ALPHA Documentation

Release 0.2.0

EPA

Jun 15, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is ALPHA?

The Advanced Light-Duty Powertrain and Hybrid Analysis (ALPHA) tool was created by EPA to evaluate the Greenhouse Gas (GHG) emissions of Light-Duty (LD) vehicles. ALPHA is a physics-based, forward-looking, full vehicle computer simulation capable of analyzing various vehicle types combined with different powertrain technologies. The software tool is a MATLAB/Simulink based application.

EPA has developed the ALPHA model to enable the simulation of current and future vehicles, and as a tool for understanding vehicle behavior, greenhouse gas emissions and the effectiveness of various powertrain technologies. For GHG, ALPHA calculates CO2 emissions based on test fuel properties and vehicle fuel consumption. No other emissions are calculated at the present time but future work on other emissions is not precluded.

EPA engineers utilize ALPHA as an in-house research tool to explore in detail current and future advanced vehicle technologies. ALPHA is continually refined and updated to more accurately model light-duty vehicle behavior and to include new technologies.

ALPHA (and EPA's Heavy-Duty compliance model, GEM) are built on a common platform known as "REVS" - Regulated Emissions Vehicle Simulation. REVS forms the foundation of ALPHA. This document refers to the third revision of REVS, known as REVS3. ALPHA can be considered a tool as well as a modeling process, the components of which are defined in REVS.

> For more information, visit:
>
> https://www.epa.gov/regulations-emissions-vehicles-and-engines/advanced-light-duty-powertrain-and-hybrid-analysis-alpha

1.2 What is this Document?

This documentation should provide the reader an overview of the ALPHA modeling process and serve as a starting point for understanding some of the ALPHA implementation details. Common use cases and techniques frequently used to control and modify the modeling process are presented as a way to jump start ALPHA use.

1.3 Target Audience

The target audience for this document is anyone who is interested in learning more about how to run EPA's ALPHA model. Prior modeling experience or a good understanding of vehicle powertrains and some Matlab familiarity is assumed. There are ample resources available to learn the basics of Matlab and Simulink in print and online from MathWorks and other third parties.

1.4 System Requirements and Installation

1.4.1 System Requirements

ALPHA REVS3 requires Matlab/Simulink 2020a, but should also work with later releases after library/model up-conversions.

1.4.2 Installation

Install Matlab/Simulink following MathWork's instructions. Copy the `NVFEL_MATLAB_Tools` repo (https://github.com/USEPA/NVFEL_MATLAB_Tools) to a suitable directory on your modeling machine. `NVFEL_MATLAB_Tools` is a directory of helpful Matlab scripts and functions which are commonly used for data analysis and visualization, etc. Launch Matlab and add `NVFEL_MATLAB_Tools` and its subfolders to your Matlab path (from the Matlab console, select "Set Path" from the "HOME" tab of the Matlab window, then select "Add with Subfolders…" and browse to `NVFEL_MATLAB_Tools`).

Similarly, add the `REVS_Common` and `Parameter Library` folders and their subfolders to your Matlab path. The path may be saved for future sessions or it is also possible to write a simple script to add the required folders to your path on an as-needed basis. For example:

```
addpath(genpath('C:\dev\EPA_ALPHA_Model\REVS_Common'));
addpath(genpath('C:\dev\EPA_ALPHA_Model\Parameter Library'));
addpath(genpath('C:\dev\NVFEL_MATLAB_Tools'));
```

1.5 Directory Structure

A high-level description of the `REVS_Common` directory structure follows. Use it as a rough guide to exploring the file system. Not all releases of ALPHA may contain all subfolders (for example, HIL-related files) but this should still provide the user a good idea of where common items are located.

- `REVS_Common` top level - contains `REVS_VM.mdl`, the top-level ALPHA model and the ALPHA logo.
- `config_packages` - a set of pre-defined simulation configuration scripts which define simulation config tags and associated "loader" scripts which use the defined variables to set up the simulation input workspace.
- `datatypes` - Matlab class definitions for the Matlab objects that compose REVS and various enumerated datatypes. Also contains `REVS_fuel_table.csv` that holds the fuel properties for known fuel types.
- `drive_cycles` - `.mat` files that represent various compliance or custom drive cycles in the form of `class_REVS_drive_cycle` objects with the name `drive_cycle`.
- `functions` - various Matlab functions used during the modeling process.
- `helper_scripts` - primarily contains scripts related to pre- and post-processing simulation runs.
- `libraries` - the REVS Simulink component block models, separated into various libraries by component type.

