Modelling SSPX Equilibria with Corsica

A Users Guide

R. H. Bulmer

Lawrence Livermore National Laboratory

February 27, 2004

Abstract

The Corsica equilibrium solver, embodied in the `caltrans` executable, is used to model equilibria in the SSPX spheromak with a collection of routines defined via script file `sspx.bas`. This users guide describes how to load SSPX data into a Corsica session and "fit" the equilibrium by adjusting the toroidal current and model profile parameters. Routines are available to evaluate and plot various diagnostic quantities. Profile Thomson data may also be loaded into the session and ohmic power analyses may be performed. Corsica is built with LLNL's Basis system, which provides the interactive user interface, text and binary file I/O and graphics capabilities.

Contents

List of Tables

List of Figures

1 Introduction

This user's guide describes the application of the Corsica[1] axisymmetric ideal-MHD equilibrium package to the modelling of spheromak equilibria in SSPX[2], including the importation of SSPX shot data and the adjustment of model profile parameters to reconstruct equilibria and evaluate and plot diagnostic quantities. This guide assumes the user is familiar with Unix and/or Linux operating systems. The Corsica code is installed on LLNL's Energy and Environment Directorate (EED) Solaris file system and the PAT/MFE Linux Cluster file system. The Corsica code is built with LLNL's Basis system—it provides the user interface (parser), text and binary file I/O, numerous utility routines and graphics facilities. The Corsica code is presently embodied in the `caltrans` executable. (CalTrans is a collaborative effort coupling LLNL's Corsica code with General Atomic's ONETWO transport code.)

The Corsica executable (`caltrans`) consists of compiled and interpreted modules grouped into packages, where selected routines and variable identifiers may be referenced directly by the user. The Basis system provides the user interface to these public identifiers and includes a Fortran 90-like scripting language with which the user steers the code in a flexible way.

The usual mode of operation is interactive with the Basis parser interpreting and executing input on a line-by-line basis. When user input becomes lengthy it can be put into text files—scripts—and more efficiently read into the code. In the extreme, the entire execution can be controlled by script files; thus operation in a batch-like mode is possible. The main sections of the document are: (2) SSPX equilibrium model in Corsica, (3) a Corsica session, (4) bias field configurations, (5) importing SSPX measurements, (6) equilibrium reconstruction, (7) stability analysis, and (8) ohmic power analysis. New users should refer to Appendix A for instructions on how to set up your Unix environment for using Corsica Appendix B contains general usage information about "Basis codes". Appendix C describes the SSPX generic equilibria, used as a starting point in most sessions. Appendix D describes the SSPX script files, of interest to all users. Appendix E describes auxiliary (binary) files used by the SSPX scripts, and Appendix F describes IDL procedures for extracting data from the SSPX shot database for importation into Corsica. All figures appear at the end of the document.

2 SSPX equilibrium model in Corsica

This section describes how spheromaks are modelled in Corsica. In the SSPX spheromak, measurements of the injector current, $I_{gun}(t)$, and poloidal field, $B_\theta(t)$ at several wall locations are available. In addition, measurements of the electron density and temperature from the Profile Thomson Scattering diagnostic are available. The measurements are extracted from the SSPX database with IDL procedures and can be imported into a Corsica session to facilitate equilibrium reconstruction and ohmic power analysis.

2.1 Spheromak equilibria

The SSPX spheromak[2] may be modelled under the assumption of toroidal axisymmetry in the ideal MHD equilibrium package in Corsica. The equilibrium of the spheromak is described in the force-free approximation by λ, which is known to be a flux function. We generalize the definition to include pressure by equating $\lambda(\psi) = dF/d\psi$, with ψ the poloidal flux stream function ($\Psi/2\pi$) and $F(\psi) = RB_\varphi$. At low beta, this becomes the usual definition; more generally, inclusion of the diamagnetic current yields:

$$\frac{dF}{d\psi} = \frac{\mu_0}{B^2}\left(\mathbf{j}\cdot\mathbf{B} - \frac{dp}{d\psi}F\right)$$

We use this definition in the Grad-Shafranov equation in cylindrical coordinates:

$$R\frac{\partial\psi}{\partial R}\frac{1}{R}\frac{\partial\psi}{\partial R} + \frac{\partial^2\psi}{\partial Z^2} = -\lambda\int_0^\psi \lambda d\psi - \mu_0 R^2\frac{dp}{d\psi}$$

where $\psi = 0$ on the symmetry axis ($R = 0$). The limit $\lambda =$ *constant* and pressure $p = 0$ describes the Taylor state.

The Grad-Shafranov equation is solved for SSPX (see Figure 1) as a free boundary problem with the λ profile form *prescribed*. The flux conserver is represented by a discrete number (~ 300) of toroidal current elements. To constrain the solution, the total toroidal current, I_φ, is also prescribed.

The current on open field-lines and in the private flux region in the injector is parameterized by

$$\mu_0\mathbf{j} = \lambda_{edge}\mathbf{B}$$

When relaxation processes are strong in the edge plasma, λ_{edge} can be assumed constant, independent of the magnetic flux. The injector is also characterized by the applied bias magnetic field, Ψ_{gun}, which for the SSPX pulse length (few milliseconds) is frozen into the conducting wall. We take the edge beta to be zero in all our calculations, so the total discharge current is

$$I_{gun} = \lambda_{edge}\Psi_{gun}/\mu_0\ , \quad \text{for constant } \lambda_{edge}$$

Thus, a spheromak equilibrium in Corsica is completely determined by the four elements: (1) flux conserver configuration, (2) vacuum field configuration (bias coil positions and currents), (3) total toroidal current, and (4) the λ-profile form. With the exception of a scale factor on λ, these four elements must be explicitly prescribed in the Corsica model.

The flux conserver, with two components (the inner electrode and the outer flux conserver), is assumed to be perfectly conducting. The vacuum flux, as generated by a set of bias coils, is evaluated at many points on the flux conserver and is held constant as the equilibrium is computed. The total toroidal current (I_φ) within the flux conserver is a free parameter specified by the user, as is the λ-profile, a prescribed function of poloidal flux only.

Measurements of the injected gun current, $I_{gun}(t)$, flowing in from the top of the inner electrode and the poloidal field, $B_\theta(t, \mathsf{p})$, at up to $\mathsf{p} = 19$ locations are available from the experiment as a function of time, with a $1\ \mu$s interval. These measurements are used to "reconstruct" an equilibrium in Corsica.

2.2 SSPX model in Corsica

The primary quantities for a given SSPX shot pertinent to equilibrium reconstruction are: the nine bias coil currents, which remain constant during the shot, and measurements of $I_{gun}(t)$ and $B_\theta(t, \mathsf{p})$ at many time-points t during the pulse. The magnetic field is measured at $\mathsf{p} = 1, \ldots, 19$ probe locations distributed poloidally on the outer flux conserver, as shown in Figure 1. In some cases measurements from multiple toroidal locations are available at the same poloidal location—they are averaged to provide a single value consistent with the axisymmetric model. In order to utilize the ohmic power analysis routines in `sspx.bas`, measurements of $n_e(R)$ and $T_e(R)$ from the Profile Thomson Scattering (PTS) diagnostic are usually available for each shot at a particular time-point.

The conducting shell is represented by "coil" elements with toroidal current, adjusted by the equilibrium solver to freeze-in the bias flux. Coils in Corsica are characterized by the mean radius, R_c, and vertical position, Z_c, of the current centroid. The current is uniformly distributed in filamentary current loops arrayed over a rectangular or parallelogram cross-section of size $\Delta R_c \times \Delta Z_c$. The number of filaments in each coil element is $n_c = n_{\Delta R_c} \times n_{\Delta Z_c}$. The Corsica model for parallelogram cross-sections follows the EFIT convention[1]. Two types of parallelogram cross-section models are available, as shown in Figure 2. Type-1 coils have angle $\alpha_c \neq 0$, and Type-2 have angle $\alpha_{c2} \neq 0$. The coil parameters are listed in Table 1. Except for `nc`, all of these quantities are 1D arrays of

Table 1: Corsica coil parameters (see Fig. 2)

quantity	Corsica *name*	*units*	*description*
N_c	`nc`	—	number of coil elements
R_c	`rc`	m	mean radius
Z_c	`rc`	m	vertical position
ΔR_c	`drc`	m	radial build
ΔZ_c	`drc`	m	vertical build
α_c	`ac`	rad.	Type-1 inclination
α_{c2}	`ac2`	rad.	Type-2 inclination
n_{R_c}	`nrc`	—	filaments across ΔR_c
n_{Z_c}	`nzc`	—	filaments across ΔZ_c
—	`pfid`	—	coil name

length `nc`. In the SSPX model, the first `ncplot`=12 coil elements represent the

[1] EFITis General Atomic's equilibrium fitting code, see `http://web.gat.com/efit/`.

nine bias coils, as discussed in §3.2.2.

The thickness of the conducting shell is 2 mm (representing the nominal current penetration depth for the few-ms SSPX pulse). The current is contained in discrete filaments, about 16 per coil element, arranged in a single row centered in the middle of the element (i.e., 1 mm from the plasma-facing surface). The SSPX conducting wall model is shown in Figure 3.

The plasma-facing coordinates of the conducting wall are contained in the Corsica `rplate(nplates,2)`, `zplate(nplates,2)` arrays (in centimeters), which hold the end-points of `nplates` line-segments used to describe the geometry. These plate elements are also shown in Figure 3.

The remainder of this document describes how to import the SSPX measurements into Corsica, reconstruct an equilibria, and evaluate various code diagnostic quantities.

3 A Corsica session

This section describes how to start-up a Corsica session; introduces script *routines*, usually defined as Basis functions; describes how to change an equilibrium; and how to look at results.

3.1 Start-up

The Corsica code is presently embodied in the `caltrans` executable (see Appendix A for details on getting set-up to use it). The user is free to choose any name for an alias or link to the executable, but here we will use the name `caltrans` as the name given to the operating system to launch the code. The name CalTrans is used in this document to refer to the entire code distribution: the source files, the compiled executable, standard script files (including the SSPX scripts), and a wrapper shell-script named `caltrans`. The wrapper is used to automatically set environment variables that ensure the correct set of scripts is found for each version of the compiled code, as described in Appendix A.

It is necessary to start-up the Corsica session with a previously saved SSPX equilibrium model, contained in a binary "save-file"[2]. By convention, save-file names have the suffix "`.sav`". One chooses a save-file appropriate for the analysis to be performed, although the generic save-file "`sspx.sav`" will usually suffice. If one is going to import shot data, choose a similar equilibrium as expected by the bias coil currents for the shot, as this may expedite subsequent analyses. Here we demonstrate start-up with a generic save-file, of which there are several variants as discussed in Section 4.

The minimal `caltrans` command-line argument is the name of a save-file,

[2]Save-files, and other `caltrans` binary files, are created and read by the Basis "portable-files-from-Basis" (PFB) package—they are *portable* across all Unix platforms.

but we will typically want to define a problem name string and also include the SSPX script filename, as shown below.

Typical Unix command-line to launch `caltrans` for SSPX applications...

```
caltrans -probname pname sspx.sav sspx.bas
```

The `-probname` option directs the code to use the following string (*pname*) as the file-name prefix for graphics and session-log output files. It also sets Basis variable `probname`. Corsica will next read the binary data in the save-file, and automatically execute the equilibrium solver with the information from that file. It then reads any other text files named on the command-line, in this case `sspx.bas`, which, in turn, reads all of the SSPX script files as described in Appendix D.2.

After starting up the code as above, the prompt string "`corsica> `" will be displayed then the code awaits user input.

Terminate a Corsica session with the `quit` command, or by issuing the disconnect signal: `^D` (`CTRL-D`).

3.2 Changing an equilibrium

After loading an SSPX equilibrium, either from a generic save-file or any other SSPX save-file, one may alter the equilibrium in three ways: (1) change the toroidal current, (2) change the bias coil currents (or the bias coil configuration), or (3) change the λ-profile form. Secondarily, the grid resolution may be changed, as well as the extent of the open field-line region in the injector annulus. The steps involved in making such changes are described in the following subsections, including a discussion of wall versus X-point limited equilibria.

3.2.1 Modifying the toroidal current

The total toroidal current for the equilibrium, I_φ, is specified by code variable `plcm` in MA. Variable `plcm` is an input to the equilibrium solver (when flag `ipscl`=0). Variable `plc` is an *output* quantity containing the total toroidal current in abamperes, or `plc*10` amperes. The output quantity for toroidal current in the confined region is contained in variable `placur` in amperes.

To change the toroidal current, execute:

```
plcm=new_value
run
```

for example, to double the present value:

```
plcm=2*plcm
run
```

To determine the value of the toroidal current, just enter its name: `plcm`. The `run` command in `caltrans` executes the Grad-Shafranov solver, using the

present state as the initial guess. The semicolon in Basis can be used to place multiple statements on the same line, so one may equivalently enter:

```
plcm=new_value; run
```

The Basis system has a mechanism for displaying documentation for any of the code variables that are accessible from the command-line parser. To see the documentation for `plcm`, execute:

```
list plcm
```

Entering just `list` will display the documentation for the `list` command.

3.2.2 Modifying bias coil currents (`setcc`)

Changing the bias coil currents is a bit more involved. Corsica variable `cc` is a vector of coil currents. Listing variable `cc` will show that it is a statically-allocated array of length 500, contains "coil currents" in units of MA (actually, MA-turns), and is *limited* by variable `nc`. To get the contents of a variable, just type its name, so entering `nc` we find that it has the value 327 for a generic SSPX save-file. This means that the vector `cc` of coil currents can contain up to 500 values, but the number of values used in this model is 327. The coil current vector `cc` contains elements both for the bias coils and the flux-conserver wall elements. The number of elements used for bias coils is contained in variable `ncplot` (12 for SSPX). There are 9 bias coils in SSPX as shown in Figure 1, but coil 3 is comprised of 4 sub-coils, so in Corsica 12 coil elements are needed to represent the 9 bias coils. Displaying the string variable `pfid(1:12)` will show the relationship between Corsica coil elements 1-12 and the SSPX coil names: 1, 2, 3A, . . . , 3D, 4, . . . , 9.

To change a bias coil current, change one or more elements of the coil current vector `cc(1:12)` then execute the `run` command. To update the vacuum flux in the flux conserver, however, an additional step is required. Script function `wall_sph` must be executed to impose the modified vacuum flux as a boundary condition for the flux conserver. This must be followed by another `run` command to make the equilibrium consistent with the modified vacuum flux. So, for example, the steps required to double the current in the injector solenoid (SSPX coil 9 and Corsica coil element 12), are:

```
cc(12)=2*cc(12)
run
wall_sph
run
```

This gets more complicated if one wants to change the current in coil 3, as it is comprised of elements `cc(3:6)` which have different values because the number of turns is different (the sub-coils are in series with the same circuit current, I_3). The number of turns in the bias coil elements is contained in variable `ntc(1:12)`, which is defined in one of the SSPX script files. Therefore, the syntax for changing the current in coil 3 to say, 350 A, is:

```
cc(3:6)=ntc(3:6)*350*1e-6
```

To simplify the process of changing bias coil currents, SSPX script function `setcc`[3] is available.

Set bias coil currents with...

`call setcc(;I1,I2,I3,I4,I5,I6,I7,I8,I9)`

`I1,...I9` : specifies the desired circuit current [A] in each bias coil, which default to their present value.

Note that the arguments to `setcc` are optional[4] and default values will be provided.

Function `setcc` takes the desired bias coil circuit currents, converts them to MA-turns, maps the 9 coils to the 12 Corsica coil elements, updates the vacuum flux in the conducting shell, and then updates the equilibrium. The following demonstrates changing the current in coil 3 to 350 A and in coil 9 to 400 A, leaving the other coil currents as-is.

```
setcc(,,350,,,,,,400)
```

Coils may be turned off by setting their current to exactly zero in `setcc`, but note that no elements of `cc` may be exactly zero—`setcc` assigns a small value to `cc` when zero current has been requested.

In most cases, one will be working with SSPX shot data, where the bias coil currents will be set automatically when data is loaded into a session, as described below in §5.1.

Script function `coils` will display a table of bias coil parameters: SSPX coil name, number of turns, ampere-turns, circuit current and measured value, if known.

Display bias coil parameters with...

`call coils`

3.2.3 Changing the λ profile

The λ-profile model built into Corsica allows one to specify the variation of λ as a function of normalized poloidal flux over three regions: (1) confined, λ_c,

[3]Script routines are defined as Basis *functions* (or sometimes macros). Basis functions may or may not have return values (see App. B.7). In this document, the syntax "`call name()`" is used to emphasize that the function does not provide a return value. When invoking a script function that has no return value in an interactive session, the `call` token is redundant and is seldom used.

[4]The semicolon is used in the *definition* of Basis script functions to signal the beginning of optional arguments, for which default values will be provided—the semicolon itself is not entered by the user when invoking the function—they are used in this document to emphasize that default arguments exist.

(2) external or open field-lines, λ_e, and (3) the private flux region, λ_p. Normalized poloidal flux $\tilde{\psi}$ (or x, as used below) is defined with:

$$\tilde{\psi} \equiv x \equiv \frac{\psi - \psi_{axis}}{\psi_{edge} - \psi_{axis}}$$

Thus, normalized poloidal flux is zero at the magnetic axis and unity at the edge of the confined region.

In the confined region ($0 \leq x \leq 1$) the model profile form is:

$$\lambda_c(x) = \lambda_0(1 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^n)$$

where λ_0 (at the magnetic axis) is determined self-consistently by Corsica. At the edge of the confined region $x = 1$, so

$$\lambda_{edge} = \lambda_c(1) = \lambda_0(1 + \sum_{i=1}^{4} a_i)$$

In the external region ($x > 1$), with $x_\varphi = x(R = 0)$ the profile form is:

$$\lambda_e(x) = \lambda_{edge}\left[\frac{1 + b_1(x/x_\varphi) + b_2(x/x_\varphi)^2 + b_3(x/x_\varphi)^3 + b_4(x/x_\varphi)^m}{1 + b_1/x_\varphi + b_2/x_\varphi^2 + b_3/x_\varphi^3 + b_4/x_\varphi^m}\right]$$

In the private flux region ($x < 1$), when $g \neq 0$, the profile form is:

$$\lambda_p(x) = \lambda_{edge} x^g$$

The Corsica variable names corresponding to the profile variables introduced above are given in Table 2. If `asph=bsph=gsph=0`, then λ is uniform over the

Table 2: Corsica spheromak profile parameters

variable	*description*
`asph(1:4)`	coefficients $a_i, i = 1, 2, 3, 4$
`nasp`	exponent n
`bsph(1:4)`	coefficients $b_i, i = 1, 2, 3, 4$
`nbsp`	exponent m
`gsph`	exponent g
`bsph_flag`	integer variable (see text)

entire region inside the flux conserver—the "flat-lambda" condition or Taylor state, which is the condition in generic save-files.

Variable `bsph_flag` is an integer in {0,1,2,3,4,12,13,14,23,24,34} that specifies, if $1 \leq i \leq 4$, that b_i will be varied to maintain continuity (in $d\lambda/d\psi$) at the edge ($x = 1$) of the confined region; if `bsph_flag` > 4 it specifies the doublet i, j and will use b_i and b_j to preserve continuity at $x = 1$ *and* constrain $d\lambda/d\psi = 0$

at the geometric axis ($R = 0$); if `bsph_flag`=0 (the default) no constraints are placed on $d\lambda/d\psi$.

The most common way to affect the λ-profile is to specify n and a_4. So, for example, to make λ drop-off sharply to nearly zero at the edge of the confined region, do something like:

```
nasp=10; asph(4)=-0.9; run
```

Generally, one will use the equilibrium fitting routine `fit` discussed in §6.1 to change the lambda profile to be consistent with measurements.

3.2.4 Changing the grid resolution (`gridup` and `griddown`)

Two routines (`gridup` and `griddown`) are available to double or halve the grid resolution. The default number of grid points for an SSPX equilibrium is $N_R \times N_Z = 33 \times 65$ where N_R and N_Z are contained in Corsica variables `jm` and `km`, respectively. These grid specifications result in $\Delta R \simeq \Delta Z \simeq 2\,\text{cm}$ (the actual values are contained in code variables `dr` and `dz`). Changing the grid resolution involves changing these two parameters (`jm` and `km`) and executing `wall_sph` to update the vacuum flux at the shell. The process is somewhat complicated in that other parameters need to be set for self-consistency, so the procedure is encapsulated in the two routines.

Double the R-Z grid resolution with...

```
call gridup
```

Halve the R-Z grid resolution with...

```
call griddown
```

Since increasing the resolution will increase the memory requirements of the code, the number of times one may grid-up is limited by computer memory. Conversely, decreasing the resolution may result in such a coarse grid that the equilibrium cannot be resolved. If either type of grid resolution change fails, one must start a new session with a valid equilibrium.

Both of these routines adjust all necessary parameters, compute a new equilibrium, execute `wall_sph` to update the vacuum flux, then re-execute the equilibrium solver to produce a self-consistent solution.

Changing the grid resolution to something other than the standard 33×65 will necessitate re-evaluation of the Greens functions, which may take time to evaluate.

3.2.5 Changing the open field-line region (`zcutoff`)

In addition to imposing the vacuum flux on the conducting shell, the `wall_sph` routine also sets some grid variables that specify the extent of the open field-line region.

The grid variables that specify the open field-line region are `jwl(1:km)` and `jwu(1:km)`, which are integer arrays containing the radial grid index to begin the open region (variable `jwl`) and the radial grid index to end the open region (variable `jwu`). The `pgrid` plotting routine can be used to view the contents of these two variables graphically. Sample output is contained in Figure 4.

The elements of `jwl` and `jwu` are initialized by `wall_sph` to conform to the SSPX conducting wall. A function called `zcutoff` may be used to alter the open field-line boundary in the injector region or divertor region.

Modify the open field-line region...

`call zcutoff(z_cutoff;cutoff_type)`

`z_cutoff` : the axial position or cutoff-point [cm], measured in the Corsica coordinate system, at the inner electrode, above which λ is "cutoff" (i.e., set to zero) if `z_cutoff` > 0, or if `z_cutoff` < 0, λ will be set to zero below the cutoff-point (i.e., the divertor region).

`cutoff_type` : if non-zero, specifies that the cutoff will be horizontal. If zero (the default), the cutoff will approximately follow the vacuum field-line passing through the cutoff-point.

Figure 5 shows the results of executing

```
zcutoff(50)
pgrid
zcutoff(55,1)
pgrid
```

3.2.6 X-point versus wall-limited configurations

An SSPX equilibrium may be limited by the conducting shell or, more usually, an X-point, depending upon the bias field configuration, plasma current, etc. The close proximity of the SSPX confined plasma region and the current-carrying wall exacerbates resolution of the topology, resulting in the following artificial constraints within the equilibrium solver:

1. **X-point search-box**—a search box is used to define a region where an X-point is allowed to limit the confined plasma boundary, to avoid extraneous X-points that exist in the filamentary wall model (or outside the conducting shell). The search box is defined by variable `rxpr(2)` representing the minimum and maximum radial extent, and `zxpr(2)` representing the minimum and maximum axial extent, both in units of centimeters. The search box is unique for each type of SSPX configuration as described in §4.1.

2. **Static limiter**—if the equilibrium is limited by the conducting wall, the limiting point must be *prescribed* by the coordinates `rlim` and `zlim` [cm], which must lie on the plasma-facing side of the conducting wall (i.e., 1 mm from toroidal current filaments of the conducting wall).

What these artificial constraints mean is that if the topology changes significantly, one must take extra steps to make sure: (1) the code works, and (2) it finds the "correct" solution.

Sessions always start with a good solution, but if topology problems occur when parameters are changed, we can take smaller steps and make appropriate corrections to either the search box or the limiter point until the desired end state is reached. In practice, these extra steps have only been required when exploring new configurations, such as changing from a bias field configuration which utilizes only the injector solenoid to one where all the bias coils are used.

The Corsica plotting routine `pb` can be used to examine the X-point search box (shown as a rectangle with dashed lines) relative to the conducting shell and boundary of the confined flux region. The limiter point will be indicated with a circle marker and labeled `L0`—the circle is filled if the point (`rlim`,`zlim`) is limiting the confined region. Figure 6 shows sample `pb` output for wall-limited and X-point limited configurations.

The X-point search box is usually located in the injector annulus, but sometimes the limiting X-point will appear in the diagnostics slot, necessitating a change in the `rxpr`, `zxpr` values.

If the configuration is limited by the conducting shell, one must adjust the position of the limiter point to be consistent with the particular conditions at hand, as described below. The `setlimiter` routine maps the coordinates of the plasma-facing side of the conducting wall to Corsica's `rlimw`, `zlimw` arrays. As the equilibrium solver iterates, the self-consistent limiter point is adjusted as the code converges. When iterations have been completed, `rlim` and `zlim` will contain the coordinates of a new limiter point.

Adjust the limiter point with...

`call setlimiter(;`*`offset`*`,`*`n1`*`,`*`n2`*`,`*`plotit`*`)`

`offset` : offset [cm] of the limiting surface from the plasma-facing surface of the wall. This is usually 0, but a positive value will offset the limiting surface towards the plasma region.

`n1` : index of beginning wall segment number [default: 2].

`n2` : index of ending wall segment number [default: `nplates`].

`plotit` : if non-zero, plot the conducting wall and the `rlimw`,`zlimw` coordinates generated [default: 0].

Activate by setting `limw=1`.

The `setlimiter` routine is usually executed with no arguments (the optional arguments are used mainly for debugging purposes). After execution, set `limw=1`, then `run` the equilibrium (and verify that the solution is reasonable), then set `limw=0`. If `limw` is non-zero, subsequent executions will activate the

dynamic limiter feature which will exacerbate equilibrium convergence.

This process can be tedious, as the equilibrium solver may fail if parameters are changed too much. One way to ease into the desired solution is to start by manually putting a fictitious limiter point well inside the conducting shell at some convenient location (e.g., `rlim=40`, `zlim=-20`) and `run` the equilibrium. Then, in small increments, increase `rlim` until the confined boundary is (somewhere) close to the conducting wall, by viewing the topology with `pb` at each step. Then, when the confined boundary is reasonably close to the wall, turn on the dynamic limiter feature, as demonstrated in the following lines

```
zlim=-20; rlim=40; run
rlim=rlim+1; run; pb
⋮
rlim=rlim+1; run; pb
setlimiter
limw=1
run
pb
limw=0
```

3.3 Looking at results

There are several plotting routines defined in the SSPX scripts. A macro named `Layout` is a customized (for SSPX) routine based on the tokamak plotting routine called `layout`, defined in one of the standard Corsica scripts. The `Layout` macro makes a plot of the configuration (bias coils, flux conserver, contours of poloidal flux) and lists some input and output quantities. Figure 7 is an example of the output from the `Layout` macro.

The Basis system provides several plot commands, the most common one being `plot`. Knowing that Corsica variable `qsrf(1:msrf)` contains the q-profile on the `msrf` confined flux surfaces, and `psibar(1:msrf)` contains the corresponding values of normalized poloidal flux, one can plot $q(\tilde{\psi})$ with the Basis statement:

```
win
plot qsrf psibar
```

Here, the `win` command opens an NCAR X-window, so the plot will be displayed on your screen as well as copied to the NCGM file. To close an NCAR X-window, use the `win close` statement.

There is also an SSPX script function, `pq`, that plots $q(\tilde{\psi})$, which produces the plot shown in Figure 8.

The λ-profile (or F'-profile) over the *confined* region may be graphed in a similar way, with the following:

```
plot 100*fpsrf psibar
```

where the multiplier converts cm^{-1} to m^{-1}. To display $\lambda(R)$ is a bit more complicated, so an SSPX script function `plvr` ("plot-lambda-versus-R") is available. It takes an optional argument, used as the upper scale limit for the plot. Its output is shown in Figure 9. To plot contours of $\lambda(R, Z)$, use the `pl2d` ("plot-lambda-2D") routine.

Table 3 contains a list of Corsica flux surface quantities of interest for spheromak models, and Table 4 some grid quantities.

