SJP Pamela et al.

Improvements to the Flux Surface Handling Code FLUSH.

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

Improvements to the Flux Surface Handling Code FLUSH.

S.J.P. Pamela[1],
H.J. Leggate[2], D.C. McDonald[1,3], D.M. Harting[1],
D. Dodt[4], S. Wiesen[4], D.G. Muir[1] and JET contributors[*].

EUROfusion Consortium, JET, Culham Science Centre, Abingdon, OX14 3DB, UK.
[1] CCFE, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK.
[2] Dublin City University, Glasnevin, Dublin, Ireland.
[3] EUROfusion PMU, Garching, Boltzmannstraße 2, D-85748, Garching, Germany.
[4] Institut für Energie und Klimaforschung IEK-4, FZJ, TEC, 52425 Jülich, Germany.
[*] See Appendix of F. Romanelli, Proceedings of 25th IAEA Fusion Energy Conference 2014, Saint Petersburg, Russia.

ABSTRACT

On magnetic fusion facilities, it is often necessary to make use of the magnetic equilibrium obtained by fast analysis codes such as EFIT (on JET) or CLISTE (on AUG). Although equilibrium codes may provide a very advanced and precise magnetic equilirbium for a given plasma pulse/time, they do not provide tools for exploiting these equilibria. The FLUx Surface Handling code was initially developed by Nazareno Gottardi in 1989 for this purpose. Over the years, FLUSH has gone through various modifications, in particular those brought in 2006 to enable the use of FLUSH on various tokamak devices in the frame of the Integrated Tokamak Modelling project (ITM). This work brought forward the central part of the code, which converts a given magnetic equilirbium into a set of spline coefficients to be exploited by the user. The user interface, however, had since remained untouched, and most of the equilbrium exploitation functionalities, which made use of several central algorithms, were not generalised to deal with various tokamak devices. The present paper shows how the FLUSH code generalisation has been completed, and explains the improvements that have been brought to some of the central algorithms, in order to offer faster and more precise tools to the users.

Keywords: FLUSH, EFIT, Flux, Surface, X-point, Separatrix, LCFS, Grad-Shafranov, Line of Sight, Spline.

1 Introduction

1.1 The FLUSH Code

In tokamak fusion, the total magnetic field can be defined as $\vec{B} = F(\psi)\nabla\phi + \nabla\psi \times \nabla\phi$, where ϕ is the toroidal coordinate, $F(\psi)$ is the toroidal magnetic field strength, and ψ is the poloidal magnetic flux. Thus, magnetic field lines can be projected poloidally onto surfaces of constant ψ-values, defined as flux surfaces. Note that in toroidal coordinates, $\nabla\phi = R^{-1}\vec{e_\phi}$, so that $B_\phi = F/R$. The balance between the plasma pressure ∇p and the magnetic pressure $\vec{J} \times \vec{B}$ can be used to derive the Grad-Shafranov equilibrium [1], given by

$$\Delta^\star\psi = -\mu_o R^2 \frac{dp}{d\psi} - F\frac{dF}{d\psi} \tag{1}$$

where the Grad-Shafranov operator is defined by $\Delta^\star\psi = R^2\nabla \cdot (R^{-2}\nabla\psi)$, and $p = p(\psi)$ is the plasma pressure. More elaborate models exist to take into consideration plasma velocity, pressure anisotropy and other effects [2]. It is also important to note that, considering the fast parallel transport along field lines in fusion devices, the assumption $p = p(\psi)$ holds to some extent (ignoring turbulence and MHD instabilities).

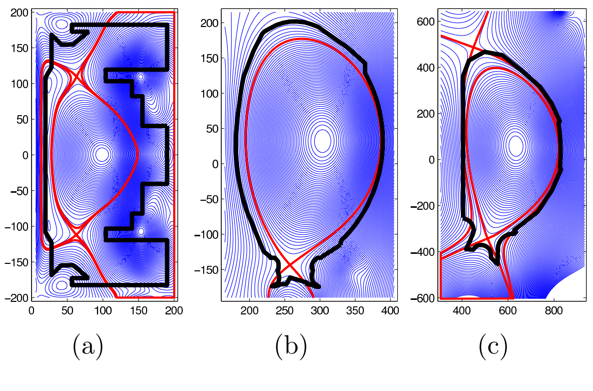

Figure 1:
Three examples of diverted plasmas:
(a) *MAST Double-Null Discharge (DND) #24763.*
(b) *JET Pulse #72569.*
(c) *ITER equilibrium (CORSICA).*

The magnetic equilibrium, here represented by the Grad-Shafranov equation, is solved for tokamak plasmas using various codes, depending on the device, such as EFIT [3,4], CLISTE [5], EQUINOX [6] or CORSICA [7]. Once an equilibrium has been computed, it may be used by other analysis codes to map various plasma profiles (eg. temperature, density, radiation etc.) between real space and magnetic space. In particular, such operations are extensively used by diagnostic codes that need to express collected data in terms of magnetic confinement (eg. to determine the position of a probe measurement compared to the edge of the magnetically confined plasma).

A few examples of tokamak magnetic equilibria are shown in Figure-1. The most basic magnetic configuration in tokamaks is the limiter plasma, for which magnetic surfaces are all cylindrical, and the edge of the plasma is determined by the surface that touches material components of the machine (the limiter). A more advanced magnetic configuration consists in diverting the plasma by creating a null point in the toroidal current, thus introducing an X-point in the magnetic topology. The surface that has an X-point is called the separatrix, and it may be considered as the edge of the confined plasma, beyond which plasma is transported directly to the first-wall (or divertor) along magnetic field lines. It should be noted that, although FLUSH can deal with double X-point configurations, it cannot yet handle snowflake configurations [8], which will not be discussed in this paper; consideration of snowflake configurations remains one of the generalisations required for FLUSH to be truly portable.

The FLUSH code may be considered in two parts: the core part, which produces the spline of a given magnetic equilibrium, and the exploitation part, which offers various sets of tools for the user. In order to present a clear understanding of the improvements brought to FLUSH in recent years, it is necessary to introduce those two fundamental parts of the code.

The central part of FLUSH consists in a set of actions taken during the initialisation of the code through the routine `flushinit`. In essence, there is only one input crutially required by FLUSH: the magnetic equilibrium itself. This needs to be provided by the user, or retrieved from some database, and for every new plasma equilibrium, FLUSH needs to be re-initialised. These actions, carried out during the initialisation, can be listed as follows:

(a) Clearing internal data.
FLUSH is often used for data analysis of whole plasma pulses, which may contain several thousands of different time slices, each of which has its own magnetic equilibrium, computed by EFIT or any other code. At JET, for example, the FLUSH splines are calculated once, for all time slices of a given pulse, so that a pulse/time equilibrium does not need to be splined several times by each analysis code that needs to use the equilibrium data. Hence, FLUSH caches some of its data, which may be used at the next initialisation, provided this data is compatible between the two equiliria. At the begining of the initialisation, compatible data sets are kept, while the rest is cleared for the new equilibrium.