- `log_packages` - scripts that are used in conjunction with the batch modeling process in order to control the datalogging and post-processing of datalogs into a standardized data object.
- `plots` - can be used to store plots of common interest to REVS3 development.
- `postprocess_scripts` - simulation post-processing scripts.
- `publish_tools` - tools related to publishing NCAT Test Data Packages, particularly for publishing engine data.
- `python` - Python scripts related to the implementation of multi-core and/or multi-machine parallel modeling processes on a local network using Python packages.

A high-level description of the `Parameter Library` directory structure follows. In particular, the engine and emachine files are part of the NCAT Test Data Package publishing process.

- `EMotor` - parameter files and data for common model components such as electrical motor/generators which can be used across multiple modeling projects.
- `Engines` - parameter files and data for various Internal Combustion Engines tested at EPA or made publicly available from other sources
- `Extras` - parameter files for various batteries (lead-acid and otherwise).

1.6 Design Principles

This section will lay out of the some high-level design principles that guide ALPHA development.

1.6.1 Object Oriented Design

REVS3 makes significant use of Matlab classes and objects in order to provide a well-defined, maintainable and reusable set of data structures and model functionality. Class definitions start with `class_` and enumerated types start with `enum_`. With a few exceptions, most of the classes start with `class_REVS` so that Matlab auto-completion provides a useful list of the available classes.

1.6.2 Component Reuse

The use of Matlab classes and objects aids in the maintenance of the code base by allowing easier addition of new elements and behaviors to existing data structures. Using classes (instead of structures) also ensures that data structures have known and reusable definitions.

Generally speaking, model components have class definitions that correspond to the required parameters and data necessary for their intended function. There are rare exceptions for a few legacy components that came over from REVS2 (which did not generally use Matlab classes and objects). New components should be added to the model following the object-oriented paradigm whenever possible.

1.6.3 Datalogging and Auditing

Datalogging enables post-simulation data analysis and debugging. Significant effort was applied to the creation of a datalogging framework that is both flexible and fast. For that reason there are controls available to limit the amount of data logged by the model (excess datalogging significantly slows the model down and is therefore to be avoided). For example, datalogging may be limited to the bare minimum required to calculate fuel economy, or datalogging may be limited to the bare minimum plus everything related to the engine or transmission. It is also possible to log every available signal in the model if desired and the associated performance slowdown is acceptable. Datalogging should generally be limited to the signals or components required for the investigation at hand. Datalogs are found in a workspace object named `datalog` at the end of simulation.

Simulation "results" are available in the simulation output workspace in the `result` object which contains scalar values by drive cycle phase. Values of common interest can be displayed in the command console using the `result.print` command.

Audit Notes

If new components are added to the model then new audit blocks also need to be added and the corresponding audit scripts will require updating in order to capture the new energy source or sink in the audit report. Adding audits to the model is somewhat of an advanced topic, primarily because the block layout of the model and the mathematical structure of the model are not the same - although sometimes they are! The primary principle is to remember that the purpose of the audit is to monitor the physical energy flows and not the energy flow through the Simulink blocks which may be distinct from the physics.

The model is also set up to be able to audit the energy flows throughout the model. If auditing is enabled then a text file (or console output) is created that shows the energy sources and sinks that were simulated. The total energy provided and absorbed should be equal if the model conserves energy. Since the model runs at discrete time steps and since modeling is an exercise in approximation there is commonly some slight discrepancy which is noted as the Simulation Error in the audit report. The Energy Conservation is reported as a percentage ratio between the Net Energy Provided and the Total Loss Energy. Audit summaries can be displayed in the command console using the `audit.print` command when auditing is enabled.

Auditing the energy flow in the model is a key factor in ensuring the plausibility and function of the model.

1.6.4 Conventions and Guidelines

There are several conventions and guidelines that enhance the consistency and usability of the model, see *ALPHA Development*.

CHAPTER

TWO

OVERVIEW

This chapter is meant to give a quick overview of how to run a pre-configured ALPHA simulation and understand the modeling process.

2.1 Running ALPHA - Quickstart

Launch Matlab and make sure `NVFEL_MATLAB_Tools`, `REVS_Common` and the `Parameter Library` folders are on the Matlab path as described in the installation instructions. As a quick check execute the following Matlab command and if successful, the path to the top-level ALPHA model should be returned:

```
which REVS_VM
```

If the command fails, double check the path setup.

