Separatrix library user manual

Johan Carlsson, Tech-X Corporation
teq-users@fusion.txcorp.com

$Id: usermanual.tex,v 1.1.2.2 2006/01/14 08:55:54 carlsson Exp $

The interface to the separatrix library is summarized in the “Introduction” section. A detailed description follows in the “Interface specification” and commented usage examples can be found in the “Examples” section.

Introduction

CGS units are used for all quantities. The floating-point precision is determined by the kind parameter `rq`, which is defined in the file `utils.f90`. By default `rq` is chosen to make `real(rq)` an 8-byte floating-point type. By changing line 7 of `utils.f90` from `rq=r8` to `rq=r4`, `real(rq)` becomes a 4-byte floating-point type instead.

The separatrix library takes a numerical MHD equilibrium (solution to the Grad-Shafranov equation) as input and finds the X- and O-points, the separatrix and an arbitrary set of flux surfaces. The library is accessed through the interface defined in the module `seplib`. The interface is constituted by the four subroutines `init_sep`, `exit_sep`, `find_separatrix` and `find_surfaces`.

`init_sep` takes 4 arguments: `msrf`, `mls`, `geom` and `sep`. `msrf` is the number of flux surfaces and `mls` is the number of coordinate points used to define the separatrix and each of the flux surfaces. The variables `geom` and `sep` are of derived types. `geom` holds information about the magnetic geometry (including the set of `msrf` flux surfaces) and `sep` holds the coordinates of the separatrix. `init_sep` allocates memory for the variables `geom` and `sep`.

`exit_sep` takes the same 4 arguments as `init_sep` and it does the opposite: it releases the memory allocated by `init_sep`.

`init_sep` must be called before any other operations on the separatrix library are performed. `exit_sep` should be called after all other operations on the separatrix library have been performed.

`find_separatrix` takes 4 arguments: `eq`, `geom`, `xpoint` and `sep`, where all are of derived types. `eq` holds the numerical equilibrium, `geom` and `sep` are as described above, and `xpoint` holds the coordinates of the X-point. The primary input to `find_separatrix` is the numerical equilibrium contained in `eq` and the primary output is the X-point and separatrix coordinates contained in `xpoint` and `sep`, respectively.

`find_surfaces` takes `geom` as its single argument and fills in the coordinates of the flux surfaces held by `geom`. It must be called after `find_separatrix`.

Interface specification

The data passed through the interface is encapsulated in the four variables `eq`, `geom`, `xpoint` and `sep`, which are all of derived type. The type declarations are in the `seplib` module in the file `seplib.f90`. For `eq` we have:

```
! Derived type that describes a numerical equilibrium and its meta data
type :: eq_type
   ! R and Z are the gridpoints where psi is given.
   real(rq), dimension(:), pointer :: R, Z
   ! psi is the stream function (poloidal flux / 2pi).
   real(rq), dimension(:), pointer :: psi
   ! Coordinates of magnetic axis, psi at magnetic axis.
   real(rq) :: raxis, zaxis, psix
   ! Limiter point coordinates and psi.
   real(rq) :: rlim, zlim, psil
   ! nsym = 1 for up/down symmetric equilibrium, 2 for asymmetric.
   integer :: nsym
end type eq_type
```

