURN_FAIL ) goto 800
! normal exit
answer = RETURN_SUCCESS
return
! error traps
300 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname, &
custom_1 = "Argument pi out of range.")
return
400 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname, &
custom_1 = "Argument lambda_1 or lambda_2 out of range.")
return
500 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname, &
custom_1 = "Invalid value for argument maxits.")
return
600 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname, &
custom_1 = "Invalid value for argument eps.")
return
700 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname, &
custom_1 = "Attempted division by zero;", &
custom_2 = "EM algorithm aborted at iteration \\
// trim(sInt) 
return
800 call err_handle(err, 1000, &
called_from = subname//" in MOD "/modname)
return
end function run_em_exponential
!******************************************************************************

20
A SIMPLE CALLING PROGRAM

program em_test
  ! quick and dirty test program
  use error_handler
  use program_constants
  use em_exponential_engine
  implicit none
  ! declare variables
  type(error_type) :: err
  real(kind=our_dble) :: y(50), pi=.30, lambda_1=0.5, lambda_2=2.0,
    loglik, score(3), hessian(3,3)
  integer(kind=our_int) :: iter, i
  logical :: converged
  ! begin
  open( 10, file="good50.dat" )
  read( 10, * ) ( y(i), i = 1, 50 )
  close( 10 )
  if( run_em_exponential( y, pi, lambda_1, lambda_2, iter, &
    converged, loglik, score, hessian, err ) == RETURN_FAIL ) &
    goto 800
  print *, "Iterations:", iter
  print *, "Converged:", converged
  print *, "Pi = ", pi
  print *, "Lambda_1 = ", lambda_1
  print *, "Lambda_2 = ", lambda_2
  print *, "Loglik = ", loglik
  print *, "Score:"
print *, score(:)
print *, "Hessian:"
  do i = 1, 3
    print *, hessian(i,:)
  end do
800 continue
  ! report error message
  if( err_msg_present(err) ) then
    call err_get_msgs(err, msg_string, platform)
    print "(A)" , trim(msg_string)
    print "(A)" , "Aborted"
  end if
end program em_test

!***********************************************************************
Data file good50.dat:

5.6 0.7 2.4 2.2 4.5 0.6 2.3 3.1 1.6 2.2
0.1 4.9 9.0 7.4 1.8 9.7 0.9 1.0 0.7 3.4
1.8 0.5 0.1 0.7 0.1 6.6 1.6 8.6 0.3 0.1
4.2 0.8 3.1 0.2 1.0 2.0 2.3 0.8 6.6 1.2
0.3 2.7 0.5 0.7 1.8 1.5 2.8 18.3 1.2 0.6

This is what we see when we run the program from a command line.

D:\jls\software\em_exponential>em_test
Iterations: 278
Converged: T
Pi = 0.464573293051795
Lambda_1 = 0.237951775412863
Lambda_2 = 0.678085120778212
Loglik = -99.3683135052164
Score: 
-1.934833778779978E-003 -8.597519091892991E-004
-4.53079743427907E-004
Hessian:
-40.3950981345350 87.5771631568222 21.4024109210376
87.5771631568222 -328.830858103097 -21.9849166839296
3. Interfacing Fortran with R

Making a DLL

• The easiest and most reliable way to call Fortran/C from R is to create a DLL

• The R function `.Fortran` will pass data to and retrieve data from a subroutine

• Be sure to know what data types you are passing

• Be sure to pass integer dimensions for every array

• No fancy data types or optional arguments; only what would be allowed in FORTRAN 77
Wrapper functions

• We cannot directly call the procedure `run_em_exponential` directly from R, because it uses assumed-shape arrays, optional arguments and one fancy data type (`err`)

• We need to create a wrapper function that takes only standard arguments
subroutine em_exponential(n, y, maxits, eps, pi, lambda_1, lambda_2, &
   iter, converged, loglik, score, hessian, msg_len, msg)
!
use error_handler
use program_constants
use em_exponential_engine
implicit none
!
! declare arguments
integer, intent(in) :: n
real(kind=our_dble), intent(in) :: y(n)
integer, intent(in) :: maxits
real(kind=our_dble), intent(in) :: eps
real(kind=our_dble), intent(inout) :: pi, lambda_1, lambda_2
integer, intent(out) :: iter
logical, intent(out) :: converged
real(kind=our_dble), intent(out) :: loglik, score(3), hessian(3,3)
integer, intent(in) :: msg_len
character(len=msg_len) :: msg
!
! declare locals
integer(kind=our_int) :: ijunk
!
! begin
!
call err_reset(err)
!
i Junk = run_em_exponential( y, pi, lambda_1, lambda_2, &
   iter, converged, loglik, score, hessian, err, maxits, eps )
!
if( err_msg_present(err) ) call err_get_msgs(err, msg, "UNIX")
!
return
end subroutine em_exponential

• Notice the compiler directive (line 3) which exports the subroutine when creating the DLL. This will be compiler-specific.