Change the Matlab working directory to the `ALPHA_DEMO` folder and run `run_ALPHA_quickstart`. The `REVS_VM` model will open up (to watch the vehicle speed trace in real-time), compile and then run a drive cycle. When the simulation is complete there will be three files in the output folder. The file names are prefixed with a timestamp, YYYY_MM_DD_hh_mm_ss_, followed by `ALPHA_quickstart_results.csv`, `ALPHA_quickstart_1_console.txt` and `ALPHA_quickstart.log`. For example, `2022_02_01_09_36_23_ALPHA_quickstart_results.csv`, `2022_02_01_09_36_23_ALPHA_quickstart_1_console.txt` and `2022_02_01_09_36_23_ALPHA_quickstart.log` for files created on February 1st 2022, 23 seconds after 9:36 AM. The `results` file contains a summary of the simulation inputs, settings and outputs. The `console.txt` file captures anything that would have been output to the Matlab console window. In this case the console file contains the cycle phase summaries and the energy audit. The `.log` file has information regarding the batch process itself such as which simulation configuration tags are available and which pre- and post-process scripts were run.

Examining the Matlab workspace after the model runs reveals some string variables used to define the simulation configuration and the `sim_batch` object. To populate the top-level workspace with the simulation input and output data structures, execute the following command:

```
sim_batch.sim_case(1).extract_workspace
```

2.2 Modeling Processes

The fundamental modeling process consists of creating a Matlab workspace that contains all the variables necessary to run the `REVS_VM` (REVS Vehicle Model) Simulink model. There are several ways to accomplish this. The first approach below will be the primary focus of this document due to its numerous advantages as outlined below.

1. Create and execute a batch run using an instance of `class_REVS_sim_batch`.
    - Consistent approach to the modeling process
    - Ability with sim batch to run any number of simulations
    - Standard output summary results
    - Framework for pre- and post-processing simulations
    - Convenient capability to sweep variables and define multiple simulation scenarios
    - Capability to run simulations in parallel, on one or multiple computers
    - Automatically collates the results into a single output summary file
    - Framework for controlling simulation datalogging and auditing
    - Framework for controlling the amount ("verbosity") of output summary data
    - Framework for saving Matlab workspaces at various points in the modeling process
    - Easy and convenient method to define and reuse sim batches across multiple projects
2. Load a saved workspace available in a `.mat` file and then run the model. In this case the pre- and post-processing must be handled by the user, this is somewhat of an advanced use case but can be very useful under the right circumstances.
3. Create an ad-hoc script to load individual param files (Matlab scripts containing component data structures) and manually perform the pre- and post-processing. This was the process prior to the standardized batch process, which can lead to duplication of effort and inconsistent approaches across users and therefore should be avoided.

2.3 What is a Sim Batch?

A `class_REVS_sim_batch` object actually contains a vector of `class_REVS_sim_case` objects stored in the `sim_case` property. A `sim_case` could be created and run without a batch but there is no advantage to doing so since the batch process provides all the necessary pre- and post-processing and is much easier to use. Typically the only reason to manipulate a particular `sim_case` would be to extract its local workspace to populate the top-level workspace for direct access. This will be covered in more detail later in the discussion on working with workspaces.

2.4 Understanding the ALPHA Quickstart Script

The `run_ALPHA_quickstart` M-script demonstrates a simple batch process - a single simulation run with the default settings and only the minimum required input files and minimal outputs.

run_ALPHA_quickstart.m:

1. `clear; clear classes; clc;`
    - Clears the Matlab workspace, classes and console which is highly recommended before running a batch.
2. `sim_batch = class_REVS_sim_batch(REVS_log_default);`

- Creates sim_batch, an object of class class_REVS_sim_batch, and instructs it to log only the minimum required signals in the model. Datalogging will be discussed in more detail later.