`eq` is predominantly an input variable and must be initialized before `find_separatrix` and `find_surfaces` are called. Our demonstration driver initializes `eq` from an EQDSK g-file:

```
subroutine read_eqdsk(ioun, goun, eq)

  ! We need the derived type for the equilibrium (eq_type) from seplib
  use seplib, only : rq, eq_type

  implicit none

  type(eq_type) :: eq
  integer, intent(in) :: ioun
  character*(*), intent(in) :: goun
  real(rq), dimension(:), allocatable :: dummyd
  real(rq) :: dr, dz, dummy, ro, rdim, zdim
  integer :: i, jm, km, j, k

  print *, ioun, goun

  open(unit = ioun, file = goun)

  read(ioun, '(52x,2i4)') jm, km

  allocate(eq%psi(jm * km), eq%R(jm), eq%Z(km), dummyd(jm))

  read(ioun, '(5e16.9)') rdim, zdim, dummy, ro, dummy
  read(ioun, '(5e16.9)') dummy, dummy, dummy, dummy, dummy
  read(ioun, '(5e16.9)') dummy, dummy, dummy, dummy, dummy
  read(ioun, '(5e16.9)') dummy, dummy, dummy, dummy, dummy
  read(ioun, '(5e16.9)') (dummyd(i), i = 1, jm)
  read(ioun, '(5e16.9)') (dummyd(i), i = 1, jm)
  read(ioun, '(5e16.9)') (dummyd(i), i = 1, jm)
  read(ioun, '(5e16.9)') (dummyd(i), i = 1, jm)
  read(ioun, '(5e16.9)') (eq%psi(i), i = 1, jm * km)

  close(ioun)

  deallocate(dummyd)

  dr = rdim / (jm - 1)
  dz = zdim / (km - 1)

  eq%R = (/(i, i = 0, jm - 1)/) * dr + ro
  eq%Z = (/(i, i = 0, km - 1)/) * dz - zdim / 2_rq

  ! Convert to CGS
  eq%psi = eq%psi * 1.0e+08_rq
  eq%R = eq%R * 100_rq
  eq%Z = eq%Z * 100_rq

end subroutine read_eqdsk
```

For codes that do not load EQDSK g-files, the initialization of `eq` will obviously be implemented differently.

The only components of `eq` that are not strictly input variables are `eq%raxis`, `eq%zaxis` and `eq%psix`. If `eq%raxis` is non-zero, the coordinate (`eq%raxis`, `eq%zaxis`) is used as an initial guess for the axis location. If `eq%raxis` is set to zero, the geometric axis is used instead. In either case `eq%raxis` and `eq%zaxis` are overwritten with the values found by `find_separatrix`.

`geom` has the following type declaration:

```
! Auxilliary types for geom_type
type :: R_type
   ! R coordinate, partial derivatives of R WRT theta and psi, respectively,
   ! and poloidal magnetic field
   real(rq), dimension(:), pointer :: coord, theta, psi, Bpol
end type R_type

type :: Z_type
   ! Z coordinate, partial derivatives of Z WRT theta and psi, respectively,
   ! and Jacobian
   real(rq), dimension(:), pointer :: coord, theta, psi, J
end type Z_type

! Relates cylinder coordinates in configuration space to flux coordinates
type :: geom_type
   type(R_type), dimension(:), pointer :: R
   type(Z_type), dimension(:), pointer :: Z
   ! Normalized psi values at the flux surfaces
   real(rq), dimension(:), pointer :: psibar
   ! Poloidal angles along each flux surface where R, Z are calculated
   real(rq), dimension(:), pointer :: tswpts
   ! if wvert is inputted as non-zero then its outputted value is the
   ! elongation on axis and the bi-polar coordinate used for the flux
   ! geometry is stretched by that amount:
   ! z=zaxis+wvert*rho(psi,theta)*cos(theta)
   real(rq) :: wvert
end type geom_type
```

geom is both an input and an output variable. The input components are geom%psibar and geom%tswpts. In the demonstration driver they are initialized like this:

```
! Note psibar coordinate is stretched at the axis and the edge
geom%psibar=(/(i, i = 0, msrf - 1)/) / (msrf - 1.0)
geom%psibar=(sin(geom%psibar * pi2))**2

! Note tswpts must always be uniform (pi2 = pi / 2)
geom%tswpts=(/(4.0 * pi2 * i, i = 0, mls - 1)/) / (mls - 1.0)
```

geom%psibar is used to determine how the msrf flux surfaces should be distributed. Similarly, tswpts determines the mls poloidal angles at which the flux surface and separatrix (R, Z) coordinates are calculated.

