GENERAL DESCRIPTION

The following document contains the definition of various functions and how they can be called from an external program after they have been compiled. Calls from different platforms/environments/compilers are suggested: the list is of course not exhaustive. We recommend users to run the command "nm" or something equivalent to it on the compiled library to determine what is the exact name mangling used by the compiler that made the library. The mangling reported here was observed with gcc-4.1 to gcc-4.3.
We are referring explicitly to dynamic linked libraries, shared objects and run-time libraries (referred to .dll on windows, .so on Apple OSX and Linux -- examples of names are: libroc.so, roc.dll and so on); other types of libraries can in principle be constructed from static libraries (.a) to other slightly less known entities. We will not be discussing in detail the nature of such entities.

The list is not exhaustive, which means that many more functions are actually available from our libraries than the ones described here. The list is updated and with time more descriptions will be included.

When discussing a decision-variable to be used to plot an ROC curve, it is important to define which is the "direction" that we expect to "correlate" with a signal being present, also known as positivity; here it is always assumed to be for larger values, unless stated otherwise.

DESCRIPTION OF VARIABLES AND OTHER NUMERICAL ISSUES

Floating point variables, called reals or doubles, are coded in our routines as floating point IEEE T_floats.
integer, parameter :: double = selected_real_kind(p=15) ! IEEE T_float +- < 10^308, 15 digits
While we have in principle S_floats, we never found a reason to use them.

Integers are defined simply using the declaration integer as they are less critical in terms of interfacing (integers create only problems,
and rarely, during summation of many terms and mostly to people who don't know how to program).
We have never found any issues associated with their use in this form.

Complex numbers are not used in these libraries.

Characters are used only internally, because we could not find a way to utilize them consistently across multiple operating systems and
computational enviroments (e.g., R, SAS, IDL, Matlab).

Arrays should be handled with care, as not all environment store or represent them in the same way; our examples should be sufficient to
understand how to use them.

Pointers are for the most part avoided explicitly and used implicitly (with the exception of the Java libraries, not described here).
The reason again is portability; we found it to be too difficult to have a working version for all systems.

Generally the procedures produce essentially the same output (within prespecified numerical accuracy) on all operating systems.

GENERAL SYNTAX

function_name -> it is the name of the function as appears after the mangling produced by the compiler. It can be found, for example, but
doing (OS X command line or Linux/Unix or Windows from a Command shell where the appropriate functions have been installed):
nm library_name | grep <name before mangling>

E.g., for libraries built using gcc-4.1.1 on Linux.

```
% nm libroc.so | grep auc_pbm
00000000000242bf T __proproc_functions__auc_pbm
```

At the end of the description of each function there is a section with a pseudocode description of the call. In that description it is explicitly stated which are the calling parameters that are input and which are output.

PBM -> proper binormal model
CvM -> conventional binormal model
nonparametric -> non-parametric model (e.g., some statistic usually called with one of the following names proportion, U-statistic, Wilcoxon, Mann-Whitney)

AUC -> area under the roc curve
pAUC-> partial area under the ROC curve
TPF -> true positive fraction
FPF -> false positive fraction

When a variable is defined as "intent(IN)" it needs to be defined on call, if it isn't it might result in an error message or an exception or at the very least produce the wrong answer. We are using for this Fortran 90 syntax, which is for the most part the language used to code the calculations described here. Intent(out) on the other hand means that the variable is an output.

error general coding:
0 -> operation concluded successfully (might still be wrong, but the calculation passed our tests and appears correct)
-1 -> input is unacceptable, e.g., parameters out of bounds, negative variances, negative number of cases and so on
+1 -> fit failed
We use numbers because they appear to be the only approach that is stable across platforms and operating systems.

We decided to code all these procedures as Fortran subroutines because "R" and many other environments cannot handle how a fortran function returns the result of its execution.

Most of the calling syntax is defined from R, we assume that the user will be able to determine the correct calling scheme for their specific language and operating system.
Only dynamically linked library type libraries will be discussed here.

R requires variables to be initialized in some way before utilizing them. this is why in some descriptions we associate directly a number
with these calls. To the best of our knowledge these procedures work more or less as well with SAS and other statistical manipulation
software (or IDL or Matlab). Instructions about how to use them from those environments are available elsewhere.

CURRENTLY THERE ARE FOUR GROUPS OF METHODS/PROCEDURES:
-VERIFICATION/IMPLEMENTATION,
-INDICES CALCULATION,
-PLOTTING, AND
-ESTIMATION

<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
HOW TO USE THE LIBRARIES:
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

EXAMPLES FOR HOW TO LOAD THE LIBRARIES

R:

din.load(<file_location>/libroc.so) #Should that be dyn.load?
e.g., (my macintosh): dyn.load("~/ROC/FORTRAN_LIBS/SPECIAL_LIBS/libroc.so")

C++, to be used by program TestGetAzValue:

e.g., g++ -o TestGetAzValue TestGetAzValue.cpp libroc.a

Using the matlab “wrapper” functions:

As in R, a dynamically loaded library must first be loaded; the relevant matlab function is “loadlibrary” (equivalent to R's dyn.load above). However, matlab also places the burden on the user of converting their (matlab) data structures into pointers, which are then sent to the library function being called. The entities returned by the library function are again pointers which must be converted back into matlab data structures.

This administrative bookkeeping may be needed by matlab, but it is distracting to the user and diverts time and energy that could be directed to the actual ROC problem at hand. The wrapper functions we have written for the matlab environment accept and return ordinary matlab data structures; the loading of the ROC library, and conversion of the data structures to and from library pointers, is then done internally within the wrapper functions.

The price of this convenience is that we currently have no means of generating these wrapper functions automatically from the library source code. (In principle adding a wrapper function for another library function entails adding the function's “signature” to the libroc.h file required by matlab to load the library, and then writing the wrapper function itself to accommodate this signature. Users who need a library function added to our list of functions supported in matlab are urged to contact us so that we may do so.) Furthermore, the locations of the libroc.h and libroc.so files, as well as the library version number (checked for compatibility of the wrapper functions with the library interface version), are all currently hard-coded and need to be modified to suit the user's environment. We hope to address these issues in the future with the assistance of user feedback.

<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
DESCRIPTION OF THE FUNCTIONS
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

##
##
##
##
VERIFICATION IMPLEMENTATION
##
##
##
##

#####################Check the library version in R
##
###
This command allows the users to determine which version number the are using, as defined by major changes (used when some big implementation
change has been included in this version, in our case we reserve it for new approaches e.g., when going from single modality to multi
modality), minor changes (when adding some new functionalities or modifying a a function such that it won't be back-compatible anymore), and
version (bug removal not expected to have any other effect apart from removing the unwanted behavior).
It is important to check this all the time because version change and bugs are removed regularly.