Table 3: Corsica flux-surface quantities

variable	*description*
`msrf`	number of confined flux surfaces
`psibar(msrf)`	$\tilde{\psi}(i)$, $i = 1, \ldots,$`msrf`
`qsrf(msrf)`	$q(\tilde{\psi})$
`fpsrf(msrf)`	$F'(\tilde{\psi})$ $\left(= \lambda(\tilde{\psi})\right)$, in cm^{-1}
`cusrf(msrf)`	$\oint B \cdot dl$, in G cm

The variable `cusrf` is proportional to the current contained with the closed flux surfaces ,so

```
cusrf(msrf)/(0.4*pi)
```

represents the confined toroidal current, also contained in variable `placur`.

Table 4: Corsica grid quantities

variable	*description*
`jmkm`	number of R-Z grid points
`psi(jmkm)`	$\psi(R, Z)$ in G cm^2/radian
`psiv(jm,km)`	$\psi(R, Z)$ 2D representation of `psi`
`br(jmkm)`	$B_R(R, Z)$ in G
`bz(jmkm)`	$B_Z(R, Z)$ in G
`bt(jmkm)`	$B_\varphi(R, Z)$ in G
`bmod(jmkm)`	$\lvert B\rvert(R, Z)$ in G

Most grid quantities in Corsica are one-dimensional, of length `jmkm` = `jm`$\times$`km`. The Basis `shape` routine can be used to map one-dimensional arrays to two-dimensional arrays "on-the-fly", for example

```
real b2d = shape(bmod,jm,km)*1e-4 # |B| in Tesla
```

creates new 2D array `b2d(jm,km)` and contour plots can be made with the Basis `plotz` command:

```
plotz b2d,r,z
```

Some predefined variables are already defined in 2D form, so the following will make contour plots of the poloidal flux in units of Wb/radian.

```
plotz psiv*1e-8,r,z
```

The user may create any number of new variables during a session and it is possible to "clobber" (re-define) existing variables. This may corrupt the code so it is best to avoid doing it by making sure that a new variable you wish to create does not already exist. One way to do this is by executing the `list` command on the variable name. If `list` reports it as an unknown name, the name is safe to use. You can also instruct Basis to undefine the variable by using the `forget` command:

```
forget variable_name
```

The `list` command can be used to explore information contained in the Corsica variable descriptor database. Corsica, as with any Basis code, has a "variable descriptor file" (VDF) associated with each code package. The VDF contains variable definitions, organized into groups (group names begin with an uppercase letter, all variables begin with a lowercase letter). As of this writing, there are 28 packages in the code, some of which are predefined by Basis, but most of them unique to Corsica. The package most relevant for spheromak equilibrium modelling is the equilibrium package, `eq`. The `list` command:

```
list eq.variables
```

will display the names of all variables defined in the equilibrium package that are accessible to the user, organized by group name. Variable `cusrf` is a member of group `Dgns` ("diagnostic quantities"), and listing with a group name will display each variable along with its built-in comments, for example:

```
list Dgns
```

Thus, the `list` command provides a way to browse the Corsica database of variables during a session.

3.3.1 Features available with `Layout`

The `Layout` macro and associated routines provide some flexibility in controlling the appearance of the plot.

Display the equilibrium configuration with macro `Layout` with...

`Layout(;coil_style)`

`coil_style=0` : shows all the bias coils with their coil names (default).

`coil_style=1` : draws the bias coil cross-sections in proportion to their current.

`coil_style=2` : draws all the bias coil outlines and shows their filamentary sub-elements.

`coil_style=3` : draws all the bias coil outlines and shows their index numbers (as opposed to their coil names).

`coil_style=4` : like 2, but also shows the R-Z grid lines.

The *`coil_style`* is retained, in a variable of the same name.

The R-Z range of the `Layout` image is controlled by global variables `rclmin`, `rclmax`, `zclmin` and `zclmax` [cm], which are carried in save-files, but can be changed by the user to affect the `Layout` image. A function named `zoom` can also be used to "zoom-in" (or out).

Change the image range in `Layout` with `zoom`...

`call zoom(;r1,r2,z1,z2)`

`r1,r2,z1,z2` : specify new plot range values [m] (i.e., change `rclmin`, etc.), but remember the initial values. To restore to the initial values, do `zoom("reset")`. The statement `zoom("grid")` will automatically set the R and Z range to conform to the R-Z grid domain used by the Grad-Shafranov solver.

For example, to reduce the image range to that of the R-Z grid, execute

```
zoom("grid"); Layout; zoom("reset")
```

The color of `Layout` components may also be altered. The Basis variable `color` contains a list of valid NCAR color names. Script routine `colors` will show these colors in an NCAR X-window, and display a list of component names and their current color setting. Use the `cmap` routine to change the color of individual components.

Change the color of components in `Layout` with...

`call cmap("name","color")`

`name` : component name (from the `colors` list).

`color` : new color for the component (from the Basis `color` list).

To change the color of the poloidal flux contours to white, for example, execute

```
cmap("plasma","fgcolor")
```

The Basis color names `fgcolor` and `bgcolor` refer to the foreground and background colors, which are white and black respectively when displayed in an X-window. The foreground and background colors are interchanged in NCGM files so hardcopy will have a white background.

Flux contour levels in the `Layout` graphic are shown in increments specified in variable `delta_psi_contour` [Wb], with a default value of 5×10^{-3}. An alternative is to show a particular number of uniformly spaced contour levels, specified in variable `nlevels`. In order for this option to take effect, one must "forget" the contour level increment specifier with the Basis statement

```
forget delta_psi_contour
```

Later, to reinstate the contour level specifier, do something like

```
real delta_psi_contour=5e-3
```

To eliminate the legend, set logical variable `layoutLegend` to `false`.

3.4 Saving an equilibrium to disk (`saveit`)

Script function `saveit` will save an equilibrium in a disk file in the current directory. It is a wrapper for the general purpose Corsica save-file writer: `saveq` but with optional automatic file-name generation.

Save an equilibrium to disk with...

`chameleon` *`file_name`*`=``saveit(;`*`string`*`)`

`file_name` : the return value of this function is the explicit save-file name.

`string` : character string—interpreted as an explicit save-file name if it ends in `".sav"`. If shot data has been loaded with `lsd` (see §5.1) then *`string`* defaults to a name of the form *`shot_time`*`.sav`, where *`shot`* is taken from code variable `shotName` and *`time`* is taken from code variable `shotTime` (both of which are set by `lsd`). If argument *`string`* exists but does not end in `".sav"`, then it is interpreted as a *modifier* and the save-file will be named *`shot_time_string`*`.sav`.

This function has a return value—the full name of the save-file. Note the use of the `chameleon` type in Basis, which causes the left-hand side of an assignment statement to take on the type of the right-hand side (in this case, a character string). This is preferable to using the Basis `character` declaration, as it requires an explicit character count.

The return value allows one to capture the name of the save-file in a variable, which is sometimes useful if it has been embedded in a user's script file, but in most cases it is executed with just the syntax "`saveit("...")`".

Examples of using `saveit` are

```
saveit("some_name.sav") # save as file "some_name.sav"
saveit                  # use default name:  shot_time.sav
saveit("good_case")     # save as shot_time_good_case.sav
```

3.5 Restoring an equilibrium from disk

One may restore an equilibrium from disk with the Basis `restore` statement as follows

```
package eq
restore "shot_time.sav"
run
```

The `package` statement is necessary if `caltrans` happens to be in a package *other* than the equilibrium package[5]. Note that file names containing certain

[5] If the prompt string is `"corsica> "`, the code is in the equilibrium package: `eq`.

character combinations must be quoted. Make sure to execute the restored equilibrium with `run` to evaluate all quantities self-consistently.

There is also an SSPX script routine, called `reload`, which is easier to use if `lsd` has already been invoked.

Restore an equilibrium from disk with...

`call reload(;t)`

t : an optional time-point (in ms or μs). The `reload` routine uses the current shot number in `shotName` and, if argument *t* is not specified, the time value in `shotTime` to construct a save-file name to restore from a previously saved equilibrium reconstruction.

3.6 Summarizing save-files (ss)

The `ss` routine can be used to make a summary of save-files in the current working directory.

Summarize save-files with...

`call ss(;regexp)`

regexp : regular expression string specifying which save-files to summarize. Specifying "`*.sav`" for *regexp* will select all save-files in the current working directory [default: "*shotName*`_*.sav`"].

This routine is designed for use when reconstruction save-files have been created for particular shots, using the routines described below in Sections 5 and 6. The routine offers a quick way to identify special cases where input parameters may have been changed.

The following example shows the default `ss` output when executed in a directory where 16 save-files exist for SSPX shot 12098. In most cases, the λ-profile exponent `nasp` was 2 and the λ-continuity option turned off, but note that case 4 is unique (`bsph_flag=1`) and case 9 has `nasp`=10.

```
corsica> ss
Loading 16 save-files...

Parameters for SSPX #12098 save-files matching "12098_*.sav"
No.    time    plcm    nasp asph(4)    nbsp bsph(4)    gsph bsph_flag
  1   0.400   0.454   2.000   1.294   6.000   0.000   0.000     0
  2   0.600   0.364   2.000  -0.154   6.000   0.000   0.000     0
  3   0.800   0.338   2.000  -0.237   6.000   0.000   0.000     0
  4   1.000   0.353   2.000  -0.148   6.000   0.000   0.000     1
  5   1.200   0.356   2.000  -0.160   6.000   0.000   0.000     0
  6   1.400   0.354   2.000  -0.158   6.000   0.000   0.000     0
  7   1.600   0.355   2.000  -0.152   6.000   0.000   0.000     0
  8   1.800   0.347   2.000  -0.145   6.000   0.000   0.000     0
```

```
 9   2.000   0.344  10.000  -0.106   6.000   0.000   0.000     0
10   2.200   0.338   2.000  -0.112   6.000   0.000   0.000     0
11   2.400   0.337   2.000  -0.072   6.000   0.000   0.000     0
12   2.600   0.337   2.000  -0.025   6.000   0.000   0.000     0
13   2.800   0.333   2.000   0.001   6.000   0.000   0.000     0
14   3.000   0.320   2.000  -0.042   6.000   0.000   0.000     0
15   3.200   0.293   2.000  -0.186   6.000   0.000   0.000     0
16   3.400   0.254   2.000  -0.391   6.000   0.000   0.000     0
```

4 Bias field configurations

To use Corsica for an SSPX application, one must start-up the code with a save-file named on the command-line. Save-files are binary files in so-called PFB (Portable Files from Basis) format and contain all necessary data to model an equilibrium. Bias field configurations are usually established automatically when one loads data for a particular SSPX shot, as described in §5.1. There are so-called generic save-files available for SSPX which contain nominal bias coils currents for popular field configurations and benign (flat) λ-profiles.

4.1 Generic save-files

There are several generic save-files for SSPX. They are identical except for the bias field configuration, which has a strong effect on the configuration of the equilibrium. The generic save-files all have a toroidal current of $I_\varphi = 500\,\mathrm{kA}$ and a uniform (flat) λ-profile.

Table 5 contains a summary of the present set of generic save-files. Details are given in Appendix C. It is often possible to start-up an SSPX session using

Table 5: Bias coil configurations (generic save-files)

	name	*save-file*	*description*
0	ZERO	`sspx_zero.sav`	zero-bias-flux
1	SOL	`sspx_sol.sav`	solenoid-only
2	STD	`sspx_std.sav`	standard-flux
3	MF	`sspx_mf.sav`	modified-flux
4	BCS	`sspx_bcs.sav`	bias-coil-standard
5	BCM	`sspx_bcm.sav`	bias-coil-modified
6	BCV	`sspx_bcv.sav`	vertical-field
7	NOZ	`sspx_noz.sav`	nozzle-field
8	LG	`sspx_lg.sav`	lower-gun
–	—	`sspx.sav`	link to `sspx_mf.sav`

the save-file name `sspx.sav`, which is a symbolic link to the "modified-flux" save-file. However, in some cases, it will be necessary to choose a generic save-file that is close to the case of interest.

4.2 Bias field routine (biascoils)

Script function biascoils is available during a session to get an up-to-date list of the generic bias coil configurations.

Display information about generic save-files with...

call biascoils(;"*name*")

name : can be one of the bias coil configuration names (from Table 5), in either upper or lower-case, the string "list" to list all configurations or the string "all" to list the bias coil nominal currents for all configurations. Without an argument, biascoils will display a help message.

Script function coils will list the bias coil circuit currents, number of turns and total coil currents for the present equilibrium, as described in §3.2.2.

Script function pvac is available to plot the flux contours for the bias field.

Plot contours of the vacuum flux from the bias coils...

call pvac(;*highlight*,*delta_psi*)

highlight : a flux value [mWb] to highlight (default: none).

delta_psi : increment for contour levels (default: 1000*delta_psi_contour).

4.3 Evaluating bias flux (biasflux)

A *stand-alone* code called biasflux is available to evaluate the flux and field for the SSPX bias coil configuration, as a function of coil currents. It is located in directory /sspx/bin which must be in your Unix search path (see §A.2). Start-up biasflux at the Unix prompt and execute bf("help") for usage information. It defines the vector cc(1:9) which contains the circuit currents (in amperes) for the 9 bias coils (not to be confused with the Corsica vector of the same name, that contains the total current in MA-turns). Set this vector to the desired bias coil currents and execute bf to evaluate and plot the $|B|$ and flux surfaces. This routine writes graphical output to PostScript™ files with names of the form bf.*nnn*.ps.

5 Importing SSPX measurements

There are two routines available to load shot data into a Corsica session. The primary data (bias coil currents, magnetic probe measurements, gun current, etc.) are loaded with the "load-shot-data" routine, lsd, and PTS data is loaded with lpts, "load-PTS data". These routines are described in the following subsections.

5.1 Magnetic probe data (`lsd`)

The shot data written by IDL procedure `d4c.pro` (see Appendix F) will initially be contained in a text file named *`shot`*`.d4c`. This file must reside in the directory in which Corsica is initialized. To load the data into a Corsica session, execute the `lsd` routine.

Load shot data: $I_{gun}(t), B_\theta(t,p)$, etc. with...

`integer` *`error_code`* `= lsd(;`*`shot`*`,`*`time_point`*`,`*`plotit`*`,`*`sdate`*`)`

`error_code` : returned non-zero if an error occurred.

`shot` : integer shot number. If the starting equilibrium was already created after loading shot data, then its `shotName` variable from the save-file will contain the correct shot number which will be used as the default.

`time_point` : integer or real time-point, in μs or ms, for which data is to be interpolated for. As with the shot number, if the starting equilibrium was created for a specific shot, the variable `shotTime` will contain the default time-point.

`plotit` : if non-zero, creates a graphics frame with plots of $I_{gun}(t)$ and $B_\theta(t,p)$. The plot is made by calling `psd(,,`*`time_point`*`)`, see §5.2.

`sdate` : date string, with format `"YYMMDD"`, which specifies the shot date. This date is only used if the shot date does not yet exist in the database (see Appendix E.2), i.e., if the shot is relatively new. The string `"today"` may also be used to select the current date.

The `lsd` routine performs many functions:

1. Reads a *`shot`*`.d4c` file if it exists, or the binary file *`shot`*`-d4c.pfb` (§5.1.1).
2. Applies calibration factors to the data (§5.1.2).
3. Smoothes the data (if `w_smooth`> 0) (§5.1.3).
4. Performs baseline correction to the data (§5.1.4).
5. Assigns weight factors used by the fitting routines (§5.1.5).
6. Installs the measured bias coil currents and updates the vacuum flux in the equilibrium model (§5.1.6).
7. Interpolates for the gun current, integrated gun energy and magnetic field at the specified time-point (§5.1.7).

When loading data for the first time in a session when a generic save-file has been used to launch the code, specify the shot number and a time-point:

```
lsd(123,1.5)
```

Code variables `shotName` and `shotTime` will then contain the shot number and time-point. Thereafter, in the same session, the shot number contained in `shotName` will be used as the default, so only a time-point need be specified:

```
lsd(,1.6)
< do some calculations for t = 1.6 ms >
lsd(,1.7)
< do some calculations for t = 1.7 ms >
⋮
```
etc.

If the shot data has not yet been entered into the shot-date database, `lsd` will prompt for the shot date. One may also supply the date, or the string `"today"`, as the 4th argument to `lsd`.

5.1.1 Binary shot data files

When shot data is obtained from a *shot*`.d4c` text file, the data is written back to disk in PFB format in a new file named *shot*`-d4c.pfb`. Future `lsd` executions will then read the *binary* file, which greatly expedites data loading. The *shot*`-d4c.pfb` file, as opposed to the *shot*`.d4c` file, need not be in the directory of Corsica execution. The `lsd` routine will look in the directory named by the users SSPX_SHOTDATA environment variable for *shot*`-d4c.pfb` files, if they are not found in the current working directory.

The shot data files contain scalar and array quantities of both the measured data and the parameters used to *process* the data in the IDL session (e.g., `nfit` as used in the SSPX `get_mp090pxx` procedure). The names of the quantities in the shot data file are suffixed with "`_d4c`", so the value of `nfit` is contained in variable `nfit_d4c`. Use the `d4c` command to get a listing of the parameters used in the IDL session.

5.1.2 Calibration factors

Calibration factors (multipliers) are applied to the magnetic probe measurements each time a *shot*`-d4c.pfb` file is loaded. The calibration factors (and their standard deviations) are stored in a binary database file which is part of the CalTrans distribution. This file gets loaded into the session automatically by `sspx.bas`.

The calibration factors and standard deviations may be graphically displayed by executing the `calib` routine, the output of which is shown in Figure 10.

If the SSPX probes are re-calibrated, the calibration database must be updated as described in Appendix E.4.

5.1.3 Smoothing data

It is usually desirable to *smooth* the probe data, to eliminate rapid (few μs fluctuations). Smoothing is performed in `lsd` by the `boxcar`[6] algorithm in Corsica. Given a data set $y_i = y(t_i), i = 0, \ldots, N-1$, the "boxcar average" $\tilde{y}_i$ using a "boxcar" width of w is:

$$\tilde{y}_i = \frac{1}{w} \sum_{j=0}^{w-1} y_{i+j-w/2}, \quad i = w/2, \ldots, N - w/2$$

and $\tilde{y}_i = y_i$ for other values of i. Global integer variable `w_smooth` is the parameter w in `lsd`, and is constrained to be an odd number.

The smoothing parameter `w_smooth` is initialized to zero, so typically one inspects the $B_\theta(t, n)$ wave forms, sets `w_smooth` then re-executes `lsd`, for example:

```
win
lsd(,,1)
w_smooth=101; lsd(,,1)
```

After executing `lsd`, the 2D arrays `bprobe_smoothed`, `bt_smoothed`, etc., will contain the smoothed data.

Global variables created by the SSPX scripts (such as `w_smooth`) are not preserved in the binary save-file, so one must specify a value in each Corsica session. To enhance consistency from session to session, place any desired variable settings in a file named `customize.sspx` in the working directory (see §5.1.8). This file, if it exists, will be read when `sspx.bas` is read, ensuring that Corsica will be initialized with the same parameters for every session.

5.1.4 Baseline correction

Signal drift will introduce error in the measurements. When `lsd` is executed the first time, variable `t_baseline` will be initialized to two time-points (0 and the last time-point for which data is available). These two time-points are used to construct a linear baseline correction where each probe signal is adjusted to be zero at the two time-points. The value of `t_baseline` may be modified by the user to adjust this baseline correction. Figure 11 shows the graphical output from `lsd`, produced by the commands:

```
caltrans sspx.sav sspx.bas
lsd(12098,,1)
w_smooth=101
t_baseline(1)=0.137
lsd(,,1)
```

To change both time-points, use the syntax: `t_baseline=[0.137,3.85]`.

[6] The `boxcar` algorithm is, for consistency between Corsica and IDL, based on the `SMOOTH()` function in IDL.

5.1.5 Assigning weights

The `lsd` routine initializes the weight factors in `bprobe_wt(1:19)` for the poloidal field measurements, which are used by the fitting routine (see §6.1). If the magnitude of the signal for the `p`th probe is zero, or if its magnitude is greater than the quantity defined in global variable `bogus_probe`, then `bprobe_wt(`p`)` will be set to zero. Otherwise, the weight is assigned the corresponding value from global variable array `default_wt(1:19)`, which is initialized to 1.

5.1.6 Bias coil currents

The vacuum poloidal flux created by the bias coils will be imposed on the conducting shell model by `lsd` using the `setcc` routine. There are several generic save-files corresponding to specific bias field configurations. If the shot data being loaded by `lsd` has significantly different bias coil currents than the starting point, `lsd` will issue a message like:

```
Will try to morph from SOL to MF bias field.
```

which, in this example, means the starting equilibrium had a bias field using only coil 9, the solenoid-only configuration (SOL), but the shot data are based on the modified-flux configuration (MF). If this morphing process fails, it will be necessary to start with a more relevant save-file, which in this case would be the one named `sspx_mf.sav`. Appendix C contains more information on bias field configurations and their corresponding generic save-files.

5.1.7 Interpolation for quantities at particular time

The `lsd` routine interpolates for quantities at the time point specified in its argument list (or the default time, `shotTime`, extracted from a save-file). Since an equilibrium state represents one instant, $t = t^\star$, the variables listed in Table 6 are the experimentally measured (but perhaps smoothed and/or re-baselined) quantities to be compared to the equilibrium model.

Table 6: Interpolants returned by `lsd` at time-point $t = t^\star$

variable	num	units	description
`bprobe_meas`	19	T	$\langle B_\theta(t^\star, \mathtt{p})\rangle$ poloidal field (toroidal average)
`bprobe03_meas`	2	T	$B_\theta(t^\star, 3)$ poloidal field at probe `p03`
`bprobe09_meas`	6	T	$B_\theta(t^\star, 9)$ poloidal field at probe `p09`
`bprobe17_meas`	2	T	$B_\theta(t^\star, 17)$ poloidal field at probe `p17`
`bt_meas`	19	T	$\langle B_\varphi(t^\star, \mathtt{p})\rangle$ toroidal field (toroidal average)
`bt03_meas`	2	T	$B_\varphi(t^\star, 3)$ toroidal field at probe `p03`
`bt09_meas`	9	T	$B_\varphi(t^\star, 9)$ toroidal field at probe `p09`
`bt17_meas`	2	T	$B_\varphi(t^\star, 17)$ toroidal field at probe `p17`
`igun_meas`	1	A	$I_{gun}(t^\star)$, gun current
`wgun_meas`	1	J	$W_{gun}(t^\star)$, integrated gun energy

Scalar quantities `vfb_meas` (formation bank charge voltage), `vsb_meas` (sus-

tainment bank charge voltage), and the vector `cc_meas` of measured coil currents will also be available as global variables.

5.1.8 Keeping things straight

When working with specific SSPX shots, a customization feature is available to maintain consistency between sessions. Each time `sspx.bas` is read into a `caltrans` session, the script looks for files named `customize.sspx` or *`shot`*`_customize.sspx` in the current directory. These files can be used to ensure consistency between sessions by including the settings for parameters such as `w_smooth`, `t_baseline`, etc. so that the same settings are used for each session.

For example, suppose we are working on SSPX shot 12098 in a particular directory for which we have at least one saved equilibrium, say `12098_1.000.sav`. We decide we always want to smooth the data with `w_smooth=101` and adjust the baseline correction for the gun current ramp-up at $t = 0.137$ ms for all work on this shot. We create a file named `12098_customize.sspx` with contents:

```
w_smooth=101
call lsd
t_baseline(1)=0.137
call lsd
```

then, each time the code is launched the customization file will be read, executed, and echoed to the screen

```
caltrans 12098_1.000.sav sspx.bas
⋮
Reading "12098_customize.sspx"...
w_smooth=101
call lsd
t_baseline(1)=0.137
call lsd
```

This feature allows one to perform consistent analyses across sessions without having to remember parameter settings. Of course, they can be changed or overridden during the session.

5.2 Plotting shot data (`psd`)

The "plot-shot-data" routine (`psd`) can be used to plot $I_{gun}(t)$, $B_\theta(t)$ and $B_\varphi(t)$ in more detail than available with the plot option to `lsd`. It can be used to graph individual probe signals (both poloidal and toroidal). If multiple values (same poloidal location, but multiple toroidal locations) are available, such as for probes `p03`, `p09` and `p17`, they will also be shown. The `lsd` routine calls `psd` when its *`plotit`* argument is non-zero.

Plot shot data with...

call psd(;*plot_type*,*probe_number*,*time_point*)

plot_type : character string, one of "igun", "bpol" or "btor" or doublets with a "+" character (e.g. "igun+bpol") indicating the quantities to be plotted and whether one or two plots are to be displayed in the frame.

probe_number : integer probe number. If a probe number is not specified, the signals at all 19 positions will be displayed. If a probe number is specified, then only data for that probe will be displayed, but where multiple measurements are available (e.g., p09) then all the signals at that position will be shown.

time_point : if specified, then a vertical line will be drawn on the plots of $I_{gun}(t)$, $B_\theta(t)$ and/or $B_{\varphi(t)}$ at the indicated time.

The plot will span the full time range given in the shot data file (variable `t_d4c`), but one may select a narrower range by specifying values in global variable `psd_time(1:2)`, the beginning and ending times, in ms.

Figure 12 shows the output from executing

```
psd("bpol",9)
```

which displays the poloidal field at the 6 toroidal locations for probe p09.

5.3 Profile Thomson scattering data (`lpts`)

Function `lpts` loads PTS data, $n_e(R)$ and $T_e(R)$, from a text file with a name of the form `ptsfit_`*`shot`*, created by IDL script `ptsfit.pro` as described in Appendix F.2.

Load PTS data with...

integer *number_of_channels* = lpts(;*shot*,*npoly*,*plot_type*)

number_of_channels : this return value is the number of PTS channels; zero indicates an error occurred, or the `ptsfit_`*`shot`* file could not be found.

shot : integer SSPX shot number, defaults to the contents of `shotName`.

npoly : integer order of the polynomial used to fit the PTS data, initialized to 4.

plot_type : integer plot type, one of {0, 1, 2, 3} [default: 1].

The 1st argument is an integer shot number for which a `ptsfit_`*`shot`* file must be available, either in the current working directory or the directory specified by the user's SSPX_SHOTDATA or SSPX_PTSDATA environment variables. The default value of *shot* is taken from the Corsica `shotName` variable for the current equilibrium, so in most cases one need only enter:

```
lpts
```

The `lpts` routine returns the number of PTS data channels. A return value of zero indicates that the file for the designated shot could not be found, so in user scripts the following syntax is suggested.

```
if (lpts(shot) <> 0) then
  # do something useful here
else
  remark "cannot load PTS data for "//format(shot,0)
endif
```

The data will be smoothed and mapped to flux-surface-based arrays using Corsica's `polyfit` routine. Use the 2nd argument, ***`npoly`***, to change the number of Legendre polynomial basis-functions to alter the degree of smoothness of the fit. This argument also sets Corsica variable `npoly`, which is used as the default in `lpts`. The initial value of Corsica's `npoly` is 4. If ***`npoly`*** is 0, the data will be fit with spline functions with no smoothing.

The 3rd argument can take the value 0, 1, 2 or 3 to specify the type of plot frames to create. A plot type of 0 creates no plots, type 1 creates plots of n_e and T_e versus R, type 2 creates plots of n_e and T_e versus normalized poloidal flux (Corsica variable `psibar`) and type 3 creates both frames from types 1 and 2.

Data channels may be masked off by setting elements of a mask array, named `pts_mask`, to zero. The elements of `pts_mask` are initialized to 1.

The values of (uniform) density and temperature in the open field-line region will, by default, be assigned to match the most inboard measurement (selection 1 in the Table 7). However, the user may change the way in which values are assigned by setting the "open" density and temperature (integer) selectors `ne_select` and `te_select`. Since: (1) negative values of the temperature

Table 7: Open field-line value selection in `lpts`

`ne,te_select`	for $n_{e,open}$ and $T_{e,open}$...
0	Use `ne_edge` and `te_edge` (input parameters)
1	average of PTS data for $R \leq R_0 - a$ (i.e., "inner")
2	average of PTS data for $R \geq R_0 + a$ (i.e., "outer")
3	average of PTS data outside confined region (i.e., "average")

and density profiles may result from the polynomial fit, and (2) the fit will not generally match the measurements on the open field-lines, the `lpts` routine will, by default, blend the T_e and n_e profiles near the edge of the confined region to match the open field-line values, as determined by `te_select` and `ne_select`. The scale-length over which the profile modifications are made is specified by the user in array `blend(1:2)` [cm], where the 1st element specifies the scale-length on the inboard side and the 2nd element the scale-length on the outboard side. This scale length variable is initialized to `[10,10]`. If blending is not desired, set `blend=0` prior to executing `lpts`.