The output components are geom%R and geom%Z, two arrays with msrf elements each. The elements of geom%R and geom%Z, one per flux surface, are themselves arrays with each of their mls elements corresponding to a poloidal point along the flux surface. In addition to the (R, Z) coordinates, these elements contain $(\partial R/\partial\theta, \partial R/\partial\psi, B_p)$ and $(\partial Z/\partial\theta, \partial Z/\partial\psi, J)$ (where J is the Jacobian), respectively. The values of geom%R and geom%Z are calculated and filled in by find_surfaces. Note that find_separatrix must be called before find_surfaces.

geom%wvert is most likely only of interest to expert users, who are undoubtedly capable of parsing the source code to understand how geom%wvert is properly used.

xpoint holds the location of the X-point and associated information.

```
! Derived type that describes an X-point
type :: x_point_type
   ! X-point coordinates and psi.
   real(rq) :: rxpt, zxpt, pxpt
   ! rxpr(2) and zxpr(2) define the box in which to search for the X-point
   ! If rxpr(1) = 0 the search is not limited
   real(rq), dimension(2) :: rxpr, zxpr
   ! If ixpt=2: uses X-point to define plasma boundary
   ! If ixpt=1: uses X-point or limiter point, whichever is active
   ! If ixpt=0: does not look for X-point and it better not be active
   integer :: ixpt
end type x_point_type
```

`xpoint` is both an input and an output variable. The input components are the flag `xpoint%ixpt` and the size-2 arrays `xpoint%rxpt` and `xpoint%zxpt`. The flag `ixpt` determines how the plasma boundary is defined. If there is no limiter, we can use the X-point to define the plasma boundary (`ixpt = 2`). If there is a limiter, `ixpt = 1` tells the separatrix library to use the limiter point to define the plasma boundary, unless the X-point is inside it. Finally, `ixpt = 0` can be used if we are not interested in the X-point but need to call `find_separatrix` before we can call `find_surfaces`. `xpoint%rxpr` and `xpoint%zxpr` can be used to box in the X-point for particularly difficult equilibria, like SSPX. If `xpoint%rxpr(1) = 0.0`, `xpoint%rxpr` and `xpoint%zxpr` are ignored; otherwise the search for the X-point is limited to the box with the corners (`rxpr(1)`, `zxpr(1)`), (`rxpr(1)`, `zxpr(2)`), (`rxpr(2)`, `zxpr(2)`) and (`rxpr(2)`, `zxpr(1)`). The output components are `xpoint%rxpt`, `xpoint%zxpt` (coordinates of the X-point) and `xpoint%pxpt` (flux at the X-point).

`sep` is primarily an output variable that holds the coordinates of the separatrix:

```
! Derived type that holds the coordinates of the separatrix
type :: sep_type
   real(rq), dimension(:), pointer :: R, Z
   ! theta is the increment in normalized flux away from the separatrix
   ! at which the plasma boundary is prescribed. That is if theta=0, the
   ! separatrix is the boundary; if theta=0.01 the 99% flux surface is
   ! the boundary
   real(rq) :: theta
end type sep_type
```

The input component is `sep%theta`. For the default value of `sep%theta = 0.0`, the library calculates the true separatrix. By slightly increasing the value of `sep%theta`, we can tell the library to instead find a flux surface just inside the true separatrix. For `sep%theta = 0.01`, it finds the 99% flux surface, etc. This might be useful in the rare cases where `find_separatrix` fails to find the true separatrix. After a successful call to `find_separatrix`, the coordinates of the `sep%theta`-designated surface are in the arrays `sep%R` and `sep%Z`.

The complete interfaces of the four subroutines provided by the separatrix library are as follows:

```
  subroutine init_sep(msrf, mls, geom, sep)
    integer, intent(in) :: msrf, mls
    type(geom_type) :: geom
    type(sep_type) :: sep

  subroutine exit_sep(msrf, mls, geom, sep)
    integer, intent(in) :: msrf, mls
    type(geom_type) :: geom
    type(sep_type) :: sep

  subroutine find_separatrix(eq, geom, xpoint, sep)
    type(eq_type), intent(inout), target :: eq
    type(geom_type), intent(inout) :: geom
    type(x_point_type), intent(inout) :: xpoint
    type(sep_type), intent(inout), target :: sep

  subroutine find_surfaces(geom)
    type(geom_type), intent(inout) :: geom
```