(b) Reading the equilibrium data.
In the usual case where a new equilibrium is given as input to FLUSH, it needs to be extracted from its original format into standard FLUSH variables. This is one of the advantages for the FLUSH user: it is not required to provide data in one specific FLUSH format, rather FLUSH has been developed to read most conventional equilibrium formats from different institutes, such as EFIT data, CLISTE data, MAST idam data, text format data EQDSK, ITM data format, etc.

(c) Fitting splines to the equilibrium data.
Spline coefficients may be passed directly to FLUSH (as is often the case on JET), but the most standard process is to have a poloidal flux map over a spatial grid of coordinates. In this case, both for stability and speed purposes, the grid data is splined using NAG tools [9]. Such a spline is both representative of the initial equilibrium data, but also provides a smoothness which ensure well-behaved flux derivatives, which is not the case for a map of finite discrete resolution.

(d) Computing the main plasma parameters.
Once the equilibrium has been splined, some plasma characteristics are computed, mainly the magnetic axis, the X-point(s) and the separatrix value. These are then used to provide a normalised poloidal flux ψ_n to the user, such that $\psi_n|_{axis} = 0.0$ and $\psi_n|_{separatrix} = 1.0$.

The second part of FLUSH, which surrounds the central core described above, is the user interface, best described as the user's tools. These range from simple spline interpolation of the flux ψ at a given (R, Z) point, to complex calculations, such as performing flux surface averages of given functions. Effectively, most user routines make use of one central tool: getting intersections between a given line of sight and flux surfaces. Where this routine fails, generally the rest of the more elaborate routines will fail too. The larger part of these routines have been developed over the years, in order to answer specific needs of physicists or diagnosticians. Although the central flux surface intersection routine had been generalised at some point to include the X-point geometry, all routines were clearly hard-coded for JET use only. Also, an important aspect of FLUSH is that it is used by the software SURF, which provides an IDL-based GUI interface to plot flux surfaces in JET. This program needed to be reliable and fast, so that some of the routines developed in FLUSH had a clear emphasis on rapidity rather than precision.

1.2 Initial Improvements and Motivation on Further Improvements

The central part of FLUSH has been generalised in 2006 in order to accommodate the equilibrium to any given machine. In this improvement, several hard-coded parameters were generalised, such as the plasma dimensions, the physical units, or the spline parameters (eg. the number spline knots). In addition, FLUSH had previously used only the NAG library [9] to spline equilibria, which is not practical for code portability, since NAG is not open-source; one of the main tasks was to adapt FLUSH to a open-source spline library, PSPLINE [10].

This new core was written using the C-language (the previous FLUSH was fortran 77), making use of modules in a manner close to Object-Oriented architecture. In particular, different structures were used for distinct equilibrium aspects, such as the `metaData` for all prespline inputs and quantities, the `fittedData` for all post-spline quantities (eg. spline coefficients, X-point, axis etc.), and the `tokGeometry` for all device-related data, such as the divertor and the first-wall. These separated structures are especially useful when caching data over several initialisations, or just when using generic parameters internally for coding purposes.

Although the central part of FLUSH was, at that point, generic enough to be usable for any tokamak device, and for a wide range of input formats, the rest of the code, namely the user tools, were still strongly JET-oriented, if not simply JET hard-coded. Thus, FLUSH was still not usable for devices such as ITER, MAST or DEMO, or at least most of the more elaborate user routines were not. In fact, since 2006, the old FLUSH had not been replaced by the newer version at JET, simply due to lack of manpower to complete the development of the new FLUSH and ensure its reliability regarding large data analysis codes such as CHAIN1 [11-13].

There were other motivations initially to develop FLUSH further, from a practical and aesthetic point of view. From the user's perpective, the old FLUSH routines were often named in the Fortran77 fashion, with short names that did not clearly relate to the role of the routine, at least not to a FLUSH beginner. For example, the routine to get the flux ψ at an (R, Z)-point was called `flupn`, the routine to get intersections between flux surfaces and a line of sight was called `flul2`, the routine to get the tangent flux surface to a line of sight was called `flul1`, etc. For practical reasons, such routines would best be renamed as `getFlux`, `getIntersections` and `getTangents`.

From the FLUSH developer's perspective, the new FLUSH version was half way through, with half of the code written in C, the other half in Fortran77, sharing common blocks between Fortran77 and C. It is, in general, not good practice to rewrite a code in a new language just for the sake of the language itself. One would ideally provide wrappers to the old code in the new language. However, in the case of FLUSH, several routines did need rewriting, to generalise the code to more global tokamak use. Some routines in fact needed rewriting from scratch. Once these central tools were generalised, adapting the rest of the code to these new tools was straight forward.

Additionally, as pointed out above, some of the routines of FLUSH were strongly speed oriented, sometimes lacking in precision. This was the case, for example, of the flux surface routines, one of the main tools of FLUSH. First of all, there were separated routines to retrieve closed flux surfaces (inside the separatrix), open flux surfaces (outside the separatrix), and the separatrix itself. There was not a single routine to get surfaces anywhere. Although this could be achieved through a wrapper calling all three routines successively, there would still be some drawbacks. The first one is that the closed flux surface routine was approximating surfaces using a cubic spline, based on a dozen points along the surface itself. This was quick, but not at all precise. The other drawback was that the open flux surface routine needed a starting point for the surface, which was, in some cases, very limiting. At last, both the separatrix and the open surface routines required fix-sized arrays, with a maximum number of surface points. Dynamic allocation was not available in Fortran77, but there are now elegant tools to remedy to such issues.

The next sections describe how the FLUSH code was developed in the years 2010-2013. The next section-2 explains how the intermediate version of FLUSH, developed in 2006, was adapted to replace the old version of FLUSH at JET. It gives a detailed outline of the implications of modifying a public code at JET, and the testing and releasing procedures involved. The following section-3 focuses on the work done to generalise the FLUSH tools to any tokamak configuration. The several steps required to achive this are described successively. The last section-4 gives an overview of the work and discusses future improvements to be considered.

2 Replacing the Old FLUSH at JET

2.1 The Use of FLUSH at JET

The first release of the new FLUSH at JET, in 2008, required large modifications to the library. Most of those were legacy issues to make sure the library was backward compatible, but there were some cases where generic algorithms had been improved for speed/precision but now failed for some cases specific to JET. Nevetheless, the new FLUSH library was eventually released, but the important aspect of this release was not the modifications required to comply to JET standards. The main step forward was the set of tools that were developed to provide FLUSH with a high-level, user-friendly testing procedure. These two aspects (high-level and user-friendly) were at the centre of the development of the testing procedure: high-level, because FLUSH is extensively used at JET, and therefore cannot afford to fail; and user-friendly because the FLUSH developer position is a so called JET-Operating-Contract (JOC) position that is replaced at least every 4 years, so the testing procedure needed to be quick and easy to use. Ultimately, scripts were developed to enable a new FLUSH officer to release a new version of the library at JET only after reading approximately two pages of the manual and running a few sets of simple command lines.