First one needs to initialize the values (in R)

```
maj <- 0
min <- 0
vers <- 0

.C("__libroc_version_MOD_get_version_number", as.integer(maj), as.integer(min),
as.integer(vers))
```

In matlab:

```
[major, minor, version] = libroc_version;
```

##
##
##
##
 INDICES CALCULATION ROUTINES

#Do you not want descriptions of the cvbmroc and pbmroc routines? I don't have
#a lot of the other component routines implemented yet.

In matlab:

```
[Az, a, b, Az_Variance] = cvbmroc(neg, pos);
[AUC, da, c, AUC_Variance] = pbmroc(neg, pos);
```

############### AUC ###
Computes the area under the curve for the proper/conventional binormal model.Here we describe only functions that are called based upon parameter estimation as opposed to directly from the data, as it is done

by U-statistic, the Wilcoxon and other non-parametric methods, which are also available, but shoud be under fitting.

TESTING: Both tested for various input values on May 18th 2009, the testing is not particularly problematic because the functions that
define it are reasonably stable and not too complex. -- LP, U of C.

function_name with expected mangling (check using nm on the library name)

LINUX/WINDOWS: __proproc_functions__auc_pbm
OSX: __proproc_functions_MOD_auc_pbm
LINUX/WINDOWS: __labroc_functions__auc_cvbm
OSX: __labroc_functions_MOD_auc_cvbm

R syntax call
Define the following variables -- PROPER BINORMAL MODEL variables, users should refer to the sections from the proper binormal model is described

da <- real value between 0 and infinity, depending upon the curve, the values is usually obtained by fitting the routine, as described below
c <- real value between -1 and 1, depending upon the curve, the values is usually obtained by fitting the routine, as described below
auc <- real 0.0 -- R requires an initialization value
error <- integer 0 -- default 0, if it failed could have different values, but it never happens

Define the following variables -- Conventional BINORMAL MODEL
a <- real value between 0 and infinity, depending what the fit is
b <- real value between 0 and infinity, depending upon what the fit is
auc <- real 0.0 R requires an initialization value
error <- integer 0 default 0 if it failed -- never happens

E.g., of calling the function, R syntax

.C("__proproc_functions_MOD_auc_pbm", as.double(da), as.double(c) , as.double(AUC), as.integer(error))
.C("__labroc_functions_MOD_auc_cvbm", as.double(a), as.double(b) , as.double(AUC), as.integer(error))

Pseudocode call from other languages/environments

function_name(par1, par2, auc, ierror)

double, intent(in):: par1 ! da or a
double, intent(in):: par2 ! c or b
double, intent(out):: auc
integer, intent(out):: ierror ! ierror:: 0 -> OK, 1 == failed, 2 == c too small, used only phi part, used only for proproc

############ variance of AUC ##################################
Compute the variance of the area under the curve for the proper/conventional binormal model

TESTING: Both tested for various input values on May 18th 2009 -- LP
var AUC for proproc was extensively tested in Pesce LL, Metz CE. Reliable

and computationally efficient maximum-likelihood estimation of "proper" binormal ROC curves. Acad Radiol. 2007;14(7):814-29.

function_name with expected mangling

LINUX/WINDOWS: __proproc_functions__var_auc_pbm
OSX: __proproc_functions_MOD_var_auc_pbm
LINUX/WINDOWS: __labroc_functions__var_auc_cvbm
OSX: __labroc_functions_MOD_var_auc_cvbm

R syntax

Define the following variables PROPER BINORMAL MODEL

da <- real value between 0 and infinity, depending what the fit is
c <- real value between -1 a,d 1, depending upon what the fit is
varda <- real value from MLE or other estimation procedure
varc <- real value from MLE or other estimation procedure
covdac <- real value from MLE or other estimation procedure
varauc <- real 0.0 R needs initialization
error <- integer 0 default to successful estimation

Define the following variables CONVENTIONAL BINORMAL MODEL

a <- real value between 0 and infinity, depending what the fit is
b <- real value between 0 and infinity, depending upon what the fit is
vara <- real value from MLE or other estimation procedure
varb <- real value from MLE or other estimation procedure
covab <- real value from MLE or other estimation procedure
varauc <- real, R initialization, e.g. 0.0
error <- integer 0 for successful estimation

Call the functions from R
.C("__proproc_functions_MOD_var_auc_pbm", as.double(da), as.double(c) , as.double(varda), as.double(varc), as.double(covdac), as.double(varauc), as.integer(error))

.C("__labroc_functions_MOD_var_auc_cvbm", as.double(a), as.double(b) , as.double(vara), as.double(varb), as.double(covab), as.double(varauc), as.integer(error))

Pseudocode call from other languages/environments

```
function_name(par1,par2,var_par1,var_par2,cov_par1_par2, var_auc, ierror)
double, intent(IN):: par1, par2  ! parameters d_a or a; and b or c
double, intent(IN):: var_par1, var_par2, cov_par1_par2  ! variance par1, variance par2, covariance ...
double, intent(OUT):: varauc ! output, the variance
integer, intent(OUT):: ierror ! error flag, ierror (-1 August 2009) if wrong input,  0 if right, 2 if c is very small and approximations are used
                                (that to the best of our knowledge are OK -- only for proproc labroc doesn't return a 2).
```

########## Partial_auc ##
Computes the partial area under the curve for the proper and conventional binormal

models
for testing see file ROC/FORTRAN_LIBS/Verification of functional values.nb
(mathematica file)
We tested for values of c = -1, -.5, -.25, 0, +.25, +.5, +1, for values of da =
10000, 100, 12, 5, 3,1,0
We tested for values of b = 0, .5, 1, 2, +5, for values of a = 10000, 100,
12,5,3,1,0
And they correspond both to numerical and analytical values within 6 decimal
places. Both vertical and horizontal partial AUCs

function_name with expected mangling

LINUX/WINDOWS: __proproc_functions__partialauc_pbm
OSX: __proproc_functions_MOD_partialauc_pbm
LINUX/WINDOWS: __labroc_functions__partialauc_cvbm
OSX: __labroc_functions_MOD_partialauc_cvbm

Call from R
Define the following variables PROPER BINORMAL MODEL
da <- real value between 0 and infinity, depending what the fit is
c <- real value between -1 a,d 1, depending upon what the fit is
frac1 <- real lower bound for the partial AUC, between 0 and 1
frac2 <- real upper bound for the partial AUC, between frac1 and 1]
FPF_flag <- integer, 1 means a vertical partial AUC, 0 means an horizontal
partial_auc <- real 0.0 R requires initialization initialization, value of partial
AUC on exit]
error <- integer 0, see below for details