The quantities listed in Table 8 will be created by `lpts`.

Table 8: Quantities created by `lpts`

variable	*description*
`t_pts`	trigger time-point for PTS data [ms]
`n_pts`	number of data channels
`r_pts(n_pts)`	radial position for data [cm]
`ne_pts(n_pts)`	density measurements [cm^{-3}]
`te_pts(n_pts)`	temperature measurements [eV]
`r_exp(2*msrf-1)`	radial coordinate of smoothed data [cm]
`ne_exp(2*msrf-1)`	smoothed density at `r_exp` [cm^{-3}]
`te_exp(2*msrf-1)`	smoothed temperature at `r_exp` [eV]
`nesrf(msrf)`	smoothed and averaged $n_e(\tilde{\psi})$ [cm^{-3}]
`tesrf(msrf)`	smoothed and averaged $T_e(\tilde{\psi})$ [cm^{-3}]

In addition, some other global variables will be created, such as the relative errors `te_errL`, `te_errU`, etc., where `L` and `U` refer to lower and upper bounds.

5.4 Plot PTS data (`ppts`)

This function is called by the `lpts`, `mpts` and `apts` routines when the plot type is greater than 0, but it can also be called by the user directly after PTS data has been loaded.

Plot PTS data with...

```
call ppts(;plot_type)
```

plot_type=1 : plot one frame showing $n_e(R)$ and $T_e(R)$, both the measurements and smoothed profiles.

plot_type=2 : plot one frame showing $n_e(\tilde{\psi})$ and $T_e(\tilde{\psi})$, both the measurements and smoothed profiles.

plot_type=3 : display both plot types.

Figure 13 shows sample type 1 and type 2 output from the `ppts` (or `lpts`) routines.

5.5 Modify PTS data (`mpts`)

Function `mpts` displays existing PTS data on the screen and allows the user to modify the values of one or more n_e or T_e entries. This is useful when bogus values are contained in the `ptsfit` file.

Modify PTS data with...

`call mpts(;npoly,plot_type)`

`npoly` : integer order of the polynomial used to fit the PTS data (as in `lpts`).

`plot_type` : integer plot type, one of {0, 1, 2, 3} (as in `lpts`).

This routine allows the user to inspect each data element, stored in 1D arrays with names `ne_pts` and `te_pts`, and possibly modify the data interactively by specifying new values for one or more data elements and re-executing `mpts` to re-smooth with the new data; for example:

```
lpts(123)          # load PTS data for shot number 123
mpts               # display data table for user inspection
te_pts(3)=new_value # alter te entry for 3rd channel
mpts               # re-smooth using modified data
```

Use the `wpts` routine, described below, to save the modified data in a disk file for use in a future session.

5.6 Write (modified) PTS data to disk (`wpts`)

Function `wpts` writes the current set of PTS data—presumably modified by `mpts`—to a text file named `ptsfit_shot`. The purpose of this routine is to record changes made to the data by the user for possible re-use in a future session.

Write modified PTS data to disk with...

`call wpts()`

The file containing the modified data will be written into the current working directory irrespective of whether the original data came from a `ptsfit_shot` file in the current directory or one in the `SSPX_SHOTDATA` directory.

If a `ptsfit_shot` file already exists in the current directory, it will be re-named `ptsfit_shot.orig` to preserve original data as extracted from the SSPX database.

5.7 Analytic PTS data generator (`apts`)

Function `apts` generates n_e and T_e profile data as a function of radius, over the confined region, and fills the PTS data arrays using the analytic models:

$$n_e(r) = (n_{e,peak} - n_{e,edge})\sqrt{1-(r/a)^{\alpha_{n_e}}} + n_{e,edge}$$

$$T_e(r) = (T_{e,peak} - T_{e,edge})\sqrt{1-(r/a)^{\alpha_{T_e}}} + T_{e,edge}$$

The profile parameters are defined in global variables `alpha_te`, `te_peak`, `te_edge`, `alpha_ne`, `ne_peak` and `ne_edge`. The units of temperature are eV

and cm^{-3} for density. The number of data points, specified in variable `n_pts`, will be distributed from the outer edge to the inner edge (as in actual PTS data) but over only the confined region.

Generate analytic PTS data with...

`call apts(;npoly,plot_type)`

`npoly` : integer order of the polynomial used to fit the PTS data (as in `lpts`).

`plot_type` : integer plot type, one of {0, 1, 2, 3} (as in `lpts`).

The data will be smoothed and mapped to flux-surface-based arrays in the same manner as `lpts`, with parameters *`npoly`* and *`plot_type`* having the same effect.

5.8 Shot information (`shots`)

The `shots` routine displays information from the `shot_dates.pfb` database (described in Appendix E.2), which gets automatically loaded into each session. The database contains variable `shot_date`, a string array containing date entries of the form `"YYYYMMDD"`. The index into array `shot_date` is the shot number, so referencing `shot_date(`*`shot`*`)` will return the date for integer shot number *`shot`* (an entry set to `"????????"` indicates that the shot number was not used).

The `shots` command will display the entire list of SSPX shots, with the format

`YYYY/MM/DD` *`shot1-shotn n`*

where *`shot1`* represents the 1st shot number for the day listed, *`shotn`* the last shot for the day, and *`n`*, the number of shots for the day.

An optional shot number argument to `shots`, will start the listing from that shot number, e.g.:

```
corsica> shots(12000)
2003/09/16 12000-12042 43
2003/09/18 12043-12061 19
2003/09/19 12063-12111 49
2003/09/23 12112-12150 39
2003/09/24 12151-12191 41
2003/09/25 12192-12254 63
2003/10/07 12255-12320 66
2003/10/08 12321-12355 35
2003/10/09 12357-12395 39
```

The `shot_dates.pfb` database gets automatically updated nightly but from a *manually* maintained list (see Appendix E.2), so there may be some delay before the most recent shots appear.

5.9 Multiple shots (mksadb)

When data is available from several "identical" shots, one may construct a shot-averaged database, i.e., a hybrid *shot*-d4c.pfb file, with the mksadb ("make-shot-averaged-database") routine.

Make a shot-averaged database with...

call mksadb(*shot_list*;*n_points*)

shot_list : integer array of 2 or more shot numbers.

n_points : number of time-points to use (defaults to the number of points for the first shot in *shot_list*).

This routine requires that a *shot*-d4c.pfb file for each shot in *shot_list* exist in the current working directory. The output (arithmetic average of data in the individual shot data files) is written to a file named

shot1shotn-d4c.pfb

where *shot1* is the 1st shot in *shot_list* and *shotn* the last.

As an example, say we wish to combine the data from "identical" shots 123, 124, 126, ... 130 into a single database; we execute the following statements:

```
integer slist=[123,124,126,127,128,129,130]
mksadb(slist)
```

which will create a file named 123130-d4c.pfb. The averaged data may then be loaded into a session by referencing the hybrid shot number:

```
lsd(123130)
```

Note that averaging multiple shots provides inherent smoothing, so executing lsd with w_smooth=0 is usually acceptable.

Alternatively, the Basis iota(n:m) function, which returns the sequence vector [n,...m], and the Basis concatenation operator, //, may be used as follows

```
mksadb( iota(123,124)//iota(126,130) )
```

Prior to constructing a shot-averaged database, it is wise to compare the data for the individual shots to identify anomalous cases that one may wish to skip. The compare_shots routine makes a graphical comparison of shot data.

Compare data for multiple shots with...

call compare_shots(*shot_list*;*time_point*)

shot_list : integer array of 2 or more shot numbers.

time_point : a particular time-point [ms or μs] at which to make a detailed comparison.

Again, the individual shot data files must exist in the current working directory.

6 Equilibrium reconstruction

Once shot data has been loaded, the Corsica model profile parameters can be adjusted to match the measurements. This generally consists of adjusting a subset of the coefficients and/or exponents of the λ profile model and the parameter `plcm` which scales the toroidal current.

6.1 Fitting to B-probe measurements (`fit`)

Some of the parameters in Table 2 can be used as independent variables by the `fit` routine, which attempts to minimize the difference between the measured poloidal field and gun current with the Corsica model. This routine utilizes the general purpose nonlinear root-finder HYBRD [3], which is part of the `ceq` package in Corsica. Equilibrium fitting is performed with zero pressure. Finite pressure fits may be performed with the `pfit` (see §6.4) routine if PTS data is available, although fitting with zero-pressure is generally sufficient.

Sometimes a reasonable "fit" (the Corsica model can produce a similar poloidal field to say, within 5%) can be achieved, but in some cases the differences between the Corsica model and the measured values will be unacceptable. The ability to match the measured probe data with the Corsica model is exacerbated by (1) the adequacy of the λ-profile model built into Corsica, (2) uncertainties and/or errors in the measurements—which are sensitive to signal processing parameters, and (3) non-axisymmetry in the experiment which is beyond the scope of the 2D Corsica model.

The `fit` routine varies the toroidal current `plcm` and one of the profile form coefficients, a_i.

Fit an equilibrium to shot data with...

`call fit(;i_asph,i_asph2,plotit)`

`i_asph` : integer, represents the index of coefficient array `asph` to use as the independent variable, thus it can be 1, 2, 3 or 4. The default value is defined in global variable `index_asph`.

`i_asph2` : integer, represents the index of another `asph` member (must not have the same value as ***`i_asph`***). If ***`i_asph2`*** is non-zero, coefficient `asph(`***`i_asph2`***`)` will be adjusted to maintain $d\lambda/d\psi = 0$ at the edge of the confined region. The default value of ***`i_asph2`*** is defined in global variable `index_deriv`.

`plotit` : if non-zero, executes the `Publish` macro (see below) after successful completion [default value: 1].

The `fit` routine utilizes the constrained-equilibrium package (`ceq`) in order

to access the HYBRD nonlinear solver. If the fitting process fails, remember to return to the equilibrium package (`eq`) by executing

```
package eq
```

The `Publish` macro is predefined as follows:

```
mdef Publish=
  pparams; pause("pparams")
  Layout; pause("Layout")
  plvr(20); pause("plvr")
  pprobe; pause("pprobe")
mend
```

That is, it executes the four graphics routines `pparams`, `Layout`, `plvr` and `pprobe`. The `pparams` routine displays a page of parameters, macro `Layout` displays the overall configuration, function `plvr` shows $\lambda(R)$ and $q(R)$, and function `pprobe` plots the measured and model poloidal field at the probe locations.

The `pause` routine as used above causes the current frame displayed in an open graphics X-window to be held for a few seconds before displaying the next frame (otherwise the images disappear too fast for the eye). If there are no graphics windows open, the plots are sent to the `ncgm` file without delay. The user may change the pause time by invoking

```
pause(n)
```

prior to executing `Publish`, where n (≥ 0) is the number of seconds to hold the frame; $n = 0$ implies no pause. If the argument to `pause` is a character string, as in the above definition of macro `Publish`, the character string is displayed on the screen after the image has been displayed and the pause duration is unchanged.

The `Publish` macro can be redefined by the user to perform any sequence of commands after the `fit` routines finishes (if `fit` argument $plotit \neq 0$). To see the present definition, enter "`list Publish`". A simple way to automatically display only the quality of the fit would be to redefine the macro using the Basis `mdef/mend` syntax:

```
undefine Publish
mdef Publish=;pprobe;mend
```

If the `fit` routine completes successfully, indicated by the message "`converged`" from HYBRD, one may make a binary save-file by executing the `saveit` (see §3.4) routine to avoid having to re-fit the equilibrium in subsequent sessions.

As the HYBRD solver executes, progress will be displayed in the terminal screen—two lines per iteration. A typical successful execution history is given below.

```
corsica> fit

PROBLEM NO. 4  ceq  Bias coil currents from 12098-d4c.pfb

 ihy  =   0 nceq =   2 msrf = 101 lsrf =  -1 thetac =  0.0000
 vo   =     igun_err  berr_ave
 vo0  =     0.00E+00  0.00E+00
 vi   =     plcm      asph(4)
 x0   =     5.00E-01  0.00E+00
  1  1 axis(16, 25)= 3.235E+01,-1.793E+01 xpt(21, 43)= 4.196E+01, 2.395E+01 *
  1 igun_err= 1.1235d-01 ( 0.0000d+00) < 11.235%>     plcm    = 5.0000d-01
  2 berr_ave=-2.0365d-01 ( 0.0000d+00) <-20.365%>     asph(4) = 0.0000d+00

PROBLEM NO. 5  ceq  Bias coil currents from 12098-d4c.pfb

 ihy  =  99 nceq =   2 msrf = 101 lsrf =  -1 thetac =  0.0000
 vo   =     igun_err  berr_ave
 vo0  =     0.00E+00  0.00E+00
 vi   =     plcm      asph(4)
 x0   =     5.00E-01  0.00E+00
  3  1 axis(16, 25)= 3.235E+01,-1.793E+01 xpt(21, 43)= 4.196E+01, 2.395E+01 *
    1 call,  Fnorm= 2.33E-01 Fmax( 2)= 2.04E-01
 38  1 axis(16, 25)= 3.235E+01,-1.793E+01 xpt(21, 43)= 4.196E+01, 2.395E+01 *
    2 calls, Fnorm= 2.33E-01 Fmax( 2)= 2.04E-01
 41  1 axis(16, 25)= 3.235E+01,-1.793E+01 xpt(21, 43)= 4.196E+01, 2.395E+01 *
    3 calls, Fnorm= 2.33E-01 Fmax( 2)= 2.04E-01
 55  1 axis(16, 25)= 3.236E+01,-1.791E+01 xpt(21, 43)= 4.196E+01, 2.390E+01 *
    4 calls, Fnorm= 2.25E-01 Fmax( 2)= 1.97E-01
 59  1 axis(16, 25)= 3.238E+01,-1.789E+01 xpt(21, 43)= 4.195E+01, 2.380E+01 *
    5 calls, Fnorm= 2.11E-01 Fmax( 2)= 1.82E-01
 63  1 axis(16, 25)= 3.243E+01,-1.783E+01 xpt(21, 43)= 4.192E+01, 2.358E+01 *
    6 calls, Fnorm= 1.82E-01 Fmax( 2)= 1.54E-01
 69  1 axis(16, 25)= 3.252E+01,-1.770E+01 xpt(21, 42)= 4.187E+01, 2.312E+01 *
    7 calls, Fnorm= 1.25E-01 Fmax( 2)= 9.82E-02
 77  1 axis(16, 25)= 3.278E+01,-1.739E+01 xpt(21, 42)= 4.172E+01, 2.212E+01 *
    8 calls, Fnorm= 2.49E-02 Fmax( 1)= 2.39E-02
 67  1 axis(16, 25)= 3.282E+01,-1.740E+01 xpt(21, 42)= 4.172E+01, 2.231E+01 *
    9 calls, Fnorm= 1.37E-03 Fmax( 2)= 9.83E-04
 52  1 axis(16, 25)= 3.282E+01,-1.740E+01 xpt(21, 42)= 4.172E+01, 2.232E+01 *
   10 calls, Fnorm= 1.03E-05 Fmax( 1)= 9.47E-06
 26  1 axis(16, 25)= 3.282E+01,-1.740E+01 xpt(21, 42)= 4.172E+01, 2.232E+01 *
   11 calls, Fnorm= 3.08E-07 Fmax( 1)= 2.99E-07

HYBRD INFO = 1: converged
  1  1 axis(16, 25)= 3.282E+01,-1.740E+01 xpt(21, 42)= 4.172E+01, 2.232E+01 *
  12 calls, Fnorm= 3.12E-07 Fmax( 1)= 3.02E-07
  1 igun_err= 3.0181d-07 ( 0.0000d+00) <  0.000%>     plcm    = 3.5494d-01
  2 berr_ave= 7.7276d-08 ( 0.0000d+00) <  0.000%>     asph(4) =-1.5228d-01

rms error = 4.834 %
```

The HYBRD problem being solved in this example has two constraints, identified by the elements of the `vo` array (`berr_ave` and `igun_err`) where `vo0` is the desired value of the constraints, and two independent variables, identified by the elements of the `vi` array (`asph(4)` and `plcm`) where `x0` are the initial values.

After successful completion of the `fit` routine, the `Publish` macro is executed (unless the 3rd argument to `fit` is zero). Typical output from the `pprobe` routine is shown in Figure 14.

Each iteration consists of a function evaluation (i.e., an equilibrium calculation) followed by a HYBRD evaluation of either a Jacobian or the determination of

the next step. Two lines wil be displayed for each iteration, for example:

```
26 1 axis(16,25)=...
11 calls, Fnorm= 3.08E-07 Fmax( 2)= 2.99E-07
```

The GS solver reports the number of iterations, $n_{GS} = 26$ in the above example, required to solve the Grad-Shafranov equation for the present function evaluation, followed by the position of the magnetic axis and x-point. The most important quantity here is n_{GS}; if it exceeds the upper limit (defined in global variable `nl`), it means the GS solver is struggling and the solution does not satisfy the convergence criterion specified in Corsica variable `epsj`. This may result in eventual failure of the `fit` procedure. The HYBRD solver reports the current iteration number (e.g., `11 calls`, here), followed by the Euclidean-norm of the R_i residuals, `Fnorm` $= ||R||_2$:

$$||R||_2 = \left(\sum_{i=1}^{N_c} |R_i|^2 \right)^{1/2}$$

and `Fmax(i)` is the particular residual with the largest magnitude.

The `fit` routine solves $N_c = 2$ equations and the residuals are the relative errors between the measured and Corsica values of the gun current and poloidal field

$$R_1 = \mathcal{E}_{I_{gun}} \mapsto \texttt{igun\_err}$$
$$R_2 = \mathcal{E}_{B_{ave}} \mapsto \texttt{berr\_ave}$$

which are evaluated at each iteration with:

$$\mathcal{E}_{I_{gun}} = \frac{I^\star_{gun_{Corsica}} - I^\star_{gun_{meas}}}{I^\star_{gun_{meas}}}$$

$$\mathcal{E}_{B_{ave}} = \frac{1}{B^\star_{meas_{max}} \sum w(p)} \sum_{p=1}^{19} w(p) \left[B^\star_{Corsica}(p) - B^\star_{meas}(p) \right]$$

where the $\star$ superscript represents quantities evaluated at $t = t^\star$, $B^\star(p)$ the poloidal field at probe index p, and $w(p)$ the weight given to probe p.

In some cases the `fit` routine will fail, either with a fatal error such as a floating-point error, or a "`not making progress`" error from the HYBRD solver, or perhaps, the code keeps running but with little apparent progress (`Fnorm` not decreasing). The following subsections describe what to do.

6.1.1 When things go wrong—equilibrium failure

If the `fit` routine fails with a fatal error—the "`yuck`" message—it usually means that the equilibrium was corrupted due to an extreme value for a profile parameter. One can try the `recoup` command to restore the last equilibrium saved in memory, but if this fails the only recourse is to restore the most relevant save-file and try a different approach (e.g., different profile variables,

different time-point, etc.). The restore should be preceded by a `package eq` statement to return to the `eq` package from the `ceq` package, and followed by a `run` command to install the equilibrium, for example

```
package eq
restore "12098_1.800.sav"
run
```

Note that double-quotes are needed around filenames with base-names that combine numerals with underscores.

6.1.2 When things go wrong—HYBRD gives up

In some cases, HYBRD will stop with the message "`...not making good progress...`" or "`...fcns calls exceeded...`". If the residual `Fnorm` is small, however, the present state may indeed be acceptable as a solution, and it can be captured and saved by the commands:

```
package eq
run
saveit
```

6.1.3 When things go wrong—HYBRD doesn't stop

If HYBRD continues to iterate but with little reduction in `Fnorm` but it is acceptably small, one may interrupt in the Unix way (i.e., CTRL-C) and capture the current state:

```
^C
package eq
run
saveit
```

6.2 Fitting at multiple time-points (`mfit`)

The `mfit` routine will execute the `fit` function at a series of uniformly-spaced time-points from t_1 to t_2 with interval Δt.

Fit equilibria (by calling `fit`) at multiple time-points with...

`call mfit(;t1,t2,dt)`

`t1` : beginning time-point [μs].

`t2` : ending time-point [μs].

`dt` : time increment [μs], negative if $t_1 > t_2$.

The `mfit` routine assumes that shot data has been loaded at least once, i.e., the relevant shot number is known to `lsd`. A log file (*`shot`*`.mfitlog`) is written by `mfit`, containing a one-line summary for each time-point.

Two data files will be created upon successful completion of the `mfit` routine: binary file *`shot`*`-results.pfb` and text file *`shot`*`-results.dat`. The PFB file contains collected data as a function of time and the text file contains a subset written as a table, suitable for importation into a spreadsheet application.

Two graphics files will also have been created, with names like

`probname``.001.ncgm` and *`probname`*`.002.ncgm`

The 1st file will contain the `Publish` plots for each time-point. The 2nd file will contain summary plots of various parameters, plotted as a function of time. Figures 15 and 16 show a sample of summary plots from the `mfit` routine.

6.2.1 Fitting an entire pulse

It is recommended that logical variable `saveAll` be set to `true` prior to executing `mfit`, to automatically call `saveit` at the end of each call to `fit`, saving the equilibrium for each time-point. The `mfit` routine will usually fail near the end of a pulse (or near the beginning if $\Delta t < 0$). By saving each equilibrium to disk, one can re-issue the `mfit` routine and it will display a prompt asking if it is to use existing save-files. By answering "`yes`" to the prompt, one may salvage previous solutions (by restoring them) as opposed to re-executing the fit—this will save considerable time. In this way, the `fit` routine will only be executed if a save-file does not exist for a particular time-point.

For example, one could start `mfit` with t_1 somewhere in the middle of the sustainment phase where a solution is easily found, and, with $t_2 = 0$ and $\Delta t < 0$ let the function process time-points until it fails (at some yet-to-be-determined time). Then, restore the equilibrium at the 1st time-point—in the middle of the pulse—with the command

```
reload(t1)
```

and direct `mfit` to run with *positive* Δt and t_2 at the end of the pulse, which will eventually fail when the probe signals become so small that the equilibrium cannot be resolved—again, at some yet-to-be-determined time-point. An `mfit` session with $\Delta t = 50\,\mu$s, might look something like this

```
lsd(12098)
saveAll=true
mfit(2500,0,-50)
reload(2500)
mfit(2500,5000,50)
```

and let's say the 1st `mfit` execution failed at $t = 150$ so its last save is for $t = 200$, and the 2nd `mfit` execution failed at $t = 4200$, so its last save occurred at $t = 4150$.

After running `mfit` backwards and forwards in this manner, one will have a series of save-files that begin at the earliest feasible time-point for which an equilibrium could be found all the way to the maximum feasible time-point, in

this example, from $t = 200$ through $t = 4150$. Then, `mfit` can be re-executed with the now-known time-point limits, with each equilibrium being restored from disk instead of requiring a fit, as follows

```
mfit(200,4150,50)
```

Since this process will be successful (i.e., the equilibrium solver will not fail), at the completion of restoring the last time-point `mfit` will write selected results to the "`*-results.pfb`" and "`*-results.dat`" files and make summary plots.

6.2.2 Other `mfit` options

Auxiliary routines `mfit_accept`, `mfit_retry`, `mfit_skip` and `mfit_quit` may be invoked during an `mfit` session as a work-around for difficult cases when HYBRD stops or is perhaps interrupted with `^C`.

If the HYBRD solver does not converge but `Fnorm` is acceptably low, then enter `mfit_accept` and `mfit` will accept the present state as a valid solution and continue processing.

If `Fnorm` is relatively large, one might try changing a parameter and asking `mfit` to retry the case. For example, if λ profile exponent `nasp=2` and the solver fails, one may do something like:

```
nasp=10
mfit_retry
```

and `mfit` will try to find a different solution.

If none of the above work. one may simply skip the time-point by issuing the command: `mfit_skip`.

Finally, if `mfit` has successfully processed most time-points but fails near the end of the pulse (or near the end of the specified time range) one may issue the command `mfit_quit` to indicate that `mfit` should use what it has to make the summary plots and data files.

6.2.3 Fitting arbitrarily-spaced time-points

The `mfit_restore` routine is available to process previously saved equilibria. This routine simply restores specified save-files so one may work with *arbitrarily*-spaced time-points.

Process equilibria (by restoring) at arbitrary time-points with…

`mfit_restore(;regexp,time_increment)`

`regexp` : a regular expression used to select save-files in the current working directory [default: "*.sav"].

`time_increment` : if non-zero, specifies a time increment [μs] for which `Publish` is to be executed.

The regular expression can be used to select a sub-set of save-files in the current working directory. For example, if save-files for shot `12098` exist at 10 μs intervals from $t = 130$ to $t = 4200\,\mu$s, but we are only interested in the interval $t = 2000 \rightarrow 3990\,\mu$s, we could execute:

```
mfit_restore("12098_[23].*.sav")
```

On the other hand, if we are interested in the entire range (408 cases), but do not want to `Publish` each case, then we can specify a `Publish` time interval, say $500\,\mu$s, at which to make the plots with:

```
mfit_restore("12098*.sav",500)
```

which will `Publish` the first and last time-points and all other time-points that are a multiple of $500\,\mu$s.

The `mfit_restore` routine can also be used to patch together saved equilibria from individual `mfit` executions performed with different time increments. For example, say we wish to fit shot `12098` with a $10\,\mu$s interval from $t = 200$ to $t = 1000\,\mu$s, with an interval of $100\,\mu$s from $t = 1000 \rightarrow 3000$, and with a $25\,\mu$s interval from $t = 3000$ on. The following demonstrates how to do this:

```
mfit(200,1000,10)
mfit(1000,3000,100)
mfit(3000,4500,25)
mfit_restore("12098*.sav")
```

Note that the individual `mfit` executions could also be performed in separate sessions.

6.3 Profile parameter scans

The `fit` routine is only able to accommodate one profile parameter and the toroidal current, `plcm`. To enable the exploration of additional profile parameters, some scanning routines are available that allow one to scan one or more profile parameters and execute `fit` at each point in the scan.

Three scanning routines are available: (1) `scan` to vary a profile parameter over a specified range, (2) `xscan` to vary a profile parameter with specified increment until a minimum in the r.m.s. error is bracketed, and (3) `mscan` to vary up to four additional profile parameters, executing `xscan` for each case. These routines are described in the following subsections.

6.3.1 One-parameter scans (`scan`)

The fitting routine is only capable of varying one profile parameter, usually `asph(4)`. The `scan` routine can vary (scan) a 2nd profile parameter, executing `fit` at each step.

Perform a one-parameter profile scan with...

`call scan(vname,v1,v2,nv;plotit)`

vname : character string representing the name of a variable, which may be a member of an array, e.g., "`asph(1)`".

v1 : beginning value of *vname*.

v2 : end-point value of *vname*.

nv : number of points (if type integer) or increment, Δv, if type real.

plotit : integer plotting option: 1 (default) to plot r.m.s. error versus *vname* at the end of the scan; 2 update plot at each point in the scan.

Figure 17 shows sample output for

```
scan("asph(1)",-0.5,0.5,0.1)
```

for a case where `asph(4)` is used as the independent variable in `fit` (with exponent `nasp=2`). Notice that the the range 0.5 to 0.5 does not bracket a minimum of the r.m.s. error, and since the 4th argument is type real, the scan interval $\Delta a_1 = 0.1$. The following will perform a similar scan but with 6 points:

```
scan("asph(1)",-0.5,0.5,6)
```

with $\Delta a_1 = 0.2$.

6.3.2 Minimization scans (`xscan`)

The `xscan` routine does a minimization scan, executing `fit` at each point, by varying a profile parameter until a minimum in the r.m.s. error is bracketed.

Perform a one-parameter minimization scan with...