In order to understand the implications of modifications in the public FLUSH library at JET, one needs to consider all codes that use FLUSH, and why they use FLUSH. For every JET experimental discharge, all data collected by diagnostics during the few seconds of plasma need to be processed, and made available to public users for analysis and, eventually, for publication. This data processing is done using a suite of codes called CHAIN1 [11-13]. One of the central codes of CHAIN1 if EFIT, which provides the equilibrium data for all remaining codes. Thus, unless using very basic data directly from EFIT (eg. magnetic axis position, strike point position etc.), numerous codes will rely on FLUSH to interpret the magnetic field equilibrium.

Currently, there are 51 codes using FLUSH in CHAIN1. A large part of those codes use FLUSH in a simple manner, just making use of the spline interpolation to retrieve the poloidal flux or the magnetic field at given (R, Z)-positions, but other codes make use of the more advanced routines. For example, many codes that measure radial profiles (of density, temperature or other) will call FLUSH to project these measures into flux-space, eg. to go from $n_e(R, Z)$ to $n_e(\psi)$. A few codes use even more complex FLUSH tools, to retrieve flux surfaces, compute geometric quantities (plasma elongation, trangularity etc.), or integrate plasma quantities over flux surfaces. If FLUSH is not functioning properly, a large part of the experimental data will not be available.

There are other codes than CHAIN1 that use FLUSH throughout JET, some of which are officially public, many of which are private. In practice, it is not possible to check that all codes relying on FLUSH at JET are working as expected. Hence, the FLUSH philosophy is to make use of CHAIN1's extensive reliance on FLUSH and to assume all FLUSH-using codes at JET are fine so long as CHAIN1 is reproducing data accurately.

The other aspect of FLUSH at JET is that wrappers to the user's functions are available in many languages, namely `FORTRAN`, `C`, `IDL`, `MATLAB` and, more recently, `PYTHON`. It is also available externally via `MDSPLUS`. These wrappers are also released publicly. Thus, some testing is also required for all wrappers.

2.2 Testing Procedure

There are several tests now available for FLUSH. Historically, the only test for FLUSH was a single test code that called all FLUSH routines with standard parameters, and dumped the output into files. The code was run for the public FLUSH library and the development library, so that output files could then be compared. In case everything was exactly identical, such a procedure was good, but there were no tools for measuring the extent of any differences. Comparisons between the public and the development library were not automated.

One of the first step in setting up a rigid testing procedure for FLUSH was to develop a `PERL` script to categorise and measure all differences from each FLUSH routine output. This required the classification of outputs depending on their units (eg. `cm`, `radian`, `Tesla`, normalised data etc.) in order to measure the weight of differences accordingly. For example, a difference of 1% for a distance of $5m$ is $5cm$, which is unacceptable (as far as JET is concerned), whereas a difference of 1% for an angle given in the range $[0, 2\pi]$ may be tolerable. Another particular case is if the relative weight of a difference has to be measured between zero and non-zero values, because the relative difference is infinite compared to the zero value. Hence, several class of outputs and difference measurements have been organised, and the `PERL` script returns to the developer a clear report with a section for each routine, giving the average difference and the maximum difference. Typically , the test is run over a set of standard JET discharges (eg. a limiter pulse, a divertor pulse, an old EFIT pulse, a pulse with a secondary X-point at the top of the plasma, etc.). The script also judges whether differences are acceptable or not. For example, if a routine gives spatial differences whose maximum is smaller than $1mm$, and whose average is smaller that $0.1mm$, then the routine is judged accurate for the new library. Similar upper limits are used for percentages. If all differences are judged negligible, the test will be judged successful, otherwise the developer will be pointed to the differences report.

The test described above is efficient when testing the global reliability of a development library, but is is not sufficient for testing whether codes like CHAIN1 will produce sensible output. Therefore, for completeness of the test procedure, another script was written to test all FLUSH-related CHAIN1 codes. In recent years, elegant testing procedures were developed by D.Dodt for CHAIN1, enabling developers to run accurate tests rapidly, and measure differences between data outputs for different CHAIN1 versions. The architecture of the `PERL` codes that were developed for CHAIN1 has been exploited in this FLUSH script, not only to give an appropriate test of the CHAIN1 codes (as done by the CHAIN1 test-scripts themselves), but also to make sure that the FLUSH script remains up to date with the CHAIN1 `PERL` architecture in the long term. This FLUSH script uses CHAIN1 functions to clone the latest versions of the CHAIN1 codes related to FLUSH, and run them using the public and the development FLUSH libraries. The two sets of outputs are then compared using CHAIN1 scripts.

2.3 Release Procedure

Although routine development does not require CHAIN1 testing, since the initial test script that compares all FLUSH routines outputs is quick and gives a good overview of the impact of changes, any release of FLUSH at JET has to be preceeded by a complete test, using both the generic FLUSH test and the CHAIN1 test. These are in fact not the only requirements for a release, which involves several other actions. The release procedure for FLUSH is the following:

(a) Do the tests
For any FLUSH release, both the standard FLUSH test and the CHAIN1 test need to be run.

(b) Update the release number
Each release comes with a minor/medium/major release number, eg. `libflush.so.1.2.3`, where `1`, `2` and `3` are the major, medium and minor release numbers respectively. These numbers are included inside the code, inside the release record file, and at the end of the names of the `.so` and `.a` files that are copied into the release directory to sent to the official public release on JET (ie. copied to the `/usr/local` directory).

(c) Update `SVN`.
FLUSH is currently under `SVN`. The `SVN` revision number is included in the code, and in the release record file.

(d) Recompile code with updated release number.
Since both the `SVN` revision number and the release number are included in the code, the code needs to be recompiled with those new numbers before release.

(e) Check for uncommitted files.
Some of the files are always committed to `SVN` during a release, such as the release record file and the `version.c` file, which contains the release version number and the `SVN` revision number. In case other files have a non-standard `SVN` status (eg. modified "M" or added "A") they need to be committed.

(f) Final commit.
All files, including the release record file and the binary copies of the library are committed to `SVN`.

For security (ie. to ensure that all actions are delivered) as much as simplicity, all these actions have been integrated into a single `PERL` script. Note that since the code and the release record file, which both contain the `SVN` revision number, have to be committed to `SVN`, the upper `SVN` revision number has to be used for those files (ie. `svn_rev`+1). The script has been adapted so that even someone new to FLUSH (ie. not doing the release properly) will be prompted with enough information to complete the release eventually. The only thing that needs to be done separately by the person doing the release is the testing procedure, which is also explained by the release script in case the test results are missing. Note that this is the only input to the release script: the locations and names of the test summaries. Thus, executing the release simply looks like

```
release_flush.perl standardtest.txt chain1test.txt
```

The general practice is that most modifications of the FLUSH library will not result in failures of the standard and CHAIN1 tests. Currently, the majority of FLUSH issues have been dealt with, so that only special cases remain, each of which generally does not affect JET data as a whole, but only a small fraction of the data. As an example, a typical JET pulse comprises 10,000 EHTR time-slices (ie. 10,000 Grad-Shafranov equilibria), where EHTR is simply EFIT at high time resolution. The sort of special cases that now have to be dealt with in FLUSH occur perhaps for a few hundred time-slices per experimental campaign (say at least 1000 pulses). Hence, these special cases affect data with a frequency of the order of 10^{-3}%. Nevertheless, these are spread out over time and regularly require FLUSH releases.