OR Define the following variables CONVENTIONAL BINORMAL MODEL
a <- real value between 0 and infinity, depending what the fit is
b <- real value between 0 and infinity, depending upon what the fit is
frac1 <- real lower bound for the partial AUC, between 0 and 1
frac2 <- real upper bound for the partial AUC, between frac1 and 1
FPF_flag <- integer, 1 means a vertical partial AUC, 0 means an horizontal
partial_auc <- real 0, R requires initialization, value of partial AUC on exit
error <- integer 0, see below for details

Actual call to procedures from R:
.C("__proproc_functions_MOD_partialauc_pbm", as.double(da), as.double(c) ,
as.double(frac1),as.double(frac2), as.integer(FPF_flag), as.double(partial_auc),
as.integer(error))

.C("__labroc_functions_MOD_partialauc_cvbm", as.double(a), as.double(b) ,
as.double(frac1),as.double(frac2), as.integer(FPF_flag), as.double(partial_auc),
as.integer(error))

Pseudocode call from other languages/environments

function_name(par1, par2, fraction_1, fraction_2, FPF_flag, partial_auc, ierror)

 double, intent(in):: par1 ! First Curve parameters
 double, intent(in):: par2 ! Second Curve parameters
 double, intent(in):: fraction_1, fraction_2 ! These are called fractions because
the can be FPF or TPF depending upon which area are we computing
 integer, intent(in):: FPF_flag ! If it is true, it means that the area computed
will

```
       ! be between FPF_1 and FPF_2, otherwise it means that it will be between
TPF_1 and
       ! TPF_2
  double, intent(out):: partial_auc
  integer,           intent(out):: ierror  ! = 0, computatio is OK
                                           ! = bad_input (-1 as of August 2009)
the values in input are wrong (see source code for details)
                                           ! = 1 computation failed (e.g., normal
deviates could not be computed -- it never happened of late)
                                           ! = 2 currently not used (had a
different purpose before)
                                           ! = 3 , fractions are almost identical
or identical

In matlab:
[partial_AUC, error_flag] = partial_auc_pbm(da, c, frac1, frac2, ...
    fpf_flag);
[partial_AUC, error_flag] = partial_auc_cvbm(a, b, frac1, frac2, ...
    fpf_flag);

########## variance of partial auc ########################################
Computes the value of the variance of the partial AUC both for the conventional
and the proper binormal model
as described in Pan, X., Metz, C.E., 1997. The proper binormal model: parametric
receiver operating characteristic curve estimation with
degenerate data. Acad. Radiol. 4, 380â “389. (note that while the equations are
described in that paper, they are also for the most part
full of typos, the only reliable source of those equations we know our program
itself)

For testing see file ROC/FORTRAN_LIBS/Verification of functional values.nb
(mathematica file)
We tested for values of c = -1, -.5, 0, +.5, +1, for values of da = 5, 3,1,0
We tested for values of b = 0, .5, 1, 2, +5,  for values of a = 10000, 100,
12,5,3,1,0
and they correspond both to numerical and analytical values within 6 decimal
places. Both vertical and horizontal partial AUCs
for values at  the boundaries (i.e., da = 0 c = +-1, a = 0 and b = 0) the variance
may not be returned because the
conditions implicit in the series expansion and normality of estimates are clearly
violated. When it is returned
care must be paid to whether the assumptions behind the delta method here applied
are appropriate (the delta
assume normal distributions for both variables -- which means we are assuming the
relationship is linear for the
range of values spanned by the variances, the assumption is violated if the
variances are large or if the estimates
are too close to the boundaries. We decided not to force a control for the
assumptions, which means that it is up to the
user to make sure it is likely to work.


function_name with expected mangling

function_name
-------------
LINUX/WINDOWS: __proproc_functions__var_partialauc_pbm
OSX:           __proproc_functions_MOD_var_partialauc_pbm
```

```
LINUX/WINDOWS: __labroc_functions__var_partialauc_cvbm
OSX:           __labroc_functions_MOD_var_partialauc_cvbm


Call from R

First define the following variables
da <- real value between 0 and infinity, depending what the fit is
c <- real value  between -1 a,d 1, depending upon what the fit is
frac1 <- real lower bound for the partial AUC, between 0 and 1
frac2 <- real upper bound for the partial AUC, between frac1 and 1
flag <- integer, 1 means a vertical partial AUC, 0 means an horizontal
varda <-  real value, usually computed by MLE or other estimation procedure
varc = real value, also from MLE or other estimation procedure
covdac = real value also from MLE or other estimation procedure
varPartialAUC <- real 0.0, requires initialization, value of partial AUC on exit
error <- integer 0 [see below for details]


OR Define the following variables CONVENTIONAL BINORMAL MODEL
a <- real value between 0 and infinity, depending what the fit is
b <- real value between 0 and infinity, depending upon what the fit is
frac1 <- real lower bound for the partial AUC, between 0 and 1
frac2 <- real upper bound for the partial AUC, between frac1 and 1
flag <- integer, 1 means a vertical partial AUC, 0 means an horizontal
vara <-  real value, from MLE or other estimation procedure
varb = real value, from MLE or other estimation procedure
covab = real value, from MLE or other estimation procedure
varauc <- real 0.0, R requires initialization
varPartialAUC <- real 0.0, start with some initialization, value of partial AUC on
exit
error <- integer 0 [see below for details]


Call to the functions from R
.C("__proproc_functions_MOD_var_partialauc_pbm", as.double(da), as.double(c)
,as.double(frac1),as.double(frac2), as.integer(flag), as.double(varda),
as.double(varc), as.double(covdac), as.double(varPartialAUC), as.integer(error) )

.C("__labroc_functions_MOD_var_partialauc_cvbm", as.double(a), as.double(b)
,as.double(frac1),as.double(frac2), as.integer(flag), as.double(vara),
as.double(varb), as.double(covab), as.double(varPartialAUC), as.integer(error) )



Pseudocode call

  function_name(par1, par2, fraction_1, fraction_2, FPF_flag, &
                                          var_par1,var_par2,cov_par1_par2,
var_p_auc, ierror)

 double, intent(in):: par1 ! First Curve parameters
 double, intent(in):: par2 ! Second Curve parameters
 double, intent(in):: fraction_1, fraction_2 ! These are called fractions because
the can be FPF or TPF depending upon which area are we computing
 integer, intent(in):: FPF_flag ! If it is true, it means that the area computed
will
       ! be between FPF_1 and FPF_2, otherwise it means that it will be between
```

```
TPF_1 and
       ! TPF_2
 double, intent(IN):: var_par1, var_par2, cov_par1_par2  ! variance par1, variance
par2, covariance ...
 double, intent(OUT):: var_p_auc ! output, the variance
  integer,           intent(OUT):: ierror  ! = 0, computatio is OK
                                           ! = bad_input (-1 as of August 2009)
the values in input are wrong (see source code for details)
                                           ! = 1 computation failed (e.g., normal
deviates could not be computed -- it never happened of late)
                                           ! = 2 currently not used (had a
different purpose before)
                                           ! = 3 , fractions are almost identical
or identical



############# TPF (FPF) and FPF (TPF) ##########################################
These are two functions the compute the value of the TPF when the FPF is known or
the value
of the FPF when the FPF is known. Of course the parameters are assumed to be
known. The
The known value is considered to be fixed as opposed to have to be estimated and
therefore
contain a measurement error.

function_name with expected mangling
-------------
LINUX/WINDOWS: __[proproc/labroc]_functions__fpf_find_tpf_[pbm/cvbm]
LINUX/WINDOWS: __[proproc/labroc]_functions__tpf_find_fpf_[pbm/cvbm]
OSX        : __[[proproc/labroc]_functions_MOD_fpf_find_tpf_[pbm/cvbm]
OSX        : __[proproc/labroc]_functions_MOD_tpf_find_fpf_[pbm/cvbm]

Call from R

First define the following variables
da <- real value between 0 and infinity, depending what the fit is
c <- real value  between -1 a,d 1, depending upon what the fit is
PF <- real, input value, FPF for the first function, TPF for the second, between 0
and 1
OPF <- real, output value, TPF for the first function, FPF for the second, between
0 and 1, further restrictions might apply to specific models
error <- integer 0 [default 0 if it failed -- never happens]

Call to the function from R
.C("__[proproc/labroc]_functions_MOD_fpf_find_tpf_[pbm/cvbm]", as.double(da),
as.double(c) , as.double(FPF), as.double(TPF), as.integer(error) )

Pseudocode example of call to the functions

 function_name(d_a_par, c_par, fpf , tpf, ierror)
double, intent(in):: d_a_par, c_par ! parameters of the current fit, used as input
double, intent(in):: fpf ! value of the fpf for the single point available for the
fit
double, intent(OUT):: tpf ! value of TPF for that fpf
double, intent(OUT)::  ierror ! error flag. If ierror = -1, the value of FPF is
out of bounds
```

