Integrated Plasma Simulator (IPS) Design Description

Version 1.3

8-22-06

D. B. Batchelor

Abstract

This is intended as an evolving document to provide the published description of the Integrated Plasma Simulator (IPS) that SWIM developers and users can rely upon to guarantee compatibility with design IPS in their work. This is based on a number of meetings [1, 2, 3], and brings together and updates several previous write-ups [4, 5].

The IPS (at least the physics layer of it) consist of five main parts 1) A collection of input and results data structures. Initially this is a directory of files, 2) An overall controller script, 3) A set of components that implement the simulation functionality, 4) A distinguished component Plasma State, and, 5) an infrastructure of services. This document describes the general philosophy of the IPS and describes the functionality of each of the parts. Plus it describes the content of the Plasma State data structure, and gives examples of how to use the plasma state.

Section 0 contains some general discussion, definitions and conventions. Section I describes the large scale data structures that are required to set up and control the simulation and that are required to record the results of the simulation. Section II describes the functionality of the controller script.

Section III describes the interface and structure of implementation for the Plasma State component in terms of Fortran90 modules. Appendix 2 lists the data contents of the plasma state and the public interface of the of the plasma state which is a Fortran90 module containing the plasma state data. The list is not complete yet.

Section IV describes the structure and responsibilities of the components. Appendix lists the physics components and codes to provide the implementation. Appendix 3 gives a simple example of the use of a Plasma State component for a case with a single using component (RF) and where there are only 3 variables in the plasma state.

Appendix 4 contains an informal list of topics needing further discussion, Appendix 5 gives the layout of the simulation file directory, and Appendix 6 is a pushdown list describing modifications to this document that should enable a reader to tell quickly if anything has changed that affects them.

Table of Contents

0. General considerations

Our initial objectives in the design of the IPS framework were to:

- Get some rudimentary system up and running quickly, while not blocking the way to a more efficient or refined framework later. This is so we can test the framework without a heavy physics code burden, and address operability issues
- Have minimal impact on physics code developers. In particular not to have separate branches of the actual computation code – i.e. a "real" branch and a "SWIM" branch that rapidly goes out of date. To do this each physics code that implements a simulation component can be runnable as a separate process. Rationale: having each code supplied as a separate executable makes remote launch on HPC platforms easier, and can support code providers just supplying an executable for each platform.
- Keep the data needed to allow a code owner to debug or analyze a run the code makes within SWIM, without having to run entire SWIM framework up to the point of failure
- Reuse successful facilities and services that already exist, both within the fusion community (e.g., the Fusion Collaboratory) and within the CS community (e.g., TOPS, portal technology)

We believe we have an approach that will allow us to get started rapidly, with rather loose, file based coupling between the components, but that will allow progressively tighter coupling as required for the Slow MHD campaign. We have maintained the component approach as called out in the proposal, and as discussed at previous meetings, but have not adopted a formal component-based architecture such as CCA for the first implementation. Using separate executables is intended for the fast MHD campaign, where the coupling between components is fairly light-weight in size of data and most of the HPC required is within each component. For the slow campaign physics models must be more tightly coupled and it is likely we'll require subroutine access to the codes. We may move more toward a formal component model as it makes sense

Component-based software engineering builds software applications by composing self-contained "components" that interact with the outside world only through well defined interfaces. The logic of a component-based approach to fusion simulation is that a simulation requires a set of separate, or separable, physics processes and associated mathematical operations (e.g equations to be solved) that can be assigned to independent components. Once those separate, abstract mathematical operations are determined, the input information required to perform these operations, and the output information resulting from them is determined and is generic to the component. That is to say, the *interface* between the component and the rest of the system is the same regardless of what method, or specific code is used to do the work. Of course carrying out practical solutions with real codes requires other information that is code-specific. In the Component Definition Documents (see our web site) we identified (at some level of approximation) the abstract mathematical function of each of the components, defined the generic input-output interface, and made the separation of generic versus code-specific data. In the IPS, components are wrappers around codes that implement the actual

mathematical functions. The components always look the same outside the component independent of what code is inside, and they provide services to the codes that make it easy to substitute one code for another inside the component.

The benefits of the component approach we expect to gain are 1) effective division of labor, people can work simultaneously on different components and need not be expert on aspects of the system they are not responsible for, 2) the ability to support a range of levels in detail of description for each physics processes, including best-in-class codes, and 3) robustness to changes and improvements in the implementing codes. Appendix 1 contains a list of IPS components, the responsible component developer, a list of codes to initially implement the components, and the responsible code developer. At latest count there were 13 components

A key feature of our IPS design is that there is a distinguished component, the Plasma State Component, which contains and communicates the time varying data of the simulation. For the most part such data consists of plasma variables appearing in the simulation equations such as densities and magnetics, but could include other, non-plasma, generic information like filenames or source control settings. This component will initially be implemented using the XPLASMA system available in the NTCC.

The overall execution model uses scripts on at least two levels – the overall simulation controller is a script, and each component is implemented as a wrapper script that manages input/output data and launches the specific physics codes. We'll use Python for this because it is available on all platforms we know about, even when Java is not installed. Code contributors won't have to become Python experts; stubs will be provided that can be used as templates to be filled in.

There are four classes of developers who are involved in the IPS project:

1) Developers of the framework Simulation Controller script (section II) – these are CS people with support from physicists. The Controller script should be entirely generic, i.e. should not need any information about what codes are implementing the components. Of course the controller logic does depend on the physics and could be very complicated.
2) Developers of Component Scripts (section III) – These should be physicists with support from CS people who know Python. The component scripts should be written to be generic, in that any physics code that provides the required physics/mathematics functionality can easily be fit into the component. The part of the component script that deals with plasma state data is generic. It is written once and can be used no matter what physics code does the component work. Furthermore the structure of the plasma state part is the same for all physics components. Different variables need to pass in and out of the plasma state for different components, but the work that must be done to make the data accessible is exactly parallel. The component scripts also have a code-specific part that will have to be tailored for each code. This will require involvement of the code developers.
3) Developers of the Plasma State component (section IV)– These are physicists who know something about XPLASMA with help from CS.

4) Physics code developers – These are physicists who have codes that will implement the functionality of the components.

The cardinal rule of the IPS design is: Developers in one of these categories need know little or nothing of what is required for the other categories.

The strategy whereby this is possible hinges on a careful, abstract definition of the interface to the Plasma State component.

Section I of this describes the large scale data structures that are required to set up and control the simulation and that are required to record the results of the simulation. Section II describes the functionality of the controller script.

Section III describes the interface and structure of implementation for the Plasma State component in terms of Fortran90 modules. There are two layers to this structure, a semi-public layer such that only the Plasma State developers need to know about the implementation (e.g. XPLASMA), and a public layer to be implemented by the component developers, who only need to know about the interface of the semi-public layer. In principle, the code developers and the Controller Script developers don't need to know anything about the Plasma State. Appendix 2 lists the data contents of the plasma state and the public interface of the of the plasma state which is a Fortran90 module containing the plasma state data. The list is not complete yet.

Section IV describes the structure and responsibilities of the components. It is envisioned that each component will be required to implement just three methods: <COMP>_INITIALIZE, <COMP>_STEP, and <COMP>_FINALIZE. The most complicated of these is the step method, which must interact with the Plasma State to get the current input data for the component, must launch the implementing code, and must interact with the Plasma State to store the code results for use by the other components. Interaction with the Plasma State can be made generic to the component, i.e. independent of the implementing code. Furthermore the interaction with the Plasma State can be localized for each component to a single Fortran90 module that implements the public layer of the Plasma State interface. Appendix 3 gives a simple example of the use of a Plasma State component for a case with a single using component (rf_solve) and where there are only 3 variables in the plasma state.

Appendix 4 contains an informal list of topics needing further discussion. Readers are encouraged to add to this list. The intention is that the list will grow and shrink as topics are added and resolved. Appendix 5 gives that layout of the simulation file directory. Appendix 6 is a pushdown list describing modifications to this document that should enable a reader to tell quickly if anything has changes that affects them.

The IPS consist of five main parts 1) A collection of input and results data structures. Initially this is a directory of files, 2) An overall controller script, 3) A set of components that implement the simulation functionality, 4) A distinguished component Plasma State, and, 5) an infrastructure of services. See Figure 1.