Thus, most corner cases do not lead to failures of the standard and CHAIN1 tests, but those tests were particularly efficient when large modifications were brought to FLUSH in recent years. In fact, these tools enabled the evaluation of the efficiency of the development library compared to the old library, or in some cases, the inefficiency of the old library compared to the new one. Test reports that can point to a specific library Application Program Interface (API) provide direct information on the flaws of algorithms being tested. However, differences may occur because the newer library is more precise than the old one, in which case a release is done using tests that have failed, which usually requires clear explanations in the FLUSH records and `SVN`.

3 Completing the FLUSH Generalisation

3.1 Improving the User Interface

The first aspect of the generalisation of the FLUSH routine was to make the library user-friendly. There are numerous cases to which simple conventions were applied, and in order to describe these conventions, a few representative cases will be used. The first one is the main FLUSH routine, `flushinit`, for which the standard `FORTRAN` call is

```
call flushinit(igo, shot, time, lget, seq, uid, dda, lmsg, ier)
```

where we list and explain the arguments here:

(a) igo
Integer determining the way FLUSH will be initialised (ie. using JET, MAST, eqdsk, NAG or other splines etc.) See Flush-manual for full list and details.

(b) pulse
The pulse number of the experiment.

(c) time
The time at which equilibrium data is required.

(d) lget
`FORTRAN` entry file number for data input (eg. 3 for `fort.3`).

(e) seq
JET PPF-data [14] sequence number (in case JET equilibrium data is required).

(f) uid
Identification name under which PPF-data requested has been written (eg. `JETPPF` or `flush` or any).

(g) dda
Name of the PPF-data DDA requested (eg. `EFIT` or `EHTR` or any).

(h) lmsg
`FORTRAN` output file number for any output data (eg. 6 for `fort.6`).

(i) ier
Error identification number.

The sole purpose of introducing this call here is to clearly present the level of intricacy required simply to obtain some equilibrium data. The `lget` and `lmsg` have long been obsolete but cannot be removed due to backward compatibility. Currently the `igo` parameter may take up to 28 different values between -2 and 92 (one of the JET values is 15). Of course, all cases need to be included, different machines, different data inputs, different splines, different initialisation levels etc. Ultimately, this `igo` parameter should be broken into separate routines, such as

`Flush_setMachine`,
`Flush_setSplineType`,
`Flush_setUID`,
`Flush_setDDA`,
`Flush_setSeq`,
etc.

to be called before the initialisation call, in case a user does not want to use the default settings. This was unfortunately not done at the time of development, due to some disagreements among the team members regarding the true usefulness of such a modification, and how it should be formulated. In the meantime, for JET users, this crucial call was simplified to,

`flushQuickInit(pulse, time, ier)`.

Of course, this is only valid for JET, but it could easily be generalised to any common default settings for each machine. Eventually, breaking up the `igo` parameter into different routines should really be done.

The main FLUSH routine `flushinit` represents perfectly the sense of clarification that was desired by FLUSH users. In addition to this isolated simplification, several other conventions were applied to the code:

(a) All FLUSH API's should have the prefix "`Flush_`"
In order to avoid any clash with other public libraries, and to make FLUSH calls easily pinpointable when debugging users codes, all new user routines implemented where named with the prefix "`Flush_`", such as `Flush_getFlux`, or `Flush_getMagAxis`.

(b) Users APIs to have self-explanatory names
Some old FLUSH APIs were not intuitive for someone who was not intimately working with the code. For example, `flupx` was renamed to `Flush_getXpoint`, `flusu2` was renamed to `Flush_getClosedFluxSurfaces`, etc. Note that the APIs `flupx` and `flusu2` still exist, but they are now wrappers to the newer routines. In fact `flusu2` is a good example for this, since it is now a wrapper to `Flush_getClosedFluxSurfaces`, which finds flux surfaces in a manner entirely different to what was done in the old library.

(c) Users APIs to return an error number
All user calls should return some error number `ier`. Returning a character string that explains the error is too complex, especially when dealing with different levels of wrapping for different programing languages. Likewise, printing the error character string directly on the user's prompt could result in pollution of code output. Instead, integers are used, and the user may then call the routine `Flush_getError(ier)`, which will print the error identification name and information related to the library failure.

(d) No maximum array size
Several old FLUSH routines needed size-limited arrays. For example, `flseparatrix` and `flouter`, which were used to retrieve the separatrix surface and outer flux surfaces respectively, could not determine the number of surface points (which depends on the accuracy required) before they were called. This is still the case, but now allocatable pointers are sent to FLUSH and memory is internally allocated accordingly.

(e) No combined APIs
A large number of the old FLUSH APIs were combinations of other APIs. For example, `flupn` was used to get the poloidal flux ψ at multiple points; `flupn2` to get ψ and the poloidal magnetic field components B_R and B_Z; `flupn3` to get ψ, B_R, B_Z and the toroidal field B_ϕ; and `flupn4` to get ψ, B_R, B_Z, B_ϕ and the current field components j_R, j_Z, j_ϕ. The new APIs only retrieve one quatity at a time, to avoid long lists of routine arguments, so that functions such as `Flush_getFlux`, `Flush_getBr`, `Flush_getBz` etc. are now used. The user is then free to create his own wrappers to combine different APIs together.

Although such conventions do not in any way justify rewriting a working library, they are only the forefront of some more significant modifications needed in the code to ensure reliable performance on all types of magnetic equilibria.

3.2 Flux Extrema Along Lines of Sight

One of the main tools, available to the user, but also used extensively internally in FLUSH, is the routine that finds intersections between a given line of sight and given flux values. The line of sight (l.o.s.) is typically given as a spacial point and an angle, such as (R, Z, α). For a plasma that has a co-injected current, the poloidal flux is monotonically increasing from the magnetic axis up to the separatrix and beyond, except below a lower X-point (or above an upper X-point), in the so-called private region, where ψ decreases again. Hence, given any flux-value ψ_o, there may be up to 2 intersections with the l.o.s., if the later goes through the core plasma, but does not enter any private region (and if $\psi_o <= \psi_{xpoint}$). If the plasma has a lower X-point and the l.o.s. also crosses the the private region below that X-point, then there may be up to 3 intersections. Likewise, if the plasma has a lower and an upper X-point, then there may be up to 4 intersections.

In order to find all intersections, the extrema along the l.o.s. first need to be found. Once all minima/maxima have been found, since ψ is monotonically increasing between each minimum/maximum pair, finding a flux-value ψ_o is a simple iterative process using the stepping increment δl along the l.o.s. given by:

$$\delta l = \frac{(\psi_o - \psi_i)}{\frac{d\psi_i}{dl}}$$

where the i subscript represents evaluation at the previous position found on line.