In matlab:

```
[tpf, error_flag] = fpf_find_tpf_pbm(da, c, fpf);
[tpf, error_flag] = fpf_find_tpf_cvbm(a, b, fpf);
```

############ variance of TPF (FPF) and FPF (TPF)
##
These are two functions the compute the value of variance of the TPF when the FPF is known or the value
of the FPF when the FPF is known. Of course the parameters are assumed to be known. The
The known value is considered to be fixed as opposed to be estimated therefore it does not affect the
variance.

function_name with expected mangling

LINUX/WINDOWS: __[proproc/labroc]_functions__var_fpf_find_tpf_[pbm/cvbm]
LINUX/WINDOWS: __[proproc/labroc]_functions__var_tpf_find_fpf_[pbm/cvbm]
OSX : __[proproc/labroc]_functions_MOD_var_fpf_find_tpf_[pbm/cvbm]
OSX : __[proproc/labroc]_functions_MOD_var_tpf_find_fpf_[pbm/cvbm]

Call from R
First define the following variables
da <- real value between 0 and infinity, depending what the fit is
c <- real value between -1 a,d 1, depending upon what the fit is
varda <- real value from MLE or other estimation procedure
varc = real value from MLE or other estimation procedure
covdac = real value from MLE or other estimation procedure
PF <- real input value, FPF for the first function, TPF for the second, between 0 and 1
varopf <- real 0.0, R requires initialization, return the value of the variance of the estimated fraction
error <- integer 0, default 0, wrong input -1, if it failed 1 -- never happens

Call the function from R
.C("__[proproc/labroc]_functions_MOD_var_fpf_find_tpf_[pbm/cvbm]", as.double(da), as.double(c) , as.double(varda), as.double(varc), as.double(covdac),
as.double(PF), as.double(varopf), as.integer(error))

Pseudocode call for one of the two

```
function_name(d_a_par, c_par, var_d_a, var_c, cov_d_a_c, fpf , var_tpf, ierror)
double, intent(in):: d_a_par, c_par ! parameters of the current fit, used as input
double, intent(IN):: var_d_a, var_c, cov_d_a_c ! parameters of the current fit, used as input
double, intent(in):: fpf ! value of the fpf for the single point available for the fit
double, intent(OUT):: var_tpf ! value of TPF for that fpf
double, intent(OUT):: ierror ! error flag. If ierror = -1, the value of FPF is out of bounds
```

############ TPF (FPF) at cutoff value
##
these are functions that compute the value of the TPF or FPF when the cutoff value is known (either in the
actual or in the latent space. Of course the parameters are assumed to be known --

if the functions are parametric.
There is no return error because we decided that the possible errors are just too stupid to bother and
checking of their consistency was going to be expensive for simulations and resampling procedures.

function_name with expected mangling, for the semi-parametric models
---
LINUX/WINDOWS: __[proproc/labroc]_functions__fpf_[pbm/cvbm]
LINUX/WINDOWS: __[proproc/labroc]_functions__tpf_[pbm/cvbm]
OSX : __[proproc/labroc]_functions_MOD_fpf_[pbm/cvbm]
OSX : __[proproc/labroc]_functions_MOD_tpf_[pbm/cvbm]

Call from R
First define the following variables
da <- real value between 0 and infinity, depending what the fit is
c <- real value between -1 a,d 1, depending upon what the fit is
PF <- real input value, FPF for the first function, TPF for the second, between 0 and 1
OPF <- real input value, 1-FPF for the first function, 1-TPF for the second, between 0 and 1

Call the function from R
.C("__[proproc/labroc]_functions_MOD_fpf_[pbm/cvbm]", as.double(da), as.double(c), as.double(PF), as.double(OPF))

function_name with expected mangling -- non parametric
The non-parametric functions associate a value of 1/2 for values equal to the threshold.

LINUX/WINDOWS: __roc_nonparametric__empirical_fpf
LINUX/WINDOWS: __roc_nonparametric__empirical_tpf
OSX : __roc_nonparametric_MOD_empirical_fpf
OSX : __roc_nonparametric_MOD_empirical_tpf

Call from R
First define the following variables
N <- integer value, between 1 and infinity, it is either the number of actually-positive or the number of actually-negative cases]
AP/AN <- array(0, c(1, N)), real array of size N [with the values associated with each of the actually-positive or actually-negative cases]
fpf/tpf <- real value, the estimated sensitivity of 1 - specificity
ierror <- integer value # checks at least whether the value of N makes sense, returns -1 if it does not

Call the function from R
.C("__roc_nonparametric_MOD_empirical_tpf", as.integer(N), as.double(AP) , as.double(threshold), as.double(tpf), as.integer(ierror))

