MP-Sim 1.0
User's Manual

Haeyong (David) Shin Ray D. Zimmerman

April 17, 2018

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

MP-Sim is a package of MATLAB language M-files[1] for simulating a variety of scheduling problems. The MP-Sim project page can be found at:

https://github.com/MATPOWER/mpsim

MP-Sim was developed by Haeyong (David) Shin as an undergraduate student at Cornell University under the supervision of Ray D. Zimmerman of PSERC[2] at Cornell University for running simulations pertaining to electric power systems such as unit commitment and economic dispatch problems. For purposes of illustrating the structure of MP-Sim, a burger shop example is used throughout this manual.

[1] Also compatible with GNU Octave [1].
[2] http://pserc.org/

1.2 License and Terms of Use

The code in MP-Sim is distributed under the 3-clause BSD license [2]. The full text of the license can be found in the `LICENSE` file at the top level of the distribution or at https://github.com/MATPOWER/mpsim/blob/master/LICENSE and reads as follows.

1.3 Citing MP-Sim

While not required by the terms of the license, we do request that publications derived from the use of MP-Sim explicitly acknowledge that fact by citing this manual [3].

> H. Shin and R. D. Zimmerman, "MP-Sim User's Manual," 2017. [Online]. Available: http://www.pserc.cornell.edu/matpower/docs/MP-Sim-manual-1.0.pdf

1.4 MP-Sim Development

The MP-Sim project is based on an open development paradigm, hosted on the MP-Sim GitHub project page:

`https://github.com/MATPOWER/mpsim`

The MP-Sim GitHub project hosts the public Git code repository as well as a public issue tracker for handling bug reports, patches, and other issues and contributions. There are separate GitHub hosted repositories and issue trackers for MATPOWER, MOST, MIPS, MP-Sim and the testing framework used by all of them, MP-Test, all available from `https://github.com/MATPOWER/`.

2 Getting Started

2.1 System Requirements

To use MP-Sim 1.0 you will need:

- MATLAB® version 7 (R14) or later[3], or
- GNU Octave version 3.4 or later[4]
- MP-Test, for running the MP-Sim test suite.[5]

For the hardware requirements, please refer to the system requirements for the version of MATLAB[6] or Octave that you are using.

In this manual, references to MATLAB usually apply to Octave as well.

[3]MATLAB is available from The MathWorks, Inc. (`http://www.mathworks.com/`). MATLAB is a registered trademark of The MathWorks, Inc.

[4]GNU Octave [1] is free software, available online at `http://www.gnu.org/software/octave/`.

[5]MP-Test is available at `https://github.com/MATPOWER/mptest`.

[6]`http://www.mathworks.com/support/sysreq/previous_releases.html`

2.2 Installation

Installation and use of MP-Sim requires familiarity with the basic operation of MATLAB or Octave, including setting up your MATLAB path.

Step 1: Clone the repository or download and extract the zip file of the MP-Sim distribution from the MP-Sim project page[7] to the location of your choice. The files in the resulting `mpsim` or `mpsimXXX` directory, where `XXX` depends on the version of MP-Sim, should not need to be modified, so it is recommended that they be kept separate from your own code. We will use <*MPSIM*> to denote the path to this directory.

Step 2: Add the following directories to your MATLAB or Octave path:

- <*MPSIM*>`/lib` – core MP-Sim classes and functions
- <*MPSIM*>`/lib/t` – test scripts for MP-Sim

Step 3: At the MATLAB prompt, type `test_mpsim` to run the test suite and verify that MP-Sim is properly installed and functioning.[8] The result should resemble the following:

```
>> test_mpsim
t_mpsim_shared_x_numeric....ok
t_mpsim_shared_x_queue......ok
t_mpsim_process.............ok
t_mpsim.....................ok
t_burger_shop...............ok
t_burger_shop_2d............ok
t_opf_sim...................ok (8 of 8 skipped)
All tests successful (186 passed, 8 skipped of 194)
Elapsed time 1.03 seconds.
```

Step 4: *(optional)* Edit the <*MPSIM*>`/lib/mpsim_config.m` file to specify the path to the base directories for MP-Sim inputs, outputs and temporary work files, found in the variables `inputdir`, `outputdir` and `workdir`, respectively. The values of these paths are denoted by <*INPUTDIR*>, <*OUTPUTDIR*> and <*WORKDIR*>. If the `inputdir` variable is left blank, `'`<*MPSIM*>`/sim_data'` will be used by default as the <*INPUTDIR*>. If `outputdir` and `workdir` are left blank, the default is to use the value of <*INPUTDIR*>.

[7] https://github.com/MATPOWER/mpsim

[8] The tests require a functioning installation of MP-Test.

2.3 Running a Simulation

Running a simulation based on MP-Sim requires access to (1) a simulator, consisting of the set of subclasses that implement the simulator and its various process and state objects, and (2) any input data required by the simulation.

2.3.1 Simulator

The subclasses that implement the simulator and its process and state objects are user-defined and may be placed anywhere in your MATLAB or Octave path. It is recommended that they be kept outside of <*MPSIM*>, separate from the MP-Sim distribution, which should not need to be modified by the user. For instance, the `burger_shop` and `opf_sim` classes in <*MPSIM*>`/lib/t` are examples of simulators.

2.3.2 Input Data

For a given simulator, such as `burger_shop`, the inputs for a given batch of simulation runs are identified by a *simulation name* which we will denote <*SIMNAME*>. The simulator will look for the input data inside the <*INPUTDIR*> directory specified in <*MPSIM*>`/lib/mpsim_config.m`, more specifically, in the simulation-specific input directory <*INPUTDIR*>`/`<*SIMNAME*>`/inputs`, which we denote <*SIMINPUTDIR*>. The organization and format of the input files within this directory are completely user-defined. The only requirement is that they be consistent with what is expected by the simulator code. The inputs for the burger shop example can be found in <*MPSIM*>`/sim_data/burger_shop_example/inputs`, for instance.

2.3.3 Executing the Simulation

To run a simulation, instantiate the simulator object and call its `run` method with the name of the simulation. For example, to run a sample simulation called `'burger_shop_example'` using the `burger_shop` simulator, type:

```
sim = burger_shop();
sim.run('burger_shop_example');
```

or more succinctly:

```
burger_shop().run('burger_shop_example');
```

Simulation options can be supplied to `run()` via additional arguments as name/value pairs, or as a struct.[9] For example, to turn on verbose display of simulation progress and turn off the post-run pretty printed summary of burger shop activity and inventory levels, type:

```
burger_shop().run('burger_shop_example', 'verbose', 1, 'post_run_on', 0);
```

For a list of all options, please see Table 4-7.

2.3.4 Accessing the Results

If the simulator creates output files in the course of execution, they will be found in <*SIMOUTPUTDIR*>. The organization and format of the output files within this directory are also completely user-defined.

2.4 Documentation

There are two primary sources of documentation for MP-Sim. The first is this manual, which gives an overview of the capabilities and structure of MP-Sim and provides a reference for the classes, properties and methods, and tutorial for creating your own simulation. This manual can be found in your MP-Sim distribution at <*MPSIM*>`/docs/MP-Sim-manual.pdf` and the latest version is always available at: `https://github.com/MATPOWER/mpsim/blob/master/MP-Sim-manual.pdf`.

The second source of documentation is the built-in `help` command. As with the built-in functions and toolbox routines in MATLAB and Octave, you can type `help` followed by the name of a command or M-file to get help on that particular function. All of the M-files in MP-Sim have such documentation and this should be considered the main reference for the calling options for each function.

[9]Or as a combination of the two, with the options struct as the last argument.

3 Structure of MP-Sim

MP-Sim is a general-purpose simulator that is easily adaptable to a myriad of situations, including those pertaining to electric power systems such as unit commitment and economic dispatch problems. MP-Sim runs one or more simulations across a sequence of discrete time steps, calling on a set of processes that execute at a certain frequency to update a well-defined state from one step to the next throughout each simulation.

3.1 Terminology

The following list of terms will be used according to their given definition:

Simulation: An operation consisting of updating a state x throughout a sequence of discrete time steps, where new input may become available and additional outputs may be computed at each step.

Simulator: A tool used to run a simulation, specifically an instance of an `mpsim` object. A subclass of `mpsim` is used to define the components and behaviors of a given simulator and the simulations it runs.

Simulation Run: The execution of a single instance of a simulation, starting from a given initial state, along with its inputs and outputs.

Simulation Batch: A set of simulation runs grouped together under a single name and with a shared initial state.

Simulation Name: The name assigned to identify a particular simulation batch.

Simulator State $\boldsymbol{x}$: The set of information that fully describes the current conditions at a given time step, summarizing any history of past actions. The state consists of two kinds of information, process-specific state and shared state.

Input $\boldsymbol{u}$: Data from outside the simulation that becomes available to the simulation at a particular time step, used to determine the state at the same time step.

Output $\boldsymbol{y}$: Any additional information, besides the state, computed at each time step in a simulation. The simulation "results" generally consist of a summary of the individual per-step outputs y.

Update function $\boldsymbol{f}(\cdot)$: A function that determines the state x at each time step based on the state at the previous time step and the current input.

Output function $g(\cdot)$: A function that generates simulation output y at each time step based on the state at the previous time step and the current input.

Process: A task or operation performed in a simulation run with specified frequency and duration. The operations performed by a simulator are grouped into processes. Both the state x and the output y, along with the update and output functions $f()$ and $g()$ are partitioned according to the set of processes defined by the simulator. Each process has its own output function which determines its portion of the output y, and its own update function which updates its portion of the state x and possibly shared portions.

Process-specific state: The portion of the state x corresponding to a specific process. This portion is updated by the update function of the corresponding process, and only by that function. While the update function of a process may modify only *its own* process-specific portion of the state (and shared portions), it has read access to the entire state of the previous time step, including the process-specific portions corresponding to other processes.

Shared state: A portion of the state x that can be updated by more than one process. The update function of a specific process can optionally update any portion of the shared state in addition to its own process-specific state.

User: The person using MP-Sim to construct a simulator and/or set up and run simulations. "User-defined" refers to something that the user must implement or define.

Trigger: To initiate the execution of the update function of a process. Each process triggers at particular time steps within the simulation. For a process triggering at time step t, it is the state at $t - 1$ and the input at t that are available as inputs to the update and output functions of the process. Depending on the duration of the process, it may "run" for more than one time step, in which case the updated state values are not applied to the state until the time step in which the process completes or *finalizes*.