• In R, all arguments are passed by reference

• Be careful when compiling; you may need to specify certain options, such as align:sequence and nothreads and assume:underscore and names:lowercase

• These things are not well documented and often change when compilers are versions of R are updated.

• This may involve some trial and error

• Funny thing about R: The only kind of character data that can be passed to Fortran is a character string of length 255
THE R CLIENT

Write a function in R that will make it easy for the R user to access the routine.

```r
em.exponential <- function( y, start, eps=.00001, maxits=10000 ){
  # client function for the em exponential dll server
  if( missing(start) ){
    # generate starting values by a random split
    w <- sample( 1:2, length(y), replace=T )
    start <- list(
      pi = mean( w==1 ),
      lambda.1 = 1./mean( y[w==1] ),
      lambda.2 = 1./mean( y[w==2] )
    )
  }

  msg.len <- 255
  msg <- ""
  for(i in 1:msg.len) msg<-paste(msg," ",sep="")
  tmp <- .Fortran("em_exponential",
    n = length(y),
    y = as.double(y),
    maxits = as.integer(maxits),
    eps = as.double(eps),
    pi = as.double(start$pi),
    lambda.1 = as.double(start$lambda.1),
    lambda.2 = as.double(start$lambda.2),
    iter = integer(1),
    converged = logical(1),
    loglik = numeric(1),
```
score = numeric(3),
hessian = matrix(0.,3,3),
msg.len = as.integer(msg.len),
msg = msg)
est <- list( pi = tmp$pi, lambda.1 = tmp$lambda.1,
lambda.2 = tmp$lambda.2 )
list( est = est, converged = tmp$converged )
msg <- tmp$msg
if( all(strsplit(msg,"")[[1]]==" ") ) # all spaces
  msg <- NULL
else{
  # trim off the white space from msg and print to screen
  i <- msg.len
  while( ( substring( msg, i, i) == " ") & ( i >= 0 ) ) i <- i-1
  msg <- paste( substring( msg, 1, i ), "\n", sep="" )
cat(msg)}
result <- list(
est = est,
iter = tmp$iter,
converged = tmp$converged,
logliklihood = tmp$loglik,
score = tmp$score,
hessian = tmp$hessian,
msg = msg)
return(result)}
Using the R Client

> dyn.load("em_exponential.dll")
> y <- c(5.6, 0.7, 2.4, 2.2, 4.5, 0.6, 2.3, 3.1, 1.6, 2.2,
+ 0.1, 4.9, 9.0, 7.4, 1.8, 9.7, 0.9, 1.0, 0.7, 3.4,
+ 1.8, 0.5, 0.1, 0.7, 0.1, 6.6, 1.6, 8.6, 0.3, 0.1,
+ 4.2, 0.8, 3.1, 0.2, 1.0, 2.0, 2.3, 0.8, 6.6, 1.2,
+ 0.3, 2.7, 0.5, 0.7, 1.8, 1.5, 2.8, 18.3, 1.2, 0.6)
>
> tmp <- em.exponential(y)
> print( tmp$est )
$pi
[1] 0.5354197

$lambda.1
[1] 0.6780908

$lambda.2
[1] 0.2379533

> print( tmp$score )
[1] 0.0019629725 -0.0004596604 -0.0008722596
> print( solve( -tmp$hess ) )
[,1]       [,2]       [,3]
[1,]  0.19482037 -0.14959040 -0.04188542
[2,] -0.14959040  0.16424997  0.02885927
[3,] -0.04188542  0.02885927  0.01226698
4. Creating an R package

• A *package* is a set of R functions that is not part of the R base distribution but has been arranged and documented in a standard fashion.

• Once a package is installed on an R user’s machine, it becomes a *library* which can be attached and used in an R session.

• CRAN has a vast archive of user-contributed packages. They are all open-source and can be built on any platform.

• Usual package-building routine requires that all Fortran code be *FORTRAN 77*-compliant.
What about Fortran 95?

• I have figured out how to create an R package calling native routines in Fortran 95.

• My procedure is not sanctioned by the R Core Development Team.

• My package is not open source and runs only in Windows.

• The package is for multiple imputation of missing data. It is available at

  http://www.stat.psu.edu/~jls/norm3/

  and the procedures that I used to create it are explained in the written report cited on p. 2.