In the previous FLUSH version, only the standard case of a conventional tokamak with at most two X-points was treated, so at most one minimum and two maxima along any line. This treatment is rather robust for JET, but it is not adequate for any ψ-domain that also includes poloidal field coils or X-point coils, which results in other localised minima outside the plasma (eg. Figure-1a). If all extrema along the l.o.s. cannot be found accurately, then finding all requested ψ-values along that line may prove very tedious. Thus, the tool used to find extrema along lines had to be generalised.

The algorithm implemented considers the two ends of the l.o.s. on the spline domain. There can be two cases, as shown in Figure-2. If the ψ-gradient is of the same sign at both ends of the line, then there will be either no extrema between those two ends (black line), or at least one pair of minimum-maximum (blue line). Hence depending on which end has the lowest flux, those two extrema are found by stepping along the line starting from both ends. Once these have been found, the same procedure can be applied again, since there will be either no extrema between those two new points, or at least one pair of minimum-maximum (red line). The algorithm is applied until no more extrema are found between the new pair of minimum-maximum.

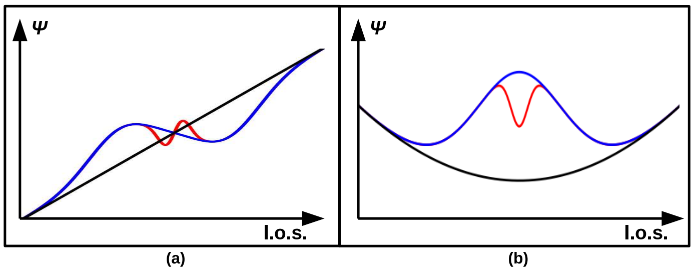

Figure 2:
(a) *Line of sight with ψ-gradients of same signs at both ends of the line. The flux ψ will have either no extremum on the l.o.s. (black case), or one pair of minimum-maximum (blue case), or two pairs of minimum-maximum (red case), and so on.*
(b) *Line of sight with ψ-gradients of opposite signs at both ends of the line. The flux ψ will have either one extremum (black case), or three (blue case), or five (red case), an so on.*

Likewise, if the ψ-gradients at the two end points are of opposite sign, then there will be at least one extremum along the line (black line). By stepping along the l.o.s., starting from both ends, if both extrema found are at the same location, then the unique extremum has been found, else it means that there must be at least another extremum between those two new points (blue line), so the same procedure can be applied again (red line), over and over until two matching extrema are found.

The algorithms used to find each extrema when stepping along the line from one end are a mixture of Newton-Raphson and Steepest-Descent/Ascent [15,16]. If the first derivative of ψ and its non-zero second derivative are of opposite sign, then Newton-Raphson will be faster, otherwise steepest descent/ascent is used. However, if Newton-Raphson is used but the extremum being searched for is almost (or close to) a saddle point, or if there is a change of sign in the second derivative of ψ on the way to the extremum, then the algorithm must switch back to a steepest descent/ascent.

3.3 Finding Flux Surfaces

There were three different tools to find flux surfaces in the old FLUSH library. These tools will be described in detail to justify the new flux surface tool that was developed for the new library.

The first tool, `flusur`, was used to retrieve closed flux surfaces, inside the separatrix. The method used was to find the flux intersections for each surface with lines of sight originating from the magnetic axis, at different angles. Ten lines of sight were taken at equidistant angles in the range $[0, 2\pi]$ for circular cases, and in the range $[\alpha_{xpoint}, \alpha_{xpoint} + 2\pi]$ for X-point cases (α_{xpoint} being the angle between the X-point and the magnetic axis). The intersections found along the surfaces for these l.o.s. were then used as a basis for a cubic spline to interpolate as many surface points as required by the user. The advantage of this routine was its speed, but regardless of the number of points the user would ask for, the precision was always the same, restricted to the cubic spline interpolation. In addition, had one l.o.s. intersection not been found, or found with an error, the cubic spline interpolation would result in oscillations around the mistaken intersection.

The second tool, `flseparatrix`, was used to retrieve the separatrix surface. This was done in two steps. The first step was to get the closed part of the surface, using the tool described just above, but with 15 l.o.s. instead of 10 for the spline interpolation. The second step was to retrieve the legs of the separatrix, connecting the X-point to the divertor targets. This was done by taking a set of horizontal lines of sight between the X-point and the bottom of the FLUSH spline domain, which were used to find intersections with the flux value ψ_{xpoint}. Although this works for most JET cases, it is not acceptable for many other configurations, or for larger domains where the separatrix legs would not be approximately vertical.

The last tool, `flouter`, was used to retrieve open flux surfaces, outside the separatrix. This was done by giving a point outside the separatrix as input to the routine, which would then follow the flux contour with the requested accuracy until it reached the edge of the domain. The contour following was done using the Ralston and Runge-Kutta methods [17,18]. Although this method is rather efficient, it requires an input point to start the surface, and assumes the surface will eventually intersect the edge of the domain. In other words, the user has to know where the surface starts and make sure it is an open surface. Another issue with this method is that all surfaces outside the separatrix are not necessarily open, some of them are closed (eg. around PF-coils).

The three methods described above were rather efficient for standard JET cases. For many other plasma configurations, however, they would fail. Firstly because the closed surface and separatrix methods required finding intersections with lines of sight, which could fail unless the flux configuration was very close to that of EFIT at JET. Secondly because finding the separatrix legs requires those legs to be approximately vertical and intersecting the edge of the domain. Thirdly because the outer flux surface method, although based on a powerful contour following algorithm, was restricted to well defined cases of surfaces going around the main plasma and intersecting the domain.

In order to provide the user with a rapid and robust tool for flux surface interpolation, a new method was developed, based on a discretization of the flux domain. The flux domain is discretized into a rectangular grid, the size of which is typically `5cm` for JET plasma, but may be reduced or increased. At present the size of the grid elements is set by the FLUSH developer, but making it available to the user would be straight forward. As will be shown later in this section, a `5cm` grid is typical for JET and provides enough accuracy and robustness while remaining fast enough for rapid execution.

The grid elements are not all equally sized, they are constructed so that the magnetic axis lies on a grid node (or element corner) and so that any X-point lies in the centre of an element. This ensures that small, closed flux surfaces near the axis may be resolved, and that the four surface lines near an X-point may be found accurately without being too close to one-another.

Each grid element is then treated individually. The first step is to find the minimum and maximum flux values ψ_{min} and ψ_{max} for each element. Then, if the user requested a flux surface with values ψ_o, and if that value belongs to the interval $[\psi_{min}, \psi_{max}]$, intersections of that ψ_o-value are found along the sides of the element. The flux surface piece inside this element is then interpolated according to the requested accuracy. Finally, for a given flux surface, all surface pieces among the grid elements are joined together in the right order and returned to the user. The ordering of surface pieces is done by stepping from piece to piece on adjacent grid elements.