Finalize: To complete the execution of the update function of a process. For a previously *triggered* process that finalizes at time step t, the state updates computed by the corresponding update function are applied at t, and therefore only available as inputs to processes triggered at $t + 1$ and beyond.

Shared state value: The value that is stored in the shared part of the state x.

Shared state object: The object used to implement a shared state and manage and update the corresponding *shared state value* contained in x.

Shared state update: A set of values and corresponding operations used by the shared state object to update the current shared state value. To avoid overwriting changes made by other processes to a shared state value, each process computes an update consisting of a shared state update value along with the operation used to apply it. All such updates are then applied together.

3.2 Model Overview

MP-Sim provides the framework for a *user* to construct a *simulator* and run *simulations* consisting of a set of scheduled tasks or *processes* that run with a user-specified frequency and duration. These processes operate on a *state* that is updated from time step to time step as each *simulation run* proceeds. The processes can produce *output* at each time step as well. The state x is updated by the *update function*, denoted by $f(\cdot)$ and the output y is produced by the *output function*, denoted by $g(\cdot)$. The processes and the simulator state are initialized when the simulator object is instantiated.

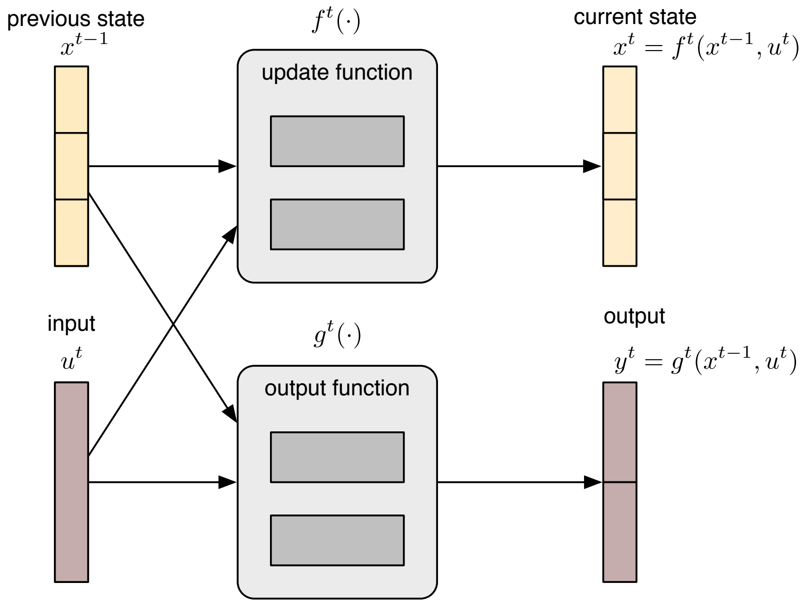

Figure 3-1: Simulator Update and Output Functions

For a given time step t, Figure 3-1 illustrates a high-level view of the update and output functions used to compute the state x^t and output y^t from the previous state x^{t-1} and current input u^t.

$$x^t = f^t(x^{t-1}, u^t) \tag{3.1}$$
$$y^t = g^t(x^{t-1}, u^t) \tag{3.2}$$

3.2.1 Partitioning by Process

The state x consists of two types of components, *process-specific state* updated only by the corresponding process and *shared state* updated by more than one process.

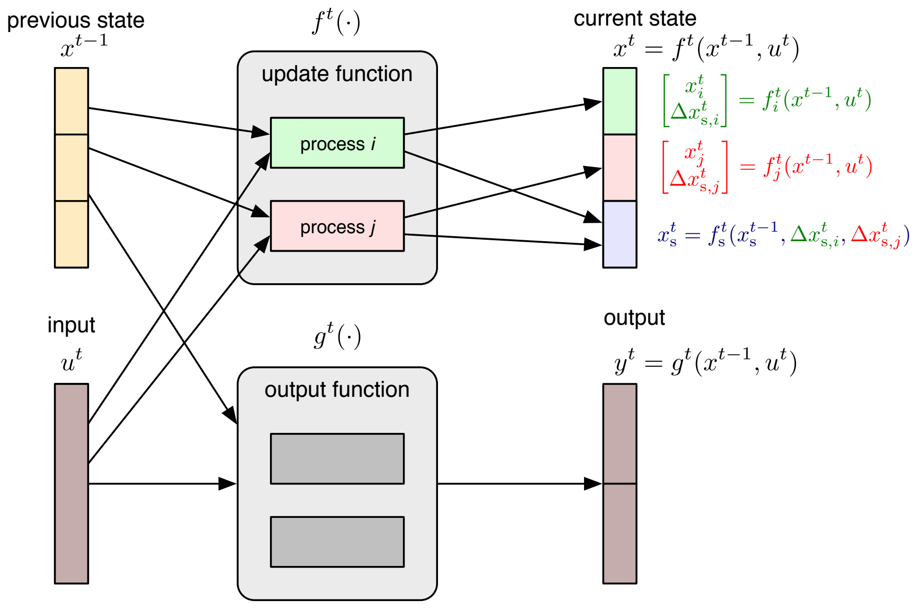
previous state
x^{t-1}
$f^t(\cdot)$
update function
process i
process j
current state
$x^t = f^t(x^{t-1}, u^t)$
$\begin{bmatrix} x_i^t \\ \Delta x_{s,i}^t \end{bmatrix} = f_i^t(x^{t-1}, u^t)$
$\begin{bmatrix} x_j^t \\ \Delta x_{s,j}^t \end{bmatrix} = f_j^t(x^{t-1}, u^t)$
$x_s^t = f_s^t(x_s^{t-1}, \Delta x_{s,i}^t, \Delta x_{s,j}^t)$
input
u^t
$g^t(\cdot)$
output function
output
$y^t = g^t(x^{t-1}, u^t)$

Figure 3-2: States and Update Functions Partitioned by Process

The overall update and output functions $f(\cdot)$ and $g(\cdot)$ are partitioned by process, as is the process-specific portion of the state and the output y itself. For a simulator with two process i and j, both of which execute in a single time step (no delay between trigger and finalize), Figure 3-2 and Figure 3-3 illustrate the process-specific update and output functions, respectively. Note that both processes can participate in updating the shared state x_s^t, by computing *shared state updates*, namely $\Delta x_{s,i}^t$ and $\Delta x_{s,j}^t$ which are then applied together.

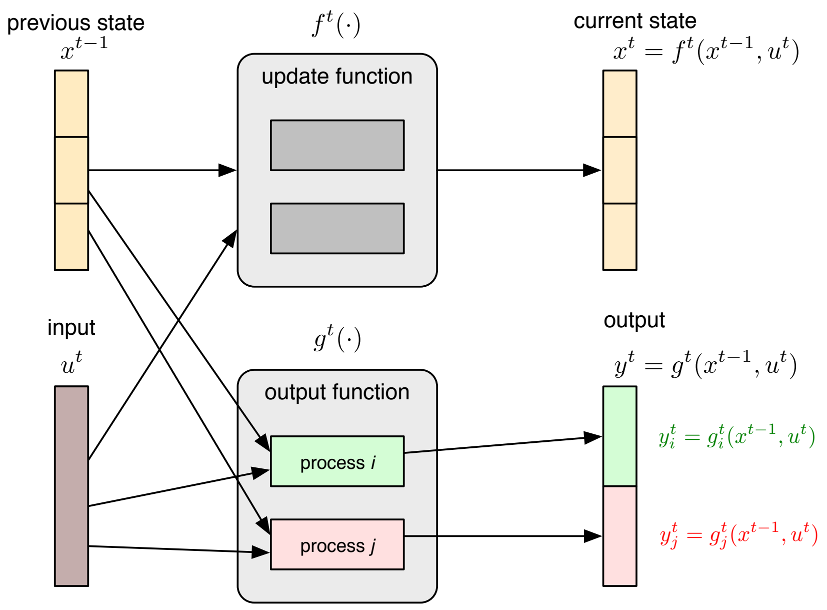

Figure 3-3: Outputs and Output Functions Partitioned by Process

3.2.2 Process Timing

Each process *triggers* at specified time steps and then *finalizes* at later time steps based on the duration of the execution of the process. The trigger time determines what information is available as input to the process when it begins and the finalize time determines when modifications to the state performed by the process are available to be seen by other processes. The timing of each process is controlled by the following three parameters:

- the time t_0 at which the process is first triggered,
- the time f between consecutive triggers of the process, controlling the frequency with which the process executes, and
- the execution time τ that determines the amount of time between when the process triggers and when it finalizes.[10]

Figures 3-2 and 3-3 assumed a simplified context in which all processes are finalized in the same time step in which they are triggered (i.e. $\tau = 0$), which need not be

[10]Note that multiple executions of a process can overlap in time. That is, a second instance of a process can be triggered before the first has completed and been finalized.

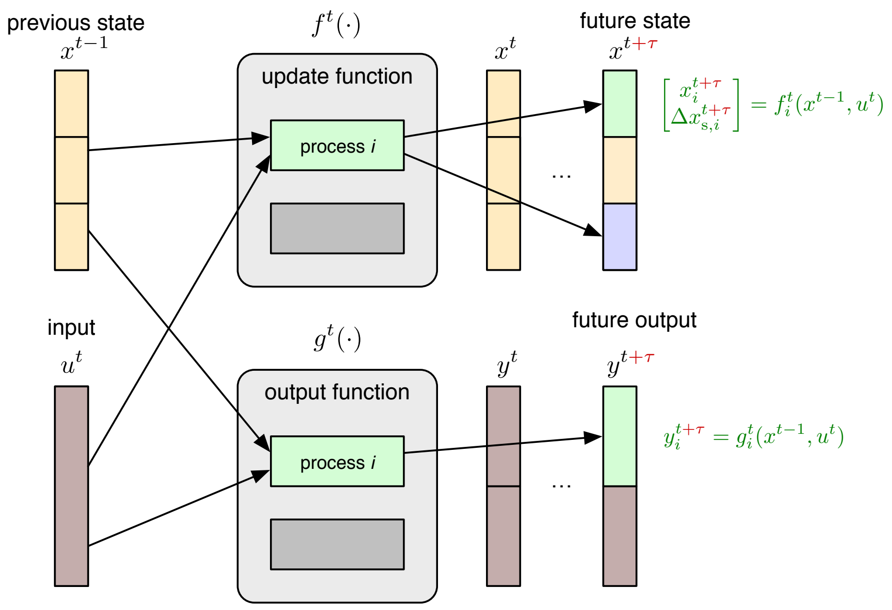

Figure 3-4: Process with Run-time τ

the case. Figure 3-4 illustrates the general form of the update and output functions for a process, where there is an arbitrary delay of τ time steps between when it is triggered and when it is finalized. In this case, it is the state and output at time $t + \tau$, rather than at t, that are affected by the process triggered at time t.