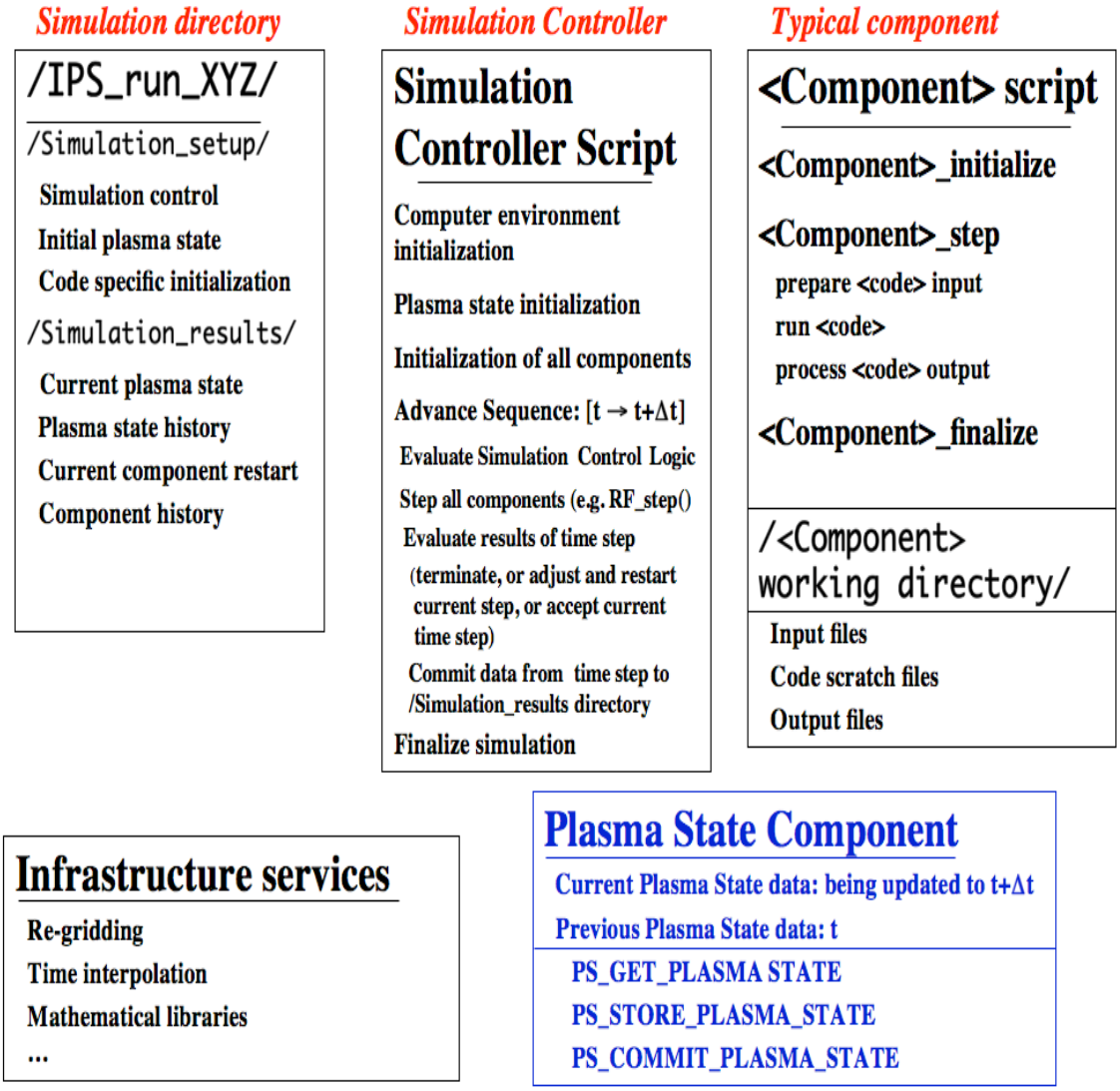

Figure 1

The Δt referred to in Figure 1 is the time-step on which simulation results are to be recorded. Presumably this is the longest time step in the process. Other, shorter time steps maybe be required inside this loop at the controller level, or internal to some of the components.

Notation convention for this document: Fortran90 module names are written in Monaco font (`plasma_state_mod`). Subroutine names are written in all caps (PS_GET_PLASMA_STATE). When reading a name like <COMP>_INITIALIZE or <COMP>_STEP, the expression <COMP> is a placeholder for the name of any component, e.g. RF_SOLVE, so that we would have RF_SOLVE_STEP.

Standards:

As this document develops we should be more definite about the standards, but I propose the following general rules to get started. Initially we should consider these "standards"

as desiderata. We should work toward them, and apply them to new code development but not delay the project while all legacy codes are brought up to standard.

1) We will use F95, free format. We will not use any deprecated Fortran95 usage.

2) We will use F95 KIND to specify precision of all variables, e.g. no REAL*8. We'll put the `KIND` definition in a module `SPEC_KIND_MOD` as is done for example in AJAX

3) We'll formalize a variable and function naming convention. For starters we'll use the European Standards for Writing and Documenting Exchangeable Fortran 90 Code [*6*]. For example lower case for variable names, caps for function and FORTRAN key words. (*I may have to go back and clean up my own sins in this draft*)

4) We'll adopt any relevant standards from NTCC

5) Units are SI and keV

I. Large Scale Data Structures – for now this means files

I.1 Simulation Directory

There are two distinct classes of files in the record of a simulation: 1) Input files of several different kinds, and 2) Results files. We require that input files are never changed during the simulation, so that the simulation can easily be repeated with exactly the same data. Results files change or in some cases are created and added as the simulation proceeds. There is more detail about the structure of this directory and the component working directories in Appendix 5.

Simulation Directory (e.g. /IPS_run_XYZ/)
<u>Setup and Input sub-directory</u> (e.g. /simulation _setup/ **simulation_control_file_XYZ(t_0)** –top-level control file **machine_definition_data_file** – standard file for DIII-D, ITER, JET … **initial_plasma_state_data(t_0)** – numbers and paths to EQBM and profile files **simulation event files and source waveform files** – time dependent simulation controls **code_A_input_XYZ(t_0)** – code specific input for component (e.g. AORSA) **code_B_input_XYZ (t_0)** …
<u>Simulation results sub-directory</u> (e.g. /simulation _results/ **plasma_state_data(t_n) –** state data for present time step **plasma_state_history (t_0, t_n) –** validated record of simulated plasma state **code_A_restart (t_0, t_n) –** record of component internal state needed to restart **code_B_restart (t_0, t_n) …** **code_A_history_XYZ (t_0, t_n) – record of** other data generated by specific code **code_B_history_XYZ (t_0, t_n) …**

I.1.1 Simulation Setup and Input Sub-directory – read only

Simulation control file: This is the top-level control file for the simulation It contains information about the execution of the simulation – settings for the simulation, names and/or paths to other needed files. This would include information about which actual components are needed for the simulation and which specific codes are to be used to implement the components. Read by controller script.

Machine definition file: Contains information about the particular tokamak that is being simulated so that all components use the same data. Ultimately this might be one of the XML schemas that the Europeans are developing. We might delay implementation of this.

Initial plasma state data file: Plasma data for initial conditions for start of simulation. This might or might not be an actual plasma-state file. It could be something like a namelist file with the data in it, or file paths to plasma profile data. It would be read by the Plasma State Initialization function of the controller script. This function might then produce an actual plasma-state file, which it stores in the simulation results subdirectory as the starting value of the Current Plasma State file.

Simulation event and source waveform files: These contain information about events that are scheduled during the simulation such as injection of pellets, waveforms of time dependent plasma actuators – like time varying power levels, or instructions to control algorithms – like targeting a specified plasma current ramp rate or a given plasma shape at a certain time. For example the control algorithms would program the solenoid and poloidal field coil voltages to meet the current and shape targets during the simulation. Read by the Evaluate Simulation Control Logic function of the controller script.

Code-specific initialization input data files for each component code used: This includes code-specific data that does not go through the plasma state component – initial grid resolutions, algorithm settings, instructions on what additional information to calculate and store beyond that generic information required for the component. This latter could include synthetic diagnostic data, code diagnostic data (like residuals), or plot data. This would be read by the component initialization function of the controller script and communicated to the component implementation script. These files might be copied into the component working directories or maybe just the path to the originals passed into the components.

I.1.2 Simulation results files directory – read and write

Current plasma state file: Contains state data for the present time step (and an additional data set for the previous completed time step, see section III). It is initialized by the Plasma State Initialization function of the controller script. It is read by the PS_GET_PLASMA_STATE function of the Plasma State component (see section III). After a time step each component adds its plasma state output to this file with its PUT_PLASMA_STATE_<COMP> function (see Section IV), which in turn invokes the generic function PS_STORE_PLASMA_STATE of the Plasma State component.

Plasma State History: This file, or perhaps it is a directory of files (one for each time step), is the record of the plasma state evolution during the simulation. It is the essential physics result of the simulation. It is updated by the controller script each time a validated time-step is taken.

Code restart files: Each code that has internal state needed to restart from a previous time step is required save that internal state. These files are written by the component script inside the <COMP>_STEP function, after it has determined that is has produced a valid time step.

Code history files: This is the record of additional code-specific output data that does not go through the plasma state component. It would include changes to code-specific settings from the values on initial input, like grid resolutions and algorithm settings. The additional data could include synthetic diagnostic data, code diagnostic data (like residuals), or plot data. These files are written by the component script, inside the <COMP>_STEP function, and after it has determined that is has produced a valid time step (see Section IV).

I.2 Component working directories

This is the volatile working area for each component. The controller script sets up this directory in its computer environment initialization function. Exactly what files are in this working area may depend on the component and the specific codes implementing the component.