Exception handling

Exception handling was added to the separatrix library in early 2006. The purpose of exception handling is to return control to the driver when an exception occurs in the separatrix library, i.e when something goes wrong. The interface to the exception handling consists of the two subprograms `catch_xcpt()` and `throw_xcpt()` declared in the module `seperr`:

```
module seperr

  logical :: exceptions_enabled = .false.
  logical :: stop_on_error = .true.
  integer :: error_code = 0
  character(len = 128) :: error_msg = 'none'
  interface

     integer function catch_xcpt(arg)
       logical, intent(inout) :: arg
     end function catch_xcpt

     subroutine throw_xcpt(arg)
       integer, intent(inout) :: arg
     end subroutine throw_xcpt

  end interface

end module seperr
```

Exception handling is enabled by setting `stop_on_error = .false.` and passing the variable `exceptions_enabled` to the function `catch_xcpt()`. If exception handling is supported on the build platform (Linux with either lf95, ifort or pgf90 for now), `exceptions_enabled` is set to `.true.`. When `catch_xcpt()` is explicitly called it returns the integer zero. **When the subroutine `throw_xcpt()` is called, the execution of the code will appear to return from `catch_xcpt()`!** The return value of `catch_xcpt()` is the argument passed to `throw_xcpt()`, unless zero is passed to `throw_xcpt()` in which case `catch_xcpt()` returns the integer one. Here's an example from `driver.f90`:

```
  ! Catch errors that might occur when we initialize the separatrix library
  stop_on_error = .false.
  ierr = catch_xcpt(exceptions_enabled)
  if (ierr /= 0) then
     print *, 'The call to init_sep failed, the error message is: ', &
          trim(error_msg), '. Bailing out...'
     stop
  end if

  ! Initialize the separatrix library
  call init_sep(msrf, mls, geom, sep)
```

Here we just print an error message and terminate execution when an exception occurs, but we could try to handle the exception depending on exactly what went wrong (determined by the value of `ierr`), and call `init_sep` again.

The exception handling can be made as fine grained as desired (set up for every single call to the separatrix library), or more coarse grained (set up for a block of calls). In the remainder of the driver we use a more coarse-grained approach and group the calls done for each equilibrium. `catch_xcpt()` can be called an arbitrary number of times, a call to `throw_xcpt()` from anywhere in the code, no matter how deeply nested the subprogram calls are, will always return to immediately after the last call to `catch_xcpt()`. The only exception (no pun intended) is when `catch_xcpt()` is called in a subprogram (subroutine or function) that returns before `throw_xcpt()` is called, in which case the behavior is undefined.

Examples

For full details about how to use the separatrix library, look at the source files `seplib.f90` and `driver.f90`. `seplib.f90` contains the `seplib` module with declarations of the derived types and the subroutines used to access the library. `driver.f90` contains four commented examples of how to call the library. Here we will give a summary.

From the file `driver.f90`:

```
program driver

  ! The seplib module defines the interface (subroutines and
  ! associated derived types) to the separatrix library
  use seplib

  ! Some less important lines snipped out...

  ! The numerical equilibrium
  type(eq_type) :: eq

  ! The set of flux surfaces
  type(geom_type) :: geom

  ! The X-point
  type(x_point_type) :: xpoint
  ! The separatrix
  type(sep_type) :: sep

  ! Some less important lines snipped out...

  ! Initialize the separatrix library
  call init_sep(msrf, mls, geom, sep)
```

We first `use seplib` to get access to the separatrix library. Next we declare the 4 input/output variables `eq`, `geom`, `xpoint` and `sep`, and then we call `init_sep(msrf, mls, geom, sep)` to make the library ready for use. `msrf` and `mls` can be any positive integers, variables or constants. In the driver they are constants with values 101 and 127, respectively.

Next we set the values of the normalized flux for the set of `msrf` flux surfaces and the values of the poloidal angle at the `mls` coordinate points along the separatrix and each flux surface:

```
  ! Note psibar coordinate is stretched at the axis and the edge
  geom%psibar=(/(i, i = 0, msrf - 1)/) / (msrf - 1.0)
  geom%psibar=(sin(geom%psibar * pi2))**2

  ! Note tswpts must always be uniform (pi2 = pi / 2)
  geom%tswpts=(/(4.0 * pi2 * i, i = 0, mls - 1)/) / (mls - 1.0)
```

The next step is to load the numerical equilibrium into the variable `eq`. In our demonstration driver this is done by reading in an EQDSK g-file (see the subroutine `read_eqdsk` for details).