`call xscan(;delta_x,plotit)`

delta_x : the initial step-size for the scan, which may change sign or magnitude as the scan proceeds. The name of the variable to scan is defined in variable `xname`.

plotit : integer plotting option: 0 to make no plot, 1 to plot r.m.s. error versus *vname* at the end of the scan; 2 update plot at each point in the scan.

The `xscan` routine executes `Publish` at the nearest scan point to the minimum (if ***plotit*** > 0).

Figure 18 shows sample output from

```
xname="asph(1)"
xscan(0.1,1)
```

which picks up where the previous `scan` example left off. Note that this scan finds a minimum in the r.m.s. error for $a_1 \simeq 4$.

6.3.3 Multi-parameter scans (mscan)

The mscan routine facilitates multiple parameter scans of selected parameters, by executing the minimization scan (xscan) routine on up to four profiles parameters: gsph, bsph(4), nasp and nbsp.

Perform a multiple-parameter scans with...

call mscan(*;dgsph,dbsph,dnasp,dnbsp*)

dgsph : initial increment for gsph (or 0, the default, to hold fixed).

dbsph : initial increment for bsph(4) (or 0, the default, to hold fixed).

dnasp : initial increment for nasp (or 0, the default, to hold fixed).

dnbsp : initial increment for nbsp (or 0, the default, to hold fixed).

This routine varies each of the four profile parameters, one at a time unless its increment is specified as zero, finding an approximate minimum of the r.m.s. error. A plot of the xscan progress is displayed for each variable, and after the four scans are complete, the Publish macro is executed. The above process is then repeated continually, until the error reduction is insignificant. Variables gsph_bounds(1:2), etc. contain the lower and upper bounds for the four independent variables.

As an example, the following

```
mscan(1,0,1,0)
```

will vary gsph and nasp (both in increments of 1) until the r.m.s. error can no longer be reduced. It produces a log of the progress as listed below, and will also save the final solution in a file name of the form: *shot_time*_mscan.sav.

```
SSPX #12098 @ 1.600 ms [030919]
scan    gsph bsph(4)    nasp    nbsp asph(4)    plcm   Error     Fnorm
              fixed           fixed
 000  ~0.000   0.000   2.000   6.000  -0.152   0.355   4.834  8.40e-09
 001  22.000   0.000  20.000   6.000  -0.070   0.339   4.325  7.52e-07
 002  15.000   0.000  20.000   6.000  -0.072   0.341   4.321  9.60e-09
 003  15.000   0.000  20.000   6.000  -0.072   0.341   4.321  1.05e-08
mscan converged
```

Figure 19 shows the xscan progress for each of the 3 (in this case) passes, each pass varying the 2 quantities.

6.4 Finite pressure fits (pfit)

The pfit routine facilitates a crude fit to the measured (from PTS data) pressure by adjusting the peak pressure, p_0, to match that from the measurement. The pfit routine automatically calls lpts to load the PTS data. The pressure

profile is modelled with the following form:

$$p(\tilde{\psi}) = p_0 \left(1 - \tilde{\psi}^{b_p}\right)^{a_p}$$

where exponents a_p and b_p may be specified by the user.

Perform finite pressure fits with...

```
call pfit(;a_p,b_p)
```

a_p : exponent a_p, if zero, evaluate from PTS data [default: 0].

b_p : exponent b_p [default: 1].

If $a_p = 0$ (the default), the value will be estimated from the PTS data and placed in code variable `alfa(1)` and `betp(1)` will contain argument b_p.

After executing `pfit`, the measured and Corsica pressure profiles may be graphically displayed by executing `ppts(2)` (see §5.4). The volume averaged electron beta will be returned in code variable `betap(1)`.

It is generally not necessary to perform finite pressure fits, as the difference between the zero-beta and finite-beta spheromak equilibria is negligible. However, `pfit` is useful to evaluate the electron beta and the resulting finite-pressure equilibrium can be used as input for stability analyses (see §7).

6.5 Batch mode fitting (`bfit`)

A special fitting routine, `bfit`, is available to perform equilibrium reconstruction, including finite-pressure, with `caltrans` executed in a batch-like (non-interactive) way.

Perform equilibrium reconstruction in batch-mode...

```
caltrans -probname shot bfit.sav sspx.bas -exec "bfit(n)"
```

Note the problem name *must* be the SSPX shot number.

The `bfit` routine assumes both the shot data and PTS data are available. On the `caltrans` command-line, the `probname` string must identify the shot number. The special save-file name `bfit.sav` is used here, if it exists from a prior execution of `bfit` (otherwise, use an appropriate generic save-file). The `-exec` directive on the command-line instructs `caltrans` to execute the `bfit` routine, which will: (1) execute `lsd` with the time-point taken from the PTS data, (2) perform a zero-pressure fit to the probe data with `fit`, and (3) perform a finite pressure fit, using the `pfit` routine. The argument *n* to `bfit` is an integer number of seconds to display graphics frames on the screen (use *n=0* to only send graphics to the NCGM file).

If all is successful, two save-files will be created:

shot_time.sav and *shot_time*_beta.sav

Error trapping is turned off, so if an error occurs, the caltrans process will be terminated. A log file, bfit.log, will contain a one-line summary for each bfit execution.

7 Stability analyses

Once SSPX equilibria are available, stability analyses may be performed using routines in Corsica or its dcn package (DCON), as described below. The stability routines introduced below are part of the Corsica distribution—they are not defined in the SSPX scripts.

7.1 Inverse equilibria

Stability routines usually require an *inverse* equilibrium be available, which in turn requires that the boundary of the confined plasma region does not contain an X-point. If the configuration is limited by the wall, then it may be used as-is to create an inverse equilibrium. If the configuration is X-point limited, however, then specify an interior flux surface at which to truncate the confined region a distance θ_c (in normalized flux) from the edge, and compute a new free-boundary equilibrium, for example:

```
thetac=0.01; run
```

The inverse equilibrium is then constructed from the free-boundary solution with

```
start_inv
```

The convergence criterion for script function start_inv is specified with variable epsrk and its iteration limit with nht. Use the Basis list command on the underlying compiled inverse solver routine, teqinv, to see what options are available for the solver.

7.2 Mercier limit

If the equilibrium has finite pressure, the Mercier criterion can be evaluated with

```
mperd=25; balloon
```

where mperd specifies the number of periods used by the balloon routine, which evaluates ballooning and Mercier stability criteria on the confined flux surfaces. Use the plot_ball routine to graphically display the stability integral as a function of flux.

To adjust the pressure profile for marginal Mercier stability, the marg_merc script routine can be executed. The marginally stable value of β_p will be contained in betap(1).

7.3 Using DCON

The DCON package in Corsica may be used; as of this writing, however, it does not accept current in the open field-lines, so one must prepare an equilibrium with $\lambda_e = 0$. The conducting shell must also be mapped to the DCON wall model using the `sewall` ("smoothed experimental wall") routine in script file `sewall.bas`. The procedure is essentially

```
read sewall.bas
sewall
dcon
```

but in the case of SSPX, the DCON wall model may need to be adjusted. Execute `sewall("help")` for details on preparing the DCON wall model.

8 Ohmic power analyses

As described in Section 5, data from the PTS diagnostic can be loaded into a session with the `lpts` function, questionable data points can be modified with `mpts`, or simulated PTS data can be generated from an analytic profile model with the `apts` routine. Table 9 contains a summary of the ohmic power analysis functions and macros, which includes the PTS data routines introduced in Section 5.

Table 9: Summary of ohmic power routines

`lpts`	load SSPX PTS data (n_e, T_e as a function of radius)
`mpts`	modify PTS data previously loaded with `lpts`
`wpts`	write `ptsfit_shot` file using results of `mpts`
`apts`	construct analytic PTS data
`ppts`	plot PTS data
`pohmic`	evaluate ohmic power quantities on confined flux surfaces
`p1d`	plot 1D (flux-surface) quantities
`wtaue`	write `pohmic` results to disk
`pohmic2d`	evaluate ohmic power quantities on RZ grid
`tauW`	evaluate magnetic energy decay time
`tauK`	evaluate helicity decay time
`p2d`	plot 2D (RZ mesh) quantities as contours
`p2dvr`	plot 2D quantities as a function of radius

Once n_e and T_e data are available via `lpts`, `mpts` or `apts`, ohmic power dissipation on the RZ grid (both in the confined region and in the external, open field-line region) can be evaluated by executing the `pohmic2d` function. Power dissipation, $\tau_E(\psi)$ and $\chi_E(\psi)$ in the confined region may be evaluated, using flux-surface-averaged quantities, with the `pohmic` function. Additional routines are available to evaluate the magnetic energy decay time, τ_W, and the helicity decay time, τ_K.

The results of the `pohmic2d` evaluation of 2D quantities may be displayed

as contour plots with the `p2d` macro and as a function of radius with `p2dvr`. Quantities evaluated with `pohmic`, on confined flux surfaces, can be viewed with macro `p1d`, and they can be written to disk with function `wtaue`.

Detailed descriptions of the ohmic power routines are given in the sections below.

8.1 Ohmic power quantities on confined flux surfaces (`pohmic`)

Function `pohmic` uses PTS data to evaluate the ohmic power dissipation on each confined flux surface, returned in array `powdensrf`, and the integrated total power confined within each flux surface, array `powersrf`. The energy confinement time $\tau_E(\psi)$ and thermal diffusivity $\chi_E(\psi)$ are also evaluated. All of these quantities are arrays of length `msrf`, the number of flux surfaces.

Perform ohmic power analyses on confined flux surfaces with...

```
real power=pohmic(;plotit)
```

power : integrated ohmic power dissipation [W] over the confined region.

plotit : if non-zero, display four plots of $\tau_E(r/a)$, $\chi_E(r/a)$, $P/V_E(r/a)$, and $P(r/a)$ in one frame [default: 0].

Surface quantities evaluated by `pohmic` can be graphed individually with the `p1d` macro. For example, `p1d(tauE)` will plot, in the same frame, the energy confinement time versus normalized poloidal flux and versus r/a.

The relevant `pohmic` variables accessible to the user are shown in Table 10.

Table 10: Quantities evaluated by `pohmic`

`nesrf`	cm^{-3}	surface-averaged electron density
`tesrf`	eV	surface-averaged electron temperature
`zeff`	—	effective ionic charge
`tauE`	s	energy confinement time
`chiE`	m^2/s	thermal diffusivity
`powdensrf`	W/m^3	ohmic power density
`powersrf`	W	integrated power

The effective charge, Z_{eff}, is specified in script variable `zeff`, which is a flux surface quantity (`msrf` values). It is initialized to a uniform value of 2.3 for SSPX, but may be changed by the user prior to executing `pohmic`.

8.2 Write results of pohmic to disk (`wtaue`)

The function `wtaue` writes the results of a `pohmic` execution to a text file, suitable for importing into spreadsheet applications.

Write pohmic output to disk with...

```
call wtaue(;filename)
```

filename : the name of the output file, which defaults to *probname*_taue.txt.

The output file contains the values of r/a, n_e, T_e, τ_E, χ_E, P/V and P at each msrf flux surface.

8.3 Ohmic power quantities on R-Z grid (pohmic2d)

Function pohmic2d uses PTS data (either from lpts or created by apts) to populate the equilibrium R-Z grid with n_e and T_e. It then evaluates the total ohmic power dissipation and dissipation in the confined and open field-line regions, as well as several other quantities.

Evaluate ohmic power quantities on RZ grid with...

```
real power(3)=pohmic2d()
```

power : vector of length 3 containing the (1) total, (2) confined and (3) open field-line ohmic dissipation [W].

The 2nd return value from pohmic2d (ohmic power in the confined region) should agree with the flux-surface-based result from pohmic.

The edge value of Z_{eff} (zeff(msrf)) is applied uniformly to the external, open field-line region; Z_{eff} in the confined region may be non-uniform, if desired.

The grid quantities computed or used by function pohmic2d can be displayed as contour plots in R-Z space. The pohmic2d variables and related quantities are listed in Table 11.

The macro p2d can be used to plot these 2D quantities, by supplying the macro with the variable name, e.g., p2d(lambda2d) or equivalently p2d(lambda), i.e., the "2d" suffix need not be supplied. The user may specify the number of contour levels by setting global variable p2d_levels. The plot range may also be modified through global variables p2d_rmin, p2d_rmax, p2d_zmin and p2d_zmax [m].

Another macro, p2dvr, can be used to plot radial scans of the 2D arrays at the elevation of the magnetic axis. It is invoked in the same manner as the contour plotter, e.g., p2dvr(lambda).

8.4 Energy and helicity decay times (tauW and tauK)

There are two functions available to evaluate the magnetic energy and helicity decay times, τ_W and τ_K, given by the following expressions developed by

Table 11: Quantities evaluated by `pohmic2d`

`bmod2d`	T	total magnetic field
`bp2d`	T	poloidal magnetic field
`br2d`	T	radial magnetic field
`bt2d`	T	toroidal magnetic field
`bz2d`	T	axial magnetic field
`eta2d`	Ohm-m	Spitzer resistivity
`jpar2d`	A/m^2	parallel current density
`jtor2d`	A/m^2	toroidal current density
`lambda2d`	m^{-1}	lambda
`ne2d`	m^{-3}	electron density
`powden2d`	W/m^3	ohmic power density
`pressure2d`	Pa	plasma pressure
`psi2d`	Wb	poloidal flux
`te2d`	eV	electron temperature

Bick Hooper:

$$\tau_W = \frac{\mu_0 \int B^2 dV}{2 \int \eta \lambda^2 B^2 dV} \quad \text{and} \quad \tau_K = \frac{2\mu_0 \int \psi \frac{B_\varphi}{R} dV}{\int \eta \lambda B^2 dV}$$

where the total and toroidal magnetic field (B and B_φ); resistivity, η; eigenvalue (λ); and stream function, ψ, are all evaluated on the R-Z grid.

Evaluate the magnetic energy and helicity decay times with...

```
real value=tauW(;region,ne,te)
real value=tauK(;region,ne,te)
```

value : the return value of these functions is the decay time [s].

region : string, one of {`"total"`, `"confined"`, `"open"`}, specifying the region over which to evaluate the decay time integral [default: `"total"`].

ne : uniform density [m^{-3}], if PTS data not available.

te : uniform temperature [eV], if PTS data not available.

If PTS data has been loaded and `pohmic2d` executed, the resistivity $\eta = \eta(R, Z)$ will be available in script variable `eta2d`. Otherwise, one must specify average electron density and temperature through arguments `ne` and `te`.

References

[1] James A. Crotinger, et al., *CORSICA: A Comprehensive Simulation of Toroidal Magnetic-Fusion Devices*, Report UCRL-ID-126284, Lawrence Livermore National Laboratory, CA, April, 1997.

[2] E. B. Hooper, L. D. Pearlstein and R. H. Bulmer, *MHD equilibria in a spheromak sustained by coaxial helicity injection*, Nucl. Fusion **39** (1999) 863.

[3] M. J. D. Powell, *A hybrid method for nonlinear equations*. In P. Rabinowitz, editor, "Numerical Methods for Nonlinear Algebraic Equations", pages 87-114, London, 1970. Gordon and Breach.

Appendices

A Using Corsica (CalTrans)

The Corsica code is presently embodied in the `caltrans` executable installed on the Energy and Environment Directorate (EED) Solaris[7] and the PAT/MFE Linux Cluster[8] file systems. It is actually a "distribution" of compiled code, standard script files and a shell-script wrapper which is the user interface. The compiled executable and collection of associated script files (and, all the source code) are referred to as the "CalTrans (or Corsica) distribution".

The CalTrans web page may be accessed from the LLNL Fusion Energy Program web page[9]. The `caltrans` code uses the Basis system as the user interface for text and binary file I/O. The Basis system also provides an interface to the NCAR[10] graphics library. New users are encouraged to read the documentation, especially the tutorial, available from the Basis web page[11] or its link on the CalTrans web page.

The `caltrans` code consists of several packages, most compiled with Fortran but others compiled with C or C++. There are also many standard script files (in addition to the SSPX scripts described in this document) that are part of the distribution.

A.1 CalTrans distribution

There are three versions of `caltrans` available for public usage, named, for historical reasons, `pcaltrans`, `ncaltrans` and `vcaltrans` where `p`, `n` and `v` stand for "production", "new" and "volatile", with `pcaltrans` being the oldest (most stable) version and `vcaltrans` being the newest (least stable). Most users use `vcaltrans`, however, as it has the most features. If there is a problem after these public versions have been updated, use one of the older, more stable, versions. Refer to Appendix D for information on the `caltrans` source files, including modification of the SSPX scripts.

The executable `caltrans` is actually a shell-script wrapper that sets several environment variables before executing the compiled code. One of the environment variables set by the `caltrans` wrapper is `CORSICA_SCRIPTS`, which points to the directory where standard scripts are kept, including the SSPX scripts (see App. D.2). Another environment variable, `CORSICA_PFB`, points to the directory where standard save-files and auxiliary binary files (in Basis PFB

[7] Contact an Energy and Environment Unix computer support person by following the link "Computer Support" at `http://eed-r.llnl.gov/`.

[8] Contact Bill Meyer for an account on the MFE Linux Cluster. Cluster documentation is available at `http://hrothgar.llnl.gov/`.

[9] CalTrans web page: `http://www.mfescience.org/`), under the link "Caltrans".

[10] National Center for Atmospheric Research (NCAR-Graphics) web page: `http://ngwww.ucar.edu/`.

[11] Basis web page: `http://basis.llnl.gov/`.

format) are stored (see Appendix E). The wrapper script ensures that the script files and binary files are consistent with the requested version of the compiled executable.

The three `caltrans` versions are installed in the following locations.

File system	*Wrapper-script: user interface to executable*
EED Solaris	`/mfe/theory/Caltrans/pcaltrans/bin/caltrans`
	`/mfe/theory/Caltrans/ncaltrans/bin/caltrans`
	`/mfe/theory/Caltrans/vcaltrans/bin/caltrans`
MFE Linux	`/usr/local/Caltrans/pcaltrans/bin/caltrans`
	`/usr/local/Caltrans/ncaltrans/bin/caltrans`
	`/usr/local/Caltrans/vcaltrans/bin/caltrans`

The following subsection describes how to access these code versions.

A.2 How to set-up your environment to use Corsica

The user may execute the code by explicitly entering its full pathname from the above table at the Unix prompt, but it is most commonly executed via a Unix alias or by adding the wrapper directory to your Unix command search path, as described below. Refer to the "Unix Configuration for Using Basis/Corsica/Onetwo" link on the CalTrans web-page for additional information.

One way to access the code is via a Unix alias, the syntax of which depends on your Unix shell, for example, for the C-shell and its variants, use something like

```
alias caltrans /mfe/theory/Caltrans/vcaltrans/bin/caltrans
```

and for Bourne-shell derivatives (e.g., `bash`), the syntax

```
alias caltrans=/mfe/theory/Caltrans/vcaltrans/bin/caltrans
```

associates the name `caltrans` with the `caltrans` wrapper shell-script in the `vcaltrans` distribution. Of course you may choose any name for the alias, and you might want one for each version of the code, such as (in C-shell syntax):

```
alias vc /mfe/theory/Caltrans/vcaltrans/bin/caltrans
alias nc /mfe/theory/Caltrans/ncaltrans/bin/caltrans
alias pc /mfe/theory/Caltrans/pcaltrans/bin/caltrans
```

Another approach is to include a `caltrans/bin` directory in your Unix search path. This is usually done through an (arbitrarily named) environment variable, such as `CALTRANS_ROOT`. The syntax for C-shells and its variants is

```
setenv CALTRANS_ROOT /mfe/theory/Caltrans/vcaltrans
set path = ($CALTRANS_ROOT/bin $path)
```

and for Bourne-shell derivatives

```
export CALTRANS_ROOT=/mfe/theory/Caltrans/vcaltrans
PATH=$CALTRANS_ROOT/bin:$PATH
```

When set-up through the path in this fashion, Unix will find the file `caltrans` in the wrapper directory `$CALTRANS_ROOT/bin` when "`caltrans`" is entered at the shell-prompt.

In order to use the the NCAR routines (`ctrans`, `idt`, etc.; see App. A.6) to post-process NCGM files from `caltrans`, you will also need to include their location in your Unix search path, as there are too many NCAR utilities to use the alias approach. Start by defining an environment variable, like `NCARG_ROOT`. The syntax for C-shells and its variants is

```
setenv NCARG_ROOT /usr/local/ncarg
set path = ($NCARG_ROOT/bin $path)
```

and for Bourne-shell derivatives

```
export NCARG_ROOT=/usr/local/ncarg
PATH=$NCARG_ROOT/bin:$PATH
```

The `man`-page documentation for the NCAR graphics routines are located under `$NCARG_ROOT/man`.

Put the above alias and path definitions (for both Corsica and the NCAR routines) in your appropriate shell start-up scripts, so they are always defined when you login.

Finally, the auxiliary shell-scripts (`biasflux`, `d4c`, `ncgm2pdf`) for SSPX applications are located in directory `/sspx/bin`. This directory should also be added to your Unix search path.

A.3 Starting a Corsica session

Having set-up your environment as described above—we'll assume here that the name `caltrans` is either an alias to the desired wrapper script or the name will be found in your Unix search path—launch the code as follows

Start-up with an SSPX equilibrium save-file with...

```
caltrans -probname pname sspx.sav sspx.bas
```

which instructs `caltrans` to: (1) use the string *`pname`* to name output (NCGM graphics and session log) files, (2) load an SSPX equilibrium binary save-file (`sspx.sav` in this case) and "execute" the equilibrium calculation with the information from the file, and (3) read the SSPX script file `sspx.bas`, compiling its function definitions.

The *`pname`* string is used to create NCGM graphics files with names of the form *`pname`*`.nnn.ncgm`, graphics log files with names *`pname`*`.nnn.cgmlog`,

where nnn is a sequence number, and the session log file: *pname*.log. If Basis detects errors during a session, error message files with names of the form *pname.nnn*.err will also be created.

One other file is always created at the start of each session:

```
.corsica-command-line
```

It is a "hidden" file[12], and contains an explicit listing of the command-line used to launch the code, assignments of certain code variables and the `read` statements of several standard script files that are read into every `caltrans` session.

The prompt string "`corsica> `" will be displayed and the code will be ready for user input to the equilibrium package. The script file may also be read into the session using the Basis `read` statement at the Corsica prompt:

```
read sspx.bas
```

If the prompt string changes for some reason, return to the equilibrium package by typing "`corsica`" or "`package eq`".

A.4 Session termination

An interactive session ends when the user enters `quit` or `end` (or the Unix disconnect signal: `^D`). If a batch job is being processed (see App. B.1), be sure to include a termination command in your script. The function `quit` accepts an integer argument that is passed as the exit status, so you can call `quit(1)` to signal an error exit to a controlling process—the default exit status is zero.

A.5 SSPX Environment Variables

If the user defines environment variable *SSPX_SHOTDATA*, the SSPX scripts will add the directory identified in this environment variable to its search path. In this way, one can store the SSPX data files (like shot data in *shot*`-d4c.pfb` or the PTS data in `ptsfit_`*shot*) in a common area which will be accessed if `caltrans` is executed in another directory. Then, only one copy of these data files need be retained.

If PTS data files are located in a different directory than in the directory named in SSPX_SHOTDATA, use environment variable SSPX_PTSDATA to identify that directory.

A.6 Graphics Post-Processing

Graphical output from Corsica sessions is always sent to NCAR computer-graphics meta-files (with suffix `ncgm`). The user may optionally direct output

[12] By Unix convention, hidden files begin with "`.`" and are not listed by the `ls` command unless the `-a` option is invoked or the file is explicitly named on the `ls` command-line.

to an X-window (see App. B.3) or to a PostScript™ file as described in the Basis documentation. This section describes a few ways to process ncgm files using NCAR utility routines[13].

A.6.1 Translate NCGM files (ctrans)

The NCAR ctrans command translates ncgm files to other file formats. The most common usage is to translate a file and send the output to a PostScript printer or to a file:

```
ctrans -d ps.mono file.ncgm | lpr
ctrans -d ps.color file.ncgm | lpr -P color-printer
ctrans -d ps.color file.ncgm > file.ps
```

The above examples demonstrate: (1) piping the output directly to the default printer, (2) piping the output to the named color printer, and (3) saving the output to a file. See the ctrans man-page for details.

A.6.2 Display NCGM files (idt)

The NCAR idt command provides a GUI interface to the NCAR ictrans utility. It may be used to view ncgm files and select individual frames for future processing. Execute it with an ncgm file name on the command-line:

```
idt some.ncgm &
```

Then point-and-click to find frames of interest, which can be saved in new ncgm files. To process images for documents, select and save *one* frame from an ncgm file, convert it to PostScript with ctrans, them import the file into a PostScript editor, such as Adobe Illustrator™. See the idt man-page for details.

A.6.3 Convert NCGM to PDF (ncgm2pdf)

An NCGM to PDF translator is available to extract selected frames from an NCGM file and create a PDF file. To select all but the first frame of a file (which usually contains unwanted Basis version information) and put 4 frames-per-page into a PDF file, do something like

```
ncgm2pdf -f 2- -nup 4 file.ncgm
```

Execute

```
ncgm2pdf -h
```

for usage information.

B Prerequisites for Basis codes

This section provides some general information regarding the use of so-called "Basis" codes, i.e., applications built with the Basis system which provides

[13]There are many NCAR utility routines available. The ncargintro man-page provides an overview.

the command-line user interface (parser), text and binary file I/O, graphics commands and the interface to the NCAR graphics library, and general mathematical routines.

New users are strongly encouraged to read: (1) the *tutorial*, (2) the *language reference manual*, and (3) the *EZN graphics manual*, all available from the Basis web page.

B.1 Batch-like operation

Basis-codes are typically executed in an interactive way but when repetitive or complicated commands are required it is best to put these into a text file using your favorite editor[14]. There are no restrictions on the file name, but `.bas` is a typical extension. The file can be read into the session with the Basis `read` statement or simply placed at the end of the `caltrans` command-line. Note the entire session can be executed in a batch-like way, with the last executing statement either `quit` or `quit(1)`. Function call `quit(1)` can be invoked within error testing code-blocks as a robust way to abort a run.

Execute with file of commands in batch-like mode...

```
caltrans -probname pname name.sav sspx.bas batch_job.bas
```

A common mode of operation is to build up a script file by testing parts of it in interactive sessions; when you are satisfied that the script does the intended task, use it in a batch-like way.

A simple Bourne shell script to process a `caltrans` batch job that, say, creates a text file named `batch_job.dat`, looks like this:

```
#!/bin/sh
if caltrans name.sav sspx.bas batch_job.bas
then
  lpr batch_job.dat
fi
```

B.2 Session files

Files created automatically include the session log-file, *pname*`.log`, containing user–code dialog, and NCAR graphics meta-files (see App. B.3), with names of the form *pname*`.`*nnn*`.ncgm`, where *nnn* is a sequence number. A graphics log file with a `.cgmlog` suffix will also be created for each `ncgm` file.

If Basis detects an error, then a *pname*`.`*nnn*`.err` file will be created. The error files may contain helpful information if the Basis debug-mode has been enabled (`debug=yes`), useful if you are developing a script of your own.

[14] Emacs users may be interested in the lisp code `basis.el` which aids the preparation of Basis and/or `caltrans` scripts.

B.3 Viewing and post-processing graphics

Graphical output is, by default, sent by `caltrans` to the `ncgm` file. The user may additionally direct output to other files (see the Basis EZN document) or, more commonly, to an X-window for viewing during the session.

Open and close graphical X-windows with the Basis `win` command; the window title may be optionally named:

```
win
win close
win on name
win close name
```

More than one graphical X-window may be opened during a session. There are also corresponding `caltrans` macros `ow(name)` and `cw(name)` that provide a similar function.

The Basis EZN graphics document describes the plotting facilities for Basis codes. The most commonly used command to graph a function $y(x)$ is

```
plot y
```

which will plot y as a function of its indices, $i = 1, 2, \ldots, n$; to plot y *versus* x:

```
plot y,x
```

There are plot options to specify color, line thickness, style, etc., for example:

```
plot y,x color=red thick=3 style=dotted
```

Multiple plots may be labelled:

```
plot y1,x label="Y1"
plot y2,x label="Y2"
```

Refer to the EZN document for details.