Code-specific initialization input data files (maybe): This includes code-specific data that does not go through the plasma state component – initial grid resolutions, algorithm settings, instructions on what additional information to calculate and store beyond that generic information required for the component. This information is read by the PREPARE_<CODE>_INPUT function of the component, is merged with plasma state information to write the standard code-specific input files (see Section IV). To be determined later, it might not be necessary to actually copy these files into the component working directory, rather it might be sufficient to simply pass the path back to these code-specific input files residing in the Simulation Setup and Control Files subdirectory.

Standard code-specific input files: These are the actual standard files that the physics codes read. These files are written by the PREPARE_<CODE>_INPUT function of the component, and contain code-specific initialization data merged with generic plasma state information.

Code-specific working files: Any scratch files the codes write and read as part of their normal functioning.

Standard code-specific output files: These are the actual standard files that the physics codes write. These files are read by the PROCESS_<CODE>_OUTPUT function of the component, which extracts the outputs that go into the plasma state and converts them from code-specific form into standard plasma state form. See section III.2.

Code restart and code history files (maybe): These files need to go into the Simulation Results subdirectory. It is possible that these files should first be written into the working directory and subsequently copied into the Simulation Results subdirectory when the step is determined to be valid.

II. Controller script functionality

The top-level task, that controls everything, is a simulation controller script. This should start simple, be very flexible, and allow for easy inclusion of more intelligence as we go along in evaluating the progress and control of the simulation.

Simulation Controller Script
Computer environment initialization – do computer science stuff
Plasma state initialization – set up the "current" plasma state for t_0
Initialization of components – call all <COMP>_INITIALIZE functions
Loop over time: **Take a time step, t → t + Δt** **Evaluate pre-step control logic –** general logic relevant to all components **Cycle through all components:** **Evaluate pre-step control logic relevant to this component –** component specific logic (e.g. changes in heating power) **Step component –** call <COMP>_STEP function **Evaluate post-step control logic relevant to this component –** evaluate what to do about any errors or warnings returned from the <COMP>_STEP function **End cycle through components** **Evaluate Post-step control logic –** general logic after all components have been stepped Evaluate validity of time step – step ok?, need to make adjustments and re-run step?, give up and terminate? Accept time step – call PS_COMMIT_PLASMA_STATE **End time step** **End loop over time**
Finalize simulation – put files in order, call <COMP>_finalize functions

II.1 Computer environment initialization

There are all kinds of computer science things that I'm sure need to be done here associated with authorization, setting up data flow, visualization, launching parallel jobs, publishing simulation results…. I'll leave it to others (like Randy) to add these things in. It reads the simulation_control file, determines what components it has and what codes are implementing each component, sets up file directories and makes paths known to the things that need to know them. Does needed error checking.

II.2 Plasma state initialization

The initial implementation of the plasma state is in terms of XPLASMA the data being communicated via a NETCDF file whose format is specified by XPLASMA. We can imagine (at least) two approaches to obtaining an initial plasma state:

1) We are given a plasma state NETCDF file in the simulation setup and control subdirectory that was previously generated externally by XPLASMA and is copied to the current_plasma_state file of the simulation results subdirectory.
2) We are given files in the simulation setup and control subdirectory containing data (maybe namelists and files with profile data) that describe the initial state of the simulation. And we produce a function that gathers up this data and uses XPLASMA calls to write this into the plasma state, and then writes the initial NETCDF file to the current_plasma_state file of the simulation results subdirectory.

This is a topic for further discussion (Appendix 4). However we do, it I'll assume the existence of an initial plasma state NETCDF file after execution of this function

II.3 Initialize components

Loop through and call all the <COMP>_INITIALIZE functions for each component. This includes making the file paths to the initialization data known to the initialization functions, maybe by means of arguments.

II.4 Loop over time

Take a time step $t \rightarrow t + \Delta t$:

There are several points at which control logic is to be evaluated and adjustments to parameters of the simulation might be made. This logic will certainly evolve as we gain experience. A major point for further discussion is to develop a minimal set of algorithms to get started. Another important issue to be resolved is how to communicate information back and forth between the controller script and the components in a generic way. This information would include changes to parameter settings made by the controller script, warnings and error messages generated by the components. One could imagine communicating such information through arguments to the component methods, through file exchange, and some data (such as changes in source inputs or coil currents) might logically be communicated via the Plasma State.

> **Evaluate pre-step control logic:** Control logic relevant to all components. This function will evolve. For sure it will involve looking at the simulation event and source waveform files and communicating any changes to the components. It may involve looking at the plasma state and perhaps evaluating results of the previous time step to make adjustments, e.g. adjusting Δt. Before starting the time step the

current plasma state data needs to be stored as the previous step by calling the Plasma State method PS_COPY_TO_PREV_STEP (see section III)

Cycle through components:

Evaluate pre-step control logic relevant to given component: This is essentially the same function as described above, but here it is logic relevant to specific components. It could for example specify multiple time sub-steps $\delta t < \Delta t$ for this component, or whether it is even necessary to recalculate this component

Step components: call <COMP>_STEP function. On return the component should have completed its time step and stored the results in the current Plasma State. It should also return error or quality of solution results. The component may have taken internal time sub-steps to reach Δt that the controller script doesn't even need to know about.

Evaluate the results of the <COMP>_STEP: Each component will return an error, or quality of solution results of some sort to indicate success of the step and some information about what might be done in case of problems. If the step was successful then proceed. If not, the controller has to evaluate whether to make some adjustments and try again or give up and terminate. Later we can work out the balance of the logic between this function and the "evaluate pre-step control logic" function above.

End cycle through components

Accept time step: If all components were stepped successfully declare the step to be valid and add the present plasma state to the Plasma State History by calling the Plasma State function PS_COMMIT_PLASMA_STATE. This function also copies the updated plasma state data into its previous time step data. Publish results to the simulation tracking mechanism.

End time step $t = t + \Delta t$

II.5 Finalize simulation

Report simulation result to the proper authorities. Store simulation directory wherever it goes permanently. Clean up mess.

III. Plasma State Component Implementation

The biggest organizational issue in coupling a number of disparate physics calculations as we are doing in the IPS is how to allow the different components to communicate their respective input and output data among themselves. Our concept is to do this coupling through a common "Plasma State", PS, component. The prime desideratum for the PS component is that it be a true component with a simple published interface of data and functions, and that nobody other than the PS developers need to know anything about its implementation. Here is presented a simple structure that has these features, but postpones for the present dealing with the $M \times N$ problem. The proposed implementation will be described in terms of Fortran 90 modules, although it surely could be expressed more abstractly, for example in SIDL.

The design has a two-layer interface: a public interface that is written by the component developers, and a "semi-public" interface that is written by the PS developers and only used by the component developers.

The public interface consists of a published list of all of the variables that reside in the plasma state (with conventional names), plus an F90 module for each component, `plasma_state_<comp>_mod`. The component module declares as public data the variables in the plasma state that apply to that component, and contains (in the F90 sense of 'contains') two functions GET_PLASMA_STATE_<COMP> and PUT_PLASMA_STATE_<COMP>. This public interface to the PS is to be written by the component developers. Appendix 2 contains a start on the, to be published, list of all plasma state data.

The semi-public interface for the Plasma State consists of an F90 module, `plasma_state_mod` declaring all of the variables in the plasma state as public data with slightly mangled names (i.e. prepending "PS_" to each of the variables in the published variable list, Appendix 2), and containing two functions PS_GET_PLASMA_STATE and PS_STORE_PLASMA_STATE that retrieve and store the entire plasma state data set. This semi-public interface would be written by the PS developers once and for all and would encapsulate all of the XPLASMA calls. In addition the Plasma State component must provide a function PS_COMMIT_PLASMA_STATE that is called by the controller script each time step, after all components have successfully executed their time step. This routine copies the updated time step data to the previous time step data that also resides in plasma state and stores the completed state data in the simulation history archive.

Each GET_PLASMA_STATE_<COMP> subroutine uses the `plasma_state_mod` module. It then calls the Plasma State module routine PS_GET_PLASMA_STATE, which in turn uses XPLASMA calls to open the current Plasma State NETCDF file and to load all the PS data into the `plasma_state_mod` variables using the semi-public, mangled names. The GET_PLASMA_STATE_<COMP> subroutine then copies the data relevant to that component into the `plasma_state_<COMP>_mod` variables using the public names. The PREPARE_<CODE>_INPUT routine (see section IV.2.1 below) uses `plasma_state_<COMP>_mod`. From that module it calls the GET_PLASMA_STATE_<COMP> subroutine, which makes the component relevant PS

data accessible using the public names. These variables can be safely clobbered if needed, for example if the code specific name is the same but the units are different.

Updating the Plasma State contents after a component step is the inverse of the process above. Each PUT_PLASMA_STATE_<COMP> subroutine uses the `plasma_state_mod` module. It copies the appropriate output data produced by that component into the `plasma_state_mod` variables using the mangled names. It then calls the Plasma State module routine PS_STORE_PLASMA_STATE, which in turn uses XPLASMA calls to write the data into the Plasma State and to write the updated current Plasma State NETCDF file.