For the up/down asymmetric DIII-D equilibrium used in the first example there is no limiter, so we need to tell the separatrix library to use the X-point to define the plasma boundary:

```
  eq%nsym = 2 ! Equilibrium is up/down asymmetric

  ! We don't have an estimate of the magnetic axis location
  eq%raxis = 0.0; eq%zaxis = 0.0

  ! There is no limiter
  eq%rlim = 0.0; eq%zlim = 0.0
  xpoint%ixpt = 2 ! Use X-point for plasma boundary

  ! First get axis, x_point and separatrix
  call find_separatrix(eq, geom, xpoint, sep)

  ! Some less important lines snipped out...

  ! Next get all the flux surfaces
  call find_surfaces(geom)
```

Note that `find_separatrix` must be called before `find_surfaces`. A plot of the output is shown in Fig. 1.

In the next example, with an SSPX equilibrium, we impose a limiter but still let the X-point define the plasma boundary. Also, we help the separatrix library find the X-point by narrowing down the search region.

```
  ! Set limiter
  eq%rlim =  50.222
  eq%zlim = -14.7758

  xpoint%ixpt = 1 ! Use the limiter or X-point for plasma boundary

  ! Needs X-point boxed in
  xpoint%rxpr(1) =  35.0
  xpoint%rxpr(2) =  49.0
  xpoint%zxpr(1) = -10.0
  xpoint%zxpr(2) =  50.0

  ! First get axis, x_point and separatrix
  call find_separatrix(eq, geom, xpoint, sep)
```

The output is plotted in Fig. 2.

In the third example, a FIRE equilibrium, we know that there is up/down symmetry so we pass that information to the library by setting `eq%nsym = 1` before we call `find_separatrix`.