Pseudocode example for one of the two with details about calling scheme

```
function_name(N, AP, threshold, tpf, ierror)
integer, intent(in):: N ! number of actually-positive cases
double, dimension(N), intent(IN):: AP ! value associated with each of the
actually-positive cases
double, intent(in):: threshold ! value above which a case has to be considered
positive
double, intent(OUT):: TPF ! estimated value for the sensitivity
integer, intent(OUT):: ierror ! whether the input was acceptable, there are no
known computation errors at this point


############# Exact CIs for TPF (FPF) at cutoff value
##########################################
We are not reporting functions for the semi-parametric estimates at this point
because their use is complex as it requires an estimation of
the relationship between the value of the variables used in the experiment and the
latent variables for which the estimation is computed.
The non-parametric functions associate a value of 1/2 for values equal to the
threshold.
Computes the confidence intervals for a proportion using exact confidence
intervals
starting from the number of positives calls observed (k).
From Fleiss "statistical methods for rates and proportions", third edition, Wiley,
page 22.
the search of largest (smallest) value of p (the probability of observing a
success) that
has at at least a 5% chance of generating at least as few (at most as many)
successes as k
is done by first simply bounding the value and applying bisection. More refined
approaches can
be used, but it did not seem necessary at this point.We use the log of the
probability of each
observation as basis of the calculations to avoid near constant over- ad under-
flows.

Expected function_name with mangling
-------------
LINUX/WINDOWS: __roc_nonparametric__exact_CI_empirical_fpf
LINUX/WINDOWS: __roc_nonparametric__exact_CI_empirical_tpf
OSX        : __roc_nonparametric_MOD_exact_CI_empirical_fpf
OSX        : __roc_nonparametric_MOD_exact_CI_empirical_tpf


Call from R
First define the following variables
N <- integer value, between 1 and infinity, it is either the number of actually-
positive (or the number of actually-negative cases)
k <- integer value, the number of cases called positive
Cl <- real value between 0 and 1, confidence level, e.g., .95 for 95%
TCI <- integer value, -1 between 0 and UB, 0 between LB and UB, +1 between LB and
1
lb <- real value, lower bound of the CI
ub <- real value, upper bound of the CI
ierror <- integer value, checks at least whether the value of N makes sense,
returns -1 if it does not

Call the function from R
.C("__roc_nonparametric_MOD_exact_CI_empirical_fpf", as.integer(N), as.integer(k),
```

```
as.double(CL), as.integer(TCI), as.double(lb),as.double(ub),as.integer(ierror))


Pseudocode for one of the two with details about calling scheme
 function_name(mn, k , confidence_level, type_of_CI,  lb, ub, ierror)
 integer, intent(IN):: mn ! number of actually-negative cases
 integer, intent(IN):: k  ! number of actually-negative cases cases that were
called positive at a specific
                          ! threshold or decision scheme (e.g., a combination of
one or more thresholds and a random
                          ! number). We use the integer to avoid issues that could
be created by rounding errors
                          ! by forcing the calling program to take care of it.
 real(kind=double), intent(IN):: confidence_level ! e.g., 95% confidence interval
a number from 0 to 1.
 integer, intent(IN):: type_of_CI ! -1 -> lower bound is 0 find upper (find the
"inferiory" CI) ,
                                  !  0 -> find upper and lower (find the "non-
equality" CI), and
                                  ! +1 -> upper bound is 1, find lower (find the
superiority CI)

 real(kind=double), intent(OUT)::lb, ub ! estimated lower and upper bound of the
CI.
 integer, intent(OUT) :: ierror ! error value for the CI calculation :
                                ! 0 -> Procedure did not detect any computation
issues
                                ! -1 -> input values are not acceptable



##############################################################################
############################################################
##############################################################################
############################################################
 PLOTTING ROUTINES
##############################################################################
############################################################
##############################################################################
############################################################


#############  Plotting points for a [proproc/labroc] curve
################################################################
Returns a set of points on an ROC curve specified by the [proproc/labroc] model
for parameters da and c (also to be part of the input)
The plot of empirical operating points (trapezoidal non-parametric ROC  curve
corresponding to the Mann-Whitney form of the Wilcoxon
 statistics)  is described later.

 function_name with mangling
-------------
LINUX/WINDOWS: __[proproc/labroc]_out_MOD_points_on_curve_[pbm/cvbm]
OSX        : __[proproc/labroc]_out__points_on_curve_[pbm/cvbm]

Call from R
First define the following variables
da <- real value between 0 and infinity, depending what the fit is
```

```
c <- real value  between -1 a,d 1, depending upon what the fit is
NumPts <-  integer, the number of points desired on the curve
CurvePoints <- matrix(0, 2, NumPts) real array with FPF,TPF pairs
error <- integer 0 default 0 , se below for other error messages


Call of the function from R
.C("__[proproc/labroc]_out_MOD_points_on_curve_[pbm/cvbm]",
as.double(da),
as.double(c),
as.integer(NumPts),
as.double(CurvePoints),
as.integer(error))


Pseudocode call example

 call function_name(d_a_par_in, c_par_in, num_pts, CurvePoints,ierror)
 double, intent(IN):: d_a_par_in, c_par_in ! curve parameters
 integer, intent(IN) :: num_pts ! number of curve points whose value is desired
 real(kind=double), dimension(2, num_pts), intent(OUT) :: CurvePoints ! the actual
                 !  array with the fpf, tpf values on exit
 integer, intent(OUT):: ierror  ! 0 -> OK; 1 -> Failed; -1 -> wrong input


#############  Plotting empirical operating points
################################################################
Extracts the set of the empirical operating points (corners of the ROC plot or
truth state runs
See C. E. Metz, B. A. Herman, and J-H. Shen, â˜˜Maximum likelihood estimation
of receiver operating characteristic ~ROC! curves from
continuously-distributed data,â™â™ Stat. Med. 17, 1033â“1053 ~1998 for a
description of them

function_name with expected mangling
-------------
LINUX/WINDOWS: __roc_nonparametric_MOD_empirical_operating_points_list
OSX        : __roc_nonparametric_MOD_empirical_operating_points_list

Call from R
First define the following variables
mn <- integer, number of actually-negs
ms <- integer, number of actually-pos
AN <- double array value of actually negative cases
AP <- double array value of actually positive cases
PL <- 0 or 1 whether positivity is for large values
numpts <- integer, output, the number of empirical  points found
optpts <- matrix(0, 2, Mn+Ms), output, list of opeating points notice that only
the the first numpts will contain data.
error <- integer, default 0 , se below for other error messages

Call of the function from R:
.C("__roc_nonparametric_MOD_empirical_operating_points_list",
as.integer(mn),
as.integer(ms),
as.double(AN),
as.double(AP),
as.integer (PL),
```

```
as.integer(numpts),
as.double(optpts),
as.integer(error))


Pseudocode example
 function_name( mn, ms, neg_cases, pos_cases, positiveislarge, num_pts,
operatingpts,ierror)
 integer, intent(IN):: mn      !number of actually negative cases
 integer, intent(IN):: ms      !number of actually positive cases
 double,dimension(mn), intent(IN) :: neg_cases
 double,dimension(ms), intent(IN) :: pos_cases
 integer, intent(IN):: positiveislarge ! whether positivity is for more positive
or more negative values
                                       ! 1 if it is for larger values, 0 if it is
for smaller values
 integer, intent(out) :: num_pts ! number of empirical operating points found
 double, dimension(2, mn+ms), intent(OUT) :: operatingpts ! the actual
                  !  array with the fpf, tpf values of the empirical operating
points
 integer, intent(OUT):: ierror  ! 0 -> OK; 1 -> Failed; -1 -> wrong input

##############################################################################
#########################################################
##############################################################################
#########################################################
 ESTIMATION ROUTINES
##############################################################################
#########################################################
##############################################################################
#########################################################
NOTE: The categorizer has to be called before any multinomial sampling MLE (or
else) based is called. Other types of arrangements and wrappers
are available, but they will not be described here as they are largely redundant.

#############  categorizer
################################################################
Transforms two senquences of values (one sequence for the actually negative and
one sequence for the actually positive cases) into their truth runs, which
can then be used as categorical data into a MLE roc fitting model.
See C. E. Metz, B. A. Herman, and J-H. Shen, â˜ â˜Maximum likelihood estimation
of receiver operating characteristic ~ROC! curves from
continuously-distributed data,â™â™ Stat. Med. 17, 1033â“1053 ~1998 for a
description of them

 function_name with expected mangling
-------------
LINUX/WINDOWS: __categorization_MOD_catgrz
OSX        : __categorization__catgrz

Call from R
First define the following variables

PositiveLarge  integer,  1 if positivity is for larger values, 0 otherwise
mn   integer, number of actually-negative cases
ms  integer, number of actually-positive cases
DebugLogFile,  integer  0 if no  debugfile should be written
cat0 <- matrix(0, 2, MaxNumCategories), output,  integer array with categorical
```

```
data
AP <- rnorm (ms, 1, 1),  actually-positive cases, generated from a normal
distribution
AN <- rnorm (mn), actually-negative cases, generated from a standard normal
distribution
NumCategoriesFound integer, number of categories/truth runs found in the data
MaxNumCategories <- e.g., 20 # integer max number of categories to be
                  considered. PBM/CvBM have been extensively at most 400, usually
more than 50 is not necessary
CaseCat <- matrix(0, 1, mn+ms), integer  on output, for each case, it containts
the category where it was placed

Call of the function from R

.C("__categorization_MOD_catgrz",
as.integer(PositiveLarge),
as.integer(mn),
as.integer(ms),
as.integer(DebugLogFile),
as.integer(cat0),
as.double(AN),
as.double(AP),
as.integer(NumCategoriesFound),
as.integer(MaxNumCategories),
as.integer(CaseCat) )

Pseudocode call
 function_name(POSITIVEISLARGE, NUM_NORMAL_CASES,NUM_ABNORMAL_CASES,idebug, CAT0,
&
                     NEG_INPUT,
POS_INPUT,NUM_CATEGORIES,MAX_NUM_CATEGORIES,CASE_CAT)

 INTEGER, INTENT(IN):: POSITIVEISLARGE ! Likelihood of abnormal TEST RESULT VALUE
associated with larger values
 INTEGER, INTENT(IN):: NUM_NORMAL_CASES
 INTEGER, INTENT(IN):: NUM_ABNORMAL_CASES
 INTEGER, INTENT(IN):: idebug ! whether to write a log file or not
 double,INTENT(IN),DIMENSION(NUM_NORMAL_CASES)::NEG_INPUT ! negative TEST RESULT
VALUEs are stored
 double,INTENT(IN),DIMENSION(NUM_ABNORMAL_CASES)::POS_INPUT ! negative TEST RESULT
VALUEs are stored

 INTEGER, INTENT(IN) :: MAX_NUM_CATEGORIES ! MAXIMUM NUMBER OF CATEGORIES ALLOWED
BY THE DIMENSIONING IN THE
                                           ! MAIN PROGRAM, THE MODULE WILL SEEK TO
PRODUCE MAX_NUM_CATEGORIES.
                                           ! IF LESS THAT THAT ARE AVAILABLE (SAY
N_CAT) IT WILL RETURN N_CAT
                                           ! IF MORE ARE AVAILABLE, IT WILL RETURN
MAX_NUM_CATEGORIES

 INTEGER, INTENT(out), DIMENSION(2, MAX_NUM_CATEGORIES) :: CAT0 ! CONTAINS THE
CATEGORIES
                    ! CREATED BY THIS CATEGORIZATION ALGORITHM ON EXIT
 INTEGER, INTENT(out):: NUM_CATEGORIES ! THE NUMBER OF CATEGORIES FOUND
 INTEGER,
INTENT(out),DIMENSION(2,max(NUM_NORMAL_CASES,NUM_ABNORMAL_CASES))::CASE_CAT
                  ! This array stores for each case the category where it is
```