3.2.3 Simulation Runs and Inputs

It is often the case that we want to compare multiple runs of a simulation in which one or more input parameters are being varied across the runs. MP-Sim is structured to allow multiple simulation runs with their inputs and outputs to be organized together in a *simulation batch*. This batch can be organized as a simple one-dimensional list of runs, or more generally as a multi-dimensional collection of runs, where multiple inputs are being varied and we want to simulate all combinations of the inputs.

Note that the only difference between the runs in a simulation batch is the set of inputs used. Each run begins with the same initial state, determined at the time the simulator object is instantiated.

Since many of the inputs for a simulation may be constant from run to run, or even from time step to time step, MP-Sim provides mechanisms for constructing and updating these portions of the input variable only as needed in order to make most efficient use of memory and computational resources. Depending on the context, input data on disk can be loaded at each step as needed, or pre-loaded into memory during initialization to speed up execution.

3.2.4 Output Post-Processing

Outputs can be written to disk during the simulation run or collected in memory to be handled by post-run processing code. This post-run processing code is typically used to generate summary reporting and visualizations of the simulation results.

4 MP-Sim Reference

This section provides a reference to the various classes and functions included in the MP-Sim framework, with details of the object properties, function and method names, and input and output arguments.

4.1 Notational Conventions

Several notational conventions for variable, property, and field names are used throughout the code and this manual and are summarized in Table 4-1.

4.2 MP-Sim Configuration – `mpsim_config`

MP-Sim configuration consists of paths to base directories for input, output and temporary work files, denoted by <*INPUTDIR*>, <*OUTPUTDIR*> and <*WORKDIR*>, respectively. These are determined by the values of the `inputdir`, `outputdir` and `workdir` variables in the `mpsim_config.m` file found in <*MPSIM*>`/lib`. These are string value variables containing the absolute or relative path to the corresponding directory. If left blank, the default <*INPUTDIR*> points to <*MPSIM*>`/sim_data`. Likewise, the default behavior for <*OUTPUTDIR*> and <*WORKDIR*> is to set each equal to <*INPUTDIR*>.

Upon installation of MP-Sim these values are all blank, but they can be changed for a given MP-Sim installation by editing the appropriate lines in the `mpsim_config.m` file. The only lines that should be modified are those that define the `inputdir`, `outputdir` and `workdir` variables. *Note that this configuration affects all simulations run by this installation of MP-Sim.*

Within these base directories, which may or may not point to the same directory, files are organized by simulation batch. For a simulation named <*SIMNAME*>, MP-Sim will expect to find the input files in the simulation-specific directory for input files, namely <*INPUTDIR*>/<*SIMNAME*>`/inputs`, which is also denoted by <*SIMINPUTDIR*>. This path is passed to methods that need access to input files. Similarly, the simulation writes any output files to <*OUTPUTDIR*>/<*SIMNAME*>`/outputs`, and any temporary working files to <*WORKDIR*>/<*SIMNAME*>`/work`, denoted <*SIMOUTPUTDIR*> and <*SIMWORKDIR*>, respectively. These paths are likewise passed to methods that need access to the corresponding files. This design keeps all data files for a given simulation batch together under a directory whose name is the simulation name.

Table 4-1: Notational Conventions

name	description
Values	
<*INPUTDIR*>†	path to base directory for input files
<*OUTPUTDIR*>†	path to base directory for output files
<*WORKDIR*>†	path to base directory for temporary work files
<*SIMNAME*>	simulation name used to address input, output and temporary work files
<*SIMINPUTDIR*>	path to simulation-specific directory for input files, constructed as <*INPUTDIR*>`/`<*SIMNAME*>`/inputs`
<*SIMOUTPUTDIR*>	path to simulation-specific directory for output files, constructed as <*OUTPUTDIR*>`/`<*SIMNAME*>`/outputs`
<*SIMWORKDIR*>	path to simulation-specific directory for temporary work files, constructed as <*WORKDIR*>`/`<*SIMNAME*>`/work`
Variable, Property and Field Names	
`sim`	simulator object
`sim_name`, `sim.name`	simulation name <*SIMNAME*>
`sim_inputdir`	path to simulation-specific input directory <*SIMINPUTDIR*>
`sim_outputdir`	path to simulation-specific output directory <*SIMOUTPUTDIR*>
`sim_workdir`	path to simulation-specific work directory <*SIMWORKDIR*>
`r`	index(es) of current run, 1-D cell array of scalars
`t`	index of current simulation time step
`idx`	index of process update (trigger or finalize) instance
`x`	simulator state, x, see Section 4.3
`u`	current input struct, u^t, see Section 4.4
`y`	struct of all outputs, y, see Section 4.5
`ps`	process object
`ps_name`, `ps.name`	name of process object
`x_ps`	process-specific portion of simulator state, x_i
`y_ps`	process-specific output, y_i
`sx`	shared state object
`sx_name`, `sx.name`	name of shared state object
`sx_update`	shared state update for a single shared state object, struct array with fields `'op'` and `'val'`
`sx_updates`	struct of shared state updates for multiple shared state objects, with each update in a field named according the corresponding shared state name

† Specified in the `mpsim_config.m` file found in <*MPSIM*>`/lib`, and set to `'`<*MPSIM*>`/sim_data'` by default.

4.3 Structure of Simulator State x

The simulator state x, as described in Section 3.2.1, is partitioned into two parts, the shared part and the process-specific part. It is implemented as a struct `x` with the shared part contained in the `shared` field and the process-specific part contained in fields named according to name of the corresponding process object. The `shared` field is also a struct whose field names correspond to the names of the various shared state objects. This structure is summarized in Table 4-2.

Table 4-2: Structure of Simulator State x

name	description
`x`	full simulator state struct
`.(ps_name)`	struct containing arbitrary fields defined by process object identified by name `ps_name`
`.shared`	struct of shared state fields
`.(sx_name)`	struct containing arbitrary fields defined by shared state object identified by name `sx_name`

For example, suppose a process named `'foo'` defines its process specific state as a struct with a field named `'bar'`. Then the value of this piece of the state would be found in `x.foo.bar`. Similarly, if a shared state object named `'baz'` defined a field named `'buz'`, the value of that piece of the state would be found in `x.shared.baz.buz`.

Note that it is only the shared state *value* that is contained in the shared portion of the state `x`, not the shared state *object* used to manage and update it.

4.4 Structure of Simulator Input u

The input u, also illustrated in Section 3.2.1, is an arbitrary user-defined value. More specifically, u^t is an arbitrary user-defined set of information made available to the simulator at time step t. It is assembled at each time step from multiple sources, including collections of data loaded at the beginning of the simulation batch, at the beginning of the current simulation run, and at the current time step. Each of these 3 collections of data can consist of a global portion associated with the overall simulation as well as portions associated with specific processes triggered at time t..

The user does not define the input u^t directly, but indirectly, by implementing simulator and simulator process methods[11] to handle the various preloading and loading options. MP-Sim then assembles u^t automatically from the collections of data provided by these methods. Table 4-3 describes the various variables that are defined and returned by these methods.

[11]Namely `preload_sim_inputs`, `preload_run_inputs` and `load_current_inputs`. See Sections 4.6.2 and 4.8.2 for details.

Table 4-3: Return Arguments for Input Loading Methods*

name	description
`thissim`	struct of input data pre-loaded at beginning of simulation batch
`thisrun`	struct of input data pre-loaded at beginning of simulation run
`thisidx`	struct of input data loaded at current time step
`byrun`	struct array of input data pre-loaded at beginning of simulation batch, indexed by run
`byt`	struct array of input data pre-loaded at beginning of simulation batch or run, indexed by time index `t`
`byidx`	struct array of input data pre-loaded at beginning of simulation batch or run, indexed by process update instance `idx`
`byboth`	struct array of input data pre-loaded at beginning of simulation batch, indexed by both run `r` and time step `t`, or by both run `r` and process update instance `idx`†

The input struct `u`, as passed to the `ps.update()` functions, includes data loaded by these methods.* That is, all fields from the following return args of these methods are copied into input struct `u`:

`thissim`	
`thisrun`	
`thisidx`	
`byrun(r{:})`	for current simulation run `r`
`byt(t)`	for current time step `t`
`byidx(idx)`	for current process update instance `idx`
`byboth(r{:},t)`	for current simulation run `r` and time step `t`, or
`byboth(r{:},idx)`	for current simulation run `r` and process update instance `idx`

* Namely the `preload_sim_inputs`, `preload_run_inputs` and `load_current_inputs` methods of the simulator and process classes.
† Depending on whether it is returned by a simulator method or a process method.

This design helps to prevent the user from accidentally providing a process with information that should not be available at the current time step. The key detail to keep in mind when implementing these methods is that the fields in the provided data are simply copied to `u` when it is assembled, so each type of information should use a unique field name to avoid data getting inadvertently clobbered.

4.5 Structure of Simulator Output y

At each time step the simulator may produce an output y, or more precisely y^t. Some portion of these outputs may be written to output files directly by the `output` method of a process while other portions are collected in memory inside the simulator object for post-processing (e.g. saving, printing, plotting) by a `post_run` method. This output is stored in `sim.y` as follows, where `r` is the index of the run and `idx` is the index of the process update instance.

Table 4-4: Structure of Simulator Output Struct

name	description
`y`	full simulator output struct
`.(ps_name)(r{:}, idx)`	struct containing arbitrary fields defined by process `output` methods, identified by corresponding process name `ps_name` and indexed by the current run `r` and process update instance `idx`

4.6 Simulator Class – `mpsim`

The `mpsim` class serves as the abstract base class for all simulator objects and provides the top-level interface to the user for running simulations. The properties and methods are listed in Tables 4-5 and 4-6. Further detail is provided in the following two sections for the public methods and for the private methods that are intended to be implemented or overridden by a subclass.

