INTEGRATED PLASMA SIMULATOR USER'S GUIDE

SAMANTHA S. FOLEY, AND THE SWIM IPS DEVELOPMENT TEAM

1. Overview of the IPS

The purpose of this document is to acquaint new users to the Integrated Plasma Simulator (IPS) a framework for component coupling developed for the Center for Simulation of RF Wave Interactions with Magnetohydrodynamics (SWIM) project. The rest of the document is organized into the following sections: motivation and purpose, design and implementation, what you can do with the IPS, and how to get started. For those just interested in creating components and simulations, we recommend skipping the motivation section, skimming the design and implementation and reading the last two sections. For those more interested in the computer science and design choices, the motivation and design sections will be useful, as well as the publications listed therein.

This document is a work in progress and may have varying degrees of completion and detail.

1.1. **Motivation and Purpose.**

1.2. **Design and Implementation.**

1.3. **What YOU can do with the IPS.**

The IPS framework provides a flexible environment to couple multiple components concurrently and serially. The framework provides services for file-based data management, task coupling, configuration, monitoring and resource management within a single batch allocation. It also provides an event service for adding new functionality and communication.

The execution model, that separates component method invocations and the launch of (parallel) binaries, allows for many different blocking and non- blocking task coupling scenarios. Additionally, multiple simulations may be executed within the same framework instance, within the same batch allocation.

1.4. **How to get started.**

The first thing to do after checking out the source code, is to create any components or drivers you wish to use in your simulation. See `components.txt` and the skeleton implementations in the samples directory. Details on how to add components and build and run the system are provided in `components.txt` and `simulations.txt`.

`services.txt` is to be used as a reference for component developers of what services are available and how to use them.

2. Guide to IPS Components

In the IPS, *component* refers to the python script that interacts with the framework, services, other components and (optionally) launches binaries to perform a specific function for the simulation. These components tend to be physics components that perform a specific modeling function for a coupled physics simulation.

This guide will help you construct a component (or driver) to be used in various simulations. You will most likely need to refer to services.txt for the services that framework provides for component activities. Also, some examples are provided in the sample directory and can serve as a skeleton for implementing your new components!

You will want to put newly created components in their proper directories under the `$IPS_ROOT/components/` directory (where `$IPS_ROOT` is your checkout of the IPS). The components directory is organized by class and subclass. For instance, all components that are drivers are in the class driver, and the subclass describes the implementation of the driver. This reflects the desire of the SWIM project to collect multiple implementations of different types of components. For instance, *AORSA* and *TORIC* are both RF codes, and are located in the `components/rf/` directory there are directories for `aorsa` and `toric`, respectively. In the `components/rf/ aorsa/` directory the source for the aorsa component resides, along with any other supporting files (like a makefile). The AORSA code resides in the `phys_bin/` a separate directory that contains the binaries of codes being used by SWIM. This directory lives on the target platforms and contains compiled codes that work on that platform as provided by the code owner. See the building and running sections of the `simulations.txt` file, and the `README` file in the top level of the IPS for more information on building and running the IPS and the file structure.

2.1. **Component Object Structure:**

A component implementation inherits some functionality from the IPS Component object to perform some common startup and termination tasks. Component writers do not need to worry about these calls.

These functions should be copied from the skeleton component and not modified in most cases:

`__init__(self, services, config)`:

This function gets and saves the component specific configuration information for use by the component instance throughout the simulation, and sets up the reference to the services that the component instance can use.

These functions contain the physics functionality of the component and should be written by the component writer. Please *augment* the skeleton component, as there might be some lines of code that all components need to function properly in the system.

`init(self, timestamp)`:

Any initial set up for the component is done here. "timestamp" is the simulation timestamp from the driver's time loop. This function typically gets called at the beginning of the simulation for any pre-simulation processing. Additional per-step processing may happen in the step function.

`step(self, timestamp)`:

This is where the component performs its calculations at a particular simulation level timestamp. This function typically gets called during each iteration through the time loop. Services related to launching tasks, and data management for the step should be used here. See the services section for details on how to choose the right function. Step pre- and post-processing may be done here as well.

`finalize(self, timestamp)`:

This is where the component performs any clean up after the simulation is over. It typically gets called at the end of the simulation after the simulation has completed.