allocated. Mostly to be
                    ! used by MRMC schemes, and this is why it is optional

##
##
MAXIMUM-LIKELIHOOD ESTIMATION SEMI-PARAMETRIC MODEL ROUTINES, BASED ON MULTINOMIAL SAMPLING
##
##
NOTE 1: the return error flag from the routines follow nearly identical coding. Where there is a difference between algorithms, it will be indicated.
NOTE 2: The flags are integers.
NOTE 3: Newer versions of the library might contain additional flags that are not included here yet. Feel free to make us notice any inconsistencies.
NOTE 4: Not all possible errors are considered here and sometimes a flag might be misleading, be careful how you use them.
NOTE 5: Usually it will be possible to rerun the same fit forcing the routine to write a much more extensive error logging file.
 RETURN FLAGS
 -1 => the categorical data send to the subroutinea is bad ROC data (negative number of cases, numbers don't add up and so on)
  0 => fit was successful
  1 => The routines could not converge. This *NEVER* happened to date (11/15/2010) that I know of, so please contact us if you have this problem
  2 => Note enough data to produce a unique ROC fit, e.g., there is only one point. The condition of degeneracy might be reached for
        different models in different situations
  3 => positives and negatives are perfectly separated, it is more of a warning
  4 => initial estimates did not converge; it is similar to 1, but more specific
  5 => estimates of variances did not converge or should not be trusted (variances cannot be trusted also in other situations, this is not
       exhaustive and sometimes the variances cannot be trusted, but the routine might not report it...
 6 => fit was successful, variances are pseudovariances, see Pesce LL, Metz CE. Reliable and computationally efficient maximum-likelihood estimation
        of proper binormal ROC curves. Acad Radiol 2007;14:814â “829
  7 => estimates of var are bad because the fit is too close to the boundary of the parameter space. Usually these maxima are either cusps or
       are simply created by the boundary conditions, as such the gradient is not zero and nearly every condition for the Kramer-Rao bound to
       hold is false. See reference above.
  8 => CvBM would produce an exact, but degenerate fit: the data is such that a snaky fit made of straight segments, as produced by
       some asymptotic values of a and b for the conventional binormal model, is an
       exact fit to the data, as such it is also the MLE fit as the perfect fit has the highest possible likelihood.
  9 => Data is such that a fit made of two straight segments with AUC = 0 is possible, this is a perverse fit where *all* data points are
       misclassified. Usually it implies an input error or something worse.