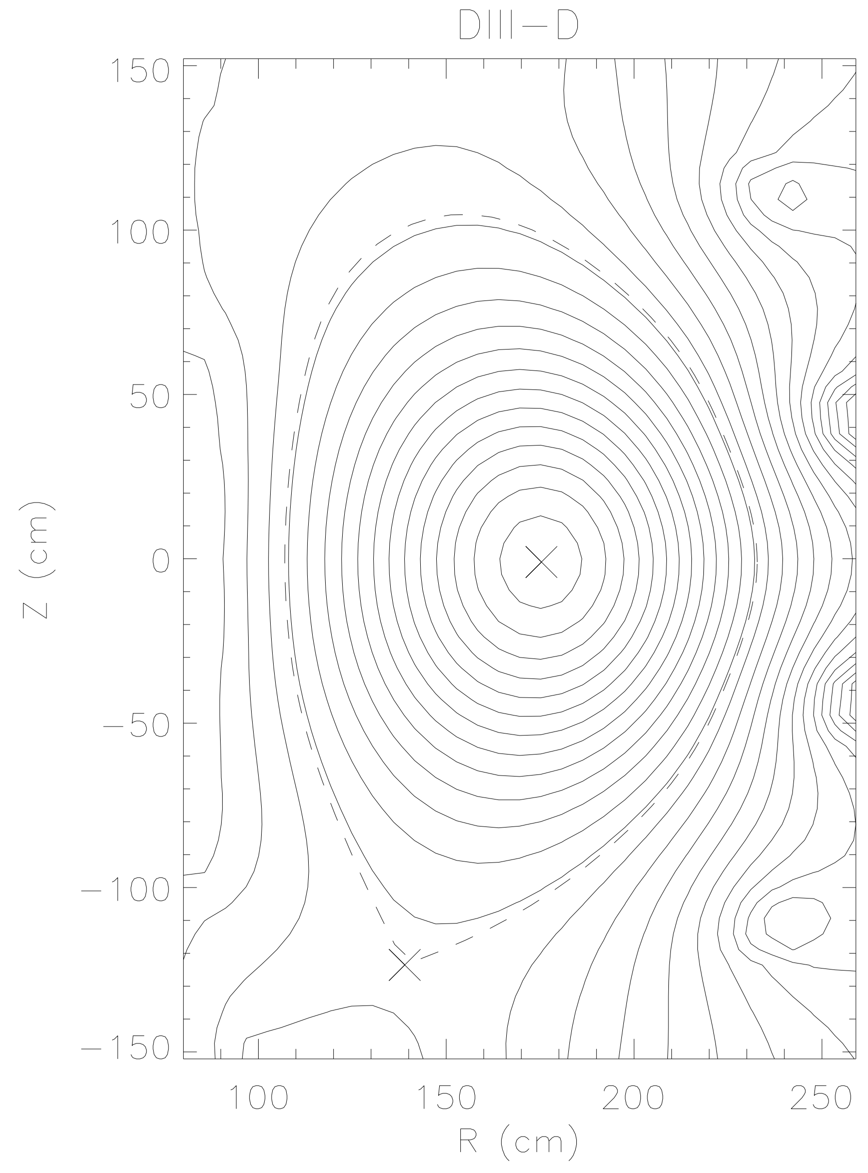

Figure 1: DIII-D equilibrium. The dashed line is the separatrix and the crosses are the O-point (magnetic axis) and X-point.