Appendix 3 shows a skeleton of use of the Plasma State model as described here for the case of an RF component, being implemented by the AORSA code. For simplicity in this example there are only 3 quantities in the plasma state, two AORSA inputs, r_axis and B_axis and one AORSA output, total rf power deposition. For illustrative purposes we suppose that AORSA uses the name r0 for r_axis, that the name B_axis is the same as the Plasma State form but uses Gauss instead of kGauss. We assume that the total power deposition is called Prf in AORSA and Prf_total in the Plasma state.

What is good about this scheme?

It is simple. Writing the `plasma_state_<comp>_mod` modules is straightforward and requires zero expertise with XPLASMA. Writing the `plasma_state_<comp>` module is straightforward and requires zero knowledge of what's going on in the components. Both of these processes can start as soon as we complete the list of variables to appear in the Plasma State, Appendix 2. Also this scheme carries over to the case of tighter, in-memory, code coupling when we get beyond file-based communication stage.

What is not so good about this scheme?

It ignores the *M*×*N* problem. This is the problem of *M* different possible implementation codes wanting to write similar data into the Plasma State in *M* different formats while *N* different codes want data out of the Plasma State in *N* different formats. Actually this proposed scheme does reduce *M*×*N* to *M*+*N* if good enough data converters can be provided. Then at most *M* providing codes would need to convert their output to the PS form and each *N* consuming code might need to covert the PS form into its needed form. XPLASMA functionality exists to provide most of the needed conversions, particularly the conversion of 1D profiles with different radial grids. We could package this functionality with simple F90 calls and include it as part of the IPS infrastructure layer. Maybe this is good enough.

Actually this may not be such a serious problem for us. There are only a few instances where radically different forms are required e.g. equilibrium (R, Z versus inverse representations), distribution functions ($v_{||}$, $v_{\perp}$, ψ grid versus Monte Carlo particle lists), RF antenna models (description of ICRF antennas is very different from ECH beams). The RF codes and CQL3D Fokker Planck solver have already been adapted and are communicating perfectly well through files. To get started we can handle these difficult conversions on a case-by-case basis and pass time-evolving data through the PS as just filenames (See Appendix 2).

Other Plasma State Considerations

The form of the data in the Plasma State as exposed by the public interface is fixed by convention. For example the specification of the electron density is [number of grid points in density profile, radial grid points of profile, density values at radial grid points]. However the number of grid points and the radius values are not fixed, and in fact may change during the simulation. So we stipulate that the component that provides the data determines the variable elements of the data description (such as number of grid points and radial grid). It is the responsibility of the consuming components to inquire about these variable elements and make sure the data is handled correctly.

The possibility that some components may take intermediate time sub-steps means that the inputs to these components may need to be available at these intermediate times, which were not computed by the providing component. Thus there is a need to have access to data from a previous time step and to have functionality for interpolating the data between time steps. The data needing interpolation is mostly from the source components, which are probably not as stiff as, for example, transport models. Therefore we need retain two time-steps of data in the plasma state (current, and previous). The "current" plasma state data will have been partially updated depending on what components have been stepped up to that point in the time step. The Plasma State component needs to implement an additional method that is called by the controller script upon successfully completing a time step (i.e. all components have been successfully stepped), that copies all of the PS data to variables with names identical to the semi-public data but with '_PREV' appended. We stipulate it to be the consuming component's responsibility to manage the interpolation. A simple interface to this interpolation functionality, which already exists in XPLASMA, is another candidate for IPS infrastructure service. The additional plasma state method (to be called PS_COMMIT_PLASMA_STATE) should also update the Plasma State history with the successfully completed time step.

IV. Component Functionality

The components are implemented as Python scripts that export three functions: <COMP>_INITIALIZE, <COMP>_STEP, <COMP>_FINALIZE. Each of these functions may be comprised of several parts and may launch other executables such as Fortran. In fact the Python part of the script can be quite thin if the developer chooses to do most of the work in executables of another language. To the outside world a typical component looks like this:

Component Script (e.g. <COMP>_A)
Initialize function – e.g. A_INITIALIZE(err_info)
Time step function – e.g. A_STEP(err_info)
Finalize function – e.g. A_FINALIZE(err_info)

IV.1 Component Initialization: <COMP>_INITIALIZE

If there is any work that is generic to the component (e.g. storing file paths, setting up directories, or doing some preliminary calculations) that needs to be done before the simulation starts, this function does it. If there is any needed work that is specific to a particular code that implements the component (e.g. doing some preliminary calculations), then we have to write a code-specific routine to do this. Such code-specific routines will be launched by the <COMP>_INITIALIZE function. It is certainly possible that, particularly during the file-based communication phase, these functions may be empty.

IV.2 Time step: <COMP>_STEP

The step script has three distinct sub-tasks: prepare the code input, run the code, process the code output including logic as to whether the run was successful, and if not what to do about it. The <COMP>_STEP script either implements these tasks itself or may launch separate executables to do so. Note that interaction with the Plasma State occurs in the INITIALIZE and FINALIZE tasks. Details of the working of the plasma state are localized to two subroutines:GET_PLASMA_STATE_<COMP>, and PUT_PLASMA_STATE_<COMP>. These are generic to the component and only need to be written once per component. In fact they should be very similar for all components and when written for the first component they should be very easy to adapt for the others. Furthermore, as described in section III, most of the interaction with the XPLASMA functionality can be localized to two functions that are generic to the entire IPS, PS_GET_PLASMA_STATE, and PS_STORE_PLASMA_STATE. These functions need only be written once for the entire project.

Below is an outline of a <COMP>_STEP function for the example of the RF_SOLVE component being implemented by the AORSA code. There is a somewhat expanded discussion in the subsections below. And there is a skeleton F90 program implementing the PREPARE_AORSA_INPUT function in Appendix 3.

RF_SOLVE_STEP

Prepare code input → Call PREPARE_AORSA_ INPUT (or this example this is a Fortran code)

Collect all needed files and subdirectories – do needed checking

USE `plasma_state_RF_mod`

Declare all input AORSA variables not already in `plasma_state_RF_mod`

Call GET_PLASMA_STATE_RF – loads relevant data into module variables

Convert PS data to form needed by AORSA

Read other AORSA specific data from input files

Combine PS data and other AORSA input data and write standard AORSA input file

Run code → call RUN_AORSA

Check to be sure everything needed for AORSA run is ready

Launch AORSA executable

Process code output → call PROCESS_AORSA_OUTPUT

Evaluate results of AORSA run – if run ok proceed, if not then:

Evaluate control logic – Are there adjustments that can be made to AORSA inputs that might make the run succeed. If so then make the changes and re-launch AORSA

If nothing can be done inside the component to fix the step, report this situation back to the controller script (error return) and quit

Update Plasma State (for this example this is a Fortran code)

USE `plasma_state_RF_mod`

Declare AORSA output variables not already in `plasma_state_RF_mod`

Read standard AORSA output files

Convert AORSA results to PS form and store in `plasma_state_RF_mod` variables

Call PUT_PLASMA_STATE_RF – stores RF results into PS

Store AORSA internal state in AORSA_restart_file in /simulation_results/ subdirectory (for AORSA in particular there is no internal state)

Store other AORSA output data in AORSA_history file in /simulation_results/ subdirectory

Clean up

IV.2.1 Prepare <CODE> Input

A. Reads required input file list, makes sure they're available. This is based on a directory being passed in telling it where to find input files.

B. Prepares the standard input data for the specific implementing code and writes standard code input files. This task is code-specific but is made considerably simpler by using a component-generic module containing a generic function GET_PLASMA_STATE_<COMP>. For the RF component we have decided to implement the data preparation as a Fortran90 executable: PREPARE_AORSA_INPUT. Here is what the PREPARE_<CODE>_INPUT function must do:

   1) Declare all of the variables that go into the standard code input files, including those that come in through the plasma state as well as the other, code-specific variables that come in from other sources. The plasma state variables needed by the component are made accessible by using a module "`plasma_state_<COMP>_mod`".

   2) Load the plasma state variables by calling the generic function GET_PLASMA_STATE_<COMP>

   3) Convert plasma state data into code-specific form if needed

   4) Read other files needed to determine the code-specific input variables. These are most likely just the initial set of standard input files for the specific code. **(Note: This, and the inverse task of putting certain results data into plasma state, are the only tasks in the whole IPS that *MIGHT* require either a component script developer or a code developer to need to know something about the implementation of the Plasma State component if for example he wants to use some other capabilities of XPLASMA. See Section III on Plasma State Component Implementation)**

   5) Combine the plasma state data and code-specific data from steps 3) and 4) and write standard code-specific input files

III.2.2 Run <CODE>

A. Check that everything needed for the code run is present and in good condition.

B. Do the work of taking the time step. For most components this will entail launching a separate Fortran executable.

III.2.3 Process <CODE> output

A. Evaluate the results of the code run. If the time step was successful then go ahead. If the step was unsuccessful then decide whether to make some adjustments to the code settings and try running the code again, or whether to report back to the controller script that the step failed. Determine if there are adjustments that can be made to the code inputs that might make the run succeed (e.g. changing tolerances or grid resolutions). For some components it might be necessary to take several smaller internal time steps. If so then make the changes and re-launch the code. Or there

may be adjustments that only the controller script can make that could allow the simulation to proceed. Such information needs to be communicated back to the controller script. If there are warnings or other messages that need to go back to the controller script, then generate those.