######## Fitting CvBM (the same model upon which were based labroc/Rocfit/RSCORE (U of IOWA) ##
Produces an MLE fit using the conventional binormal model. The input
has to be categorical data (e.g., from catgrz). The model assume that data can be modeled using a two multinomial distributions. Data-points are assumed to be

```
independent.
WARNING: If the categorical data fed to the program is not reduced to its truth
runs (for example if the data is
          categorical with categories of identical truth following each other) aka
fully-collapsed, the Hessian and
          variance covariance matrices will refer to the collapsed data. In
principle the non-collapsed matrices can
          be derived from the collapsed matrices, but I could never imagine a
reason for computing them. One can send
          the data through the categorizer first, to remove redundant
        categories. If you disagree with this decision let us know  (particulaly
why)

function_name with expected mangling
-------------
LINUX/WINDOWS: __labroc_functions__cvbmroc_mle
OSX        : __labroc_functions_MOD_cvbmroc_mle

Call from R
First define the following variables

mn  integer, number of actually-negative cases
ms  integer, number of actually-positive cases
NumCat  integer, number of categories/truth runs
k <- matrix(x, NumCat),  integer array with categorical data for actually-negative
l <- matrix(x, NumCat), integer array with categorical data for actually-positive
DebugLogFile <- 0, integer  0 if no  debugfile should be written
a, real value between -infinity  and +infinity, depending what the fit is --
negative values are associated with curves that have performance worse than
random.
b, real value between 0 and infinity, depending upon the fit. The more different
from 0 is |log[b]| the less convex-looking will be the fit.
auc, real value , AUC between 0 and 1, depending upon the fit is
var_auc, real value, the variance of AUC
vc_cutoffs <- array(0, c(1, NumCat - 1)), estimated cutoffs
logl, value of the log likelihood function at the fit
error, error message, about the fit, 0 is fine. for the other errors see above
varcov <- matrix(0, NumCat+1, NumCat+1), variance covariance matrix


 Call of the function from R
 .C("__labroc_functions_MOD_cvbmroc_mle",
as.integer(mn),
as.integer(ms),
as.integer(NumCat),
as.integer(k),
as.integer(l),
as.integer(DebugLogFile),
as.double(a),
as.double(b),
as.double(auc),
as.double(var_auc),
as.double(vc_cutoffs),
as.double(logl),
as.integer(error),
as.double(varcov))
```

```
Pesudocode call

function_name(mn, ms, num_categ, catn_in, cats_in, idebug,
                        a_par, b_par, auc, variance_auc, vc_cutoffs_out, log_like,
ierror,
                        cov_out, hessian_out)

integer, intent(in):: mn ! number of actually negative cases
integer, intent(in):: ms ! number of actually positive cases
integer, intent(in) :: num_categ ! Number of categories as created by catgrz
integer, dimension(num_categ), intent(in):: catn_in, cats_in ! arrays containing
categorical data
integer, intent(in) :: idebug               ! 0 = no debug; 1  = debug
double,intent(out)                         :: a_par, b_par ! MLE of the parameters
double,intent(out)                         :: auc ! AUC, area under the curve
double,intent(out)                         :: variance_auc ! estimated variance of
AUC

double, dimension(num_categ-1), intent(out) :: vc_cutoffs_out  ! cutoff parameter
values at the maximum found
double, intent(out)                        :: log_like ! value of the log
likelihood function at the final point
integer, intent(out)                                    :: ierror ! Error flag
about the MLE fit
! Note that the error values are set in this routine, or initialize_d_a_c so if
their numbers have
! to be changed, they have to be changed here, the rest of the subroutines use
their own numbering
! specific per routine. Look above for the different meaning. Not only 0 is
successful fit
double, dimension(num_categ+1,num_categ+1), intent(out)  :: cov_out ! these are
used because the number
double, dimension(num_categ+1,num_categ+1), intent(out), optional  :: hessian_out
! of categories can be
     ! different inside proproc because of collapsing. Note that only the reduced
hessian will be returned, the
     ! rest will be set to garbage, NOTE THAT THIS WAS NOT DESCRIBED
     !IN THE EXAMPLE ABOVE


########  Fitting PBM aka proproc
#############################################################################
produce a MLE fit using the proper binormal model. The input has to be categorical
data (e.g., from catgrz)
 WARNING: If the categorical data fed to the program is not reduced to its truth
runs (for example if the data is
          categorical with categories of identical truth following each other) aka
fully-collapsed, the Hessian and
          variance covariance matrices will refer to the collapsed data. In
principle the non-collapsed matrices can
          be derived from the collapsed matrices, but I could never imagine a
reason for computing them. One can send
          the data through the categorizer first, to remove redundant categories.

function_name with expected mangling
-------------
LINUX/WINDOWS: __proproc_functions__pbmroc_mle
```

```
OSX        : __proproc_functions_MOD_pbmroc_mle


Call from R
 First define the following variables

mn  integer, number of actually-negative cases
ms  integer, number of actually-positive cases
NumCat  integer, number of categories/truth runs
k <- matrix(x, NumCat) , integer array with categorical data for actually-negative
l <- matrix(x, NumCat) , integer array with categorical data for actually-positive
DebugLogFile <- 0, integer  0 if no  debugfile should be written
da, real value  between 0 and infinity, depending what the fit is
ce, real value,   between -1 a,d 1, depending upon what the fit is
auc, real value  AUC between .5 and 1, depending upon what the fit is
var_auc real value, the variance of AUC
vc_cutoffs <- array(0, c(1, NumCat - 1))# estimated cutoffs
logl,  value of the log likelihood function at the fit
error,  error message, about the fit, 0 is fine. see above for more details.
varcov <- matrix(0, NumCat+1, NumCat+1), variance covariance matrix.


Call of the function from R
 .C("__proproc_functions_MOD_pbmroc_mle",
as.integer(mn),
as.integer(ms),
as.integer(NumCat),
as.integer(k),
as.integer(l),
as.integer(DebugLogFile),
as.double(da),
as.double(ce),
as.double(auc),
as.double(var_auc),
as.double(vc_cutoffs),
as.double(logl),
as.integer(error),
as.double(varcov))

Pseudocode call

function_name(mn, ms, num_categ, catn_in, cats_in, idebug,
                       d_a_par, c_par, auc, variance_auc, vc_cutoffs_out,
log_like, ierror,
                       cov_out, hessian_out)

integer, intent(in):: mn ! number of actually negative cases
integer, intent(in):: ms ! number of actually positive cases
integer, intent(in) :: num_categ ! Number of categories as created by catgrz