2.2. **Driver Object Structure:** The driver has the same structure as the component with the following exceptions:

- the init happens before the "simulation" starts
- the step function contains all of the logic for the simulation. It contains three phases:
  - setup:
    - component references are obtained and initialized
    - timeloop is obtained from configuration
  - time loop:
    - in the time loop, components are called. The simulation workflow logic is implemented using the different call methods. See the services section for details.
  - clean up:
    - components are finalized

- finalize occurs after all of the components have been finalized.

3. Guide to IPS Services

In this section the functions that the services provide to components are described. It is broken down into the following groups:

- General Purpose
- Task Launch
- Component Invocations
- Data Management
- Logging
- Event Service

3.1. General Purpose Services.

This section contains information about services to access configuration information.

`get_config_param(self, param) return value`:

This function returns the value of configuration parameter `param`. Platform, simulation and dynamic configuration parameters can be retrieved in this way, however use `get_port` for retrieving component references (see Component Invocation section for `get_port` details). An exception is thrown if the parameter is not found or there is an error retrieving it.

`set_config_param(self, param, value, target_sim_name=None)`
`return retval or None`:

This function sets the value of configuration parameter `param` if it is a dynamic configuration parameter. If the parameter is a dynamic config parameters, the value is set and returned on success, an exception is raised on failure. If the parameter is not a dynamic config parameter, the return value is `None`.

`get_time_loop(self) return tlist`:

This function returns a list of simulation time values as specified in the simulation configuration file.

`get_working_dir(self) return workdir`:

This function returns the working directory for the calling component. The structure of the working directory is:

```
$SIM_ROOT/work/${CLASS}_${SUB_CLASS}_${NAME}_${INSTANCE_NUM}
```

3.2. **Task Launch Services.** These are the services for launching and managing tasks in the component. They are typically used in the step function of a physics component to launch binaries.

`launch_task(self, nproc, working_dir, binary, *args, **keywords) return task_id`:

This function launches binary `binary` in working directory `working_dir` on `nproc` processes with `*args` and `**keywords` (in python `*args` refers to any number of arbitrary arguments, and `**keywords` refers to any number of `keyword=value` pairs. In this context, the `*args` are for the binary, and the `**keywords` are used by the services and framework.). It is a non-blocking call because a new process is created to do the launch of the task, and the `task_id` (a logical ID created by the task manager) is the handle to refer to this task. There are several exceptions that can be raised:

- an exception from initializing the task (this could be related to the number of processes requested or constructing the launch string)
- an exception when launching the process

`kill_task(self, task_id)`:

This function kills task `task_id`. An exception is raised if the task id is not found, if there were problems terminating the process, or there are problems finalizing the task.

`kill_all_tasks(self)`:

This function kills all tasks that the component currently has executing. It calls `kill_task` on each one, and will raise the first exception it encounters. It returns successfully if no exceptions are raised.

`wait_task_nonblocking(self, task_id) return retval or None`:

This function checks the status of task `task_id`, if it has completed, the task is finalized and the return value returned, otherwise, `None` is returned. An exception is raised if `task_id` is invalid or there are problems finalizing the task.

`wait_task(self, task_id) return retval`:

This function waits until task `task_id` has completed, finalizes it and returns the return value. An exception is raised if `task_id` is invalid or there are problems finalizing the task.

`wait_tasklist(self, task_id_list) return ret_dict`:

This function does a blocking wait on all tasks in `task_id_list` and a dictionary of `task_id`s and return values is returned. An exception is thrown if any `task_id` is not found, or there are problems finalizing any task.

3.3. **Component Invocation Services.** These are the services necessary for calling components. They are typically used by the driver.

`get_port(self, port_name) returns component_reference`:

This function looks up and returns a reference to the component that is specified in the configuration file as `port_name`. Exceptions should be caught in component to ensure that the component reference was received successfully.

`call_nonblocking(self, component_id, method_name, *args) returns call_id`:

This function invokes the method `method_name` on component `component_id` with arguments `*args` and returns the `call_id`. You must use a `wait_call()` function (see below) to get the results of the call, or the simulation will hang until the entire simulation is terminated by an external force (most likely, the batch scheduler will notice that you are still running after your allocated time is up and kill your job).

`call(self, component_id, method_name, *args) return retval`:

This function invokes the method `method_name` on component `component_id` with arguments `*args`, then waits for the result and returns it to the caller (`retval`). This is a blocking call.

`wait_call(self, call_id, block=True) return retval`:

This function waits for the result from call `call_id` and returns the results. By default, this is a blocking call and will not return until the call has finished. If `block` is set to `False`, the function will raise `ipsExceptions.IncompleteCallException` if the call has not yet finished.