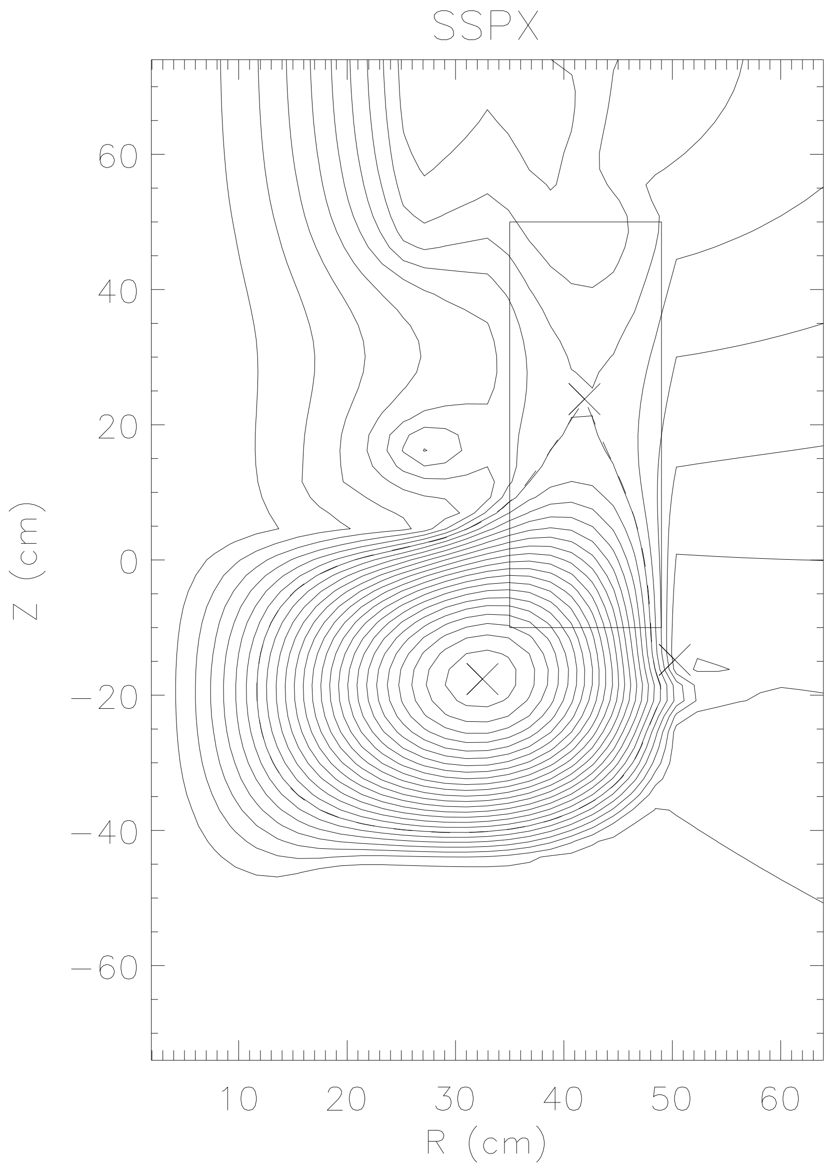

Figure 2: SSPX equilibrium. The dashed line is the separatrix and the crosses are the O-point, X-point and the limiter. The solid rectangle is the X-point search region.

```
  eq%nsym = 1 ! Equilibrium is up/down symmetric

  ! Set limiter
  eq%rlim = 150.0
  eq%zlim =   0.0

  xpoint%ixpt = 1 ! Use the limiter or X-point for plasma boundary

  ! First get axis, x_point and separatrix
  call find_separatrix(eq, geom, xpoint, sep)
```

The output is plotted in Fig. 3. The output from the fourth and final example, an ITER equilibrium, is plotted in Fig. 4.

Note that `exit_sep` is called after all other operations on the library are done:

```
! We're done, release the allocated memory
  call exit_sep(msrf, mls, geom, sep)

end program driver
```

History

The source code for the separatrix library was extracted from the Corsica code by Donald Pearlstein, LLNL, and Johan Carlsson, Tech-X Corporation, and converted from Fortran77/Basis to Fortran90.