The NCAR command `idt` can be used after a session to view `ncgm` files, select specific frames for exportation, or for printing specific frames[15]. The `ctrans` command translates `ncgm` files to many other formats, including PostScript.

Appendix A.6 describes how to post-process NCGM files in more detail.

B.4 Built-in documentation

Basis has built-in documentation allowing the user to query the code. The top-level command is `help`, which introduces the `version`, `news` and `list` commands. The most useful is `list`—invoke it without an argument to get

[15]The man-page for `ncarv_spool` describes how to customize the post processing buttons for `idt` to send frames directly to a printer.

its own documentation, then invoke it with "`list` *`name`*" to display documentation about identifier *`name`*, e.g. "`list probname`". All user accessible variables, functions (and subroutines) and macros respond to the `list` command. To get the *contents* of a variable, just enter its name.

The "`list packages`" statement is useful to get a list of the `caltrans` packages. Packages are identified, for historical reasons, with 2–3 character names. Since it may be critically necessary to reference a specific package variable, it is important to be familiar with their package names (see App. B.5).

B.5 Basis language features

The Basis language is Fortran 90-like and has the familiar constructs: `while-endwhile`, `if-elseif-else-endif`, `do-enddo`, etc. Functions can be defined with `function` *`name`*`(); <`*`body`*`>; endf` as well as macros. Multiple lines can be placed on the same physical line by separating them with semicolons, and lines can be continued by placing a backslash at the end of a line. The comment character is #. A large set of Basis built-in mathematical routines are available, as well as many other useful tools for file I/O, string processing, matrix and array processing, etc. They are all documented in the Basis reference manual.

Variables can be created (and destroyed) on-the-fly. A common task is to set up a loop to do a parameter scan, execute some `caltrans` routine within the loop (like an equilibrium solver) and capture results in user-created variable arrays. When the loop has finished, the results can be be saved to disk or graphed with the Basis plot routines. All Basis variables must be typed (`integer`, `real`, `character`, `logical` are some of the types available). There is also a `chameleon` type that, in an assignment statement, takes on the type of the r.h.s. Basis predefines the identifiers `$a`–`$z` as chameleon variables—they are commonly used in interactive sessions to avoid having to declare scratch variables.

Identifier name conflicts are resolved by qualification with the defining package name, using dot-notation. For example, variable name `r` is popular, appearing in several packages as a user-accessible quantity. You can even make your own at the `caltrans` prompt by issuing a command like "`real r`". Running the `list` command on `r` will show all package occurrences, where user-created identifiers are assigned to package `global` (unless otherwise directed). The `list` command will show the packages in which an identifier appears and its *priority*. In case of conflicts, the identifier with the highest priority is inferred. To explicitly reference an identifier preface its name with its package name; for example, the equilibrium package definition of `r` is `eq.r`, the DCON (a package in Corsica) definition is `dcn.r`, etc.

B.6 Reading script files

Script files are read into the code with the Basis "`read` *`filename`*" statement. User script files will typically be in the current working directory, but Basis has a search path of directories in which to search for files. The search path is customized by `caltrans` to include directories where standard script files are kept, and directories where binary save-files are kept. The search path is stored in Basis variable `path`[16]; to see the search path, just enter "`path`".

If you have Basis or `caltrans` scripts of your own that you would like to be able to read into any session, put them in some particular directory and add the directory name to the Basis search path with a `pathadd("dirname")` statement. Statements such as a user's `pathadd` calls are good candidates to go into a personal Basis start-up file (`$HOME/.basis`) that will get read automatically each time `caltrans` is launched. The user's current working directory is in the default search path. The `caltrans` code will also read files named `.caltrans` and/or `.corsica`, it they exist in your Basis search path.

Files interpreted with the `read` statement are processed line-by-line, with line echoing to the terminal window turned off bny default. When debugging user scripts it is often useful to turn on line echoing (to both the screen and the log-file) with an `echo=yes` statement. It can be turned off with `echo=no`, or sent to the log-file only with `echo=logonly`.

B.7 Script routines are Basis functions (or macros)

Most of the routines described in this document are Basis script functions (a few are Basis macros). Routines defined as functions may or may not have return values. Return values can be one or more data elements or one or more error condition codes. Return values may be ignored or captured. Assume a function (called `fcn` below, which takes no arguments) returns 0 if it is successful, and non-zero if it detected an error. Here are the ways it may be invoked:

```
call fcn           # ignores return value
fcn                # will display the return value
integer n=fcn      # capture return value in new integer n
if (fcn <> 0) then # use to affect control
    remark "print this warning message"
endif
```

The # character signals the beginning of a comment in Basis.

Most functions take arguments, some or all of which may have default values. The Basis syntax for declaring optional arguments is to precede the optional argument(s) with a semicolon when the function is *defined*. (The semicolon is not used when the function is *invoked*.)

[16] Basis path-related variables and routines are part of the parser package (`par`) group `Path`, so do "`list Path`" to get a complete list. Note that Basis is *case-sensitive*: `Path` is the name of a group and `path` is the name of a variable in that group.

The following function declaration has one mandatory argument (`x`) and two optional arguments (`y` and `z`):

```
function fcn(x;y,z)
```

and may thus be called with one, two or three arguments. They are position dependent, so to call the above function specifying `x=1` and `z=3`, but requesting the default value for `y`, invoke it with:

```
fcn(1,,3)
```

Most of the SSPX script functions contain a help message, which will be displayed if string `"help"` is the 1st argument, e.g.,

```
fcn("help")
```

In this document, descriptions of script functions are introduced with the syntax

```
call fcn
```

if they do not have return value(s) or as an assignment statement, e.g.,

```
integer error_code = fcn
```

if they do have return value(s). In practice, the `call` token may be omitted from user input as it is redundant—it is used in this document to emphasize that the function has no return value.

B.8 Reading and writing data

Basis has efficient facilities for reading and writing data in disk files. Text files are accessed with the stream I/O facility and binary files with the PFB (Portable Files from Basis) facility, which accesses binary data in self-describing files which are portable across all Unix platforms. These capabilities are briefly introduced by example in the following subsections; refer to the Basis reference manual for a complete description.

B.8.1 Text file I/O

Small amounts of data can be imported into the code by simply including the information in an assignment statement, either interactively or as part of a script file, for example

```
real some_data=[1.2, 3.8, 7]
```

For large amounts of data the stream I/O facility should be used: say you had 10000 measurements of some current as a function of time in text file `some_data.dat` arranged in two columns. First, create storage for the data (in one big array); open the text file for read access; read in the data "all at once" with the stream input operator `>>`; close the file; then decompose the big array into a time array and a current array:

```
integer n=10000
real big_array(2,n)
integer io=basopen("some_data.dat","r")
io >> big_array
call basclose(io)
real time(n)=big_array(1,)
real current(n)=big_array(2,)
```

Exporting data to text files can be performed in various ways. One way is to simply redirect STDOUT from the terminal window to a file using the Basis `output` command:

```
output some_data.dat2
time; current
output tty # return STDOUT to screen
```

However, the format of the data in this case is determined by Basis.

The stream output operator << and the `format` function can be used to tailor the output format. Let's say we want to write our data in 2-columns, with a header line and with the TAB character (9th ASCII character) separating data columns:

```
character*1 tab=char(9)
io=basopen("some_data.dat3","w")
io << "      time         current"
do $i=1,n
   io << format(time($i),10,3,1) << tab \
      << format(current($i),16,8,2)
enddo
call basclose(io)
```

In this example `time` is written in a Fortran-style `F10.3` format and `current` in an `E16.8` format. Note the use of chameleon variable, `$i`, as a temporary integer and use of the continuation character: \.

B.8.2 Binary file I/O

The PFB facility in Basis offers an efficient and portable way to handle large amounts of data. The underlying routines are part of LLNL's Portable Application Code Toolkit (PACT)[17].

The user may also use the PFB routines in a customized way. As an example, following those in App. B.8.1, let's say we were executing the Corsica equilibrium solver many times to produce "snap-shots" representing the time-evolution of a discharge. At each step we *append*, in variables `time` and `current`, two particular quantites of interest. We might want to do things (e.g., make plots, write formatted files) with these data in some *future* session, but don't want to bother with those things now. Initially we would declare zero-length arrays to hold our data:

[17] PACT is a comprehensive system of portable software for scientific applications, see `http://pact.llnl.gov`.

```
real time(0), current(0)
```

A loop might be used to execute the equilibrium solver and at the end of the loop we append to our storage arrays:

```
time := shotTime  # shotTime is a caltrans variable
current := 10*plc # variable plc holds Itor in abamperes
```

When the loop has finished, we write the data of interest into a binary file with statements like:

```
create some_data.pfb
write time, current
close
```

where the three commands: `create`, `write` and `close` are Basis interface routines to the PFB library. In some future session, we simply

```
restore some_data.pfb
```

and our arrays of `time` and `current` will be available.

If you have forgotten what you put in a PFB file, do:

```
open some_data.pfb
ls # to list the contents of a PFB file
```

Don't confuse this `ls` command with the Unix `ls` command, which may be executed within a Corsica session with the syntax:

```
!ls # to list file names in the current directory
```

as described in App. B.9.

B.9 Code interaction

Input to the code consists of Basis-language instructions, either coming a line at-a-time in an interactive session or read from a script file. The instructions contain combinations of routine (function or subroutine), macro and variable-name identifiers which are defined by Basis, `caltrans` and by the user. These are woven together in the Basis-language constructs and parsed by the Basis parser. The parser employs the GNU `readline` facility, so Emacs-style input-line editing features—including searchable history—are available.

Interrupts are triggered with `CTRL-C` and place the code in *debug* mode. In debug-mode you can query or alter variables or do any legitimate operation. Enter `cont` to resume an interrupted operation or enter `abort` (or any illegal token) to irreversibly interrupt the operation and return to the parser.

External processes such as the Unix `ls` command can be executed with the `basisexe` routine, e.g.,

```
if (basisexe("ls foo") <> 0) then # no file "foo"
```

and when the command's exit status is of no interest, the bang syntax

```
!ls foo
```

can be used.

Basis functions may or may not have return values and the user may or may not "capture" return values. For example, the Basis `basisexe` function returns the exit status of the process it gave to the operating system. If invoked as a function:

```
basisexe("command")                # or...
integer status=basisexe("command") # or...
if (basisexe("command") <> 0) then # error
```

the exit status will be, respectively: (a) echoed, (b) placed in new variable `status` or (c) used in an `if` test. If called like a Fortran subroutine:

```
call basisexe("command")
```

the return-value (in this case the command's exit status) will be discarded.

C Bias field configurations

The SSPX bias coil set consists of 9 coils, as shown in Fig. 1. A variety of field configurations can be produced with this coil set, some with distinct features. When modelling equilibria for a specific shot, it is recommended the user begin with either a save-file from a similar shot (same bias field configuration) or one of the generic save-files listed in Table 5 in §4.1. The nominal coil currents for these generic save-files are listed in Table 12. Note that a current of 800 A

Table 12: Bias coil nominal currents [A]

	coil→	1	2	3	4	5	6	7	8	9
0	ZERO	0	0	0	0	0	0	0	0	0
1	SOL	0	0	0	0	0	0	0	0	800
2	STD	0	0	0	0	0	0	215	-173	800
3	MF	0	0	0	0	0	0	215	267	800
4	BCS	767	767	-768	-721	699	669	6	-519	800
5	BCM	767	767	-768	-721	699	669	6	500	800
6	BCV	3	3	-6	800	-664	736	2	-1	14
7	NOZ	0	0	0	0	0	0	0	-600	525
8	LG	500	500	-800	0	0	0	215	267	800

represents the nominal capability of the SSPX coil power supplies. The generic equilibria are described in the following sections.

C.1 Solenoid-only (`sspx_sol.sav`)

The "solenoid-only" configuration utilizes only the injector solenoid (coil 9). The generic save-file name is `sspx_sol.sav`. This configuration is shown in Figure 20.

C.2 Standard-flux configuration (sspx_std.sav)

The "standard-flux" configuration, shown in Figure 21, was the first bias field configuration used in SSPX. It utilizes the three coils 7-9 in the inner electrode assembly to produce a radial field across the injector annulus with minimal field in the spheromak region. The save-file name is sspx_std.sav.

C.3 Modified-flux configuration (sspx_mf.sav)

The "modified-flux" configuration is a variant of the standard-flux configuration, where the current in coil 8 is reversed, directing more flux into the spheromak region. Better spheromak performance is attained with this configuration than with the standard-flux configuration. (The modified-flux configuration is the basis for many SSPX shots.) The save-file name is sspx_mf.sav which may also be referred to by the symbolic link: sspx.sav. The configuration is shown in Figure 22.

C.4 Bias-coil-standard configuration (sspx_bcs.sav)

The "bias-coil-standard" configuration is the design-basis for the SSPX bias coil arrangement. It was designed to conform the nominal maximum flux surface ($\Psi = 34\,\mathrm{mWb}$) to the outer shell in the spheromak region. This configuration is shown in Figure 23 and the save-file name is sspx_bcs.sav. Note the limiting X-point is in the diagnostics slot.

C.5 Bias-coil-modified configuration (sspx_bcm.sav)

The "bias-coil-modified" configuration is a variant of the bias-coil-standard configuration, where the direction of current in coil 8 is reversed (analogous to the STD→MF variant). The save-file name is sspx_bcm.sav and the configuration is shown in Figure 24. Note this configuration is limited on the outer wall.

C.6 Vertical-field configuration (sspx_bcv.sav)

The "vertical-field" configuration is designed to produce a nearly vertical field, as shown in Figure 25 (note, as with the BCS, the limiting X-point is in the diagnostics slot). The save-file name is sspx_bcv.sav.

C.7 Nozzle-field configuration (sspx_noz.sav)

The "nozzle" configuration uses only coils 8 and 9 to produce a cusp field as shown in Figure 26. Note the plasma is limited by the wall. The save-file name is sspx_noz.sav.

C.8 Lower-gun configuration (`sspx_lg.sav`)

The "lower-gun" configuration is unique in that it has an additional inner electrode in the divertor region with a corresponding section added to the outer electrode. Note the R-Z grid has been extended for this configuration, and the Corsica $Z = 0$ is aligned with the diagnostics slot, as shown in Figure 27. The save-file name is `sspx_lg.sav`.

D CalTrans source maintenance

The next subsection briefly describes how the `caltrans` source files are maintained. It is followed by a detailed description of the SSPX script files.

D.1 CalTrans source and script repository

The CalTrans developers use the Concurrent Versions System[18] (CVS) to maintain the CalTrans Fortran and C++ source files, and a large collection of standard script files—including the SSPX scripts. The source files (and a detailed record of their updates) are maintained in a common repository.

Each developer checks-out the `caltrans` source files from the repository into their private disk area. After modifications have been made and tested, they are checked-in to the repository, where they become available to all developers. An automated test suite is executed each night on all platforms to make sure (1) the updated code will "build", and (2) the test cases are successfully executed. Periodically, especially after bug fixes, the public distributions `[p,n,v]caltrans` are updated.

User's may copy any of the SSPX script files from the public distribution under the `CORSICA_SCRIPTS` directory into their private disk space and modify them for their own purposes. To determine precisely where the public scripts are located, get the contents of `CORSICA_SCRIPTS` with the Basis `getenv` routine during a session as follows:

```
getenv("CORSICA_SCRIPTS")
```

The SSPX scripts are located in sub-directory `$CORSICA_SCRIPTS/SSPX`.

Users may browse any of the standard script files and possibly copy them and make their own modifications. If changes are such that they will be useful to others, then contact a CalTrans developer to have them integrated into the CVS repository so they will be available to all.

D.2 The SSPX scripts (`sspx.bas`)

The SSPX scripts are contained in several files that are installed as part of the CalTrans distribution in directory `$CORSICA_SCRIPTS/SSPX`, which is automatically added to the Basis search path. During a session, enter "`path`" to

[18] CVS web page: `http://www.cvshome.org`.

get a list of all directories in the Basis search path. The search path is used to locate (in top-to-bottom precedence) file names that are referenced in `read` statements or any file-open requests.

The SSPX scripts are loaded into the session via a single file: `sspx.bas`, which reads all of the subsidiary script files (sub-scripts). They may be loaded into the session by including the name `sspx.bas` on the `caltrans` command-line or with the `read` statement during a session:

```
read sspx.bas
```

Script files in the CalTrans distribution call a routine named `scriptID`. This routine records the pathname and version information about each script file in variables constructed from the script file name, for example, when the SSPX scripts (like `sspx.bas`) are read into a session, variable name `sspx_bas_pn` will contain the full pathname for the script, and variable `sspx_bas_id` will contain CVS version information for the script. To get a list of all scripts which have been read into the session, execute the `scripts` command.

The following subsections describe the SSPX script files in detail.

The `sspx.bas` script reads the subsidiary script files, "sub-scripts" (plus a script named `wall.bas`) and graphics routines in `graphics.bas`. It also defines a function called `sspx`, which displays messages as shown below. This facility provides an on-line documentation facility for user-callable routines in the SSPX-specific scripts. Internal routines in the script files—those not called by the user—are not displayed.

Another function defined in `sspx.bas`: `sspx_name(`*`name`*`)`, will display the name of the script file where *`name`* is defined; *`name`* may be the name of a variable, function or macro.

To see the top-level documentation for the SSPX scripts, execute the function `sspx` with no arguments, as shown below.

```
corsica> sspx

Script file sspx.bas reads the following subsidiary files:

 1 sspx_biascoils.bas
 2 sspx_configuration.bas
 3 sspx_diagnostics.bas
 4 sspx_fitting.bas
 5 sspx_graphics.bas
 6 sspx_ohmicpower.bas
 7 sspx_pillbox.bas
 8 sspx_shotdata.bas

To obtain a list of the routines defined in one of the subsidiary script
files, execute the sspx routine with the category, e.g. sspx("fitting"),
which may be abbreviated, e.g. sspx("f"). Script versions may be displayed
with sspx("version") or sspx("v").
```

```
Use function sspx_name("name") to find the script file which defines a
variable, function or macro name.

Most sspx_*.bas functions respond to function_name("help"). Use the Basis
"list" command to obtain information about variables, macros or undocumented
functions.

Finally, sspx.bas reads a file named "cuztomize.sspx" in the current working
directory (or its parent directory). This file can contain default parameter
settings.
```

New SSPX routines may be added to the standard set by simply adding function or macro definitions to an appropriate existing sub-script file and checking it in to the CVS repository.

New modules (sub-scripts) may be installed by creating a new module file, say `sspx_new_module.bas`. The first line of new module files should contain the statement:

```
scriptID("$Id$")
```

To register the file when it is read into a session. The "`Id`" string will be expanded by CVS to include the module name, version number, date of revision and the modifier's user name.

After testing the routines defined in the new module, add it to the CVS repository and make its name known to `sspx.bas` by appending to the `module` variable in the body of function `sspx`:

```
module := new_module
```

and add the statement

```
read sspx_new_module.bas
```

to the other module reads, followed by a CVS check-in of `sspx.bas`.

D.2.1 Bias coil routines (`sspx_biascoils.bas`)

The bias coil sub-script file, `sspx_biascoils.bas`, defines two functions: `wbcc` and `biascoils`. These are used by the `coils` function, but may also be called by the user.

To display up-to-date contents of the `sspx_biascoils.bas` script:

```
corsica> sspx("biascoils")
sspx_biascoils.bas defines:
function biascoils(;name) # Bias coil information
function wbcc(;arg) # What bias coil configuration
```

The `biascoils` routine is described in §4.2 and provides descriptive information and nominal coil currents for the generic save-files. The `wbcc` ("what-bias-coil-configuration)" returns the name of the bias configuration that best matches bias coil currents in the present equilibrium model.

D.2.2 Configuration routines (sspx_configuration.bas)

The configuration sub-script file, sspx_configuration.bas, defines several routines for modifying the equilibrium model. To display up-to-date contents of the sspx_configuration.bas script:

```
corsica> sspx("configuration")
sspx_configuration.bas defines:
function griddown(;limiter) # Decrease number of grid points by 2**2
function gridup(;limiter) # Increase number of grid points by 2**2
function make_shell(;dri,dro,nis,nos) # Generate conducting shell
function psiwall(;s1,s2,plotit) # Make Psi_wall constant from s1 to s2
function setcc(;c1,c2,c3,c4,c5,c6,c7,c8,c9) # Set coil currents in amperes
function setlimiter(;offset,n_1,n_2,plotit) # Map r,zplate to r,zlimw
function zcutoff(;z_cutoff,cutoff_type) # Modify external lambda zone
```

The gridup, griddown, setcc, setlimiter and zcutoff routines are describe in the main sections of this document. The make_shell routine will modify the flux conserver configuration by changing the radii of the inner and outer electrodes in the injector region, used for sensitivity studies. The psiwall routine will alter the bias flux in the conducting chell, making it uniform over the specified distance along the outer shell.

D.2.3 Diagnostics routines (sspx_diagnostics.bas)

The sspx_diagnostics.bas sub-script contains a few diagnostic routines used by some of the SSPX graphics routines—they are seldom called by the user.

To display up-to-date contents of the sspx_diagnostics.bas script:

```
corsica> sspx("diagnostics")
sspx_diagnostics.bas defines:
function energy(;arg) # Sum Poloidal and Toroidal field energy over FC
function helicityf # Total helicity over flux conserver
function helicityp(;r1,r2,z1,z2) # Partial helicity over R,Z cylinder
function psi_gun(;r_point,z_point) # Return nominal gun flux
```

D.2.4 Fitting routines (sspx_fitting.bas)

The sspx_fitting.bas sub-script defines several equilibrium fitting routines and related functions and macros. All of these routines are described in Sections 3 or 6.

To display up-to-date contents of the sspx_fitting.bas script:

```
corsica> sspx("fitting")
sspx_fitting.bas defines:
function bfit(;n,t) # Fit equilibrium and PTS (if available) data in batch mode
function fit(;i_asph,i_asph2,plotit) # Fit Igun & Bprobe data
function mfit(;t1,t2,deltat,recordResults) # Fit a series of time-slices
function mfit_accept # If HYBRD stalls w/low residual, use eq as-is...
function mfit_quit # Terminate mfit() and mark summary plots
```

```
function mfit_restore(;regexp,time_increment) # Restore and process time-slices
function mfit_retry # If HYBRD stalls, retry to converge
function mfit_skip # Skip present time point and continue mfit()
function mscan(;dgsph,dbsph,dnasp,dnbsp) # Perform multiple-parameter fit scans
function pfit(;alfa_p,betp_p) # Fit Corsica pressure to lpts data
function reload(;time_point) # Reload save-file for time_point [ms]
function saveit(;savename) # Save this equilibrium
function scan(vname;v1,v2,nv,plotit) # Scan parameter vname from v1 to v2
function ss(;regexp,get_fit) # Summarize save-file parameters
function xscan(;delta_x0,plotit,showit) # Vary xname until minimum r.m.s. error
MDEF Layout()= # Plot configuration
MDEF Publish= # Plots made after successful fit
```

D.2.5 Graphics routines (`sspx_graphics.bas`)

The `sspx_graphics.bas` sub-script defines many SSPX-specific plotting routines. Some of these are described in the main sections of this document; most of the others are self-explanatory.

To display up-to-date contents of the `sspx_graphics.bas` script:

```
corsica> sspx("graphics")
sspx_graphics.bas defines:
function compare_fits(;regexp,y_max,t_max) # Compare errors from multiple fits
function compare_mfits(;shot,dirlist) # Compares multiple mfit results
function compare_shots(shot_list;time_point,t_cutoff) # Compare multiple shots
function pad # Plot all data
function pb917(;t1,t2) # Plot Bprobe 9 & 17 data versus time
function pbm2d(;n_levels) # Plot |B| contours
function pbp2d(;n_levels) # Plot Bpol contours
function pbr2d(;n_levels) # Plot Br contours
function pbt2d(;n_levels) # Plot Btor contours
function pbvr(;z_level,y_min,y_max) # Plot B versus radius
function pbz2d(;n_levels) # Plot Bz contours
function pdata(;t1,t2) # Plot Bprobe data spread
function pfx(;interior,r1,r2,z1,z2) # Plot flux surface topology near xpt
function pgrid(;r1,r2,z1,z2,markcoils) # Show grid & jwu,l
function pj(;delta_j) # Plot J_toroidal contours
function pjpol(;delta_j) # Plot J_poloidal contours
function pl2d(;n_levels) # Plot lambda contours
function plambda(;ymax) # Plot lambda profile
function plimiter(;viewport) # Plot limiter point & vicinity
function plvpsi(;zvalue,msize) # Plot lambda verus psibar
function plvr(;y_max) # Plot lambda versus R
function plvz(;y_max) # Plot lambda versus Z
function pmap(;delta_theta) # Plot Psi-theta mapping
function pparams # Plot measured and calculated parameters
function pprobe(;scale) # Plot Bprobe data versus Corsica
function pprobetor # Plot probe Btor data versus Corsica
function ppsibar(;n_levels) # Plot psibar contours
function ppts(;plot_type) # Plot PTS data & spline fit
function ppvr(;zlevel,y_max) # Plot psibar versus R
function pq(;ymax) # Plot q-profile
function psd(;type_code,probe_number,t_point) # Plot shot data
function psmooth(;w_list) # Plot Bprobe for specified smoothing windows
function ptip(;plotit,nxpts) # Plot TIP data & Corsica field
```

```
function pvac(;highlight,delta_psi) # Plot vacuum flux surfaces
function pwc # Plot toroidal and poloidal wall (shell) currents
MDEF p1d()= # Plot ohmic power 1D quantities
MDEF p2d()= # Plot ohmic power 2D quantities
MDEF p2dvr()= # Plot ohmic power 2D quantities versus radius
```

D.2.6 Ohmic power routines (sspx_ohmicpower.bas)

The sspx_ohmicpower.bas sub-script defines the primary routine for performing ohmic-power analyses: pohmic—evaluates ohmic-power quantities on the confined flux surfaces, and pohmic2d—evaluates quantities ovber the confined and open field-line regions. These and the auxiliary routines (tauK, tauW, and wtaue) are described in Section 8.

To display up-to-date contents of the sspx_ohmicpower.bas script:

```
corsica> sspx("ohmicpower")
sspx_ohmicpower.bas defines:
function pohmic(;plot_type) # Evaluate ohmic power over confined region
function pohmic2d(;hlp) # Evaulate ohmic power on RZ grid
function tauK(;region,ne,te) # Helicity decay time
function tauW(;region,ne,te) # Energy decay time
function wtaue(;fname) # Write results of pohmic
```

D.2.7 Pillbox routines (sspx_pillbox.bas)

The sspx_pillbox.bas sub-script defines some routines for preparing data for importation into the NIMROD[19] code. Execute wnimrod("help") for information about the content of the data file. One may also impose a simple cylindrical shell ("pillbox") on an SSPX equilibrium for use by NIMROD.

To display up-to-date contents of the sspx_pillbox.bas script:

```
corsica> sspx("pillbox")
sspx_pillbox.bas defines:
function pillbox(;arg) # Use pillbox shell for equilibrium
function rnimrod(;fname,plotit)
function wnimrod(;fname_wnimrod,ndec) # Write file for NIMROD
```

D.2.8 Shot data routines (sspx_shotdata.bas)

The sspx_shotdate.bas sub-script defines the routines which load SSPX data into a Corsica session, and some related routines. The more widely used routines are described in Section 5.

To display up-to-date contents of the sspx_shotdata.bas script:

```
corsica> sspx("shotdata")
sspx_shotdata.bas defines:
function apts(;n_poly,plot_type) # Generate PTS data from analytic model
function baseline_times(;wsmooth) # Evaluate time points for fixBaseline
```

[19] The NIMROD web page is http://www.nimrodteam.org/.

```
function calib # Display calibration information
function coils # Display coil currents
function d4c # Display parameters used to get data
function d4csum(;file_name) # Summarize d4c database files
function diddle_data # Adjust data after reading from disk (.pfb)
function files # Summarize save files etc. in CWD
function lpts(;shot,n_poly,plot_type) # Load PTS data
function lsd(;shot,time_point,plotit,shotdate)
function ltip(;shot) # Load TIP data
function mksadb(shot_list;n_points) # Make shot-averaged database
function mpts(;n_poly,plot_type) # Modify existing PTS data
function shots(;from_shot) # List shot dates
function sspx_calib(;cdate) # Make calibration file
function wpts(;hlp) # Write (modified) PTS data to disk
```

D.2.9 Vacuum flux routines (`wall.bas`)

The script file `wall.bas`, for historical reasons, is treated separately from the other `sspx_*.bas` scripts, but it is always read into the session by `sspx.bas`.