4.6.1 Public Methods

`mpsim` This is the constructor for the simulator object and is always called indirectly by the constructor of the subclass, never directly by the user. It takes no input arguments. After initializing configuration information via `mpsim_config` it calls the `initialize` method.

```
sim = mpsim()
```

Subclasses of `mpsim` will typically override the `initialize` method, but inherit the constructor itself without modification.

`add_shared_state` Adds the specified shared state object `sx` to the simulator.

```
sim = sim.add_shared_state(sx)
```

Table 4-5: Properties of `mpsim`

name	description
Public Properties	
`l`	length of simulation time step*
`units`	string value of units of `l`, for user reference only (e.g. `'minutes'` or `'hours'`)
`name`	name of current simulation batch <*SIMNAME*>, used to construct paths to directories with simulation data
`processes`	cell array of process objects registered with simulator
`R`	scalar or vector of dimension of runs in current simulation batch
`T`	number of simulation time steps per run in current simulation batch
`r`	index(es) of current run, 1-D cell array of scalars
`t`	index of current simulation time step
`x`	simulator state x, with process-specific portions in `x.(ps_name)` and shared portions in `x.shared.(sx_name)`, see Section 4.3
`y`	struct containing outputs generated during simulation, see Section 4.5
`verbose`	(0–3), option specifying level of detailed progress to be printed to screen during a simulation run (default = 0, i.e. no output)
`inspect`	(0, 1), option to automatically enter debugger at end of `input` method to allow user to inspect the input struct (default = 0)
`post_run_on`	(0, 1) option controlling whether or not `post_run` method is executed (default = 1)
`options`	struct containing custom options passed to `run`; includes all options passed except the standard `R`, `T`, `verbose`, `inspect` and `post_run_on` which are modified directly in the corresponding `sim` properties
Private Properties	
`config`	struct of paths to base directories for input, output and temporary work files in fields `inputdir`, `outputdir` and `workdir`, respectively, as specified in `mp_sim_config`
`out_args`	current set of arguments, as provided by `ps.update()`, to be passed to `sim.output()`
`shared_x_objects`	struct of shared state objects used to update shared portion of state x, fields named according to name of corresponding shared state object†
`shared_x_names`	cell array of names of shared state fields/objects
`u_preloaded`	struct containing inputs pre-loaded at beginning of simulation batch or current run‡
`x0`	copy of initial simulator state x made before first run
`x_updates`	updates to simulator state to be applied when processes finalize

* The values of `ps.f`, `ps.t0`, and `ps.tau`, for all processes `ps`, must be evenly divisible by `sim.l`.
† Object at `sim.shared_x_objects.(sx_name)` is used to update shared state value at `sim.x.shared.(sx_name)`.
‡ Pre-loaded inputs are defined by the `preload_sim_inputs` and `preload_run_inputs` methods of the simulator and/or process objects.

Table 4-6: Methods of `mpsim`

name	description
Public Methods	
`mpsim`	simulator object constructor (called indirectly by subclass)
`add_shared_state`	add a shared state object and its initial value to the simulator
`display`	display information about simulation properties and processes
`register_process`	register a process with the simulator
`reset`	set time index t to 1, reset simulator state to initial value
`run`	execute specified runs of the simulation
Private Methods – override as needed	
`initialize`	set default simulator properties, create & add any shared states, create & register processes
`load_current_inputs`	load inputs for current time step
`preload_run_inputs`	pre-load inputs at beginning of each simulation run
`preload_sim_inputs`	pre-load inputs at beginning of simulation
`post_run`	post-process simulation outputs
Private Methods – should not need to override	
`apply_ps_x`	apply queued update to process-specific state
`apply_run_options`	copy set of run options to simulator object
`apply_shared_x`	apply queued update for given process to shared state
`increment_run`	increment simulation run counter in `sim.r`
`initialize_output`	initialize the output cache in `sim.y`
`input`	construct input struct `u` for current run and time step
`inputdir`	return path to a simulation-specific input file or directory
`output`	collect output from finalizing processes that provide it
`outputdir`	return path to a simulation-specific output file or directory
`preload_all_run_inputs`	cache results of `preload_run_inputs` for simulator and processes
`preload_all_sim_inputs`	cache results of `preload_sim_inputs` for simulator and processes
`queue_out_args`	push arguments to pass from `ps.update()` to `ps.output()` to corresponding FIFO queue
`queue_ps_x`	push updated process-specific state to corresponding FIFO queue
`queue_shared_x`	push shared state updates to corresponding FIFO queue
`step`	update state for finalizing processes and increment time step
`update`	call `ps.update()` for processes triggered at current time step
`workdir`	return path to a simulation-specific work file or directory

display Displays the details of the simulator object, including its properties and processes. Automatically called to display the object when a simulator object is the result of a statement (e.g. on the command-line) that is not terminated with a semicolon.

```
sim.display()
sim
```

register_process Checks compatibility of the process and simulator timing parameters, initializes the private data structures for tracking state updates and output arguments for the process, and registers the process object `ps` with the simulator.

```
sim = sim.register_process(ps)
```

reset Sets the time index `sim.t` back to 1 and resets the simulator state to the initial value, including the values of all shared state objects. Called before the start of each new run.

```
sim = sim.reset()
```

run Executes the specified runs of the simulation using the input, output and work directories and files corresponding to the simulation batch indicated by `sim_name`, and the specified MP-Sim options.

```
sim = sim.run(sim_name, opt_struct)
sim = sim.run(sim_name, opt1_name, opt1_val, ...)
sim = sim.run(sim_name, opt1_name, opt1_val, ... opt_struct)
```

If the `'r'` option is not specified, it executes all runs specified by the `R` attribute of the simulation, which can be set at run time by the `'R'` option.

MP-Sim options can be specified as a struct (`opt_struct`) or a set of name/value pairs (`opt1_name`, `opt1_val`, etc) or a combination of both. Table 4-7 lists the standard options that are available in the corresponding public attributes of the simulator object. All other options are considered custom options and are available in `sim.options`.

Table 4-7: MP-Sim Run Options

name	default	description
`'verbose'`	0	integer from 0 to 4 specifying the level of detail of progress output printed during the simulation
`'inspect'`	0	1 to pause simulation at each step after construction of the input data `u` to allow inspection
`'r'`	*all*	index(es) of single simulation run to execute (cell array)
`'R'`	`sim.R`	scalar or vector of dimension of runs to execute
`'T'`	`sim.T`	number of simulation time steps per run to execute

4.6.2 Private Methods – *override as needed*

`initialize` Sets the default values for the simulator properties `l`, `units`, `R` and `T`, creates and add any shared states, then creates and registers the process objects. This method *must* be implemented in your `mpsim` subclass.

```
sim = sim.initialize()
```

`load_current_inputs` Called at each time period to load inputs that are not process-specific (particularly in terms of timing). Returns an empty value by default, but can be overridden by the user to load data for run `r` and time `t` specific to their simulation.

```
thisidx = sim.load_current_inputs(sim_name, sim_inputdir, r, t)
```

The input struct `u`, as passed to the `ps.update()` functions, includes the data loaded by this method at each time step. That is, in addition to data from other sources, all fields from `thisidx` are copied into input struct `u`. See Table 4-3 for details on the return argument.

`preload_run_inputs` Called at the beginning of each simulation run to pre-load inputs that are not process-specific (particularly in terms of timing). Returns empty values by default, but can be overridden by the user to load data for run `r` specific to their simulation.

```
[thisrun, byt] = sim.preload_run_inputs(sim_name, sim_inputdir, r)
```

The input struct `u`, as passed to the `ps.update()` functions, includes data pre-loaded by this method at the beginning of each simulation run. That is, in addition to data from other sources, all fields from `thisrun` and `byt(t)` (for the current time step `t`) are copied into input struct `u`. See Table 4-3 for details on the return arguments.

`preload_sim_inputs` Called at the beginning of a simulation batch to pre-load inputs that are not process-specific (particularly in terms of timing). Returns empty values by default, but can be overridden by the user to load data specific to their simulation.

```
[thissim, byrun, byt, byboth] = ...
    sim.preload_sim_inputs(sim_name, sim_inputdir)
```

The input struct `u`, as passed to the `ps.update()` functions, includes data pre-loaded by this method at the beginning of the simulation. That is, in addition to data from other sources, all fields from `thissim`, `byrun(r{:})`, `byt(t)`, and `byboth(r{:},t)` (for the current run `r` and time step `t`) are copied into input struct `u`. See Table 4-3 for details on the return arguments.

`post_run` Simulation outputs are returned by the various process `output` methods during the execution of the simulation batch and collected in an output struct `y`, described in Table 4-4. This method is called after the execution of all runs is complete.[12] Does nothing by default, but can be overridden by the user to post-process the outputs `y` specific to their simulation.

```
sim.post_run(y, sim_outputdir)
```

4.7 Shared State Class – `mpsim_shared_x`

The `mpsim_shared_x` class serves as the abstract base class for all shared state objects. The properties and methods are listed in Tables 4-8 and 4-9. Further detail is provided in the following two sections for the public methods and for the private methods that are intended to be implemented or overridden by a subclass.

The shared state mechanism in MP-Sim involves three types of data. The first is the *shared state value* found directly in the simulator state, that is, in `sim.x.shared.(sx_name)`. These values are available directly to the process `update` and `output` methods as read-only values. On the other hand, each of these values is updated only indirectly, by the corresponding *shared state object*. In particular, a process `update` method may return a set of *shared state updates* (`sx_updates`) that are

[12]Unless the `'post_run_on'` option is set to 0.

Table 4-8: Properties of `mpsim_shared_x`

name	description
Public Properties	
`name`	name of shared state object (typically denoted `sx_name`)
Private Properties	
`initial_value`	starting value for the shared state at beginning of simulation batch, specified by `initialize` method†
`value`	current value of the shared state, type is determined by the `initialize` method, and value is updated by the `update` method

† By convention, the `initialize` method is designed to load the data for the initial value from a file, located in <*SIMINPUTDIR*>`/shared_states`, with the same name as the shared state object.

Table 4-9: Methods of `mpsim_shared_x`

name	description
Public Methods	
`mpsim_shared_x`	shared state object constructor (called indirectly by subclass)
Private Methods – override as needed	
`initialize`	initialize value of shared state
`update`	update value of shared state based on the

automatically applied to the state at the proper time via the `update` method of the corresponding shared state object. These shared state updates consist of an update value and an operation used to apply that value, for instance, adding or subtracting a value to an inventory, or pushing or popping a value from a queue.