B. Do the inverse of PREPARE_<CODE>_INPUT above. As above this task is code-specific but is made considerably simpler by using a component-generic module containing a generic function PUT_PLASMA_STATE_<COMP>.

   1) Declare all of the variables that appear in the standard code output files, including those that come in through the plasma state as well as the other, code-specific variables. Again in the case of a Fortran90 implementation the plasma state variables relevant to the component are made accessible by using a module "`plasma_state_<comp>_mod`".

   2) Read the standard code output files

   3) Convert plasma state data from code-specific form to Plasma State form if needed and load into the `plasma_state_<comp>_mod` variables

   4) Store the code results back into the plasma state by calling the generic function PUT_PLASMA_STATE_<COMP>

C. Store the internal state of the component in the "code_restart" file of the /simulation_results/ subdirectory. This presupposes that the code has written the required internal state to some file (maybe directly into the /simulation_results/ subdirectory, maybe in the working directory).

D. Store other data generated by the code into the code_history file in the /simulation_results/ subdirectory. This is the record of additional code-specific output data that does not go through the plasma state component. It would include changes to code-specific settings from the values on initial input: grid resolutions, algorithm settings. The additional data could include synthetic diagnostic data, code diagnostic data (like residuals), or plot data.

E. Clean up any mess made by the code during the time step – e.g. delete scratch files.

References:

Note: SWIM documents without formal references are generally available at SWIM web site www.cswim.org

[1] Workshop Toward an Integrated Plasma Simulation, Nov 7-9, 2005, Oak Ridge, TN

[2] SWIM Design Meeting, Jan 16-19, 2006 at PPPL

[3] Working meeting on IPS interfaces May 3–5, 2006 at ORNL

[4] Report of SWIM Design Meeting, Jan 16-19, 2006 at PPPL

[5] Plasma State Component Interface Description,Version 0.2 , 5-10-06

[6] P. Andrews, G. Cats, D. Dent, M. Gertz and J.L. Ricard, "European Standards for Writing and Documenting Exchangeable Fortran 90 Code", http://www.meto.gov.uk/research/nwp/numerical/fortran90/f90_standards.html, 1996

Appendix 1. IPS components and implementing codes

Component	Responsible person	Implementing code	Responsible person
1) Plasma state	McCune	XPLASMA	McCune
2) Energetic neutral source	McCune	NUBEAM	McCune
3) Neutral gas fueling source	McCune	FRANTIC	McCune
4) RF Field source	Batchelor	AORSA	Jaeger
		TORIC	Bonoli
		GENRAY	Harvey
5) Fokker Planck solution	Berry	CQL3D	Harvey
		NUBEAM	McCune
6) Neoclassical and bootstrap	Houlberg	NCLASS	Houlberg
7) Advance profiles	Jardin	TSC-solver	Ku
		GCNM	St. John
8) Advance equilibrium	Jardin	TSC-equilibrium	Ku
		TEQ	LoDestro
9) Anomalous transport coeff	Jardin	GLF23	Ku
		MMM95	Ku
10) Linear Stability	Kruger	DCON (low-n)	Kruger
		ELITE (edge)	Snyder
		NOVA-K (kinetic)	Gorelenkov
		PEST-2 (low-n)	Chance
		BALLOON (high-n)	Chance
11) Reduced Sawtooth model	Jardin	Porcelli	Bateman
12) 3D nonlinear MHD	Kruger	M3D M3D/K	Breslau Fu
		NIMROD	Schnack
13) Waveform and feedback			

Appendix 2. Plasma State Contents

Plasma State Component interface is represented as a FORTRAN 90 module declaring all the data contained in the plasma state along with two routines to get/write that data to/from XPLASMA. In order to make the data more transparent and reduce the documentation load it was decided (at least initially) not to bundle related data into derived types except for some component-specific file types. The component-specific derived data types, that are initially implemented just as file names, allow us to pass complicated sets of data (like distribution functions, antenna models, or quasilinear operators through the Plasma State without at first having to implement them in XPLASMA. The intention is to make all names of PS data as self descriptive as possible subject to reasonable amounts of typing and avoiding the (absurd) 31 character limit in Fortran. For each variable in the list the PS module will actually contain an additional variable of the same name plus and appended '_PREV' and containing the equivalent data from the previous time step.

We assume the existence of a module, `swim_global_data_mod`, containing universally useful stuff like a derived data type for handling file names or variable names, kind specification of intrinsic data (integer, real, complex, etc), constants (pi, physical constants, …), etc.

Unless otherwise noted "real" means of type `rspec` as defined in a `SPEC_KIND_MOD` module. We will assume that in their most basic form 1D profiles, for example profile of f(rho), are specified by the number of radial points e.g. nf, the radial grid, rho_f_grid(1:nf), and the values, f(1:nf). XPLASMA provides the facility to interpolate a profile on an arbitrary grid that can easily be used if the component developer chooses. However, we thought that this might be a bit dangerous in that it hides the intrinsic resolution of the data somewhat, and also since most of the codes have their own approaches to interpolating profiles between grid points this could produce an interpolation of an interpolation.

For the present we take the radial coordinate, ρ, to be an arbitrary function monotonically increasing from the magnetic axis. It may be that there is no advantage to keeping this generality and that we will fix ρ as the toroidal flux inside a given flux surface. (See Appendix 3, Issues for Further Discussion)

Derived Data Types:

Name	Type	Providing Component	Units	Definition
`filename`	filename	swim_global_data_m	–	A character type that can contain a file name complete with path

distribution_fn	filename	init_rf	–	data type for distribution funtion
ant_model	filename	init_rf	–	data type for antenna model
ql_operator	filename	init_rf	–	data type for quasilinear operator

Timing data:

Name	Type	Providing Component	Units	Definition
t0	real	Controller script	sec	Time at beginning of current time step
t1	real	Controller script	sec	Time at end of current time step

Basic Geometry:

Name	Type	Providing Component	Units	Definition
r_axis	real	Equilibrium	*m*	Major radius of magnetic axis
z_axis	real	Equilibrium	*m*	Z of magnetic axis
r0_mach	real	PS init	*m*	Major radius of machine center
z0_mach	real	PS init	*m*	Z of machine center
r_min	real	PS init	*m*	Major radius of inside of bounding box
r_max	real	PS init	*m*	Major radius of outside of bounding box
z_max	real	PS init	*m*	Z of top of bounding box
z_min	real	PS init	*m*	Z of bottom of bounding box

Particle Species:

The number of ion species is arbitrary, nspec. We distinguish two types of species. 1) Thermal species defined by charge, mass, density temperature and flow velocity, and 2) Non-thermal species defined by charge, mass, and distribution function. Doug McCune's

XPLASMA example distinguishes between main ion species and trace impurity species. I guess there could be hundreds of these trace impurities with different isotopes and charge states. I suggest that we retain this distinction and introduce a main species called 'impurity' that is defined by density, z_{imp}, m_{imp}, all of which are radial profiles (Doug's Tokamakium). If we carry along a detailed impurity model with lots of impurity species we can calculate the average 'impurity' data from that. But the data for a more detailed impurity treatment is not in here now. I further propose that each main species have a character name, and that we have a default set consisting of ('e', 'H', 'D', 'T', '3He', '4He', 'imp') and that we store main species data in arrays of length `nspec_th`+1 and range [0:`nspec_th`] with species 0 always electrons. Ion species are 1 through `nspec_th`. (This is of course a minor detail)

Non-thermal species require numerical distribution functions. Pending a resolution of how to how to store general distribution functions in XPLASMA we will declare a distribution function to be a derived type that simply consists of a filename containing the required data. Since the density of an anisotropic distribution is not constant along a field line there is an issue of maintaining charge neutrality, i.e. the electron density probably has to be 2D reflecting a poloidal potential variation. I don't know if this makes any difference to any of the components other than RF, where dispersion relations really get unhappy if charge neutrality is not locally satisfied. For now let's take electron density in the Plasma State to be constant on a flux surface and let the RF codes figure out how to maintain charge neutrality for their calculations.

It may not be necessary to make as much distinction between thermal and non-thermal species as I have made here. We could have just one set of species arrays that includes both types. Then for the non-Maxwellian species the temperature array would be irrelevant but benign and we would supplement with a set distribution function types for those elements. This would be simpler but we can decide later.