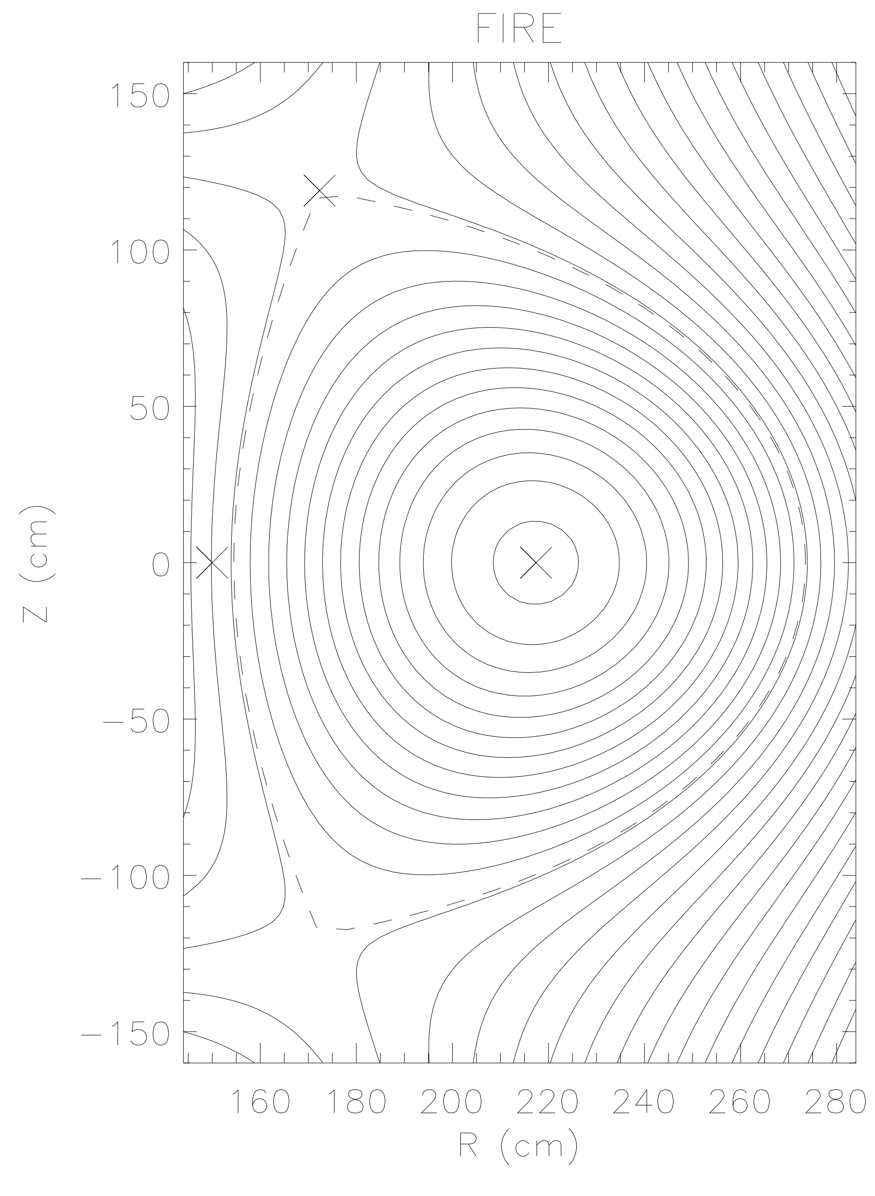

Figure 3: FIRE equilibrium. The dashed line is the separatrix and the crosses are the O-point, X-point and the limiter.

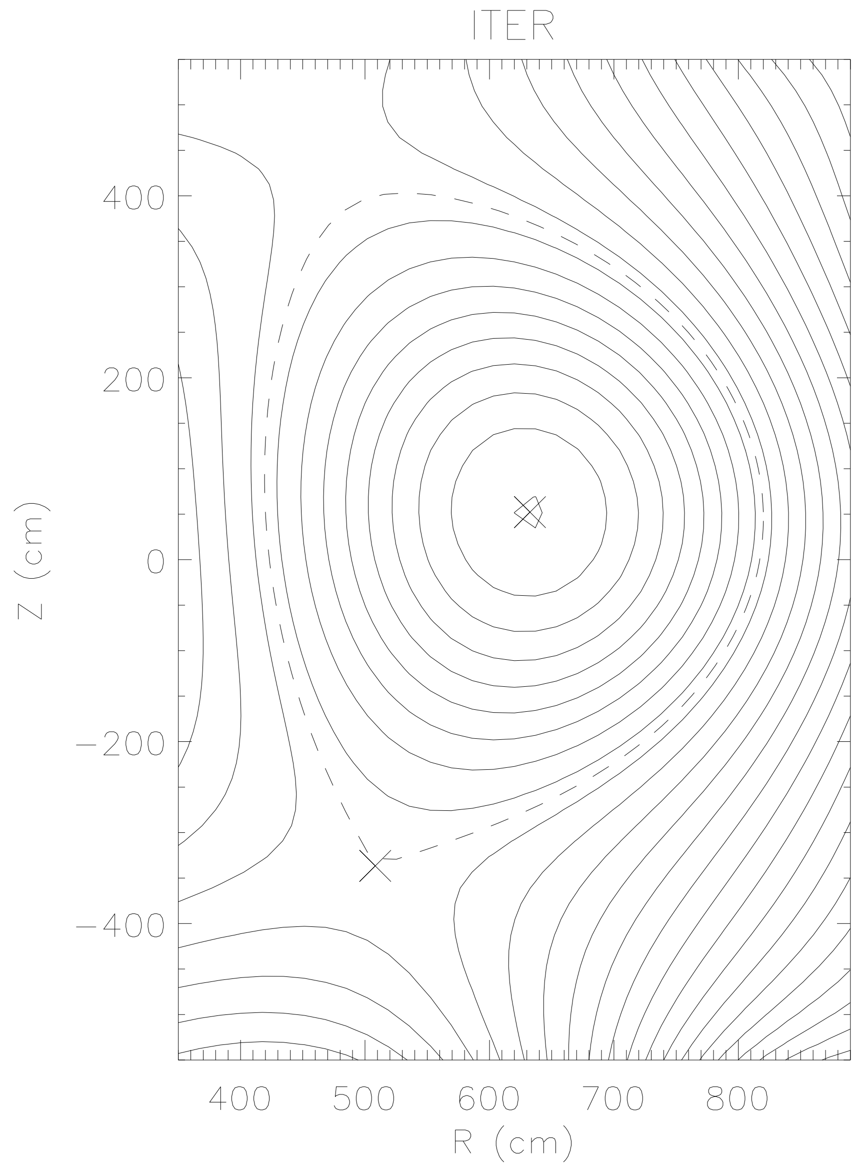

Figure 4: ITER equilibrium. The dashed line is the separatrix and the crosses are the O-point and the X-point.