4.7.1 Public Method

`mpsim_shared_x` This is the constructor for shared state objects and is always called indirectly by the constructor of the subclass, never directly by the user. The only input is the name of the shared state object `sx_name`.

```
sx = mpsim_shared_x(sx_name)
```

4.7.2 Private Methods – *override as needed*

`initialize` This method is responsible for setting the initial value of the shared state and *must* be implemented in your `mpsim_shared_x` subclass.

```
starting_value = sx.initialize(sim_inputdir, sim_name)
```

By convention, the `initialize` method is designed to load the data for the initial value from a file, located in <*SIMINPUTDIR*>`/shared_states`, with the same name as the shared state object.

`update` This method is responsible for applying a shared state update (`sx_update`) to the value of the shared state and *must* be implemented in your `mpsim_shared_x` subclass.

```
sx.update(sx_update)
```

The shared state update `sx_update` is a 1-dimensional struct array with fields `'op'` and `'val'`, containing the operation and value, respectively to be applied. If `sx_update` has more than one element, `update` should apply each of them in order.

4.7.3 Subclasses – `mpsim_shared_x_numeric`, `mpsim_shared_x_queue`

`mpsim_shared_x_numeric` This subclass of `mpsim_shared_x` implements a shared state whose value is a numerical scalar or array. The `initialize` method uses the built-in `load` function to load the initial value of the shared state from a text file, located in <*SIMINPUTDIR*>`/shared_states`, with the same name as the shared state object and with a `'.txt'` extension. For example, a shared state object named `'inventory'` would load its initial value from <*SIMINPUTDIR*>`/shared_states/inventory.txt`.

Table 4-10: Update Operations (`sx_update.op`) for `mpsim_shared_x_numeric`

operation	description
`'+'`	add update value to current value
`'-'`	subtract update value from current value
`'*'`	multipy current value by update value *(matrix multiply)*
`'/'`	divide current value by update value *(matrix divide)*
`'^'`	raise current value to power of update value *(matrix exponent)*
`'.*'`	multipy current value by update value *(elementwise)*
`'./'`	divide current value by update value *(elementwise)*
`'.^'`	raise current value to power of update value *(elementwise)*
`'='`	replace current value with update value

The valid update operations are shown in Table 4-10. So, for example, if the input text file contains a simple scalar, the following defines an update of length 2 that specifies an increment of the current value by 43, followed by doubling it.

```
sx_update = struct('op', {'+', '*'}, 'val', {43, 2});
```

The `mpsim_shared_x_numeric` class can typically be used without any further subclassing.

`mpsim_shared_x_queue` This subclass of `mpsim_shared_x` implements a shared state whose value is a FIFO queue of arbitrary data. Due to the potentially arbitrary nature of the data in the initial queue, the `initialize` method calls a function that returns the initial value of the shared state. The function is located in an M-file in <*SIMINPUTDIR*>`/shared_states` with the same name as the shared state object. For example, a shared state object named `'test_queue'` would load its initial value using the `test_queue` function found in <*SIMINPUTDIR*>`/shared_states/test_queue.m`.

Table 4-11: Update Operations (`sx_update.op`) for `mpsim_shared_x_queue`

operation	description
`'+'`	push update values to the queue, update values can be scalar or cell array
`'-'`	pop values from the queue, where update value gives number of elements to pop

The valid update operations are shown in Table 4-11. So, the following defines an update of length 1 that specifies pushing two elements to the queue: the matrix `[1 2; 3 4]`, then the string `'Hello FIFO!'`.

```
sx_update = struct('op', '+', 'val', {{[1 2; 3 4], 'Hello FIFO!'}});
```

The `mpsim_shared_x_queue` class can typically be used without any further subclassing.

4.8 Simulator Process Class – `mpsim_process`

The `mpsim_process` class serves as the abstract base class for all process objects, which define the basic tasks and behaviors performed by the simulator. The properties and methods are listed in Tables 4-12 and 4-13. The parameters, t_0, f and τ, discussed in Section 3.2.2, are properties of the process object and define the timing of the triggering and finalizing of the process. In summary, the process triggers at time step $(t_0 + fi)/l$ and finalizes at $(t_0 + \tau + fi)/l$, for $i = 1, 2, \ldots n_{\text{idx}}$, where l is the length of the simulation time step (`sim.l`).

Further detail is provided in the following two sections for the public methods and for the private methods that are intended to be implemented or overridden by a subclass.

Table 4-12: Properties of `mpsim_process`

name	description
Public Properties	
`name`	name of process
`f`	f, amount of time between instances of process update†
`t0`	t_0, time at which first process update instance is triggered†
`tau`	τ, length of process run time, or amount of time from trigger to corresponding finalize†
Private Properties	
`f_period`	$\hat{f} = f/l$, number of time steps between instances of process update‡
`t0_period`	$\hat{t_0} = t_0/l$, index of time step in which first process update instance is triggered‡
`tau_period`	$\hat{\tau} = \tau/l$, length of process run time, or number of time steps from trigger to corresponding finalize‡

† Expressed in same units as simulation time step length l `sim.l`, of which it must also be an integer multiple.
‡ Same as `ps.f`, `ps.t0` or `ps.tau`, respectively, but in units of simulation time steps.

4.8.1 Public Methods

`mpsim_process` This is the constructor for process objects and is always called indirectly by the constructor of the subclass, never directly by the user. It takes a struct `s` with fields `'name'`, `'f'`, `'t0'` and `'tau'` (all required) as an input.

```
ps = mpsim_process(s)
```

The process objects are typically instantiated and registered in the `initialize` method of the simulator object.

`display` Displays the details of the process object. Automatically called to display the object when a process object is the result of a statement (e.g. on the command-line) that is not terminated with a semicolon.

```
ps.display()
ps
```

Table 4-13: Methods of `mpsim_process`

name	description
Public Methods	
`mpsim_process`	process object constructor (called indirectly by subclass)
`display`	display the process object
`idx2t`	convert the index of a process update instance to the time step in which it is triggered or finalized
`t2idx`	convert a simulation time index into the index of the process update instance triggered or finalized at that time
Private Methods – override as needed	
`initialize`	set initial value for process-specific state
`load_current_inputs`	load inputs for current process update instance
`preload_run_inputs`	pre-load inputs at beginning of each simulation run
`preload_sim_inputs`	pre-load inputs at beginning of simulation
`input`	modify or update input struct u^t (`u`)
`output`	implement process output function $g_i(x, u)$, create process outputs
`print_finalize`	print custom output upon process finalize
`print_trigger`	print custom output upon process trigger
`update`	implement process update function $f_i(x, u)$ to update process-specific and shared state
Private Methods – should not need to override	
`finalize`	return index of process update instance if process finalizes in current period, 0 otherwise
`set_period`	check timing parameters and convert units to time steps
`trigger`	return index of process update instance if process triggers in current period, 0 otherwise

`idx2t` Given the index `idx` of a process update instance, this method returns the simulation time step `t` in which the process triggers (`'T'`) or finalizes (`'F'`), depending on the value of the second input argument.

```
t = ps.idx2t(idx, 'F')
t = ps.idx2t(idx, 'T')
```

`t2idx` Given a simulation time step `t`, this method returns the index `idx` of the process update instance that triggers (`'T'`) or finalizes (`'F'`) at time `t`, depending on the value of the second input argument. The third argument determines the handling of the case when the process does not trigger/finalize at time `t`. By default (`mode` $= 0$), it returns `idx` $= 0$ in that case. If `mode` $= 1$, it returns the `idx` corresponding to the most recent period preceding `t` in which the process triggered/finalized. If `mode` $= 2$, it returns `idx` as a non-integer value.

```
idx = ps.t2idx(t, 'F')
idx = ps.t2idx(t, 'T')
idx = ps.t2idx(t, 'F', mode)
idx = ps.t2idx(t, 'T', mode)
```

4.8.2 Private Methods – *override as needed*

`initialize` This method is responsible for setting the initial value of the process-specific state and *must* be implemented in your `mpsim_process` subclass, unless it has no process-specific state. The return argument defines both the structure of this portion of the state and the initial value to be used for all runs. Input arguments are the simluator state[13] and the path <*SIMINPUTDIR*> to the simulation inputs.

```
x_ps = ps.initialize(x, sim_inputdir)
```

`load_current_inputs` Called at each process trigger to load inputs that are process-specific (particularly in terms of timing). Returns an empty value by default, but can be overridden by the user to load data for run `r` and process update instance `idx` specific to their simulation. Arguments `R` and `nidx` are the vector of dimensions of the batch of simulation runs and the total number of process update instances per run, respectively.

```
thisidx = ps.load_current_inputs(sim_name, sim_inputdir, R, nidx, r, idx)
```

The input struct `u`, as passed to the `ps.update()` functions, includes the data loaded by this method at each process trigger. That is, in addition to data from other sources, all fields from `thisidx` are copied into input struct `u`. See Table 4-3 for details on the return argument.

`preload_run_inputs` Called at the beginning of each simulation run to pre-load inputs that are process-specific (particularly in terms of timing). Returns empty values by default, but can be overridden by the user to load data for run `r` specific to their simulation. Arguments `R` and `nidx` are the vector of dimensions of the batch of simulation runs and the total number of process update instances per run, respectively.

[13]During the initialization process, it must be assumed that the simulator state has not yet been fully constructed.

```
[thisrun, byidx] = ps.preload_run_inputs(sim_name, sim_inputdir, R, nidx, r)
```

The input struct `u`, as passed to the `ps.update()` functions, includes data pre-loaded by this method at the beginning of each simulation run. That is, in addition to data from other sources, all fields from `thisrun` and `byidx(idx)` (for the current process index `idx`) are copied into input struct `u`. See Table 4-3 for details on the return arguments.

`preload_sim_inputs` Called at the beginning of a simulation batch to pre-load inputs that are process-specific (particularly in terms of timing). Returns empty values by default, but can be overridden by the user to load data specific to their simulation. Arguments `R` and `nidx` are the vector of dimensions of the batch of simulation runs and the total number of process update instances per run, respectively.

```
[thissim, byrun, byidx, byboth] = ...
    ps.preload_sim_inputs(sim_name, sim_inputdir, R, nidx)
```

The input struct `u`, as passed to the `ps.update()` functions, includes data pre-loaded by this method at the beginning of the simulation. That is, in addition to data from other sources, all fields from `thissim`, `byrun(r{:})`, `byidx(idx)`, and `byboth(r{:},idx)` (for the current run `r` and process index `idx`) are copied into input struct `u`. See Table 4-3 for details on the return arguments.