Note also that there is a distinction between this list and the Adiabatic Species list below although some of the data is redundant. The Profile Advance component uses quantities in sufficiently different form that it makes sense to allow this redundancy. For example Profile Advance uses differential ion density rather than ion density and it uses electron entropy and total entropy rather than electron temperature and individual ion temperatures. The data in this list is more general than that provided by the advancement of adiabatic profiles in that it contains non-Maxwellian components, possible impurities, and possible minorities, which may not be advanced as adiabatic species. Models will be required to populate the data in this list from the adiabatic profiles, Fokker-Planck solutions, impurity models, and minority species models. This is a topic for further discussion (see Appendix 4)

Name	**Type**	**Providing Component**	**Units**	**Definition**
`nspec`	integer	PS init	–	Number of ion species = `nspec_th` + `nspec_nonMax`
`nspec_th`	integer	PS init	–	Number of main (thermal) ion species

s_name(0:nspec_th)	swim_name	PS init	–	Name of main species
q_s(0:nspec_th)	real	PS init	*C*	Charge of species s
m_s(0:nspec_th)	real	PS init	*kg*	Mass of species s
nrho_n	integer	PS init	–	number of rho values in density grid
rho_n_grid(1:nrho_n)	real	PS init	–	rho values in density profile grid
n_s(1:nrho_n, 0:nspec_th)	real	PS init	m^{-3}	Density profile of species s
q_impurity(1:nrho_n)	real	PS init	*C*	Effective impurity charge profile
m_impurity(1:nrho_n)	real	PS init	m^{-3}	Effective impurity mass profile
nrho_T	integer	PS init	–	number of rho values in temperature grid
rho_T_grid(1:nrho_T)	real	PS init	–	rho values in temperature profile grid
T_s(1:nrho_T,0:nspec_th)	real	PS init	*keV*	Temperature profile of species s
nrho_v_par	integer	PS init	–	number of rho values in parallel velocity
rho_v_par_grid(1:nrho_v_par)	real	PS init	–	rho values in temperature profile grid
v_par_s(1:nrho_v_par, 0:nspec_th)	real	PS init	*m/sec*	Parallel velocity profile of species s
nspec_nonMax	integer	PS init	–	Number of non-Maxwellion ion species
nonMax_name(1:nspec_nonMax)	swim_name	PS init	–	Name of non-Maxellian species
q_nonMax_s(1:nspec_nonMax)	real	PS init	*C*	Charge of species s
m_nonMax_s(1:nspec_nonMax)	real	PS init	*kg*	Mass of species s
ntheta_n	integer	PS init	–	number of theta values in 2D density grid
theta_n_grid(1:ntheta_n)	real	PS init	–	theta values in 2D density grid
n_nonMax2D_s(1:nrho_n, 1:ntheta_n, 0:nspec_nonMax)	real	PS init	m^{-3}	2D density profile of non-Maxwellian species s
n_nonMax_s(1:nrho_n, 0:nspec_nonMax)	real	PS init	m^{-3}	Flux surface average density profile of non-Maxwellian species s
dist_fun_s(1:nspec_nonMax)	distribution_f	FP Solver or		Files containing

	n	FP Solver init		distribution function data

Adiabatic Profile Advance (transport):

These are taken from the Advance Adiabatic Profiles Component Description Document 11/13/05. Note that there is an overlap between this list and the Particle Species list above. However the Profile Advance (PA) component uses quantities in sufficiently different form that it makes sense to allow this redundancy. For example PA uses differential ion density rather than ion density and it uses electron entropy and total entropy rather than electron temperature and individual ion temperatures. It will be necessary to develop models to extract the individual ion temperatures from this, but that can be done later.

For now we assume that all adiabatic variables are defined on the same ρ grid. It will be easy to allow different adiabatic variables to have different grids as we did for the particle species above.

Questions for further discussion (Appendix 4) are whether the transport coefficients (*D*, χ_e, etc) should live in the Plasma State and whether should eliminate Anomalous Transport and Profile Advance should be maintained as separate components.

Name	**Type**	**Providing Component**	**Units**	**Definition**
n_ad_ion	integer	Profile Advance (PA)	–	Number of adiabatic ion species
s_ad_name(0:n_ad_ion)	swim_name	PA	–	Name of adiabatic on species
q_ad_s(0:n_ad_ion)	real	PA	*C*	Charge of species s
m_ad_s(0:n_ad_ion)	real	PA	*kg*	Mass of species s
nrho_ad	integer	PA	–	number of rho values in adiabatic variable grid
rho_ad_grid(1:nrho_ad)	real	PA	–	rho values in adiabatic variable profiles grid
diffN_s(1:nrho_ad, 1: n_ad_ion)	real	PA		Differential density profile of species s
total_entropy(1:nrho_ad)	real	PA		Total entropy prifile
e_entropy(1:nrho_ad)	real	PA		Electron entropy profile
omegat(1:nrho_ad)	real	PA		Toroidal angular velocity profile
iota(1:nrho_ad)	real	PA		Rotational Transform profile

chi_e(1:nrho_ad)	real	PA		Electron thermal conductivity profile
dchi_e_dTeprime(1:nrho_ad)	real	PA		$\frac{\partial \chi_e}{\partial T_e'}$
dchi_e_dTiprime(1:nrho_ad)	real	PA		$\frac{\partial \chi_e}{\partial T_i'}$
chi_i(1:nrho_ad)	real	PA		Ion thermal conductivity profile
dchi_i_dTeprime(1:nrho_ad)	real	PA		$\frac{\partial \chi_i}{\partial T_e'}$
dchi_i_dTiprime(1:nrho_ad)	real	PA		$\frac{\partial \chi_i}{\partial T_i'}$
chi_omegat(1:nrho_ad)	real	PA		Toroidal angular momentum diffusivity profile
Q_ei(1:nrho_ad)	real	PA		Electron-ion energy equipartion profile
D_s(1:nrho_ad, 1:n_ad_ion)	real	PA		Particle diffusivity profile for species s
eta_parallel(1:nrho_ad)	real			Flux surface averaged parallel electrical resistivity
Source_rate_s(1:nrho_ad, 1:n_ad_ion)	real			Particle source rate for adiabatic species s

Equilibrium Data:

This list is likely to be incomplete. Both ORSA and TORIC get their equilibrium data from an EQDSK file and handle their own metric coefficients and the like. Also I have introduced a grid for toroidal flux, $\Phi(\rho)$, although ρ might actually be Φ or $\sqrt{\Phi}$. For the present the added generality won't hurt.

Name	Type	Providing Component	Units	Definition
eqdsk_file	filename	Equilibrium	–	eqdsk file
B_axis	real	Equilibrium	*Tesla*	Magnetic field at magnetic axis
geometry	filename	Intitialization	–	File or file directory containing vessel shape, coil characteristics, limiters, walls
nrho_phit	integer	Equilibrium	–	Number of rho values in toroidal flux grid
rho_phit_grid	real	Equilibrium	–	rho values in toroidal flux grid
phit(1:nrho_phit)	real	Equilibrium		Toroidal flux
vprime(1:nrho_phit)	real	Equilibrium		Differential volume between flux surfaces

R_2_flux_ave(1:nrho_phit)	real	Equilibrium		Flux surface average $\left\langle 1/R^2 \right\rangle$
gradpsi_2(1:nrho_phit)	real	Equilbrium		Flux surface average $\left\langle \lvert\nabla\psi\rvert^2 \right\rangle$
gradpsi_R_2(1:nrho_phit)	real	Equilbrium		Flux surface average $\left\langle \lvert\nabla\psi\rvert^2 / R^2 \right\rangle$
nR	integer	Equilibrium	–	Number of R points in R, Z grid for psi
R_grid(1:nR)	real	Equilibrium	*m*	R values in R, Z grid
nZ	integer	Equilibrium	–	Number of Z points in R, Z grid for psi
Z_grid(1:nR)	real	Equilibrium	*m*	Z values in R, Z grid
Psi((1:nR, 1,nZ)	real	Equilibrium		Poloidal Flux ψ(R, Z)

RF Data:

We need to allow multiple RF sources (ICRH, LH, ECH) even multiple sources in the same frequency regime, therefore things that are normally scalars like power_rf , rf_frequency, and ant_model need to be arrays (1:nrf_src). The initial RF component implementation will likely only treat one RF source, but there is no complication in allowing for multiple sources in the data structures. Then describing RF sources is almost exactly parallel to describing particle species. Specification of antennas is complicated and code-specific (at least when comparing full-wave methods to ray tracing) therefore for the present ant_model will be a derived type that just contains a file name.