There are several routines defined in this script, but the one of interest here is the `wall_sph` function, discussed in §3.2.2. This function evaluates the flux from the bias field coils at the position of each conducting wall element, and imposes that flux as a boundary condition on the Grad-Shafranov solver.

In general, when the bias coil currents (or their position) is changed, one must execute the `run` command, execute `wall_sph` to evaluate and impose the vacuum flux at the wall, and re-execute the `run` command to make the equilibrium consistent with the new vacuum flux.

This process is automatically performed by the `setcc` routine (used to change coil currents), which is called by `lsd`, so each time shot data is loaded into the session resulting in a change in the bias coil currents, `lsd` will execute `setcc` which, in turn, will execute `run; wall_sph; run`. Therefore, it is only necessary to use `wall_sph` if one changes bias coil *positions*.

E SSPX Auxiliary Files

The `CORSICA_PFB` directory (part of the CalTrans distribution) contains all the generic save-files plus other auxiliary binary files, in Basis PFB format, that are used by Corsica or the SSPX scripts. The auxiliary files are described below.

E.1 Greens functions (`greens33x65x*.pfb`)

Files with names like `greens33x65x327.pfb` for the original SSPX configuration and `greens33x65x379.pfb` for the lower-gun configuration contain arrays of Greens functions that are specific to the grid resolution (33x65 in this case) and the number of coils, which includes the bias 12 coil elements plus the wall elements. Greens function tables are only retained that are consistent with the generic save-files. This file is automatically loaded by Corsica when a save-file is named on the command-line, and need not be referenced by the

user. The Corsica start-up message

```
calculating Greens functions
```

indicates that the stored Greens functions in the PFB file are *inconsistent* with the present equilibrium (e.g., the grid resolution is different, or perhaps the grid dimensions are different) and therefore must be recomputed. This usually means the save-file is "old" and one should re-generate the equilibrium from an up-to-date generic save-file.

Greens functions files are created with the Corsica command: `store_greens`.

E.2 Shot dates database (`shot_dates.pfb`)

The binary file `shot_dates.pfb` is a database of shot dates stored as a function of shot number, which is automatically loaded by the SSPX scripts. This file is updated nightly by a Unix `cron` job[20] from the text file

```
/sspx/doc/shot.dates
```

which is manually updated[21] periodically.

This binary database is used by the `lsd` routine to include the date of a shot in the `probid` string, which is used to label plots. It is also used by the `shots` command, which displays a list of shots by date.

E.3 Probe positions (`sspx_loops.pfb`)

The binary file `sspx_loops.pfb` contains the position and orientation of the 19 poloidal magnetic probe positions and also the coordinates and orientation of points along the wall in the vicinity of the probes at which the Corsica field is evaluated.

The magnetic probe array in SSPX consists of a main set of poloidally distributed probe positions (p01-p19) located at an azimuth of 90°, and ordered top to bottom as shown in Figure 1. Two of the probe positions (p03 and p17) have 2 toroidal locations, and position p09 has 6 toroidal locations. The measurements from the multiple toroidal locations of probe positions p03, p09 and p17 are averaged by the `lsd` routine.

The following quantities are contained in `sspx_loops.pfb`, where the name refers to variable names in the Corsica session with parenthesized items indicating array sizes.

Name	*Description*
`nloop`	number of poloidal locations (19)
`rloop(19)`	radial position of probe [cm]
`zloop(19)`	axial position of probe [cm]
`thloop(19)`	probe orientation [radians]

[20] The `cron` job is maintained by Bill Meyer.
[21] The `/sspx/doc/shot.dates` file is maintained by Jeff Moller.

`sloop(19)`	distance along wall to each probe [cm]
`nwall`	number of wall coordinates (232)
`rwall(232)`	radial position of point on wall [cm]
`zwall(232)`	axial position of point on wall [cm]
`thwall(232)`	orientation of `rwall`,`zwall` point [radians]
`swall(232)`	distance along wall to each `rwall`,`zwall` point [cm]

The orientation angles (`thloop` and `thwall`) are the plasma-facing surface normals of each probe-point (`rloop`,`zloop`) or wall point (`rwall`,`zwall`). So, for example, probe position `p01` has the orientation `thloop(1)`$= \pi$.

These two sets of coordinates and orientations are used by the `pprobe` function to evaluate the Corsica field both at the SSPX probe positions and at points *in between* to make plots of the measured and Corsica magnetic field as a function of distance along the wall, $B_\theta(s)$.

E.4 Probe calibration factors (`sspx_calib.pfb`)

The binary file `sspx_calib.pfb` contains the date of calibration, calibration factors, standard deviations, probe azmuthal positions and default weights for the poloidal magnetic probes, and is automatically loaded by the SSPX scripts. Use the `calib` plotting routine to display the calibration factors and standard deviations during a session.

This file is created by script function `sspx_calib()` from a text file named `sspx_calib.dat`. As of this writing, the calibration factors are based on the *in situ* measurements of Holcomb and Woodruff made in September 2002.

The following quantities are contained in `sspx_calib.pfb`, where the name refers to variable names in the Corsica session with parenthesized items indicating array sizes.

Name	*Description*
`calib_date`	date of calibration, "YYMMDD" format
`bprobe_angle(19)`	azimuthal position of main probes
`bprobe03_angle(2)`	azimuthal position of toroidal array at position `p03`
`bprobe09_angle(6)`	azimuthal position of toroidal array at position `p09`
`bprobe17_angle(2)`	azimuthal position of toroidal array at position `p17`
`bprobe_calib(19)`	calibration factors for main probes
`bprobe03_calib(2)`	calibration factors for toroidal array at position `p03`
`bprobe09_calib(6)`	calibration factors for toroidal array at position `p09`
`bprobe17_calib(2)`	calibration factors for toroidal array at position `p17`
`bprobe_stddev(19)`	standard deviations for main probes
`bprobe03_stddev(2)`	standard deviations for toroidal array at position `p03`
`bprobe09_stddev(6)`	standard deviations for toroidal array at position `p09`
`bprobe17_stddev(2)`	standard deviations for toroidal array at position `p17`
`default_wt(19)`	default weights

F Getting shot data

Shot data consists of the bias coil currents (static during the shot) and the injector (gun) current, magnetic field (poloidal and, optionally, toroidal) at the

wall probes, and gun energy measured as a function of time. This data is obtained from the SSPX shot database with IDL procedure `d4c.pro`. Profile Thomson measurements of the electron temperature and density are obtained via McLean's IDL procedure `ptsfit.pro`. A 3rd procedure, `cc4b.pro`, can be used to extract just the bias coil currents for a particular shot. These IDL procedures are located in the SSPX directory: `/sspx/idl/`, which should be placed in you IDL search path.

Usage of these procedures is describe in the following sections, along with a shell script, `d4c`, that executes `d4c.pro` and `ptsfit.pro`.

F.1 IDL procedure `d4c.pro`

The IDL procedure "data-for-Corsica" procedure, `d4c.pro`, is used to prepare SSPX measurements $I_{gun}(t)$, $B_\theta(t)$, $B_\varphi(t)$, etc. in a file named ***`shot.d4c`*** for importation into a Corsica session via `lsd` (see §5.1). The procedure is read into an IDL session and compiled with

```
.run d4c.pro
```

Executing the `d4c` procedure without an argument will display a short help message. At a minimum an SSPX shot number must be specified, and optionally the cutoff time for the data can be specified. If toroidal field measurements are of interest, they may also be requested. Signal processing options to override the defaults can be passed.

Get shot data for Corsica in an IDL session with...

```
idl
.run d4c.pro
d4c,shot[,t_cutoff=t,include_bt=i,options]
```

`shot` : integer shot number [no default].

`t` : integer or real time, in ms, at which data retrieval will be terminated. Data retrieval begins at $t = 0$ (breakdown occurs some time after $t = 0$). The cutoff time ***`t`*** defaults to when the signal in probe `p09` falls to 5% of its peak value, rounded up to the nearest ms.

`i` : if zero, omits toroidal field measurements from the output file [default], to conserve disk space. Toroidal field measurements are not used for equilibrium reconstruction, so may therefore be omitted.

The ***`options`*** argument refers to any of the seven `GET_DIAG` signal processing options[22] (`calibrate`, `integrate`, `median`, `base0`, `nbase` `nfit` and `xfit`), which get passed to the various routines which return the processed signals (e.g., `GET_MP090PXX`).

[22]The signal processing options are defined in the file `/sspx/idl/get_diag.pro`, which gets called by all of the `GET_MP*` routines.

The text file *shot*`.d4c` is read by `lsd` using the Basis stream I/O facility. Its contents are listed below—the names refer to the variable names in the Corsica session with parenthesized items indicating array sizes.

Name	*Description*
`d4c_version`	character string containing `d4c` version
`shot_d4c`	shot number
`tor_d4c`	sentinel ($\neq 0$ means B_φ data included)
`vfb_d4c`	formation bank voltage
`vsb_d4c`	sustainment bank voltage
`cc_d4c(9)`	bias coil circuit currents
`n_d4c`	number of time-points
`t_d4c(n_d4c)`	array of time-points
`igun_d4c(n_d4c)`	$I_{gun}(t)$
`wgun_d4c(n_d4c)`	$W_{gun}(t)$
`bprobe_d4c(n_d4c,nloop)`	$B_\theta(t,p)$
`bprobe03_d4c(n_d4c,2)`	$B_{\theta,\mathsf{p03}}(t,2)$ (2 toroidal locations)
`bprobe09_d4c(n_d4c,6)`	$B_{\theta,\mathsf{p09}}(t,6)$ (6 toroidal locations)
`bprobe17_d4c(n_d4c,2)`	$B_{\theta,\mathsf{p17}}(t,2)$ (2 toroidal locations)
`bt_d4c(n_d4c,nloop)`	$B_\varphi(t,p)$
`bt03_d4c(n_d4c,2)`	$B_{\varphi,\mathsf{p03}}(t,2)$ (2 toroidal locations)
`bt09_d4c(n_d4c,6)`	$B_{\varphi,\mathsf{p09}}(t,6)$ (6 toroidal locations)
`bt17_d4c(n_d4c,2)`	$B_{\varphi,\mathsf{p17}}(t,2)$ (2 toroidal locations)
`calibrate_d4c`	`GET_DIAG` calibration value
`integrate_d4c`	`GET_DIAG` integrate value
`median_d4c`	`GET_DIAG` integrate value
`base0_d4c`	`GET_DIAG` base0 value
`nbase_d4c`	`GET_DIAG` nbase value
`nfit_d4c`	`GET_DIAG` nfit value
`xfit_d4c`	`GET_DIAG` xfit value

F.2 IDL procedure `ptsfit.pro`

The IDL procedure `ptsfit.pro` can be used to prepare the file `ptsfit_`*`shot`* containing the Profile Thomson Scattering data $n_e(R)$ and $T_e(R)$ for importation into a Corsica session via `lpts` (see §5.3).

Get Profile Thomson data for Corsica in an IDL session with...

```
idl
.run ptsfit.pro
ptsfit,shot[,options]
```

shot : integer shot number [no default].

It accepts many arguments (contact Harry McLean for details). The `out=2` directive will write the output file in a form compatible with `lpts`.

F.3 Shell script `d4c`

Shell script `d4c`, located in `/sspx/bin`, executes IDL procedures `d4c.pro` and `ptsfit.pro`, described above.

Get shot data and Profile Thomson data for Corsica with...

```
% d4c,shot
```

shot : integer shot number [no default].

The `d4c` shell-script, executed at the Unix prompt (indicated by `%` here), runs IDL with the `d4c.pro` procedure with only the shot number and the `ptsfit.pro` procedure with the following arguments:

```
ptsfit, shot, scope=1, wt4=1, ne0=2, out=2
```

It will create five files:

```
shot.idl         IDL input file
shot-d4c.ps      graphics from IDL script d4c.pro
shot.d4c         B-probe data for Corsica
ptsfit_shot      PTS data for Corsica
ptsfit_shot.ps   graphics from IDL script ptsfit.pro
```

and the `.idl` and `.ps` files may be discarded.

The `d4c` shell script will display its help message with:

```
% d4c -h
```

F.4 IDL procedure `cc4b.pro`

IDL procedure `cc4b.pro` ("Coil-currents-for-Basis") can be used to write a text file named *shot*`.cc` containing coil currents for importation into the Basis code `biasflux`, described in §4.3.

Get bias coil currents for importation into a `biasflux` session with...

```
idl
.run cc4b.pro
cc4b,shot
```

shot : integer shot number [no default].

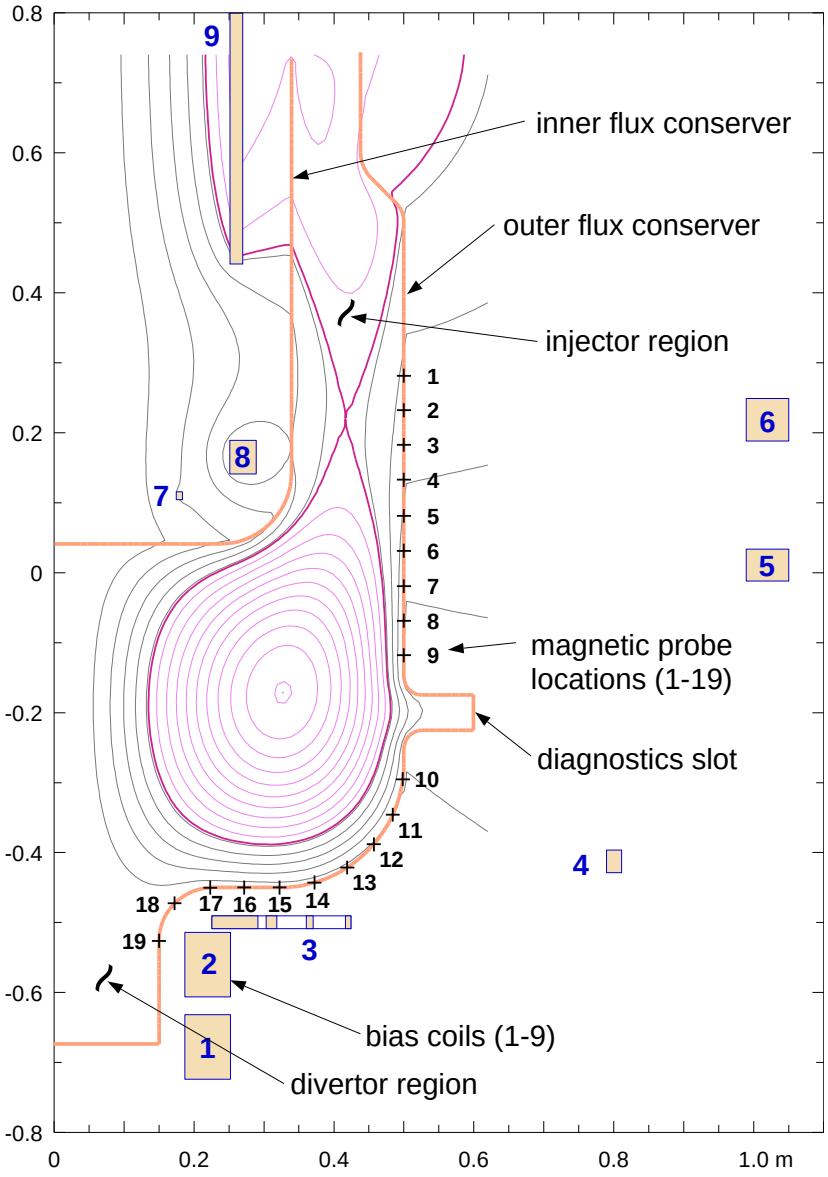

Figure 1: **Corsica model of SSPX** showing the poloidal location of the 19 magnetic probe (90° azimuth) positions, corresponding to probe signals `mp090pxx` (xx $= 01, \ldots 19$). As of Nov. 2001, probe positions 3 and 17 each have 2 probes distributed toroidally and position 9 has 6 probes distributed toroidally and the 9 bias coils (coil 9 is the injector solenoid). *Note the Corsica coordinate system origin is displaced 0.2 m from the SSPX machine coordinate system ($Z = 0$ at the diagnostics slot midplane).*

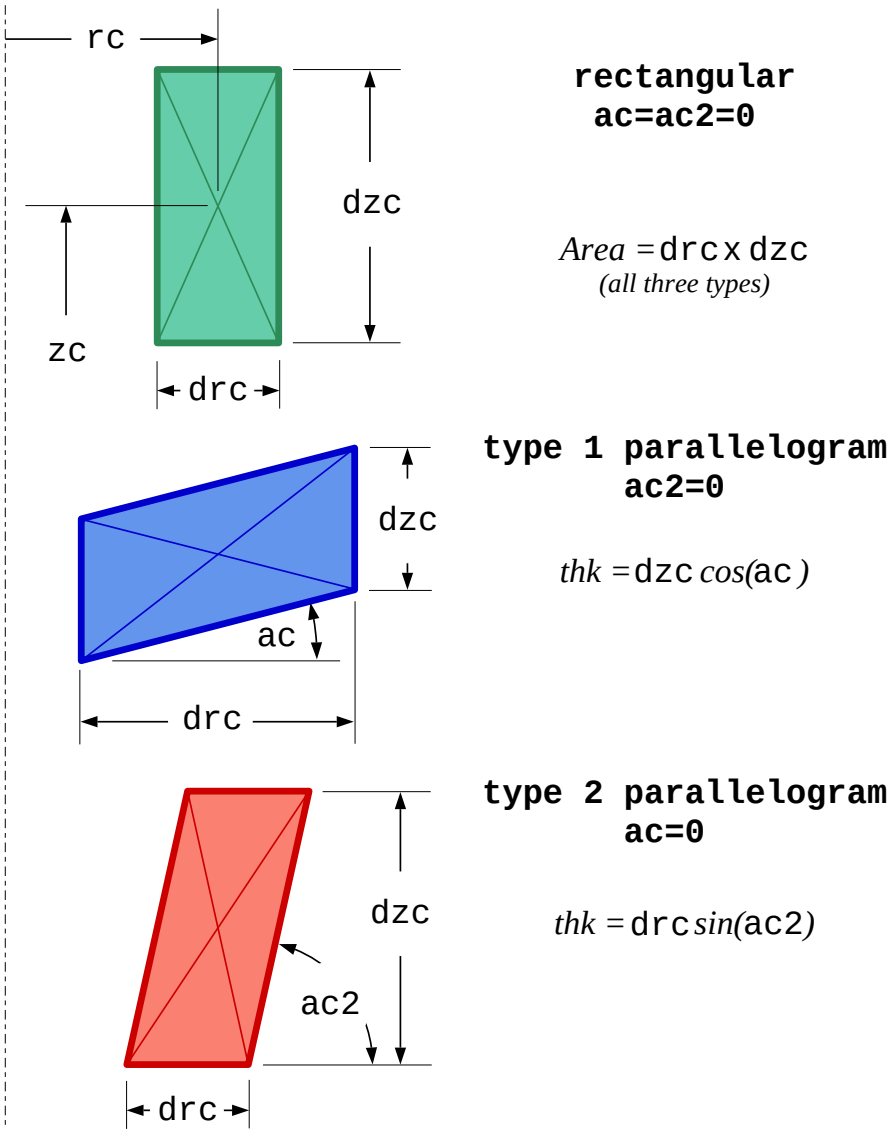

Figure 2: **Corsica coil model**; used to model the bias coils and flux conserver (conducting wall) elements, with thickness *thk*.

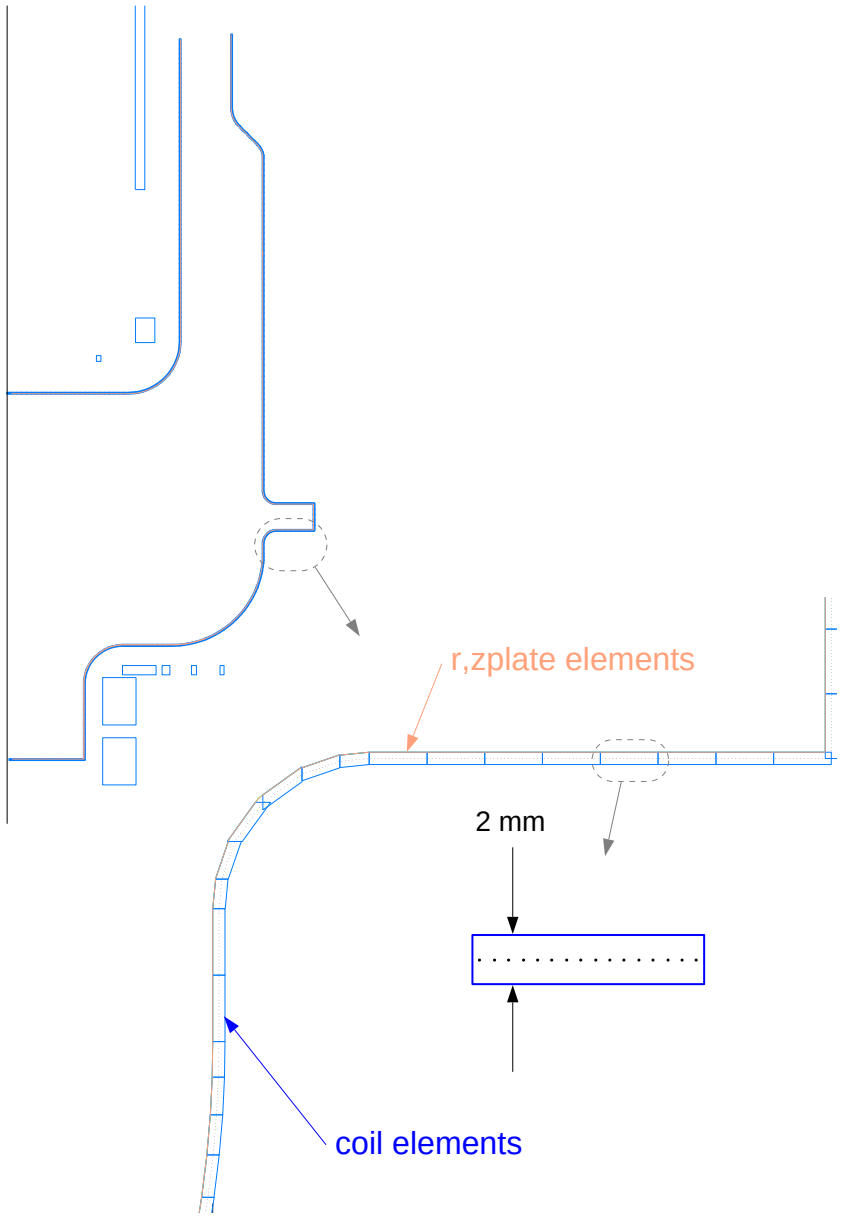

Figure 3: **SSPX conducting wall model**; the conducting wall is represented by rectangular or parallelogram coil elements of thickness 2 mm with an average length of 9 mm. A typical coil element representing the conducting wall has 16 filamentary current loops. (The bias coils are represented by the *first* `ncplot` coil elements.) The plasma-facing side of the conducting wall is described by `rplate`, `zplate` elements.

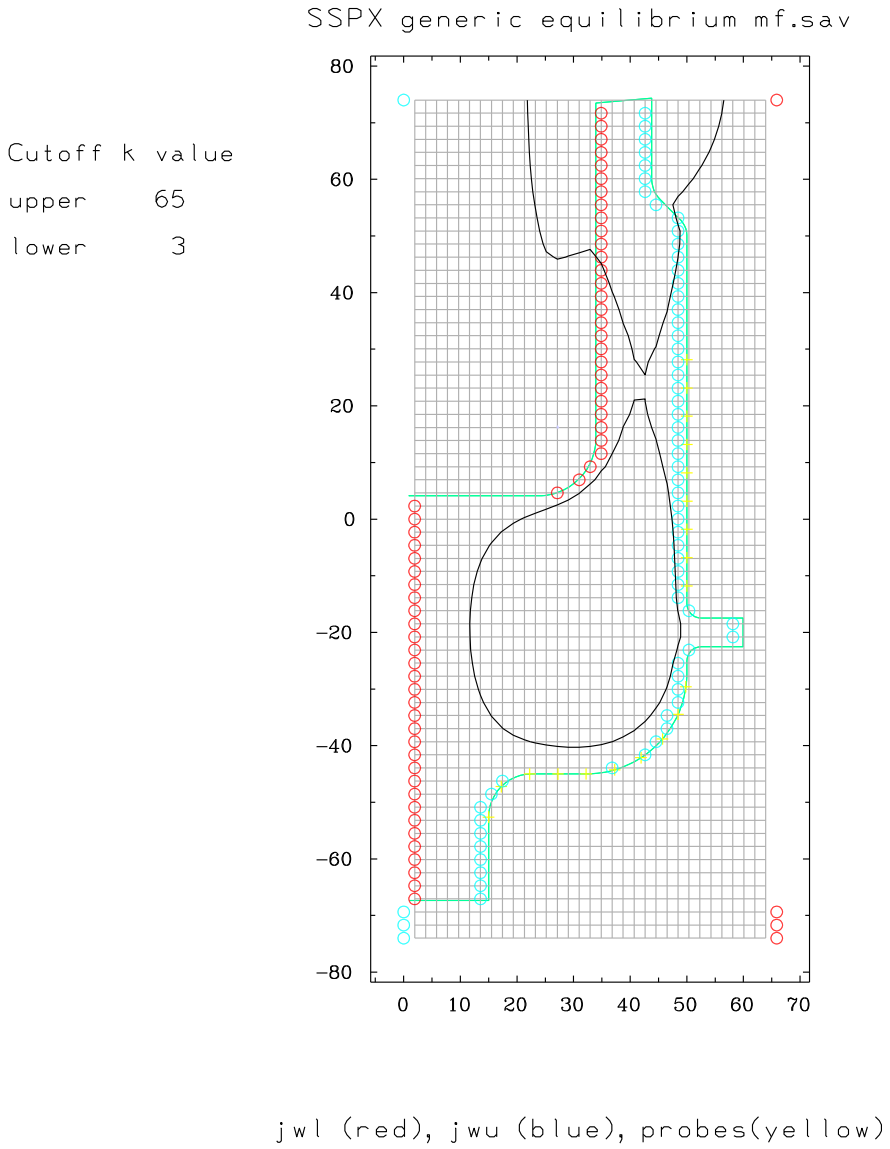

Figure 4: **Output from `pgrid` function**; grid variables `jwl(1:km)` and `jwu(1:km)`, set by `wall_sph`, specify the beginning and ending of the open field-line region.

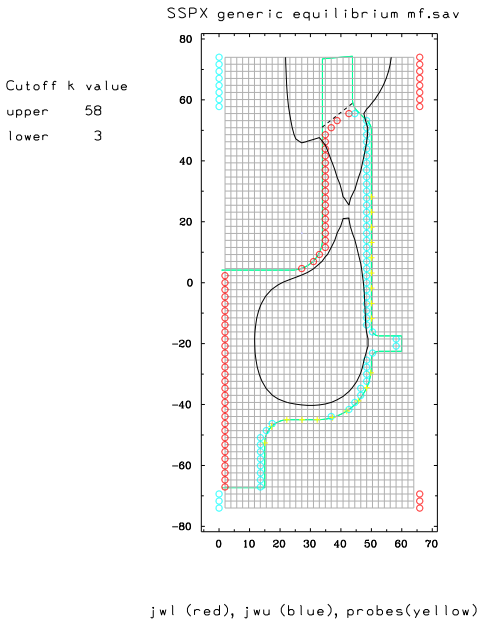

Figure 5: **Results of executing `zcutoff` function**; the upper figure was obtained with `zcutoff(50); pgrid` and the lower figure the result of `zcutoff(55,1); pgrid`.