`input` Perform any final modifications to the input struct `u` before passing it to the update functions.

```
u = ps.input(u, sim_inputdir, r, idx)
```

`output` Implements the process output function $g_i(x, u)$. Creates the outputs for a particular update instance of the process upon finalization. This method can save files to the <*SIMOUTPUTDIR*>, print output to the console or return output values `y_ps` to be cached by the simulator for post-processing by its `post_run` method.

This method is called without input arguments during simulation initialization to determine whether the process returns outputs for post-processing. In this context it should return an empty matrix if it will not be returning output for post-processing. Otherwise, it should return a non-empty scalar, if it's output will be a simple scalar, or if it will be a struct, then it should return a struct with the same fields that will be returned during the simulation. These values are stored in the simulator output struct at `sim.y.(ps.name)(r:, idx)` as described in Section 4.5.

Input arguments include the simulator state `x`, the current input struct `u`, the simulation name `sim_name`, the simulation output directory `sim_outputdir`, the run index `r`, the process update (finalize) index `idx` and any additional arguments passed from the corresponding `ps.update()` call, namely `out_args`.

```
y_ps = ps.output(x, u, sim_name, sim_outputdir, r, idx, out_args)
```

`print_finalize` Prints custom output upon process finalize.

```
ps.print_finalize(x, y, r, t, idx)
```

`print_trigger` Prints custom output upon process trigger.

```
ps.print_trigger(x, y, r, t, idx)
```

`update` Implements the process update function $f_i(x, u)$. Called with the current state `x` and current input `u` when the process is triggered, to compute updates to the process-specific and shared state. These updates are applied to the state when the corresponding update instance finalizes. It can also use `out_args` to pass along arbitrary data to the corresponding output function. Details of the input and output arguments are given in Table 4-14.

This method *must* be implemented in your `mpsim_process` subclass.

```
[x_ps, sx_updates, out_args] = ...
    ps.update(x, u, sim_name, sim_workdir, r, idx)
```

Table 4-14: Input and Output Arguments for `update` Method

name	description
Inputs	
`x`	simulator state x
`u`	simulator input u
`sim_name`	simulation name, <*SIMNAME*>
`sim_workdir`	simulation work directory
`r`	index(es) of current run, 1-D cell array of scalars
`idx`	index of process update (trigger or finalize) instance
Outputs	
`x_ps`	updated value of process-specific state
`sx_updates`	struct of shared state updates for multiple shared state objects, with each update in a field named according the corresponding shared state name
`out_args`	arguments to be passed to corresponding `output` method

5 Example Simulation

This section describes a toy example simulation used in this manual and in the MP-Sim tests to provide a concrete illustration of the concepts and implementation details for an example simulator based on MP-Sim. This example scenario is that of a greatly over-simplified burger shop with four separate processes for ordering, delivery, defrosting and grilling, respectively, running on an hourly time step for 3 weeks. To keep things small and simple, these are 3-day weeks with 6-hour days.

The simulation name used for this example is `burger_shop_example`, and the data can be found in <*MPSIM*>`/sim_data`. The code to implement the burger shop simulator is found in <*MPSIM*>`/lib/t`.[14]

5.1 State

As is typical, the state in this example includes both process-specific and shared portions. The process-specific portions contain the cumulative count of burgers that have been ordered, delivered, defrosted and grilled, respectively.[15] The shared state, on the other hand, consists of inventories of frozen, thawed and grilled burgers as well as an order queue.

5.2 Input Data

The number of burgers to be ordered, delivered, defrosted and grilled at each hour is given as input data. These data are assumed to be provided based on some external forecasts. The input for the simulation also includes the number of burgers actually sold.

5.3 Processes

The grilling process is set to run once an hour, taking the decision of the number of patties to grill directly from the input data, and subtracting that amount from the defrosted inventory and adding it to the grilled inventory. The grilled inventory is also decremented by the number of patties sold.

[14]The classes include all those whose names begin with `'bg_'`, and of course the `burger_shop` class itself.

[15]It turns out that, for this simple example simulation, these cumulative counts could be computed from outputs, and therefore need not actually be part of the state of the burger shop, which is adequately captured by the inventories and order queue. They have been included here simply to illustrate the process-specific state.

Similarly, the defrosting process runs every 3 hours, taking one hour to complete. It also receives the number of patties to defrost from the input data and subtracts that amount from the frozen inventory and adds it to the defrosted inventory.

The delivery process runs once per day, or every 6 hours, starting on the second hour of the day and completing on the third. The amount to be delivered is taken from the top of the order queue and added to the frozen inventory.

The order process runs once a week and pushes three new daily orders to the end of the order queue. The amounts for these orders are also read directly from input data.

Notice that the grilling process is able to actually do more than one task, grilling and sales, since they both occur hourly. These could have been implemented as separate processes, but are kept together here for simplicity. This is a design decision for the one implementing the simulation. On the other hand, since ordering and delivery, for example, do not occur on the same schedule, it is not possible to combine them into a single task.

5.4 Output

The output of the burger shop simulation consists of essentially two components. First, the number of burgers ordered, delivered, defrosted grilled and sold at each hour, along with the hourly inventory levels, is printed to the screen. And, second, the same data is saved to a MAT-file in the <*OUTPUTDIR*>`/burger_shop_example/outputs` directory.

5.5 Structure of Runs

The simulation is set up to run with an hourly time step for 3 weeks, with a 6-hour day and a 3-day week. Demand for burgers follows a cyclical pattern, with the peak occuring in the middle of each day, with a higher demand in the middle of the week as well. Inputs are provided for two separate runs, one in which the number sold is roughly as forecasted, and another in which there is an unexpectedly high demand for burgers.

6 Creating Your Own Simulation

This section will describe the steps required to design and implement your own simulation. It will use the burger shop example from the previous chapter to illustrate.

6.1 Primary Design Questions

There are five primary questions that need to be answered during the design of any MP-Sim simulation.

6.1.1 What information is included in the state?

One way to discern what belongs in the simulator state is to ask what is the minimal set of information needed to summarize the effects of all past action and define a starting condition from which to simulate your environment going forward. If a piece of information is not needed to simulate the future, it does not need to be in the state. If it is not a result of past action, it does not need to be in the state.

The decision of where in the state a particular piece of information belongs, on the other hand, is determined as the simulation is partitioned into its various processes.

6.1.2 What are the processes that act on the state?

This question can be answered by listing the various tasks or actions that are performed by the simulation and separating them by timing or scheduling into groups of tasks that always occur together. Each of these groups could be assigned to its own process. In some contexts it may be conceptually advantageous to further split these groups into logical sub-groups of tasks, each assigned to its own process, but that is a matter of preference.

Once the tasks are grouped into processes it becomes apparent which pieces of information in the state are acted upon by which processes. Portions of the state that are modified or updated by a single process belong in the process-specific portion of the state. State information that is modified by more than one process must be implemented as shared state.

6.1.3 What information becomes available to the simulation as it proceeds?

This is external information that is made available to the simulation at a specified time. If a piece of information depends on past simulator action, then it belongs in the state, not in the input.

6.1.4 What results does the simulation produce?

The behavior of the simulation is only available to the user through the output that it produces. This can be in the form of infomration printed to the screen during or after the execution of the simulation, or data or visualization files saved in the <*SIMOUTPUTDIR*> directory.

6.1.5 How many runs are needed and how do they relate to one another?

It is often the case that a simulation is run multiple times with results averaged or summed across the runs or comparisons made between different runs. Keep in mind that the only difference between the runs is in the input data used, since the initial state is shared among all runs. Multiple runs may be arranged as a simple list or as a multi-dimensional array of runs. For example, suppose you wanted to simulate 3 variations for each of 4 scenarios, each with and without a particular feature enabled. That would result in a 3-dimensional set of runs consisting of a total of 24 $(3 \times 4 \times 2)$ runs.

6.2 Implementation

With the above design questions answered, implementation begins by creating a subclass of `mpsim` as the top-level simulator, and subclasses of `mpsim_process` for each of the processes. Each shared state object is implemented as subclass of `mpsim_shared_x` or by using one of the provided subclasses `mpsim_shared_x_numeric` or `mpsim_shared_x_queue`.

The code for the implementation of the burger shop example described in this section can be found in <*MPSIM*>`/lib/t` in the following classes:

- `@burger_shop`
- `@bg_defrost`
- `@bg_deliver`
- `@bg_grill`
- `@bg_order`

Likewise, the input data files for the `'burger_shop_example'` can be found in <*MPSIM*>`/sim_data/burger_shop_example` in the files:[16]

- `defrost-base-run-1.txt`

[16]The `'burger_shop_example'` simulation includes data for both the single-dimensional burger shop example (`@burger_shop`) and the 2-dimensional version (`@burger_shop_2d`).

- `defrost-growth-run-1.txt`
- `grill-run-1.txt`
- `order_base_run.m`
- `order_growth_run.m`
- `shared_states/frozen_inventory.txt`
- `shared_states/grilled_inventory.txt`
- `shared_states/thawed_inventory.txt`

6.2.1 Simulator Class

It is the simulator class that defines the overall structure of the simulation, including its processes and state. This is a user-defined class that inherits from `mpsim`[17] and overrides the `initialize` method. In the burger shop example, this is the class named `burger_shop`. The `initialize` method is responsible for three things.

1. Specify the length (`l`) and units (`units`) of the simulation time step, the number of time steps (`T`), and the structure of the simulation runs (`R`).
2. Create and add any shared state objects. The shared state class is responsible to initialize the value of its shared state objects.
3. Create and register the process objects. The timing characteristics of each process are contained in the parameters used to instantiate the process object here. The process class takes care of initializing its own process-specific state.

The following is the definition of the `burger_shop` class with its `initialize` method, defining the 2 runs of 54 one-hour periods, adding the order queue and the 3 inventories, and finally registering the 4 processes described in Section 5.3.