Name	**Type**	**Providing Component**	**Units**	**Definition**
Inputs				
nrf_src	integer	PS init	–	Number of RF sources
rf_src_name(1:nrf_src)	swim_name	PS init	–	Name of RF sources
power_src(1:nrf_src)	real	PS init	*MW*	Power in RF source
rf_freq_src(1:nrf_src)	real	PS init	*MHz*	Frequency of RF source
ant_model_src(1:nrf_src)	ant_model	PS init	–	Antenna model files

Outputs				
nrho_prf	integer	PS init	–	number of rho values in RF power deposition grid
rho_prf_grid(1:nrho_prf)	real	PS init	–	rho values in power deposition grid
ntheta_prf	integer	PS init	–	number of theta values in 2D power deposition grid
rho_prf_grid(1:ntheta_prf)	real	PS init	–	theta values in power deposition grid
prf_src_s(1:nrho_prf, 1:ntheta_prf, 1:nrf_src, 0:nspec)	real	PS init	MW/m^{-3}	2D power deposition profile from RF source into species s
prf_src_s(1:nrho_prf, 1:nrf_src, 0:nspec)	real	PS init	MW/m^{-3}	Flux surface average power deposition profile from RF source into species s
prf_total_s(1:nrf_src, 0:nspec)	real	PS init	MW	Total power from RF source into species s
nrho_cdrf	integer	PS init	–	number of rho values in RF driven current density deposition grid
rho_cdrf_grid(1:nrho_cdrf)	real	PS init	–	rho values in rf current drive grid
cdrf_rf(1:nrho_prf, 1:nrf_src)	real	PS init	Amp/m^{-2}	Flux surface average drven current profile from RF source
ql_operator(0:nspec_nonMax)	ql_operator	PS init	–	Quasilinear operator file

Appendix 3 – Simple example of Plasma State with three state variables

Note added in version 1.3: There is a more complete version of this example, available separately, that compiles and executes. It consists of two Fortran90 programs Prepare_AORSA_input.f95 and Process_AORSA_output.f90 that use `plasma_state_rf_mod` module to get data out of and back into a plasma state file. There is also a toy implementation of the Plasma State component that presently only contains the data needed for RF. These should be useful as a model for preparing input data and processing output data for other codes, and for plasma_state_<component> of other components.

To build this you need the following files: Prepare_AORSA_input.f95, Process_AORSA_output.f95, Plasma_state_rf_mod.f95, Plasma_state_mod.f95, and SWIM_global_data_mod.f95

To execute you will also need an initial plasma state data file. In this toy implementation of the Plasma State component the data file is just a Fortran binary file. You can generate such an initial file using the additional Fortran90 code make_initial_PS.f95. All of these files are available on the cswim svn repository and (maybe) on the cswim web site.

Old example:

Here we show a skeleton of use of the Plasma State model as presented in section III. Here there is only one other component RF_SOLVE the RF component being implemented by the AORSA code. For simplicity in this example there are only 3 quantities in the plasma state, two AORSA inputs, r_axis and B_axis and one AORSA output, total rf power deposition. For illustrative purposes we suppose that AORSA uses the name r0 for r_axis, that the name B_axis is the same as the Plasma State form, but that AORSA uses Gauss instead of Tesla. We assume that the total power deposition is called Prf in AORSA and Prf_total in the Plasma state.

Included are skeletons of two Fortran programs:

1) PREPARE_AORSA_INPUT which uses the `plasma_state_rf_mod` module, and calls that module's GET_PLASMA_STATE_RF function, and

2) PROCESS_AORSA_OUTPUT, which also uses the `plasma_state_rf_mod` module, and calls that module's PUT_PLASMA_STATE_RF function.

Also there is a skeleton of the `plasma_state_rf_mod` module and a skeleton for the `plasma_state_mod` module. The component developers will need to work with code developers to write PREPARE_<CODE>_INPUT and PROCESS_<CODE>_OUTPUT for each of the codes to be used in the component.

```
PROGRAM PREPARE_AORSA_INPUT

    USE plasma_state_rf_mod ! This declares the RF relevant PS variables

    !-----------------------------------------------------------------------
    !
    !   Declare AORSA-specific input variable names if different from Plasma
    !   State form.
    !
    !-----------------------------------------------------------------------

        REAL (KIND = rspec) ::  r0
        ! N.B. Not necessary to declare B_axis becasue it is declared in
        ! plasma_state_rf_mod and AORSA name is the same.

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   Declare variable names for non-PS input variables (none in this example).
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !   Load Plasma State Data relevant to RF into plasma_state_rf_mod module.
    !
    !-----------------------------------------------------------------------

        CALL GET_PLASMA_STATE_RF

    !-----------------------------------------------------------------------
    !
    !   Assign AORSA variables (if different from Plasma State form) from
    !   plasma_state_rf_mod module.
    !
    !-----------------------------------------------------------------------

        r0 = r_axis             ! Change to AORSA name
        B_axis = 10000.*B_axis  ! Change units to Gauss

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   Read files to get non-PS AORSA input variables (None in this example).
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   Write standard AORSA input file.
    !
    !-----------------------------------------------------------------------

END PROGRAM PREPARE_AORSA_INPUT
```

```
PROGRAM PROCESS_AORSA_OUTPUT

    USE plasma_state_rf_mod ! This declares the RF relevant PS variables

    !-----------------------------------------------------------------------
    !
    !   Declare AORSA-specific output variable names for PS data if different
    !   from Plasma State form
    !
    !-----------------------------------------------------------------------

        REAL (KIND = rspec) ::  Prf
        ! N.B. Not necessary to declare B_axis becasue it is declared in
        ! plasma_state_rf_mod and AORSA name is the same.

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   Declare variable names for non-PS output variables (none in this
    !   example).
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   Read stadard AORSA output files
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !   Convert AORSA output variables to plasma_state form.
    !
    !-----------------------------------------------------------------------

        Prf_total = Prf             ! Change from AORSA name to PS name

    !-----------------------------------------------------------------------
    !
    !   Update Plasma State Data relevant to RF
    !
    !-----------------------------------------------------------------------

        CALL PUT_PLASMA_STATE_RF

END PROGRAM PROCESS_AORSA_OUTPUT
```

```
MODULE plasma_state_rf_mod

    !-----------------------------------------------------------------------
    !
    !   Declare public variable names relevant to RF component
    !
    !-----------------------------------------------------------------------

        REAL (KIND = rspec) ::  r_axis, B_axis, Prf_total

    !-----------------------------------------------------------------------
    !
    !   End RF relevant Plasma State data declarations
    !
    !-----------------------------------------------------------------------

    !-----------------------------------------------------------------------
    !
    !   GET and PUT functions provided by plasma_state_rf_mod
    !
    !-----------------------------------------------------------------------

    CONTAINS

    SUBROUTINE GET_PLASMA_STATE_RF

        USE plasma_state_mod    ! This delares all of the Plasma State
                                ! semi-public names

    !-----------------------------------------------------------------------
    !
    !   Load all semi-public Plasma State data into plasma_state_mod module
    !
    !-----------------------------------------------------------------------

        CALL PS_GET_PLASMA_STATE

    !-----------------------------------------------------------------------
    !
    !   Assign public variables relevant to RF component from semi-public data
    !
    !-----------------------------------------------------------------------

        r_axis = PS_r_axis
        B_axis = PS_B_axis

    END SUBROUTINE GET_PLASMA_STATE_RF

    SUBROUTINE PUT_PLASMA_STATE_RF

        USE plasma_state_mod    ! This delares all of the Plasma State
                                ! semi-public names

    !-----------------------------------------------------------------------
    !
    !   Assign semi-public variables relevant to RF component from public data
    !
    !-----------------------------------------------------------------------

        PS_Prf_total = Prf_total

    !-----------------------------------------------------------------------
    !
    !   Store all semi-public Plasma State data into the Plasma State
    !
    !-----------------------------------------------------------------------

        CALL PS_STORE_PLASMA_STATE

    END SUBROUTINE PUT_PLASMA_STATE_RF

END MODULE plasma_state_rf_mod
```

```
MODULE plasma_state_mod

    !-----------------------------------------------------------------------
    !
    !   Declare all semi-public variable plasma state names
    !
    !-----------------------------------------------------------------------

        REAL (KIND = rspec) ::  PS_r_axis, PS_B_axis, PS_Prf_total

        REAL (KIND = rspec) ::  PS_r_axis_PREV, PS_B_axis_PREV, PS_Prf_total_PRE

    !-----------------------------------------------------------------------
    !
    !   End Plasma State data declarations
    !
    !-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!
!   GET and STORE functions provided by plasma_state_mod
!
!-----------------------------------------------------------------------

    CONTAINS

    SUBROUTINE PS_GET_PLASMA_STATE

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   XPLASMA calls that load the NETCDF file and put the plasma state data
    !   into the the semi-public variables
    !
    !-----------------------------------------------------------------------

        !   XPLASMA calls -> PS_r_axis, PS_B_axis

    END SUBROUTINE PS_GET_PLASMA_STATE

    SUBROUTINE PS_STORE_PLASMA_STATE

    !-----------------------------------------------------------------------
    !
    !   Insert code:
    !   XPLASMA calls that write the data in the semi-public variables into
    !   the current Plasma State the NETCDF file
    !
    !-----------------------------------------------------------------------

        !   XPLASMA calls  PS_Prf_total -> current Plasma State file

    END SUBROUTINE PS_STORE_PLASMA_STATE


END MODULE plasma_state_mod
```

Appendix 4. Issues for Further Discussion

1. Source for initial plasma state data – We can imagine (at least) two approaches to obtaining an initial plasma state:

1) We are given a plasma state NETCDF file in the simulation setup and control subdirectory that was previously generated externally by XPLASMA and is copied to the current_plasma_state file of the simulation results subdirectory.

2) We are given files in the simulation setup and control subdirectory containing data (maybe namelists and files with profile data) that describe the initial state of the simulation. And we produce a function that gathers up this data and uses XPLASMA calls to write this into the plasma state, and then writes the initial NETCDF file to the current_plasma_state file of the simulation results subdirectory.