integer, dimension(num_categ), intent(in):: catn_in, cats_in ! arrays containing
categorical data
integer, intent(in) :: idebug               ! 0 = no debug; 1  = debug
double,intent(out)                         :: d_a_par, c_par ! MLE of the
parameters
double,intent(out)                         :: auc ! AUC, area under the curve
double,intent(out)                         :: variance_auc ! estimated variance of
AUC
```

```
double, dimension(num_categ-1), intent(out) :: vc_cutoffs_out  ! cutoff parameter
values at the maximum found
double, intent(out)                         :: log_like ! value of the log
likelihood function at the final point
integer, intent(out)                                    :: ierror ! Error flag
about the MLE fit
! Note that the error values are set in this routine, or initialize_d_a_c so if
their numbers have
! to be changed, they have to be changed here, the rest of the subroutines use
their own numbering
! specific per routine. Look above for the different meaning. Not only 0 is
successful fit
double, dimension(num_categ+1,num_categ+1), intent(out)  :: cov_out ! these are
used because the number
double, dimension(num_categ+1,num_categ+1), intent(out), optional  :: hessian_out
! of categories can be
     ! different inside proproc because of collapsing. Note that only the reduced
hessian will be returned, the
     ! rest will be set to garbage

##############################################################################
##########################################################
NON-PARAMETRIC ESTIMATION OF AUC (also known as Wilcoxon statistic, trapezoidal
AUC, empirical AUC, Mann-Whitney form of the WIlcoxon Statistic, U-statistic ...
##############################################################################
##########################################################
Here is the description of  how to call some of the available methods, their
description can be found in
Gallas BD, Pesce LL, editors. Comparison of ROC methods for partially paired
data2009: SPIE

mn, integer, number of actually-negative cases
ms, integer, number of actually-positive cases
AP, real, actually positive cases values
AN, real, actually negative cases values
AUC, real, ouput, area under the ROC curve or trapezoidal AUC, or Wilcoxon
statistics, or ...
VarAUC, real, output, variance of the AUC

Call from R:
.C("__roc_nonparametric__delonganddelong", as.integer(mn),as.integer(ms),
as.integer(1),as.double(AN), as.double(AP), as.double(AUC), as.double(VarAUC))

Other calls are possible when multiple modalities are present and  for partially
paired data (meaning that not all cases are in common
between modalities)

For those:
num_mod, integer, the number of modalities
DES_AN[ 1:mn,1:num_mod], integer,  design matrix,  basically an array with 1 when
a case is present in a modality and a 0 otherwise
DES_AP[ 1:ms,1:num_mod], integer,  design matrix,  basically an array with 1 when
a case is present in a modality and a 0 otherwise

wilc <- matrix(0,ncol= 1,nrow= num_mod ), output, array of U-statistic vallues
wilc_var <- matrix(0,ncol= num_mod,nrow= num_mod ), output, variance-covariance
matrix of the array of U-statistics
```

```
 Boostrap based method, see above.
 the last number, 100, it is the number of bootstrap samples. if it is 100, it is
a little too small.
C("__roc_nonparametric_MOD_gandpboot", as.integer(mn),as.integer(ms),
as.integer(num_mod),as.double(AN), as.double(AP),as.integer(DES_AN),
as.integer(DES_AP), as.double(wilc), as.double(wilc_var),as.integer(100))

One shot method based on moments, by B Gallas, see above
 .C("__roc_nonparametric_MOD_m_mod_one_shot", as.integer(mn),as.integer(ms),
as.integer(num_mod),as.double(AN), as.double(AP),as.integer(DES_AN),
as.integer(DES_AP), as.double(wilc), as.double(wilc_var))

Wilcoxon statistic based method
.C("__roc_nonparametric_MOD_zhouandgatsonis", as.integer(mn),as.integer(ms),
as.integer(num_mod),as.double(AN), as.double(AP),as.integer(DES_AN),
as.integer(DES_AP), as.double(wilc), as.double(wilc_car))


can still call also DeLong and DeLong but only if the data is fully-paired
.C("__roc_nonparametric_MOD_delonganddelong", as.integer(mn),as.integer(ms),
as.integer(num_mod),as.double(AN), as.double(AP), as.double(wilc),
as.double(wilc_se))

Pseudocode call, example with the bootstrap routine because it is the simplest
 WARNING: the variance covariance matrix has in the i>=j elements the variance
covariance matrix and in the i< j elements
         it will have the Var{U_i - U_j). This is done to have a more stable
estimate of that variance as opposed to
         Var{i} + var{j} - 2*cov{i,j}, this is not true for the other routines

gandpboot(mn, ms, num_mod, act_neg, act_pos, des_neg, des_pos, U_vec , U_vec_cov,
n_boot)

integer, intent(IN):: num_mod ! number of modalities or treatments analized.
integer, intent(IN):: mn ! total number of distinct negative cases (i.e., every
case that has a value for at least one of
                         ! the num_mod modalities)
integer, intent(IN):: ms ! total number of distinct positive cases (i.e., every
case that has a value for at least one of
                         ! the num_mod modalities)
real(kind=double), dimension(mn,num_mod), intent(IN):: act_neg ! actually-negative
input data for the two modalities to be analyzed
real(kind=double), dimension(ms,num_mod), intent(IN):: act_pos ! actually-positive
input data for the two modalities to be analyzed
! Design matrices. Here we assume that if there are values different from 0 or 1,
there is an input error (in general there are
! algorithms that allow the use of different flags for the design matrix, for
example to indicate clustering, however, ROCKIT
! cannot make use of them and therefore will not accept them.
integer, dimension(mn,num_mod), intent(IN):: des_neg ! actually-negative design
matrix (whether a case is present (1) or absent (0)
                                                  ! for each the two modalities to
be analyzed
integer, dimension(ms,num_mod), intent(IN):: des_pos ! actually-positive design
matrix (whether a case is present (1) or absent (0)
                                                  ! for each the two modalities to
be analyzed<E7>
```

```
Real(kind=double),dimension(num_mod), intent(out):: U_vec    ! Array of Wilcoxon
statistics, one per treatment
real(kind=double),dimension(num_mod,num_mod), intent(out):: U_vec_cov ! Variance-
covariance matrix of the U_vec in the i>=j
                                                                      ! elements the
variance covariance matrix and in the i< j elements
                                                                      ! it will have
the Var{U_i - U_j). This is done to
                                                                      ! have a more
stable estimate of that variance as opposed to
                                                                      ! Var{i} + var{j}
- 2*cov{i,j}
integer, intent(IN):: n_boot ! number of bootstrap sets
```