[17]See Section 4.6 for more details.

```
classdef burger_shop < mpsim
    properties
    end
    methods
        function initialize(sim)
            %% set default values for simulator properties
            sim.l = 1;              %% l, length of simulation time step
            sim.units = 'hours';    %% units of l, length of time step
            sim.T = 6*3*3;          %% T, number of simulation periods per run
            sim.R = 2;              %% R, dimension(s) of simulation runs

            %% create and add shared state objects
            sim.add_shared_state(mpsim_shared_x_queue('order_queue'));
            sim.add_shared_state(mpsim_shared_x_numeric('frozen_inventory'));
            sim.add_shared_state(mpsim_shared_x_numeric('thawed_inventory'));
            sim.add_shared_state(mpsim_shared_x_numeric('grilled_inventory'));

            %% create and register process objects
            sim.register_process(bg_order(...
                struct( 'name', 'order', ...
                        'f', 6*3, ...
                        't0', 1, ...
                        'tau', 0) ));
            sim.register_process(bg_deliver(...
                struct( 'name', 'deliver', ...
                        'f', 6 , ...
                        't0', 2, ...
                        'tau', 1) ));
            sim.register_process(bg_defrost(...
                struct( 'name', 'defrost', ...
                        'f', 2, ...
                        't0', 1, ...
                        'tau', 1) ));
            sim.register_process(bg_grill(...
                struct( 'name', 'grill', ...
                        'f', 1, ...
                        't0', 1, ...
                        'tau', 0) ));
        end
    end
end
```

6.2.2 Shared State Classes

The burger shop requires several inventories and a delivery queue, which are defined as shared states of the simulator. MP-Sim includes classes for two types of shared state objects, a simple numeric state (`mpsim_shared_x_numeric`), which is used for the inventories, and a FIFO queue of arbitrary data types (`mpsim_shared_x_queue`), which is used for the order queue.

Shared state objects initialize their own values when a simulation begins, by convention, from an input file located in <*SIMINPUTDIR*>`/shared_states` with the same name as the shared state object. For example, the `'thawed_inventory'` shared state object is of the class `mpsim_shared_x_numeric`, and its initial value is found in <*INPUTDIR*>`/burger_shop_example/inputs/shared_states/thawed_inventory.txt`

```
sim.add_shared_state(mpsim_shared_x_numeric('thawed_inventory'));
```

The value of a shared state object is not manipulated directly by a process to avoid modifying the value observed by subsequent process updates in the same time step. Instead, to modify a shared state, a shared state *update*, generated by the `update` method of a process, is applied to the shared state *value* together with other updates when the process finalizes. This update value is applied automatically by the `update` method of the shared state *object*.

For example, in the `update` method of the `'defrost'` process, the defrosted amount is subtracted from the `'frozen_inventory'` and added to the `'thawed_inventory'` by defining a set of shared state updates (`sx_updates`) as follows.

```
sx_updates = struct( ...
    'frozen_inventory', struct('op', '-', 'val', defrosts), ...
    'thawed_inventory', struct('op', '+', 'val', defrosts) );
```

6.2.3 Simulator Process Classes

The majority of the action happens in the implementation of the various process classes, each of which is a subclass of `mpsim_process`. The subclass for a process, such as `'defrost'`, typically inherits explicitly the constructor of its parent class.

```
classdef bg_defrost < mpsim_process
    properties
    end
    methods
        function obj = bg_defrost(s)
            obj@mpsim_process(s);
        end
    end
end
```

This constructor, as called from the `initialize` method of the `burger_shop` simulator, sets the name of the process (`'defrost'`) and the timing parameters, in this case to run every 2 hours (`'f'`), starting at hour 1 (`'t0'`), where each defrost operation takes 1 hour complete (`'tau'`).[18]

```
sim.register_process(bg_defrost(...
    struct( 'name', 'defrost', ...
            'f', 2, ...
            't0', 1, ...
            'tau', 1) ));
```

Furthermore, each process must also implement the `update()` method corresponding to $f_i^t(\cdot)$ as described in Section 3.2. An example from the `'defrost'` process illustrates.

```
function [x_ps, sx_updates, out_args] = ...
        update(ps, x, u, sim_name, sim_workdir, r, idx)
%UPDATE @bg_defrost/update

%% number to defrost is min of input value and current frozen inventory
defrosts = u.defrosts;
if x.shared.frozen_inventory < defrosts
    defrosts = x.shared.frozen_inventory;
end

x_ps = x.(ps.name) + defrosts;
sx_updates = struct( ...
    'frozen_inventory', struct('op', '-', 'val', defrosts), ...
    'thawed_inventory', struct('op', '+', 'val', defrosts) );
out_args = defrosts;
```

[18]The fact that the units are in “hours” is also specified in the `initialize` method of the `burger_shop` simulator.

First the amount to defrost (`defrosts`) is taken from the `'defrosts'` field of the input `u` where it has been placed by one of the input methods, as we will see later. Then the amount is limited, if necessary, by the existing frozen inventory, which is found in the corresponding shared state field. Finally, the defrosted amount is added to the running total of defrosted burgers tracked by the process-specific state for `'defrost'` returned `x_ps`, the shared inventories are updated, and the defrosted amount is returned in `out_args` to make it available to the `output` method.

The `initialize` method must also be implemented to initialize the value of the process-specific state to zero.

```
function x_ps = initialize(ps, x, sim_inputdir)
%INITIALIZE @bg_defrost/initialize
x_ps = 0;
```

The input data specific to the defrost process can either be loaded for the particular instance, each time the process is triggered, by implementing `load_current_inputs`, or it can be pre-loaded at the beginning of the simulation or the run by implementing `preload_sim_inputs` or `preload_run_inputs`. In this case, that latter is used to pre-load the inputs for `'defrost'` at the beginning of each run.

```
function [thisrun, byidx] = ...
        preload_run_inputs(ps, sim_name, sim_inputdir, R, nidx, r)
%PRELOAD_RUN_INPUTS @bg_defrost/preload_run_inputs

season = {'base', 'growth'};
fname = fullfile(sim_inputdir, ...
        sprintf('%s-%s-run-%d.txt', ps.name, season{r{1}}, 1));
defrosts = load(fname);

thisrun = [];
byidx = struct('defrosts', num2cell(defrosts));
```

This method looks for a file in <*SIMINPUTDIR*> whose name is based on the run index `r`. This file contains the defrost inputs for each defrost operation in the run, so the data is assigned to the `byidx` field in a field called `'defrosts'`, which we saw was used to access it in the `update` method.

Finally, the `output` method corresponding to $g_i^t(\cdot)$ is implemented to return the amount defrosted, which was passed from the `update` method via `out_args`.

```
function y_ps = output(ps, x, u, sim_name, sim_outputdir, r, idx, out_args)
%OUTPUT @bg_defrost/output

if nargin == 1
    y_ps = 0;
else
    y_ps = out_args;
end
```

Notice that when `output` is called without input arguments, it returns a non-empty scalar to indicate that space should be allocated in the simulator output struct for `'defrost'` outputs. The defrosted amount returned for normal calls will be stored in `sim.y.defrost(r{:}, idx)` for post-processing by the simulator's `post_run` method, where `r` is the run index and `idx` is the index of the defrost instance. For more information on the output see Section 4.5.

6.2.4 Input Data

There are two types of data associated with a simulation. The first is data associated with the setup of the initial state, which is common for all runs, and second, data that is made available to the simulation through the input variable u_t at a particular time t, which may differ from one run to another. All of these data are supplied by the user in the <*SIMINPUTDIR*>, which in the case of the burger shop example, is <*INPUTDIR*>`/burger_shop_example/inputs/`.

In this example, the process-specific states that track the total number of burgers ordered, delivered, defrosted, grilled and sold are all set to zero by the `initialize` method of the corresponding process, so they do not need to access any data files. The inventories, on the other hand, are shared states whose initial values are specified by files in <*SIMINPUTDIR*>`/shared_states/`, namely, in `frozen_inventory.txt`, `thawed_inventory.txt`, and `grilled_inventory.txt`. The order queue, also a shared state, starts out empty, so it does not need an input file, though it could be provided as `order_queue.m`.

As described in Section 4.4, the input struct u^t, made available to the `update` and `output` methods of each process as `u`, is assembled at each time step from multiple sources. This data consists of the number of burgers grilled and sold at each hour, the number of burgers thawed at each defrost cycle, and the orders submitted each week and delivered daily. Notice that some of the data is tied to the timing of specific processes, while others, like the amount sold, correspond to the simulation time step. The former will be loaded by a method of the corresponding process, while the latter will be loaded by a method of the simulator. All are loaded from files in the <*SIMINPUTDIR*> directory. For illustration purposes, some are pre-loaded at the beginning of the simulation, some at the beginning of each run, and others are loaded at the period when they are needed. How the files are arranged within <*SIMINPUTDIR*> and when they are loaded or pre-loaded are design decisions that should be informed by memory requirements and convenient formats for the input data.

Table 6-1: Input Loading for the Burger Shop Example

data	pre-loaded for	into field	by `@class/method`	order by
# sold	current run	`'sold'`	`@burger_shop/preload_run_inputs`	`t`
# to grill	entire batch	`'grills'`	`@bg_grill/preload_sim_inputs`	`r`, `idx`
# to defrost	current run	`'defrosts'`	`@bg_defrost/preload_run_inputs`	`idx`
weekly order	*just-in-time*	`'orders'`	`@bg_order/load_current_inputs`	–

† Footnote.

Table 6-1 summarizes the way each type of input data for the burger shop is loaded. The burger shop example is arranged with two runs, one with “base” values for the inputs and a second run representing a “growth” scenario, where the sales are higher.