`wait_call_list(self, call_id_list, block=True) return ret_map`:

This function waits on all call IDs in `call_id_list` then returns a dictionary where the keys are the `call_ids` and the values are the corresponding return values. By default this call blocks until *all* calls have finished. If `block` is set to `False`, the function will raise

`ipsExceptions.IncompleteCallException` if any call has not yet finished.

3.4. **Data Management Services.** These services are for dealing with input, output and plasma state files.

`stage_input_files(self, input_file_list)`:

This function copies files in `input_file_list` from the input file directory specified in the configuration file, to the component's working directory. Also copies files to:

```
$SIM_ROOT/simulation_setup/${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

Exceptions are raised if there are problems on the second copy.

`stage_output_files(self, timestamp, file_list)`:

This function copies output files in `file_list` to:

```
$SIM_ROOT/simulation_results/${timestamp}/components/      \
      ${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

Exceptions are raised if there are problems.

`stage_plasma_state(self)`:

This function copies the current master plasma state files to the component's working directory.

`save_restart_files(self, timestamp, file_list)`:

This function copies files needed for component restart to the restart directory:

```
$SIM_ROOT/restart/$timestamp/components/       \
      ${CLASS}_${SUB_CLASS}_${NAME}_${SEQ_NUM}
```

Exceptions are raised if there are problems.

`get_restart_files(self, restart_root, timestamp, file_list)`:

This function copies files needed to restart a component from the restart directory to the component's work directory. Exceptions are raised if there are problems.

`update_plasma_state(self)`:

This function copies component's current version of the plasma state files to the master plasma state. *Note: this will overwrite the current plasma state with the calling component's version.* This is ok for serial execution of components, however, the `merge_plasma_state()` function is best for concurrent execution. Exceptions are raised if there are problems.

`merge_current_plasma_state(self, partial_state_file, logfile=None)`:

This function merges a partial plasma state file with the current master plasma state. Exceptions are raised if there are problems.

3.5. **Logging Services.** These services allow you to log information about the simulation so you can look at it after the simulation ends and see what went wrong and when. All messages are aggregated over the simulation and printed to a single file. The functions below only differ in the conditions in which they are printed. The logging level is set at the beginning of the simulation.

```
log(self, *args)
debug(self, *args)
info(self, *args)
warning(self, *args)
error(self, *args)
exception(self, *args)
critical(self, *args)
```

Note: ∗`args` should really be a formatted string with any variables as the following arguments in order. Variables can be used in the string like so:
`services.log(‘‘this is my string with %s added to it when %s happens’’, x, y)`
(where x and y are strings).

3.6. **Event Service.** These calls are for components that are using the event service. Note that the event name, event body and topic names must be agreed upon by the communicating parties. The event service does not check for properly constructed events or active topics.

`publish(self, topicName, eventName, eventBody)`:

This function publishes an event with name `eventName` and body `eventBody` to topic `topicName`.

`subscribe(self, topicName, callback)`:

This function subscribes the component to topic `topicName` so that callback `callback` is called when `process_events` is called. The function listed as the callback should be written such that it handles exactly one event. The callback will be called for each event that is received on that topic.

`unsubscribe(self, topicName)`:

This function unsubscribed the component from topic `topicName`.

`process_events(self)`:

This function processes all events on all topics to which the component is subscribed. For each event, the callback that is registered for that event topic is called.

4. Guide to Creating Simulations using the IPS

Once you have all of the components and drivers that you want to use in order, you will need to construct a configuration file, build and run the IPS. This guide will explain the elements of the configuration, and brief instructions on how to build and run the IPS with your simulation.

4.1. **Configuration: Elements of a Simulation.**

A simulation in the IPS is defined by the contents of the configuration file. It is in the configuration files that all of the components, locations of data and source files, and other options are set. There are seven sections of the simulation configuration file. They are briefly described below and annotated in the sample directory.

4.1.1. *Paths and Run ID.* In this section the paths to the `IPS_ROOT` (framework source tree), `SIM_ROOT` (where you want the results to go) and the name of the run are specified. Some other variables relating to these items are also set for convenience. It is highly recommended that you change the name of the `SIM_ROOT` for each run so as to not clobber your previous results.

4.1.2. *Plasma State Configuration.* In this section the plasma state working directory is specified as the place for plasma state files to be recorded. The names of the files that are going to be used as the plasma state are also recorded here.