The interpolation method used between intersections of a surface and an element sides uses an inverse Taylor series expansion of second order, averaged between both intersections. Let (R_1, Z_1) and (R_2, Z_2) be the two end points (or element intersections) of the surface piece. Around the location (R_1, Z_1), the 2D Taylor expansion at second order of the poloidal flux ψ is given by:

$$\begin{aligned} & \psi\left(R_1 + \delta R, Z_1 + \delta Z\right) \\ = \; & \psi\big|_1 \\ & + \psi_R\big|_1 \delta R + \psi_Z\big|_1 \delta Z \\ & + \frac{1}{2}\left[\psi_{RR}\big|_1 \delta R^2 + 2\psi_{RZ}\big|_1 \delta R \delta Z + \psi_{ZZ}\big|_1 \delta Z^2\right] \end{aligned}$$

where the R and Z subscripts mean partial differentiation, and $\big|_1$ means evaluation at (R_1, Z_1). Note that since the aim here is to step along the surface of a given ψ-value, on the surface, the above formulation leads to

$$\begin{aligned} & \psi_R\big|_1 \delta R + \psi_Z\big|_1 \delta Z \\ & + \frac{1}{2}\left[\psi_{RR}\big|_1 \delta R^2 + 2\psi_{RZ}\big|_1 \delta R \delta Z + \psi_{ZZ}\big|_1 \delta Z^2\right] \\ & = 0 \end{aligned}$$

Then, assuming the flux surface piece is not close to vertical, the interpolation is done using the incremental steps $\delta R = x$ between R_1 and R_2 (note that if the line is close to vertical, then the same method is applied for $\delta Z = x$ instead, for which the following analytical formulation is very similar). Typically the step size x increases at each iteration with the surface spacial resolution required by the user. Substituting $\delta R = x$ into the above formulation, and rearranging, gives the second order polynomial equation

$$A\delta Z^2 + B\delta Z + C = 0$$

with

$$\begin{aligned} A = \; & \frac{1}{2}\psi_{ZZ}\big|_1 \\ B = \; & \psi_Z\big|_1 + \psi_{RZ}\big|_1 x \\ C = \; & \psi_R\big|_1 x + \frac{1}{2}\psi_{RR}\big|_1 x^2 \end{aligned}$$

Note that for a first order approximation (ignoring all second derivatives), the above reduces to the solution $\delta Z = -x\psi_R/\psi_Z$. Thus, since in the case $A \neq 0$ and $B^2 - 4AC >= 0$, there are two solutions to the above quadratic equation, the solution of choice is the one closest to this first order approximation $\delta Z = -x\psi_R/\psi_Z$ provided $psi_Z \neq 0$, otherwise the one closest to the straight line between the two end points of the surface piece, (R_1, Z_1) and (R_2, Z_2). The first order approximation $\delta Z = -x\psi_R/\psi_Z$ is also used for the case $A \sim 0$ (with the estimate that $A \sim 0$ if $|A| < 10^{-14}$), and if $\psi_Z = 0$ then the step is taken along the straight line between the two end points of the surface piece. The remaining cases are if $B^2 - 4AC < 0$ or $A \sim B \sim C \sim 0$: here too, the step is taken along the straight line between the two end points of the surface piece.

Since a similar interpolation can be done at the other end points (R_2, Z_2) of the surface piece, a weighted average is done between the two interpolations δZ_1 and δZ_2, such that

$$\delta Z = (1-\beta)\delta Z_1 + \beta\delta Z_2$$

where

$$\beta = \frac{1}{2}\left[1 - cos\left(\frac{\delta R}{R_2 - R_1}\pi\right)\right]$$

This weighted average ensures the solution is not polluted by the interpolation from (R_2, Z_2) if the step is taken very close to (R_1, Z_1) (and inversely).

Now, the standard case on a given grid element is that there are no or only two intersections corresponding to a given flux value, but this may not be true near an X-point, where there can be four intersections. For robustness, the rectangular grid is always built so that X-points are approximately at the centre of a grid element, so that the four intersections on that element will be well defined and not too close to one another. The position of the four intersections relative to the saddle point formed by the X-point is then used to determine which intersections should be joined together. For the separatrix, the two X-point surface pieces are each divided into two pieces meeting at the X-point itself. Of course, at the X-point, all first derivatives are zero and second derivatives are close to zero, so the interpolation is done only from the other end of the surface piece, not from the X-point end.

It should also be noted that four surface intersections on a grid element may also be found for closed surfaces away from any X-point. These are also treated separately. In addition, to ensure that surfaces with $\psi \sim \psi_{axis}$ are always well resolved, the rectangular grid is constructed such that the magnetic axis always lies on a grid node (element corner). This is done to avoid having a small flux surface enclosed inside an element; this way however small the closed surface is, intersections will always be found to interpolate the surface points.

There are several advantages to this new flux surface tool, when compared to the old tools. The first advantage is that there is no distinction between closed, X-point, and open surfaces, so that the user may give any ψ value and retrieve the surface whatever its topology. Another advantage is that the user does not need to know where surfaces start outside the separatrix in order to retrieve them. For example, say a given ψ value has an open surface going around the plasma, and another surface (not connected to the first one) somewhere else in the domain, the new tool will find this surface automatically from the discretization, whereas using the old tool `flouter`, the user would need to know at least one point of that surface to retrieve it. A third, important advantage is that the new tool is more precise, especially concerning closed surfaces and the separatrix, since the old tool was restricted to a cubic spline with maximum 15 knots per surface, whose coefficients had nothing to do with the background ψ-map. The new flux surface tool also has a restricted grid resolution, but the inverse Taylor expansion described above is coherent with the ψ-map, and thus ensures a precision that was lacking in the old tools. There is, however, one disadvantage to this new tool: it takes an accuracy as input, not a number of surface points, so the user will not know the exact number of surface points until the surface has been retrieved. Therefore, the old tool using the cubic spline has been kept to give this choice to the user.

The other consideration of interest, when comparing this new flux surface tool to the old one, is speed. The old closed flux surface tool only had 15 pairs of intersections (2 per l.o.s.) to find, and cubic spline coefficients to calculate, the rest was just spline interpolation. The new tool first has to determine the minimum and maximum flux of each grid element, after which it needs to find the surface intersections on each individual grid element. Once this is done, the surface pieces corresponding to each element still need to be reordered and finally interpolated using the inverse Taylor expansion. Of course, the minimum/maximum values on each grid element is calculated only once and then cached, in case the tool is used again later. The computation time effectively increases as the grid resolution is increased, hence the appropriate resolution should be found such that the tool remains precise, but computation time is not penalising. It was found that for JET a resolution of `4cm` ensures robustness of the tool, while resulting in resonable timing.