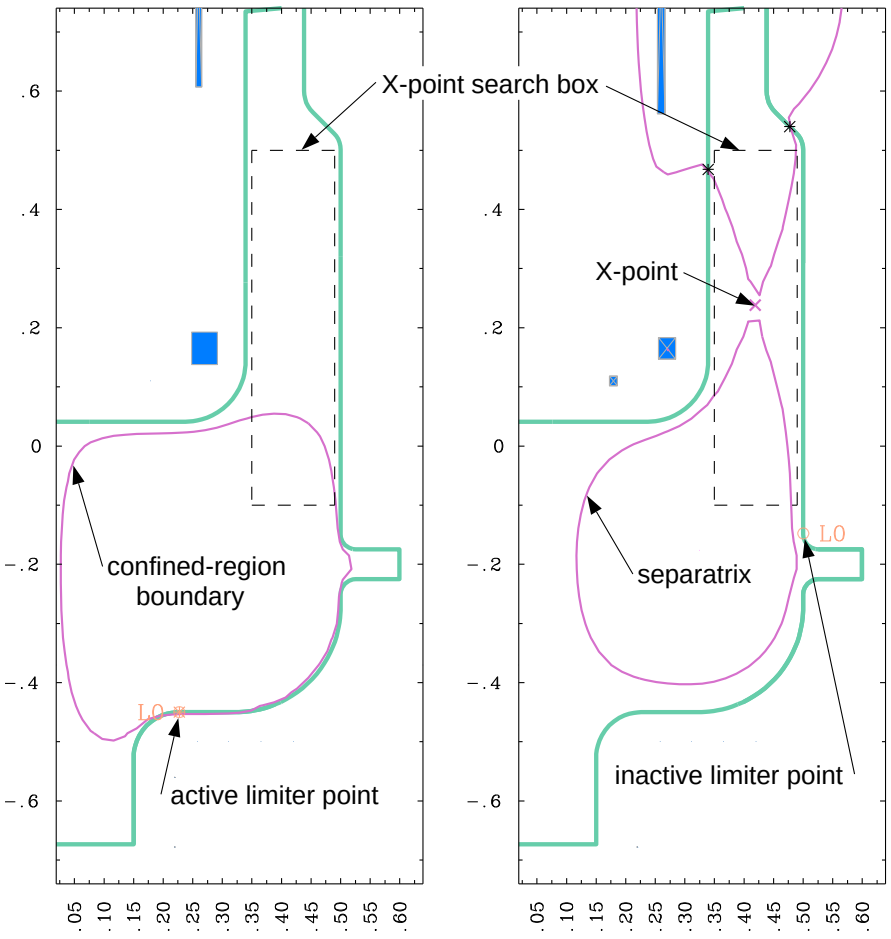

Figure 6: **Output from the pb routine**; wall-limited topology (left) with active limiter point shown with a filled circle, and X-point limited topology (right) where the limiter point is inactive. The limiter point coordinates are prescribed with variables `rlim` and `zlim`. The X-point search box is defined with `rxpr(2)` and `zxpr(2)`.

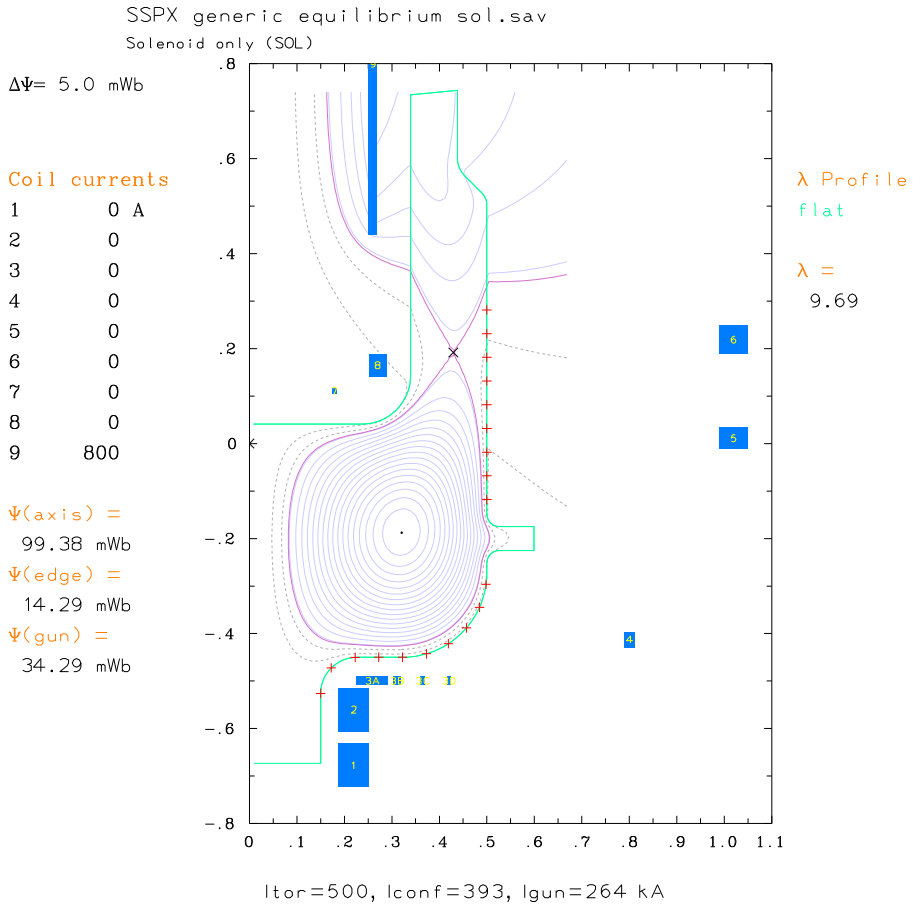

Figure 7: **Output from the `Layout` macro**; flux contours shown in increments of 5 mWb (specified by variable `delta_psi_contour`). The coil currents, toroidal current (500 kA) and λ-profile are input quantities. The confined toroidal current (393 kA), gun current (264 kA) and flux values are output quantities.

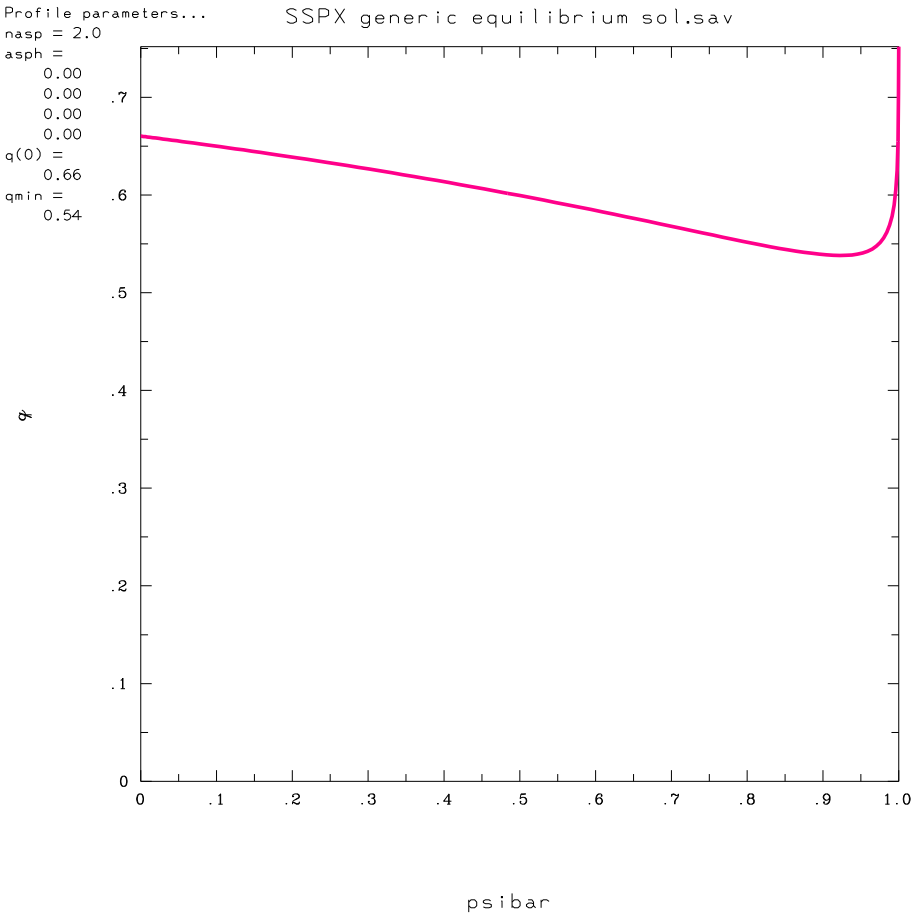

Figure 8: **Output from the pq function**, showing $q(\tilde{\psi})$ over the confined region.

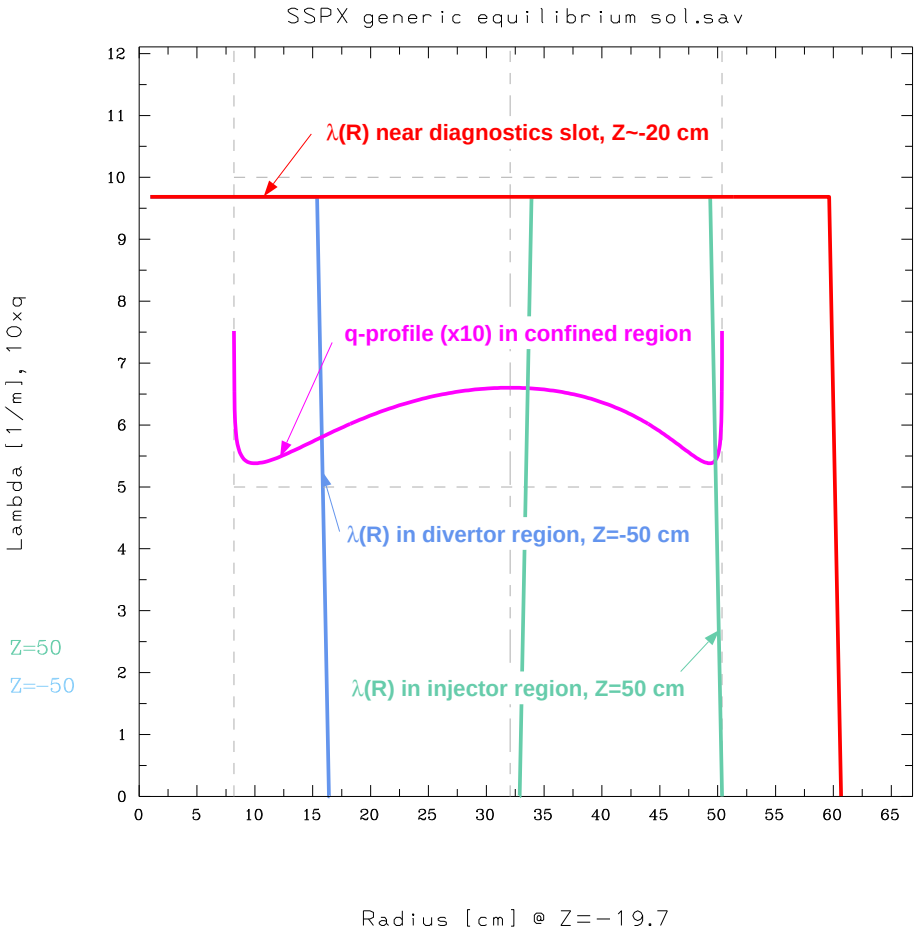

Figure 9: **Output from the `plvr` function**, showing $\lambda(R)$ at three axial positions at $10 \times q(R)$ near the magnetic axis. The vertical dashed lines represent the inner and outer edge of the confined region and the dot-dash line the magnetic axis. The horizontal dashed lines mark the $q = 0.5$ and $q = 1$ values. In this example, the lambda profile is flat everywhere, so the rapid fall-off to zero in the three λ curves mark the position of the conducting wall.

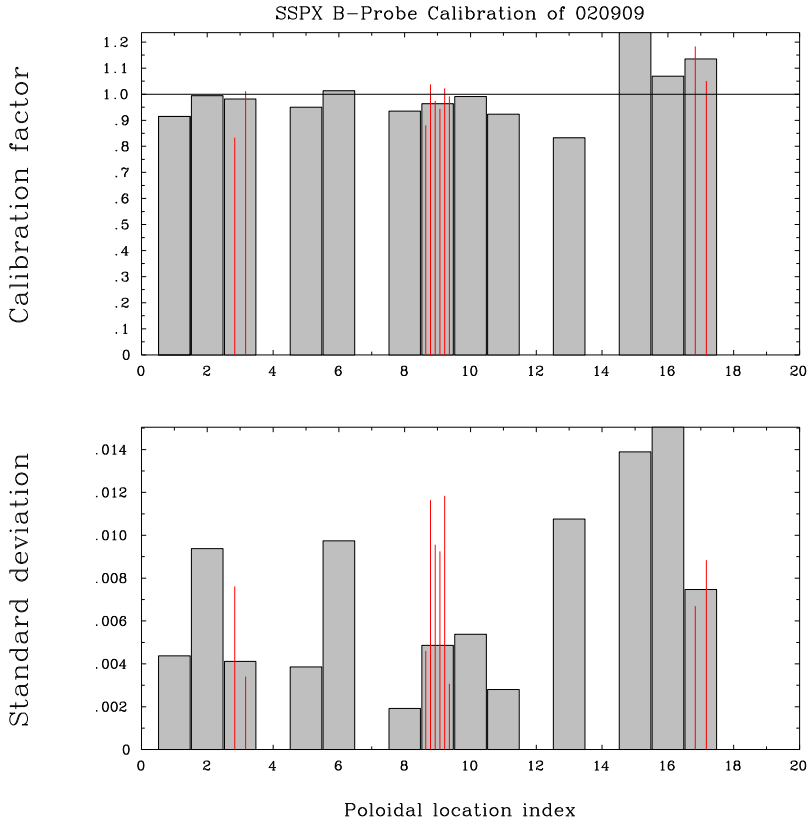

Figure 10: **Output from the `calib` function**, showing the calibration factors and standard deviations, from the *in situ* calibration performed in September 2002. The shaded bars represent the average values used by `lsd`. The vertical lines represent the multiple probes at locations `p03`, `p09` and `p17`.

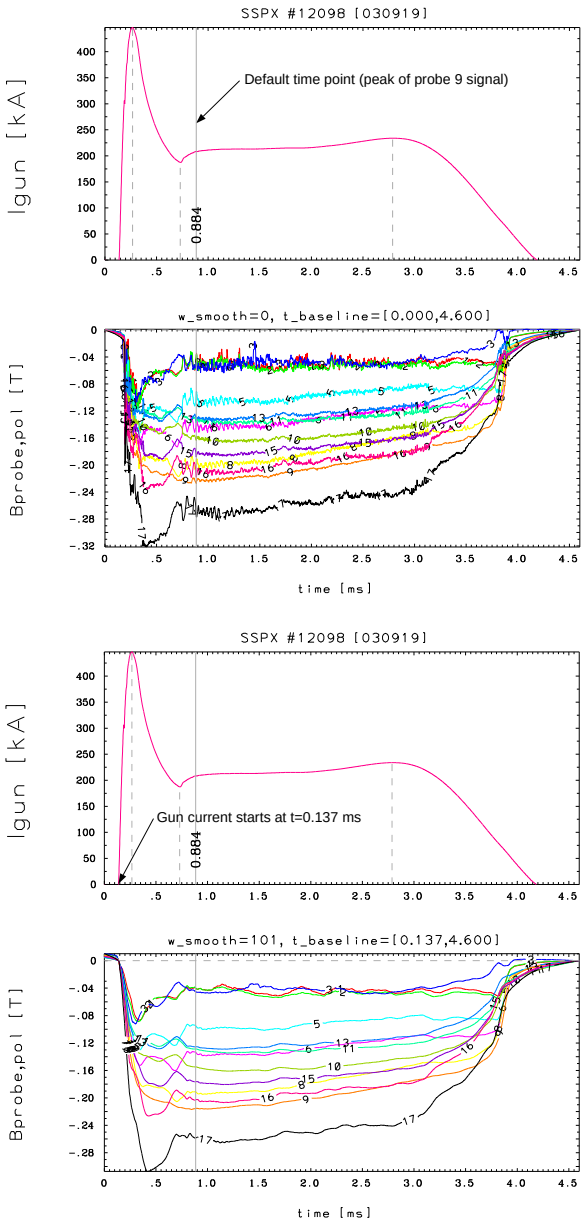

Figure 11: **Output from `lsd`**; the upper two figures shows the default result with no smoothing and no change to the baseline correction times. The lower two figures show the results of smoothing the data and modifying the default correction time of `t_baseline(1)` to match the start of the gun current rise.

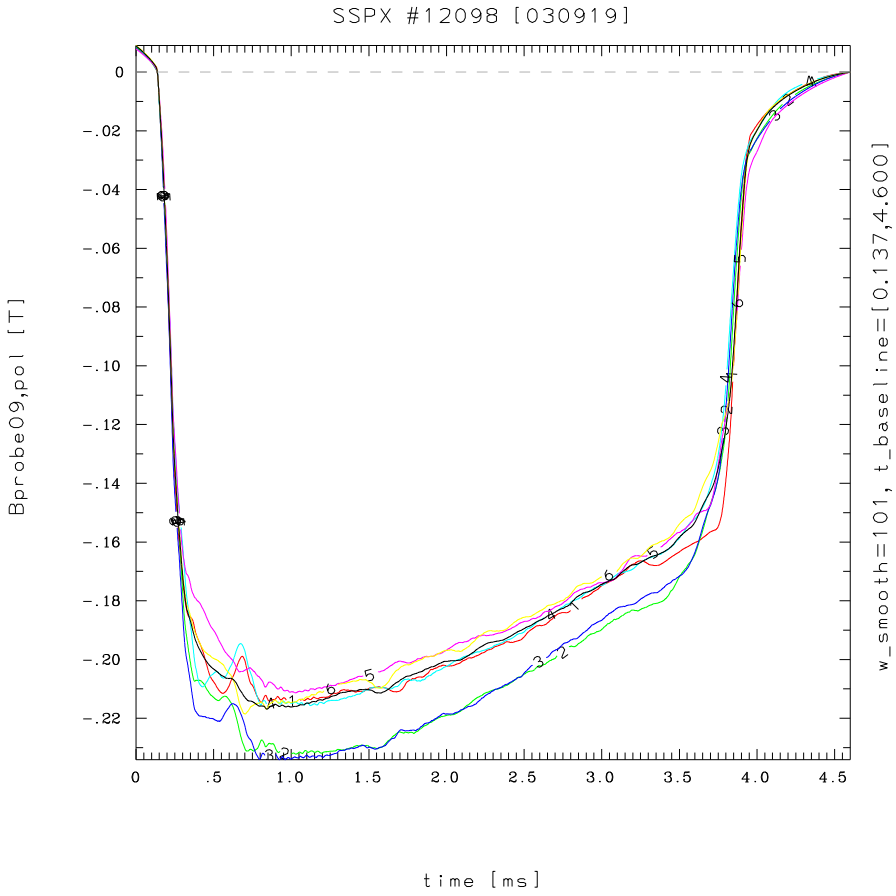

Figure 12: **Output from the `psd` function**, showing the poloidal field as a function of time for the 6 toroidal locations of probe `p09`, made by calling `psd("bpol",9)`.

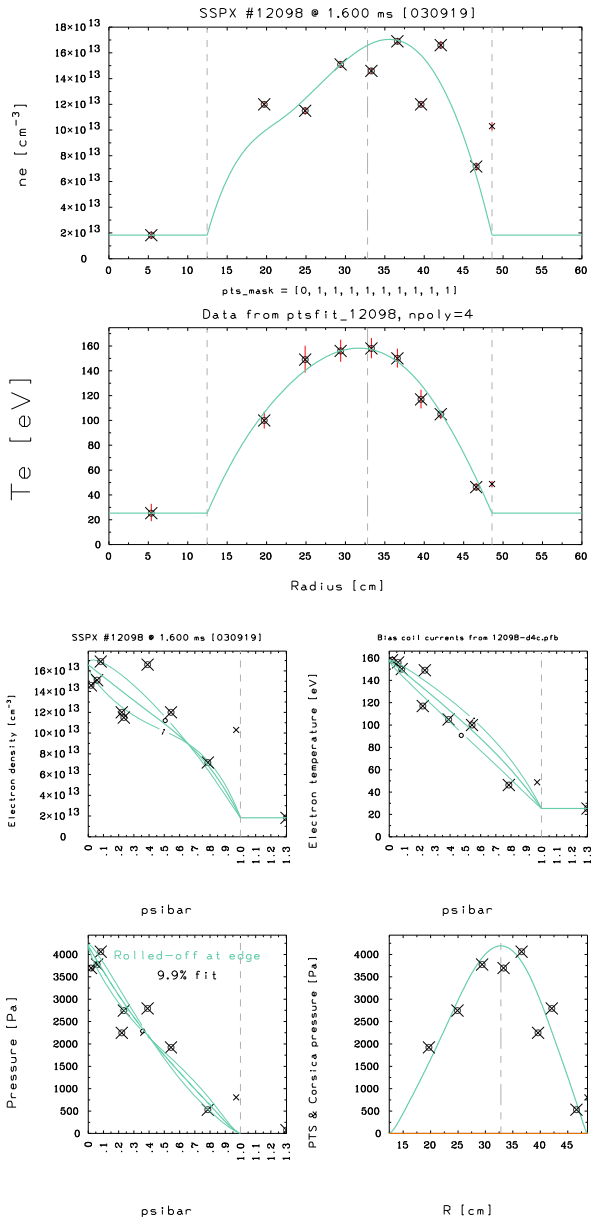

Figure 13: **Output from `lpts` or `ppts`**; the upper two plots from `lpts(,,1)` or `ppts(2)` show the $n_e(R)$ and $T_e(R)$ PTS measurements with error bars. The solid curve represents the polynomial fit to the data. The lower four plots from `lpts(,,2)` or `ppts(2)` show $n_e(\tilde{\psi})$, $T_e(\tilde{\psi})$, $p_e(\tilde{\psi})$ and $p_e(R)$, where the multiple curves indicate the inboard, outboard and average values.

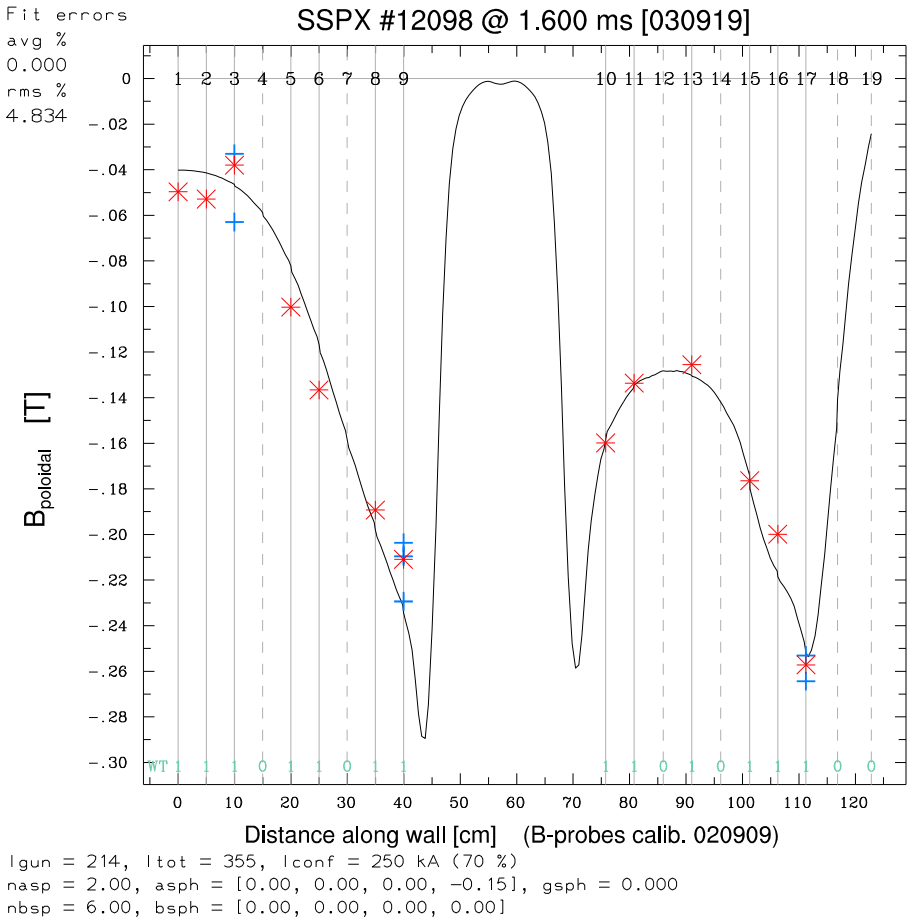

Figure 14: **Output from the `pprobe` function**, showing the tangential field at the wall from Corsica as a function of distance along the wall. The probe locations are marked by vertial lines, with the probe number listed at the top (a dashed line means the measurement is not available). The measurements are marked with "*" (multiple measurements with "+"). The probe weights are listed near the bottom.

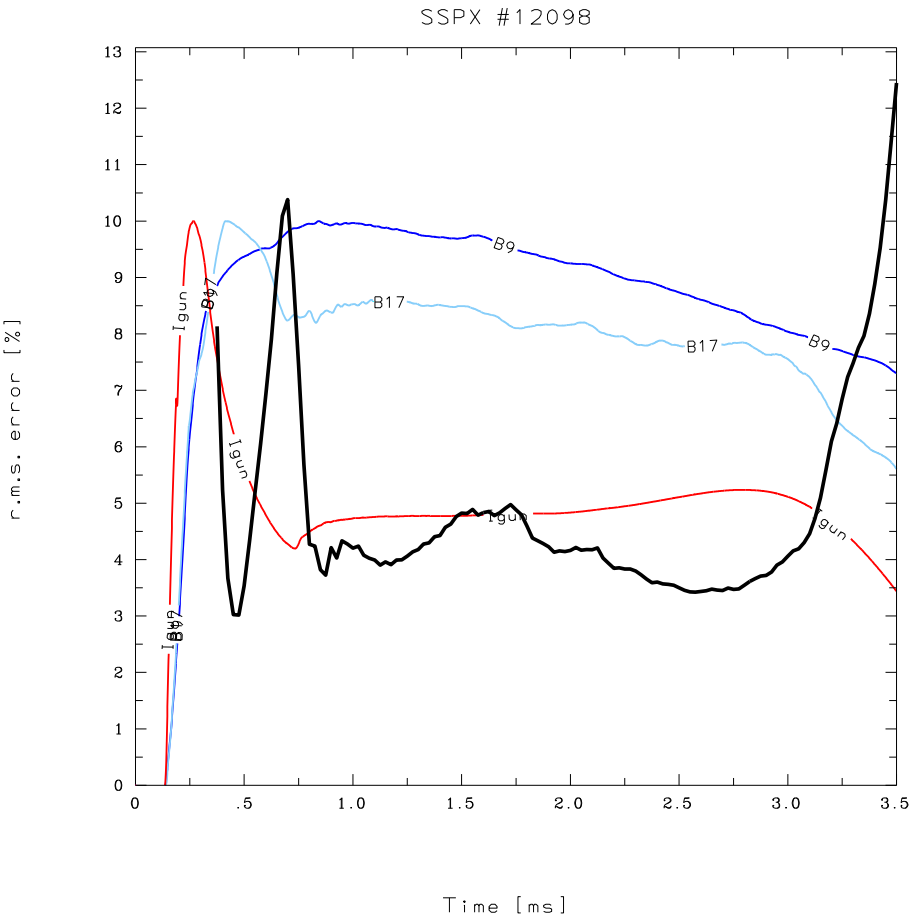

Figure 15: **Output produced by `mfit` routine**, showing the quality of the fit (r.m.s. error) as a function of time, superimposed (with arbitrary scale) over the measured $I_{gun}(t)$ and $B_\theta(t)$ at probe positions 9 and 17. The quality of fit is typical: generally good ($\leq 5\,\%$) during the sustainment phase, not too good during the formation pulse and sustainment current ramp-up, and diminishing near the end of the pulse.