2. Development of control logic – There are several points at which control logic is to be evaluated and adjustments to parameters of the simulation might be made. This logic will certainly evolve as we gain experience. A major point for further discussion is to develop a minimal set of algorithms to get started.

3. Communication of data between controller script and components – Another important issue to be resolved is how to communicate information back and forth between the controller script and the components in a generic way. This information would include changes to parameter settings made by the controller script, warnings and error messages generated by the components. One could imagine communicating such information through arguments to the component methods, through file exchange, and some data (such as changes in source inputs or coil currents) might logically be communicated via the Plasma State.

4. Definition of the simulation-wide radial coordinate ρ – For the present we take the radial coordinate, ρ, to be an arbitrary function monotonically increasing from the magnetic axis. It may be that there is no advantage to keeping this generality and that we will fix ρ as the toroidal flux or square root of the toroidal flux inside a given flux surface. Another possibility is to take ρ as the toroidal flux inside the last closed flux surface and to be some function of (R,Z) outside so as to give a simple way to specify densities, temperatures, etc still as functions of ρ outside the last closed flux surface.

5. Radial profiles as cumulative distributions – We agreed that all 1D profiles would be radial integrals of flux surface averages, i.e.

$$f(\rho = 0) = 0$$

$$f(\rho_i) = 2\pi \int_0^{\rho_i} \rho d\rho \langle f(\rho) \rangle$$

As I (*DBB*) understand it, this is desirable to guarantee conservation properties when interpolating profiles onto different grids. However perhaps we should discuss whether the data might be stored as the actual values on the grid points making the radial integration and subsequent differentiation a part of the profile interpolation process. This would probably be more intuitive for the component developers. The functionality could be packaged as part of the infrastructure layer.

6. Distinction between general particle species and adiabatically advanced species – Note that there is a distinction between the data in the Particle Species list and the Adiabatic Species list in Appendix 2 although some of the data is redundant. The Profile Advance component uses quantities in sufficiently different form that it makes sense to allow this redundancy. For example Profile Advance uses differential ion density rather than ion density and it uses electron entropy and total entropy rather than electron temperature and individual ion temperatures. The data in this list is more general than that provided by the advancement of adiabatic profiles in that it contains non-Maxwellian components, possible impurities, and possible minorities, which may not be advanced as adiabatic species. Models will be required to populate the data in this list from the adiabatic profiles, Fokker-Planck solutions, impurity models, and minority species models.

7. Separation of Profile Advance and Anomalous Transport as components – A question for further discussion is whether the transport coefficients (D, χ_e, etc) should live in the Plasma State. Will any other components use them? For that matter will the Anomalous Transport component be used outside of the Advance Adiabatic Profiles component? Since the PA component will likely take internal time steps it will need to use the Anomalous Transport component internally, so it will have to do the same task for Anomalous Transport that the controller script does for the other components. Perhaps we should eliminate Anomalous Transport as a separate component and make it part of Profile Advance. Or perhaps we should keep the strict component separation and let the controller script control the time substeps.

8. Communication of events and control variables – We have not dealt with how events such as pellet injection or predetermined sawtooth crashes and variables that might be considered controls (such as coil currents, neutral beam or RF power, RF frequency, or antenna phasing) should be passed into the components. They could be passed through some separate control-data flow that would resemble the plasma state, or they could be passed through the Plasma State component itself. The latter might be simpler since other components may also need this information. Whereas the antenna geometry doesn't change in time (at least for ICRH), other antenna quantities can change and act as controls during a shot (like relative strap phasing, equivalently n_phi, strap current, and even launch geometry for steerable ECH). Steve Jardin has suggested a Control component, I think he had in mind more of a feedback control component, but it could probably also be used in this context.

9. Order of update for Plasma State data – In the discussion of the Plasma State data above we refer to a set of data that is the value at the beginning of the time step, t_0, (<data>_PREV) and a set of current data that is being updated to $t_0 + \Delta t$ as successive components are being stepped. The order in which the components are stepped determines whether or not a given PS variable is available at $t_0 + \Delta t$ at any point. This raises two issues, 1) Care must be taken to step components that require some input data at $t_0 + \Delta t$ after the component that provides it. Is this a well-ordered set? and 2) Is this sufficient? Is it necessary for the components to know whether a given variable in the plasma state has already been updated for this time step? One can imagine at least two solutions:

a) Do nothing – Ensure in the controller script that components are only stepped when the data at the advanced time is available.

b) Include a flag for each variable that is set by the providing component when the data is updated in the Plasma State

Appendix 5. Simulation directory structure

There are three classes of files involved in a SWIM simulation

- Input files
- Results files
- Component working files

It is not necessary for all of these to be under the same directory but for simplicity we show the layout here as if that were the case. The framework figures out what to name the top level Simulation Directory (e.g. /…/`IPS_run_XYZ`), where to put it and how to make the path to this directory known to the component scripts (if in fact they need to know it). Furthermore we assume that the "computer environment initialization" step in the Controller Script (see Section II) creates the directory tree.

`IPS_run_XYZ`

|-- `simulation_setup` *(input files, this directory is never overwritten)*

|-- `simulation_results` *(output files)*

|-- `work` *(working files with a subdirectory for each component)*

To do a simulation the user will have to assemble somewhere in his user area a directory containing all of the necessary control files and initial input files for the run (described in Setup and Input sub-directory of Section I.1). There has to be a mechanism for the SWIM framework to know where this is (e.g. launching from the top level of this directory). The framework initially copies the `/simulation_setup` directory from the user area (or other simulation input staging area) into the Simulation Directory.

The current Plasma State file resides in the `/work` subdirectory. It is put there initially by the controller script and is updated after a successful time step by the Plasma State function PS_COMMIT_PLASMA_STATE. The controller script also copies the initial input data for each component from the appropriate component subdirectory of `simulation_setup` into the component subdirectory of `work`. After all of the components have successfully completed the time step the controller script creates a new subdirectory for that time step (identified by the time in milliseconds) in the `/history` and `/restart` subdirectories. It then gathers up the appropriate files in each component working directory and copies them to history and restart for the time step. It is the job of the component scripts to leave the working directories in condition to allow the controller script to effect that transfer. In particular the component leaves a list of files to be transferred in `component_outfile_list`. In this design code developers and

component developers need only be concerned about the structure of the `/work` subdirectory.

The subdirectory layout is shown below:

simulation_setup subdirectory

simulation_setup

- system_config
- simulation_wide_data
  - simulation_control_file (this is a file not a directory)
  - initial_plasma_state (t_0) (this is a file not a directory)
  - machine_definition (may be empty for now)
  - simulation_events_waveforms (scheduled events, source and control waveforms)
- component_inputs
  - RF
    - RF_component (the component may need generic files)
      - RF_config
      - RF_required_list
      - RF_outfile_list
    - code_inputs (code specific files e.g. AORSA)
      - AORSA_required_list
      - AORSA_outfile_list
      - Standard AORSA input files …
  - FokkerPlanck
    - FP_component
      - FP_config
      - FP_required_list
      - FP_outfile_list
    - code_inputs (code specific input files e.g. CQL3D)
      - CQL3D_required_list
      - CQL3D_outfile_list
      - Standard CQL3D input files…
  - …
  - <other components>…

simulation_results subdirectory

simulation_results

```
|-- history
    |-- t0 (identifier in milliseconds)
        |-- plasma_state
        |-- components
            |-- RF
            |-- Fokker Planck
            …
            |-- <other components>
    |-- t1 (identifier in milliseconds)
        |-- plasma_state
        |-- components
            |-- RF
            |-- Fokker Planck
            …
            |-- <other components>
    …
    |-- tN…<other time steps>

|-- restart
    |-- t0 (identifier in milliseconds)
        |-- plasma_state
        |-- components
            |-- RF
            |-- Fokker Planck
            …
            |-- <other components>
    |-- t1 (identifier in milliseconds)
        |-- plasma_state
        |-- components
```

```
        |-- RF
        |-- Fokker Planck
        …
        |-- <other components>
…
|-- tN…<other time steps>
```

work subdirectory

work

```
  |-- plasma_state (current plasma state t_n)
  |-- RF
    |-- RF_component (the component may need generic files)
        |-- RF_config
        |-- RF_required_list
        |-- RF_outfile_list
        |-- RF_log
    |-- code_inputs (code specific input files e.g. AORSA)
        |-- AORSA_required_list
        |-- AORSA_outfile_list
        |-- AORSA standard input files …
    |-- RF_component_script (?)
    |-- executable (?)
    |-- code_outputs
        |-- AORSA standard input files …
        |-- AORSA_log
    |-- scratch
  …
  |-- <other components>…
```

Appendix 6. Change log

Push down list that should enable the reader to quickly tell if anything has changed that affects them.

8-22-06	Added note on more complete Plasma State example to Appendix 3 and added appendix on simulation file directory layout (Batchelor).
8–14–06	Added this appendix (Batchelor)
8–14–06	Added function PS_COMMIT_STATE to Plasma State component requirements (Batchelor)
8–14–06	Editorial changes – Table of contents, abstract (Batchelor)