4.1.3. *Portal and Logging.* In this section variables used by the web portal specific to this simulation are set, as well as the simulation log level and location. The log level controls how much debugging information is provided by the framework. Each component can set their own log levels in the component configuration section.

4.1.4. *Port Configuration.* This section contains the names of the ports that will be used by the driver, along with the implementations. Ports are similar to the class names of the components, they refer to the functionality that the component implements. The port names are how the driver accesses the reference to the component. Be sure these names match what the driver is expecting.

4.1.5. *Component Configuration.* This section describes how each component should be run. The class, subclass and name are used to create the IPS name of the component and the directory structure. `NPROC` is the number of processes the binary needs to run. `BIN_PATH` is the path to the component script. Input and output files are also listed here. Any files not listed will not be captured by the data management system.

4.1.6. *Time Loop Configuration.* This section describes how the simulation level time loop is constructed. There are two modes, `REGULAR` (a list of values from start to finish (inclusive) with the specified interval are created) and `EXPLICIT` (a list of times (space separated) is given explicitly by the user).

4.2. **Platform Configuration.** In addition to the simulation configuration file, a platform configuration file must be provided to the IPS at launch time. The platform configuration file tends to remain unchanged per platform. There are examples in the top level of the IPS and in the sample directory.

Advanced users may use the `NODES` and `CORES_PER_NODE` entries to specify the node and core counts available to the simulation for arbitrary machines. Users should note that the IPS will not check these values against what the machine reports.

4.3. **Building the IPS.** Before building and running the IPS, you should read the `README` file in the top level of the IPS tree. It contains important information about dependencies and the directory structure.

You need to source `swim.bashrc.*` (the star depends on your platform, for example, franklin) to set up variables and load modules. Now you are ready to build and install the IPS.

These three steps are performed by the following commands in the top level of the IPS tree:

```
. swim.bashrc.<platform>
make
make install
```

4.4. **Running the IPS.** The IPS has been ported to the Cray XT4/XT5 machines jaguar (at NCCS) and franklin (at NERSC), as well as viz/mhd at PPPL (a shared memory machine). The IPS will run on arbitrary machines if the number of nodes available in the allocation are specified in the platform configuration file. If no resource information is obtained the IPS will assume there is only on node with one core available and will run everything on one core serially. Running the IPS in serial mode is useful for testing the component and services interactions.

To run the IPS several command line arguments must be provided as such:

```
ips [--config=CONFIG_FILE_NAME]+ --platform=PLATFORM_FILE_NAME     \
     --log=LOG_FILE_NAME [--debug] [--ftb]

Arguments:
  --config
    A simulation configuration file.  There can be multiple simulations
    executing but each must be described in separate, uniquely named
    configuration files.  At least one must be provided.
  --platform
    The platform file that matches the platform you are using.
  --log
    The name of the log file containing all the debugging output.
  --debug
    Flag to turn on debugging output.  Optional.
  --ftb
    Flag to turn on FTB notifications.  Optional, experimental, only
    available on jaguar.
```

where, `ips` is the IPS executable.

Use `create_batch_script.py` (located in `IPS_ROOT/framework/src/`) to create a batch script to submit on a particular machine. It does some error checking. There is a sample batch script in the sample directory if you wish to create and edit your own manually.

```
python create_batch_script.py --ips=IPS_EXECUTABLE             \
                              [--config=CONFIG_FILE_NAME]+     \
                               --platform=PLATFORM_FILE_NAME   \
                              [--account=CHRGE_ACCOUNT]        \
                              [--queue=BATCH_QUEUE]            \
                              [--walltime=ALLOCATION_TIME]     \
                              [--nproc=NPROCESSES]             \
                              [--debug]                        \
                              [--ftb]                          \
                              [--output=BATCH_SCRIPT]
```

```
Arguments:
  --ips
     This is the IPS executable using the full path.
  --config
     A simulation configuration file.  There can be multiple simulations
     executing but each must be described in separate, uniquely named
     configuration files.  At least one must be provided.
  --platform
     The platform file that matches the platform you are using.
  --account
     The account that will be charged for time on the machine.
  --queue
     The queue to which you will submit the job.
  --walltime
     The time to request from the batch scheduler.
  --nproc
     The number of processes to request from the batch scheduler.
  --debug
     Flag to turn on debugging output.
  --ftb
     Flag to turn on FTB notifications.  Only available on jaguar.
  --output
     The name of the batch script you want to generate.
```