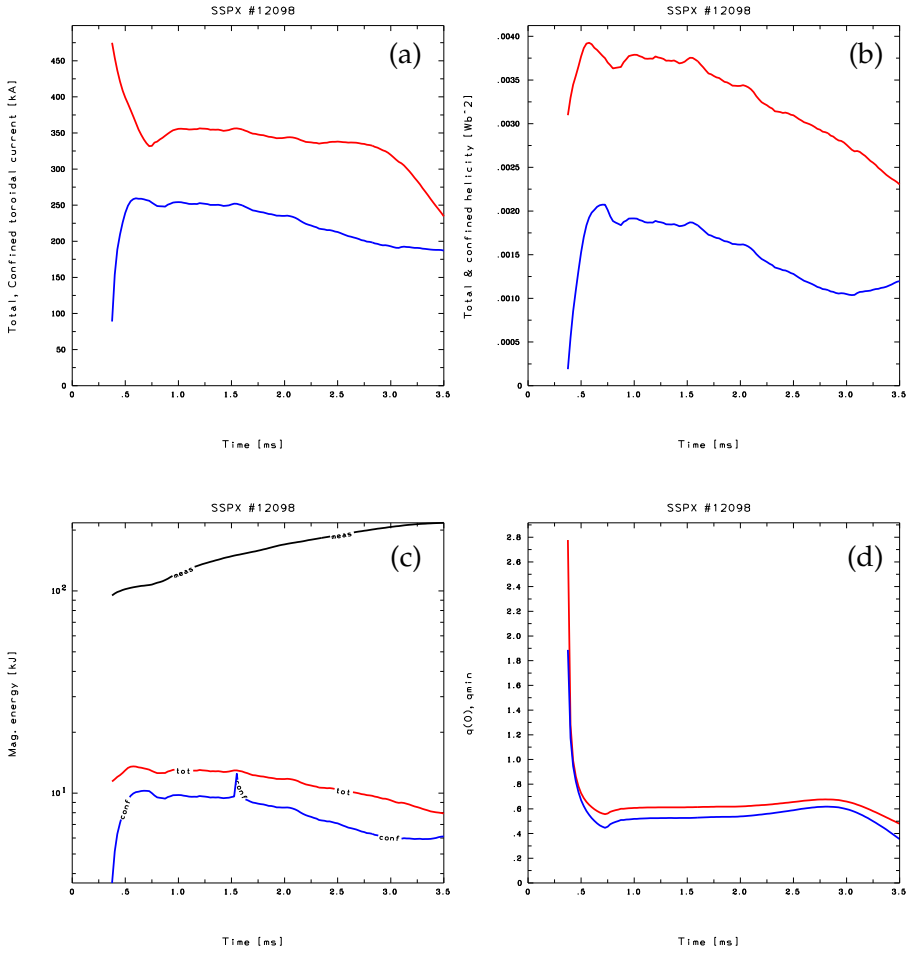

Figure 16: **Sample summary plots produced by `mfit` routine**; (a) total and confined toroidal current, (b) total and confined helicity, (c) total and confined magnetic energy (also shown is the total injected energy), and (d) the safety-factors: q_0 and q_{min}.

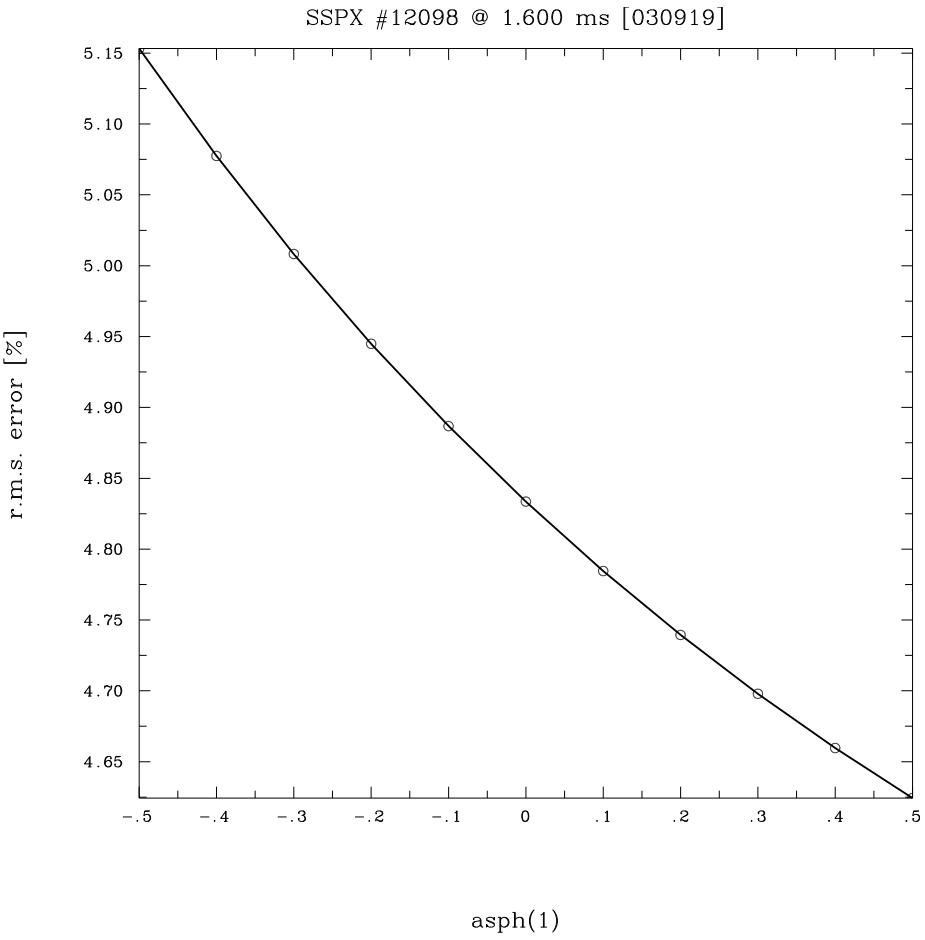

Figure 17: **Sample plot produced by `scan` routine**, showing the value of the r.m.s. error over the scan interval `asph(1)` $= -0.5 \rightarrow 0.5$ with a 0.1 increment.

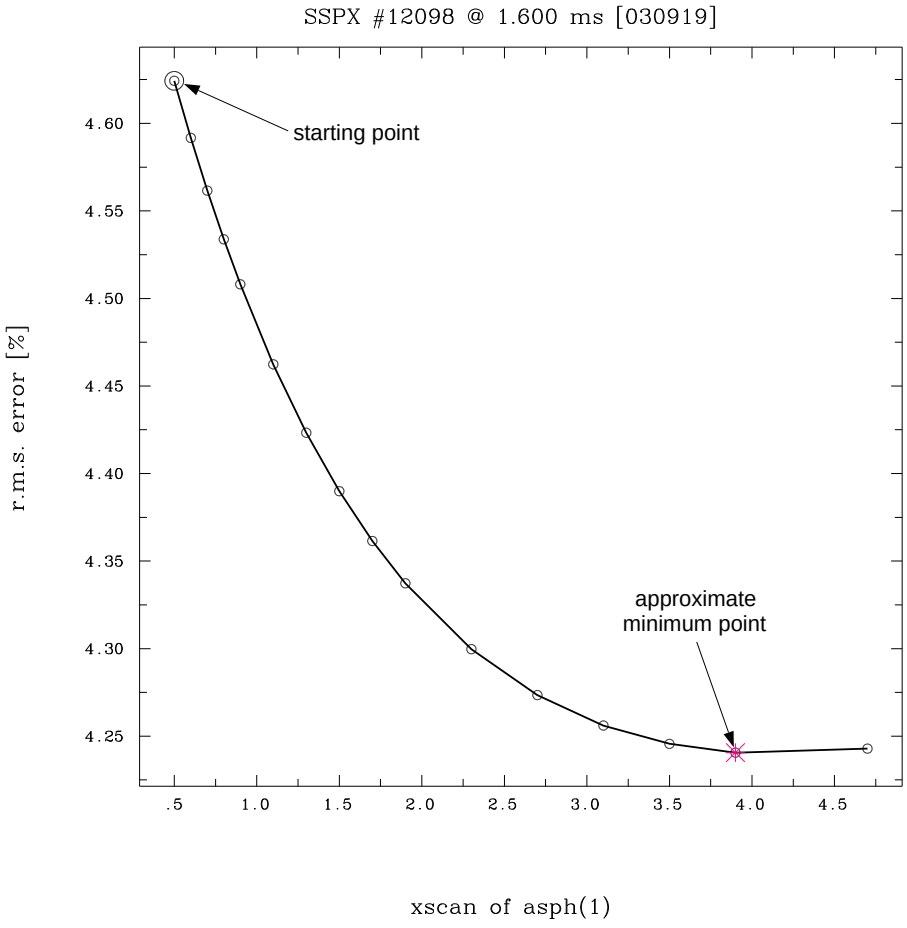

Figure 18: **Sample plot produced by `xscan` routine**, with an initial increment of `delta_x` = 0.1 for `asph(1)`.

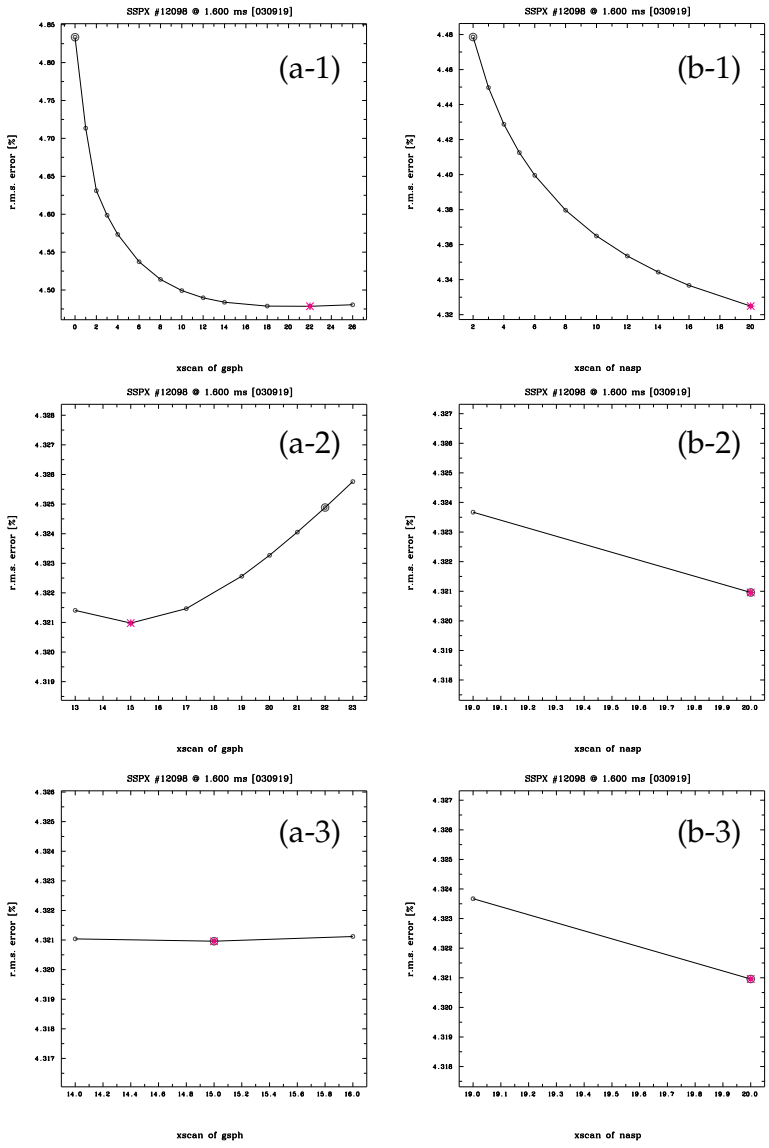

Figure 19: **Sample plots produced by `mscan` routine**, where `gsph` and `nasp` are varied until the fit error can no longer be reduced. The top row shows `gsph` $= 0 \rightarrow 22$ and `nasp` $= 2 \rightarrow 20$ (the upper-bound), with some reduction in the r.m.s. error. The 2nd pass (row 2) shows `gsph` being reduced back to 15 with small reduction in the minimum value, and the last pass verifies that (within the increments in `dgsph` and `dnasp`), a minimum has been bracketed.

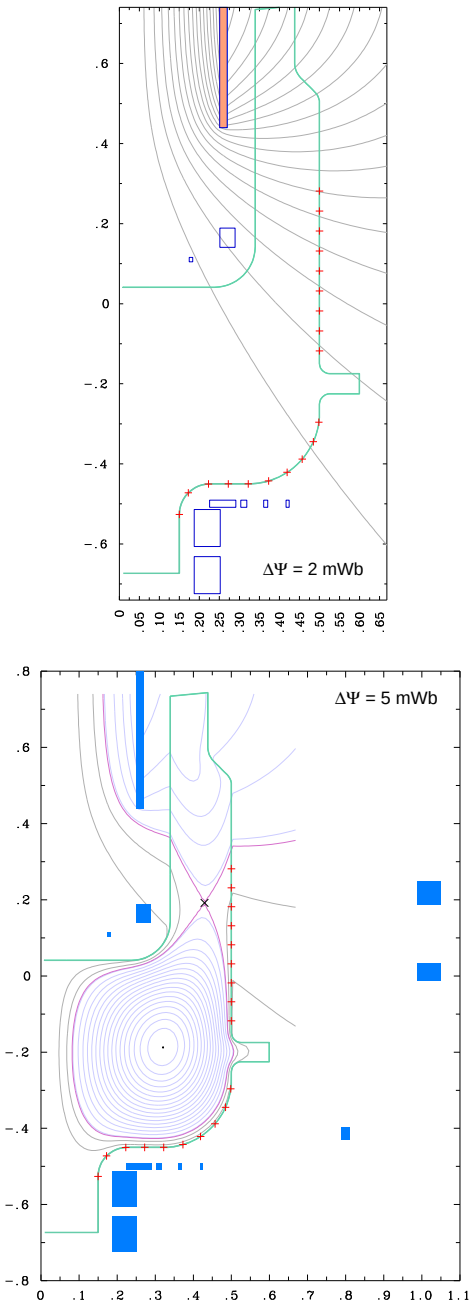

Figure 20: **Solenoid-only flux configuration**—vacuum flux (upper figure) with active bias coil shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_sol.sav`.

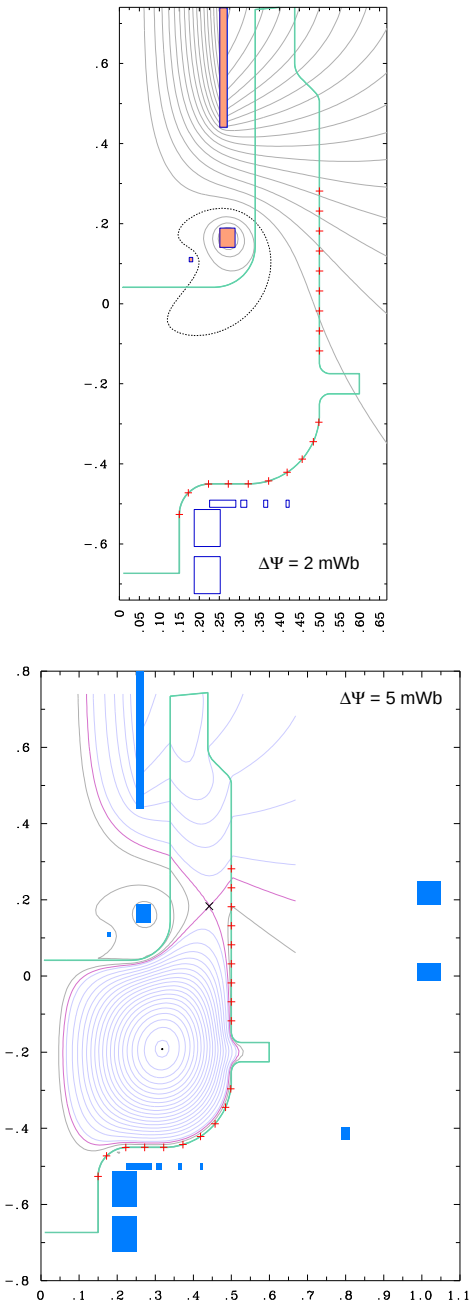

Figure 21: **Standard-flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_std.sav`.

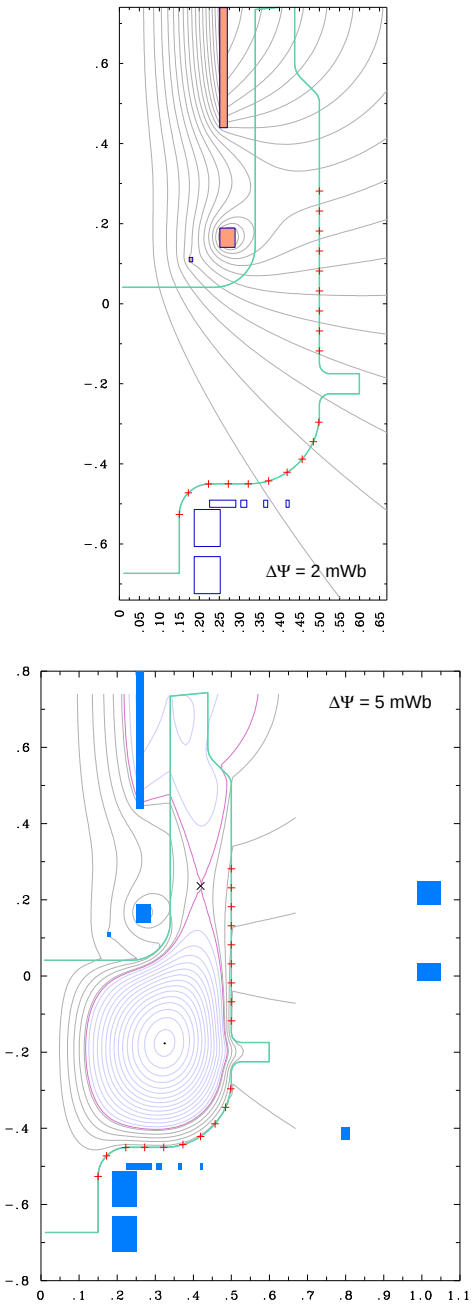

Figure 22: **Modified-flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_mf.sav`.

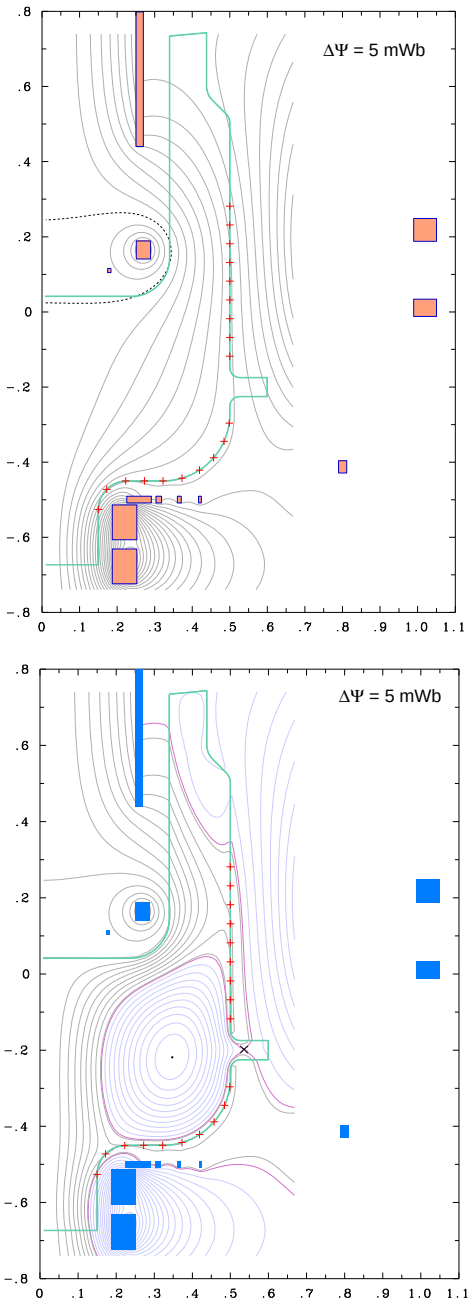

Figure 23: **Bias-coil-standard flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_bcs.sav`.

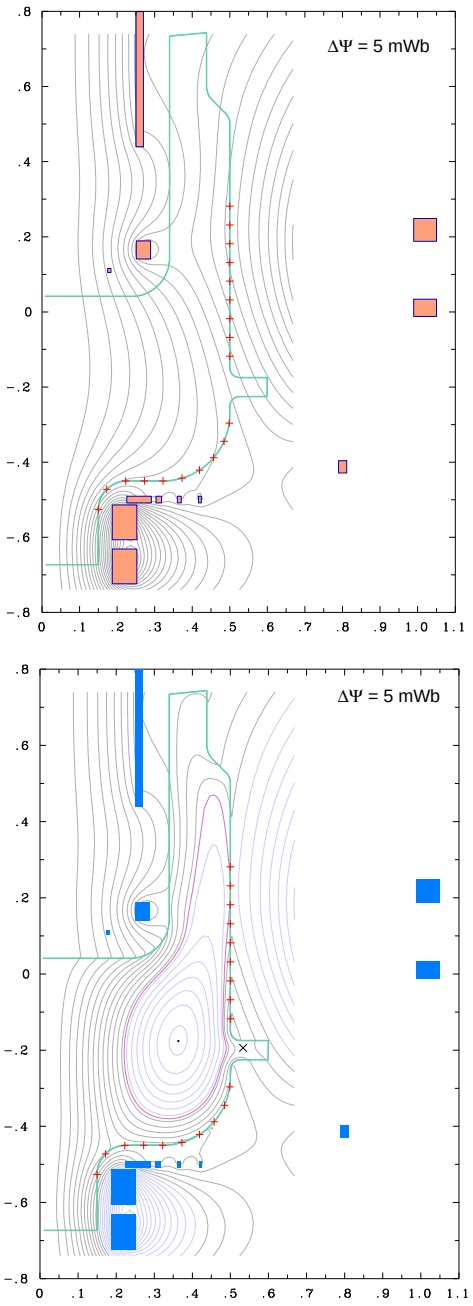

Figure 24: **Bias-coil-modified flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_bcm.sav`.

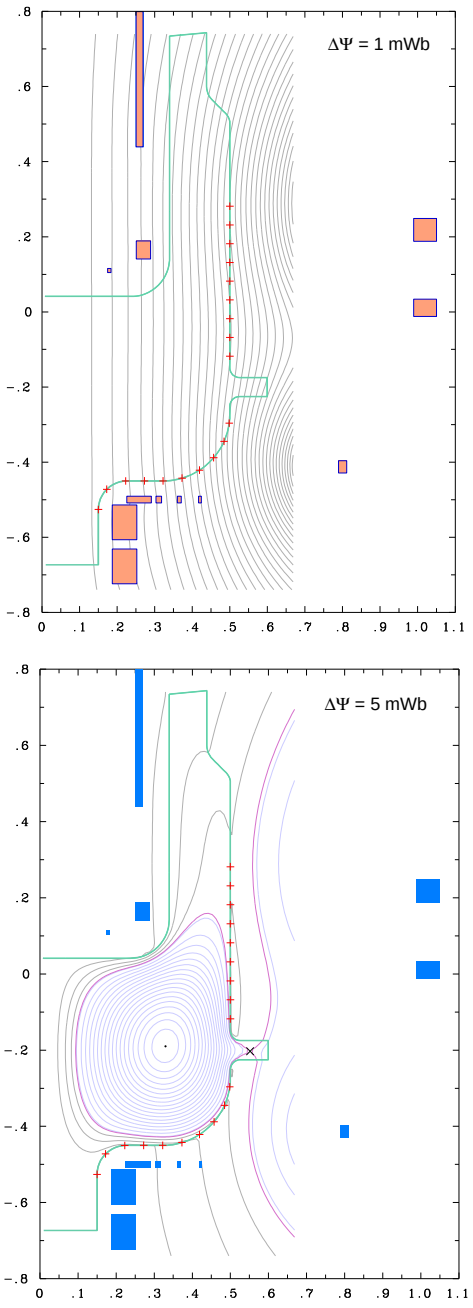

Figure 25: **Vertical-field flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corrresponding generic equilibrium (lower figure) from `sspx_bcv.sav`.

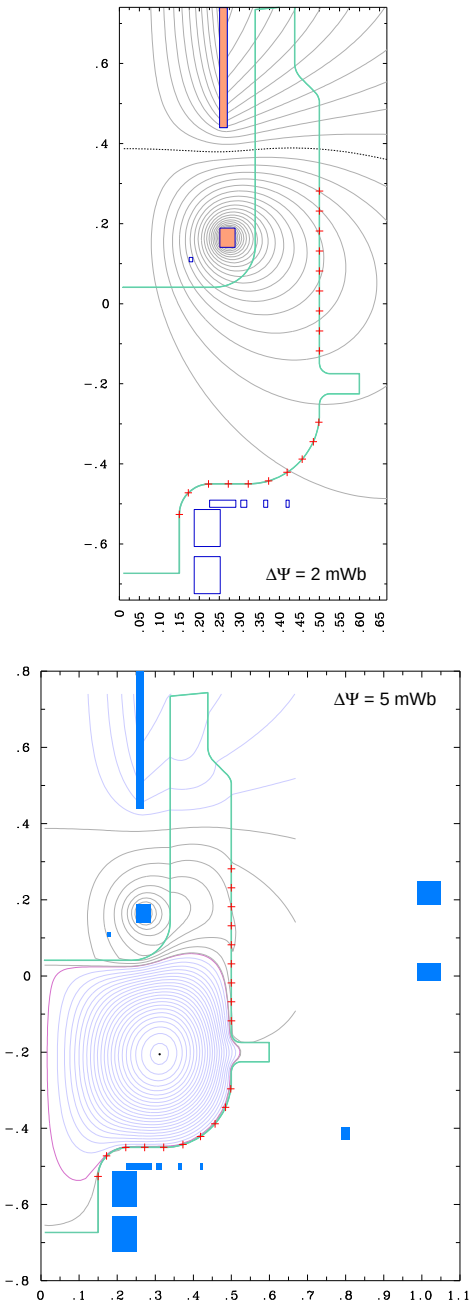

Figure 26: **Nozzle flux configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_noz.sav`.

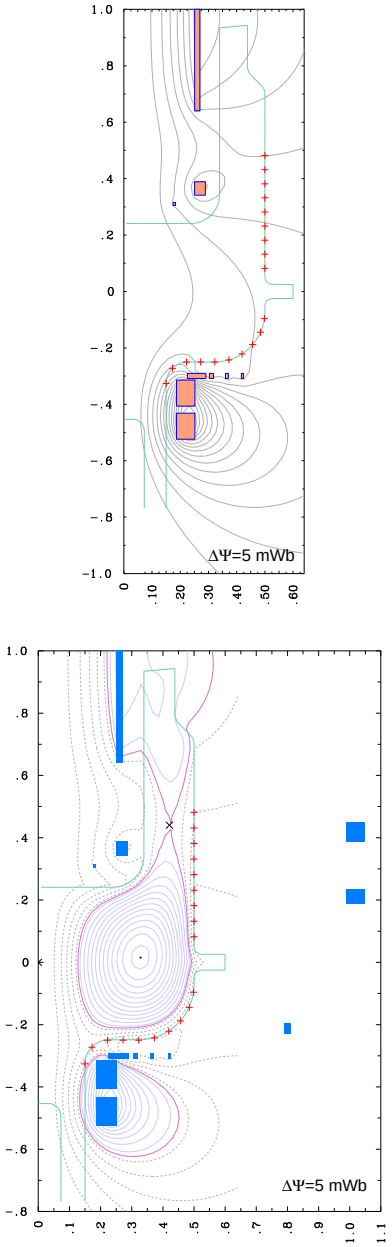

Figure 27: **Lower gun configuration**—vacuum flux (upper figure) with active bias coils shown with shaded cross-section and corresponding generic equilibrium (lower figure) from `sspx_lg.sav`. *Note: the coordinate system for this configuration has* $Z = 0$ *at the diagnostics slot.*

Index