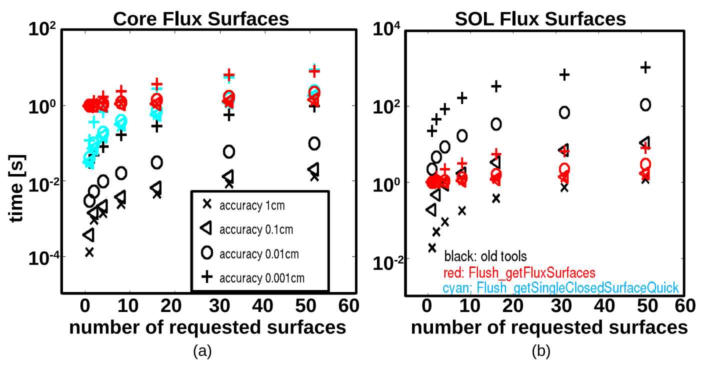

Figure 3:
(a) *Comparison of computation time for core flux surfaces tools. A scan is done over the number of flux surfaces, at 4 different levels of accuracy. The black signs show the old tool timing, the red signs show the new standard surface tool, while the cyan signs show the timing for an alternative tool, that follows surfaces along the grid, improving computation speed for low numbers of surfaces. The old tool, which finds 10 points for each surface and then interpolates the rest using a cubic spline, is clearly faster, although the difference becomes smaller at refined accuracy and large numbers of requested flux surfaces.*
(b) *This plot shows the same scan as (a), for SOL flux surfaces tools. The black signs show the old tool, the red ones the new tool. In this case, the new tool is clearly faster, except at low accuracy, low numbers of requested surfaces.*

Nevertheless, there is no comparison with the old closed flux surface tool, which is approximately 100 times faster. For SOL surfaces however, the timing is favorable to the new tool, which is up to 100 times faster, depending on the requested accuracy. Thus, between the core and the SOL, the new surface tool is on average as fast as the old one. To remedy to the slower aspect of the new tool for closed flux surfaces, an alternative is to find a point belonging to the requested surface, and follow the surface along the grid, from element to element. This way, not all grid elements need to be analysed and the computation time is improved (so long as the requested number of flux surfaces is small). The API for this alternative tool has been labelled `Flush_getSingleClosedSurfaceQuick`. Figure-3 shows the timing of the old and the new surface tools, for various accuracies and various numbers of flux surfaces. It should be noted that the old surface tools were restricted to 51 flux surfaces per call, which is the largest number shown in this scan.

It should be noted that the new flux surface tool is relatively steady in speed, for all accuracies and any flux surface number, which means that most of the computation time is spend on the analysis of the grid elements, prior to finding the flux surfaces. Another scan was performed for different surface grid resolutions. The results are shown in Figure-4. Effectively, reducing the resolution improves the computation time, although for high numbers of requested surfaces, the timing is very similar for the accuracies of `4cm`, `8cm` and `16cm`. Thus, it can be concluded that a better resolution should be retained, for the sake of accuracy and robustness. In fact, Figure-4b shows flux surfaces for a typical JET pulse, demonstrating that even with a resolution of `8cm`, surfaces are accurately reproduced as with the resolution of `4cm`, however, at the `16cm` resolution, some differences start to appear, and in some places (particularly close to the X-point), the tool starts to fail and result in non-smooth surfaces. For JET, the resolution of `4cm` is used, and has proved to be robust for the last 4 experimental campaigns C29-C32. The robustness of a `4cm` grid should nevertheless be tested in the future for other machines; it is expected that for a device of the size of ITER, a larger grid could be accurate enough, while for smaller machines, a tighter grid might be required. Eventually, adjusting the grid size dynamically with respect to the domain dimensions might be the best option, although this should be open for discussion.

Another aspect of this flux surface development was whether publicly available contour tracing codes could be used in FLUSH, rather than developing an internal set of tools to retrieve flux surfaces. However, most open source codes that provide fast contour mapping tools, such as the ROOT function `TGraph2DPainter::GetContourList` from CERN [19], use a similar method but with linear interpolation between end points. In other words, using a high-resolution grid makes linear interpolation reliable. In the present context, linear interpolation means zeroth order interpolation with respect to the inverse Taylor interpolation described above (ie. straight line between the end points). The advantage of using second order interpolation is that a low-resolution grid can be used, which is faster. In future, if it is estimated that linear interpolation with a higher resolution grid should be used instead, then the tools now available in FLUSH can be used simply by ignoring the inverse Taylor interpolation.

In summary, the new flux surface tool is useful for retrieving large numbers of flux surfaces everywhere in the plasma. There is no distinction between core, separatrix and SOL. When retrieving large numbers of surfaces in all regions for a high accuracy, the speed is approximately the same as with the old tools. In addition, the new tool is more precise in the core and at the separatrix, and more robust in the SOL. At last, the new tool may be used for any plasma configuration, whereas the old tools were significantly oriented for JET and not reliable on domains different to the usual EFIT configuration at JET.

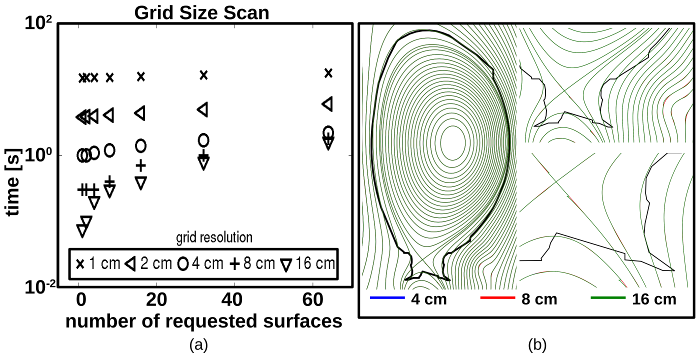

Figure 4:
(a) *Computation times for different surface grid resolutions. Timings are taken for the surface accuracy of 0.01cm, for various numbers of flux surfaces. For grid resolutions below 4cm, the computation time is dominated by the grid analysis, for resolutions above 8cm, the computation time is dominated by the surface finding and interpolation..*
(b) *This plot shows surface contours for a typical JET discharge, using three different surface grid resolutions, 4cm (blue), 8cm (red), and 16cm (green). Although the surfaces are perfectly aligned for the 4cm and 8cm resolutions, the surface tool fails in some places for the 16cm resolution, particularly near the X-point, where flux derivatives are close to zero.*

3.4 Finding the LCFS

The last aspect of improvements brought to FLUSH concerns the determination of the Last Closed Flux Surface (LCFS) at initialisation. In order to describe in detail the improvements required, the process of determining the LCFS in a plasma should first be explained.

Assuming the plasma is surrounded by a wall, we may have an X-point plasma, if the X-point surface is totally enclosed within the wall; or a limiter plasma, if there is no X-point surface or if the X-point surface is not entirely enclosed within the first wall. The ψ-value of the LCFS may be determined by stepping along the wall and recording ψ-values on wall points: the LCFS will be the lowest ψ-value along the wall (or the highest in case of counter-injected current). This process is straight forward, except if there is one or several X-points inside the wall, in which case the private region(s) (below the lower X-point and above the upper X-point) has ψ lower than the X-point surface(s). Hence, all wall points in the private region must be found, and excluded when searching for the minimum ψ-value along the wall.