The number of burgers sold at each hour of a given run is pre-loaded into the `'sold'` field at the beginning of the run by the simulator's `preload_run_inputs` method and arranged by time step (`byt`), since the data is simply a vector of numbers, one for each simulation time step.

```
function [thisrun, byt] = preload_run_inputs(sim, sim_name, sim_inputdir, r)
%PRELOAD_RUN_INPUTS @burger_shop/preload_run_inputs

season = {'base-run', 'growth-run'};

fname = fullfile(sim_inputdir, ...
    sprintf('sold-%s.txt', season{r{1}}));
sold = load(fname);

thisrun = [];
byt = struct('sold', num2cell(sold));
```

The number of burgers grilled in each hour are pre-loaded into the `'grills'` field for all runs at the beginning of the simulation by the `preload_sim_inputs` method of the `'grill'` process.

```
function [thissim, byrun, byidx, byboth] = ...
        preload_sim_inputs(ps, sim_name, sim_inputdir, R, nidx)
%PRELOAD_SIM_INPUTS @bg_grill/preload_sim_inputs

fname = fullfile(sim_inputdir, ...
        sprintf('%s-run-%d.txt', ps.name, 1));
grills = load(fname)';

thissim = [];
byrun   = [];
byidx   = [];
byboth  = struct('grills', num2cell(grills));
```

The number of burgers defrosted every two hours is also pre-loaded at the beginning of each run, but since the data corresponds to triggers of the `'defrost'` process, one for each call to `update`, they are loaded into the `'defrosts'` field by the `preload_run_inputs` method of that process and arranged by the index of the process update instance (`byidx`).

```
function [thisrun, byidx] = ...
        preload_run_inputs(ps, sim_name, sim_inputdir, R, nidx, r)
%PRELOAD_RUN_INPUTS @bg_defrost/preload_run_inputs

season = {'base', 'growth'};
fname = fullfile(sim_inputdir, ...
        sprintf('%s-%s-run-%d.txt', ps.name, season{r{1}}, 1));
defrosts = load(fname);

thisrun = [];
byidx = struct('defrosts', num2cell(defrosts));
```

And finally, the orders are simply loaded as needed, each time the order process is triggered, by the `load_current_inputs` method of the `'order'` process. These orders are loaded into the `'orders'` field of the input struct where they can be accessed by `update` and `output` methods as `u.orders`.

```
function thisidx = ...
        load_current_inputs(ps, sim_name, sim_inputdir, R, nidx, r, idx)
%LOAD_CURRENT_INPUTS @bg_order/load_current_inputs

season = {'base', 'growth'};
fcn_name = sprintf('%s_%s_run', ps.name, season{r{1}});
orders = feval_w_path_mpsim(sim_inputdir, fcn_name, idx);

thisidx = struct('orders', orders);
```

6.2.5 Post-processing of Output

The simulation output must include the history of amounts ordered, delivered, thawed, grilled and sold, and the inventory levels at each hour for frozen, thawed and grilled burgers.

Since the amount handled by each process is determined in its `update` method, this value is passed to the corresponding `output` method via `out_args` and included directly in the output, as we saw in Section 6.2.3 for the `'defrost'` process. The inventories, on the other hand, are already available as part of the shared state, so they need not be passed via `out_args` and can be accessed directly and included by the `output` method of a process like `'grill'` that updates every hour.

```
function y_ps = output(ps, x, u, sim_name, sim_outputdir, r, idx, out_args)
%OUTPUT @bg_grill/output

if nargin == 1
    y_ps = struct( ...
            'grills', 0, ...
            'sold', 0, ...
            'frozen_inventory', 0, ...
            'thawed_inventory', 0, ...
            'grilled_inventory', 0 ...
        );
else
    y_ps = struct( ...
            'grills', out_args, ...
            'sold', u.sold, ...
            'frozen_inventory', x.shared.frozen_inventory, ...
            'thawed_inventory', x.shared.thawed_inventory, ...
            'grilled_inventory', x.shared.grilled_inventory ...
        );
end
```

Table 6-2: Structure of Burger Shop Output Struct

name	description
`sim`	simulator object
`  y`	full simulator output struct
`    .defrost(r{:}, idx)`	number of burgers defrosted, for run `r`, instance `idx`
`    .deliver(r{:}, idx)`	number of burgers delivered, for run `r`, instance `idx`
`    .grill(r{:}, idx)`	additional data for run `r`, hour `idx`
`      .grills`	number of burgers grilled
`      .sold`	number of burgers sold
`      .frozen_inventory`	number of burgers in frozen inventory
`      .thawed_inventory`	number of burgers in thawed inventory
`      .grilled_inventory`	number of burgers in grilled inventory
`    .order(r{:}, idx)`	number of burgers ordered, for run `r`, instance `idx`

This results in an output struct at the end of the burger shop simulation with the structure shown in Table 6-2. The `post_run` method of `burger_shop` processes and pretty-prints this output data to the console and saves it to a MAT-file in <*SIMOUTPUTDIR*>, which in this case is <*OUTPUTDIR*>`/burger_shop_example/outputs/`.

6.3 Multi-dimensional Runs

The burger shop example described in the previous section was structured with two runs, a "base" and a "growth" run. MP-Sim allows for more complex multidimensional sets of runs. In fact, it includes an illustration of extending the burger shop example to a two-dimensional set of runs, with two sets of slightly different inputs for each "base" run and each "growth" run. It can be run using input data from the same `'burger_shop_example'` as follows.

```
burger_shop_2d().run('burger_shop_example');
```

The code for the implementation of the burger shop example with two-dimensional runs described here can be found in <*MPSIM*>`/lib/t` in the following classes:

- `@burger_shop_2d`
- `@bg2_defrost`
- `@bg2_deliver`
- `@bg2_grill`
- `@bg2_order`

Likewise, the input data files for the `'burger_shop_example'` can be found in <*MPSIM*>`/sim_data/burger_shop_example` in the files:[19]

- `defrost-base-run-1.txt`
- `defrost-base-run-2.txt`
- `defrost-growth-run-1.txt`
- `defrost-growth-run-2.txt`
- `grill-run-1.txt`
- `grill-run-2.txt`
- `order_multi_run.m`
- `shared_states/frozen_inventory.txt`
- `shared_states/grilled_inventory.txt`
- `shared_states/thawed_inventory.txt`

The `burger_shop_2d` simulator inherits everything from `burger_shop_2d`, and the `bg2_<ps_name>` classes inherit from the corresponding `bg_<ps_name>` classes, with only a few minor modifications/overrides.

First, in the `initialization` method, the simulator sets `sim.R` to a vector of dimensions for the runs, namely a 2×2 array of runs, where the first dimension corresponds to "base" and "growth" and the second to the two different realizations of each.

```
sim.R = [2 2];
```

The `initialization` method also registers the `bg2_<ps_name>` versions of the processes. And finally, the simulator overrides `post_run` with a version designed for two-dimensional runs.

The process classes all explicitly inherit the constructor of their parent and override the input loading classes to access files corresponding to the second dimension of the runs.[20]

[19]The `'burger_shop_example'` simulation includes data for both the single-dimensional burger shop example (`@burger_shop`) and the 2-dimensional version (`@burger_shop_2d`).

[20]The `@bg2_deliver` class also currently includes a `preload_sim_inputs` method that is only used for testing.

7 MP-Sim for Matpower Simulations

MP-Sim can be integrated with Matpower [4, 5] and the **M**atpower **O**ptimal **S**cheduling **T**ool [6] to perform various types of power flow, unit commitment and dispatch simulations. An example simulation of an hourly dispatch problem using an optimal power flow (OPF) is provided in the distribution. In this example, MP-Sim is used to solve an hourly dispatch using an AC OPF on a 30-bus power system model with load profiles that differ between the two runs.

The simulation is implemented using the `opf_sim` class for the simulator and a single process `opf_hourly_dispatch`, whose `update()` method solves the OPF problem. To run the OPF simulation, first install Matpower then type in the console:

```
opf_sim().run('OPF_example');
```

If Matpower is installed, running the MP-Sim tests via `test_mpsim` will also include tests for `opf_sim`.

The example code to implement the OPF simulation described here can be found in <*MPSIM*>`/lib/t` in the following classes:

- `@opf_sim`
- `@opf_hourly_dispatch`

The <*MPSIM*>`/sim_data/OPF_example` directory contains the corresponding input data for the `'OPF_example'` simulation.

- `case30_mpsim.m` – base Matpower case file
- `loads1.txt` – load profile for run 1
- `loads2.txt` – load profile for run 2

The `post_run` method of `opf_sim` generates a few plots that are saved to PDF files in <*SIMOUTPUTDIR*>, which in this case is <*OUTPUTDIR*>`/OPF_example/outputs/`.

- `congestion.pdf` – profiles of congestion prices on 2 lines
- `generation.pdf` – profiles of average generation across 2 runs
- `nodal_price.pdf` – min, max, mean nodal prices

Appendix A Release History

The full release history can be found in `CHANGES.md` or online at https://github.com/MATPOWER/mpsim/blob/master/CHANGES.md.

A.1 Version 1.0 – released April 17, 2018

The MP-Sim 1.0 User's Manual is available online.[21]

- Add to `nested_struct_copy_mpsim()` ability to copy fields that are struct arrays.
- Fix bug #1 where `t_opf_sim` failed to create outputs dir.
- Fix bug #3 that was causing `t_opf_sim` to fail on Octave, and preventing Travis-CI integration.

A.2 Version 1.0b1 – released May 19, 2017

The MP-Sim 1.0b1 User's Manual is available online.[22]

- Initial release.

[21] http://www.pserc.cornell.edu/matpower/docs/MP-Sim-manual-1.0.pdf
[22] http://www.pserc.cornell.edu/matpower/docs/MP-Sim-manual-1.0b1.pdf

References

[1] John W. Eaton, David Bateman, Søren Hauberg, Rik Wehbring (2015). *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations.* Available: http://www.gnu.org/software/octave/doc/interpreter/. 1, 4

[2] The BSD 3-Clause License. [Online]. Available: http://opensource.org/licenses/BSD-3-Clause. 1.2

[3] H. Shin and R. D. Zimmerman, "MP-Sim User's Manual," 2017. [Online]. Available: http://www.pserc.cornell.edu/matpower/docs/MP-Sim-manual-1.0.pdf 1.3

[4] R. D. Zimmerman and C. Murillo-Sánchez. MATPOWER User's Manual, [Online]. Available: http://www.pserc.cornell.edu/matpower/ 7

[5] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," *Power Systems, IEEE Transactions on*, vol. 26, no. 1, pp. 12–19, Feb. 2011. http://dx.doi.org/10.1109/TPWRS.2010.2051168 7

[6] C. E. Murillo-Sánchez, R. D. Zimmerman, C. L. Anderson, and R. J. Thomas, "Secure Planning and Operations of Systems with Stochastic Sources, Energy Storage and Active Demand," *Smart Grid, IEEE Transactions on*, vol. 4, no. 4, pp. 2220–2229, Dec. 2013, http://dx.doi.org/10.1109/TSG.2013.2281001. 7