In the old FLUSH, since mostly JET was treated, the method used to exclude those private wall points was simply to ignore all wall points below the lower X-point (and/or above the upper X-point). For JET, this is in fact quite robust, and only failed in rare corner cases, when the X-point was very close to the wall. For most machines, this assumption cannot be made, and the private region should be determined accurately. Alternatively, the closed part of the X-point surface may be determined to check that all its points are enclosed within the first wall.

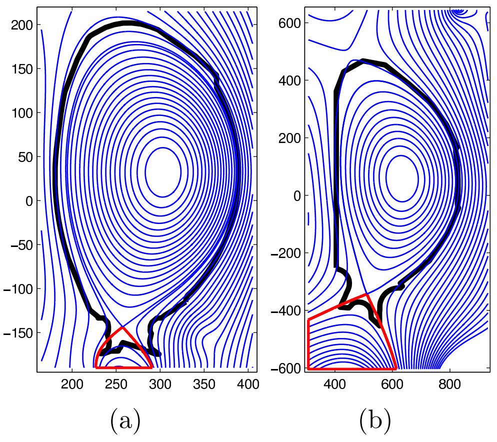

Figure 5:
(a) *JET plasma with the closed private contour.*
(b) *ITER plasma with the closed private contour, including a corner of the domain.*

The advantage of the latter method, using the closed contour of the X-point surface, is its robustness and validity on all tokamak configurations. Its disadvantage is that retrieving the complete X-point surface contour requires more computation time than retrieving only the private region of the surface, which is generally smaller. On JET EFIT equilibria, this private region method is the fastest. Thus, one needs to retrieve the X-point legs (or strike lines), up to the edge of the domain, and then create a closed contour by joining the ends of the two legs, as shown in Figure-5a. In cases where the two legs end on two different edges of the domain, closing the contour implies the addition of the domain corner, as shown in Figure-5b. Once the closed private contour has been obtained, The first wall points inside the contour are excluded from the LCFS search.

However, for other plasma configurations, such as MAST, this method cannot be used. Figure-1a demonstrates this case, where the private contour simply cannot be closed. In addition, for such cases, the private contour has a length comparable to the closed part of the X-point surface. Thus, the private contour method in FLUSH is used only for JET EFIT cases (to insure CHAIN1 speed requirements). For all other cases, FLUSH relies on the method based on the closed contour of the X-point surface.

The method implemented to retrieve the closed part of the X-point surface (or its private part) uses the rectangular grid, as for retrieving standard flux surface. In this case, the contour is followed, starting from the X-point, stepping from grid element to element. Unfortunately, although this method is more reliable and more precise, once again, the downside is loss of speed. On average, the new initialisation method has been measured to be 5-10 times slower when compared to the old FLUSH. Nevertheless, the FLUSH initialisation remains reasonably fast for use in large codes such as CHAIN1, since this initialisation is done only once per equilibrium, and generally remains only a fraction of the total FLUSH run-time (depending on the extent of the FLUSH-calculations done by the user).

4 Conclusion and Further Work

This paper presented the progress achieved in recent years with the FLUSH library. The work started in 2006 has been completed and a new, generalised FLUSH code is now available to users. Both the flux spline interpolation and the user tools are now adapted to any tokamak magnetic configuration.

In addition to the globalisation of the code, accessible test and release procedures have been implemented to ensure consistency of the code throughout any future development or maintenance. The testing procedure is directly linked to the CHAIN1 architecture to provide up-to-date CHAIN1 tests in order to certify the functionality of all FLUSH-related CHAIN1 codes at any FLUSH release.

Some of FLUSH's central algorithms and tools have also been modified, such as the line-of-sight intersection tool, the flux surfaces routines and the LCFS search method. Although the old methods are still available in the code for consistency and backward compatibility, the default methods used are the new ones. In some cases, the precision/robustness/practicality of the new tools leads to slower computation times than the old tools, but again, the old tools are still available to the user if rapidity becomes a greater concern than reliability.

In the future, some additional progress can be made, both in terms of internal coding as well as regarding user interface. For example, the `flushinit` issue should be addressed and the `IGO` fork method for initialisation should be broken into more efficient function calls, which would simplify both the coding structure and the user accessibility to various tokamak equilibria. At last, effort and manpower should be spent to update the Integrated Modelling version of FLUSH with the newer version.

ACKNOWLEDGEMENT

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

REFERENCES

[1] J.P. Freidberg, *Ideal MHD*, Cambridge University Press, ISBN: 9781107006256

[2] M.J. Hole *et al.*, *Plasma Phys. Control. Fusion* 53 (2011)

[3] L.L. Lao, H. StJohn, R.D. Stambaugh, and W.Pfeiffer, *Nucl. Fusion* 25, 1421 (1985).

[4] D.P. O'Brien, L.L. Lao, E.R. Solano, M. Garribba, T.S. Taylor, J.G. Cordey, and J.J. Ellis, *Nucl. Fusion* 32, 1351 (1992), http://dx.doi.org/10.1088/0029-5515/32/8/I05

[5] P.J. Mc Carthy, *Phys. Plasmas* 6, 3554 (1999)

[6] J. Blum, C. Boulbe, B. Faugeras, *Journal of Computational Physics* 231, 3 (2012)

[7] T. Casper *et al.*, *Nucl. Fusion* 54 (2014)

[8] R. Ambrosino *et al.*, *Nucl. Fusion* 54, 123008 (2014)

[9] http://www.nag.com/numeric/fl/FLdescription.asp

[10] http://w3.pppl.gov/ntcc/PSPLINE

[11] R. Layne, N. Cook, D. Harting, D.C. McDonald, C. Tidy, *Fusion Engineering and Design* 85 (2010), http://dx.doi.org/10.1016/j.fusengdes.2009.12.012 http://tinyurl.sfx.mpg.de/sus4

[12] J.P. Christiansen, *Journal of Computational Physics* 73 (1987), http://dx.doi.org/10.1016/0021-9991(87)90107-0 http://www.sciencedirect.com/science/article/pii/0021999187901070

[13] D. Dodt, N. Cook, D. McDonald, D. Harting, S. Pamela, *Fusion Engineering and Design* 88 (2013)

[14] R. Layne, A. Capel, N. Cook, M. Wheatley *Fusion Engineering and Design* 87 (2012)

[15] Kendall E. Atkinson, *An Introduction to Numerical Analysis*, John Wiley & Sons, ISBN 0-471-62489-6 (1989)

[16] Svetlana S. Petrova, Alexander D. Solov'ev, *Historia Mathematica* 24, (1997)

[17] A. Ralston and H.S. Wilf, *Math. Methods For Digital Comp.*, Wiley, New York/London, PP.95-109 (1960)

[18] Ralston, Runge-Kutta Methods with Minimum Error Bounds, MTAC, VOL.16, ISS.80, PP.431-437 (1962)

[19] https://root.cern.ch