3. sim_batch.output_file.descriptor = strrep(mfilename, 'run_', '');
   - Provides a descriptor string that identifies output files
4. sim_batch.retain_output_workspace = true;
   - Retains the simulation workspace in memory, for easier examination post-simulation
5. sim_batch.logging_config.audit_total = true;
   - Enables the simulation energy audit datalogging. Disabling the audit speeds up model execution
6. sim_batch.param_path = 'param_files/midsize_car';
   - The batch needs to know where to find param files that are not in the REVS_Common folder. In this case the param files are located in the midsize_car subfolder of the local param_files folder.
7. sim_batch.config_set = {'VEH:vehicle_2020_midsize_car + ENG:engine_2013_Chevrolet_Ecotec_LCV_2L5_Re
   + TRANS:TRX11_FWD + ELEC:electric_EPS + CYC:EPA_UDDS + CON:midsize_car_CVM_controls_param'};
   - The sim_batch.config_set defines the set of the simulations to be run by creating a cell array of one or more config strings. Within the config string are the tags VEH:, ENG:, TRANS:, ELEC:, CYC: and CON:. Following each tag is the name of a file that contains simulation inputs. The VEH: tag loads the vehicle information such as roadload, test weight, etc. The ENG: tag loads the engine information, in this case the engine is actually loaded from REVS_Common since it is one of the data packet engines, the other param files are loaded from the local param file directory. The TRANS: tag loads the transmission parameters, in this case for a 6-speed automatic. The ELEC: tag loads parameters that define the electrical system and accessories for this vehicle. The CYC: tag tells the simulation which drive cycle to run, in this case an EPA UDDS drive cycle. Lastly, the CON: tag tells the simulation which controls settings to use. In this case, the controls settings show that start-stop is disabled for this run. The CVM in MPW_LRL_CVM_controls_param stands for Conventional Vehicle Model. Other abbreviations that may be encountered are EVM for Electric Vehicle Model and HVM for Hybrid Vehicle Model. Electric vehicles and hybrid vehicles have their own control parameters.
5. open REVS_VM;

- This simply opens the top-level Simulink model so the simulation progress can be observed via the vehicle speed and drive cycle plot that comes from the top-level scope block. This step is optional.

6. sim_batch.run_sim_cases();

- This handles simulation pre-processing, running and post-processing.

CHAPTER

THREE

MODELING PROCESS DETAILS

This chapter contains a more detailed description of ALPHA batch simulation, how it is set up and gives an overview of how to use the provided features to conduct multiple simulations, perhaps sweeping model parameters or implementing custom pre or post processing to a batch run.

3.1 Batch Simulation Overview

ALPHA batch simulation is implemented via `class_REVS_sim_batch` which contains a variety of properties that control the simulation process. The following list is a summary of the contents of the more prominent class members which are detailed subsequently:

- Configuration Key Definitions - What options are available to construct or modify the individual simulations
- Configuration Set - List of simulations requested to be run defined via configuration Keys
- Pre-processing Scripts - Scripts used to transform the configuration set into the model input workspace
- Logging Configuration - What signals are to be logged and outputs generated
- Post-processing Scripts - Scripts used to alter or interpret the simulation output data

These many different pieces work in concert to complete the batch simulation process. An analogy may be helpful in understanding how the different pieces work together. The configuration keys define the available knobs the user can turn to influence the simulation. The configuration set is a list of settings for each knob. The scripts (pre and post) are the linkage that connects the knobs to the model and output processing.

3.2 Understanding Config Keys, Config Scripts & Config Options

The ALPHA batch simulation process, organized in `class_REVS_sim_batch`, is is controlled via configuration keys. The keys defined for a for a given batch are stored in the `sim_batch.config_keys` property and can be viewed in a tabular format via the `sim_batch.show_keys` method.

Config keys influence the simulation process through the pre- and post-processing scripts. The pre-processing scripts are stored the the `sim_batch.case_preprocess_scripts`. These scripts may handle loading data for the simulation, modifying simulation parameters. Similarly, the `sim_batch.case_postprocess_scripts` and `sim_batch.batch_postprocess_scripts` enable some post processing or aggregation of the output data.

Given that Config Keys and Config Scripts work together to produce the simulation results they are commonly organized into config option packages. The packages included with ALPHA that can provide many commonly requested operations are stored in `REVS_Common\config_packages`, with some metapackages, bundles of config options, covering each powertrain type.

The set of simulations to be run is defined using the config keys. The desired simulations (config set) are loaded into `sim_batch.config_set`. This can be accomplished via two different methods that are discussed further in this section. The config set can be entered directly, or via config strings which are interpreted is the `sim_batch.load_config_strings` method. ALPHA also includes the capability to perform full-factorial expansion for given simulation configurations. This expanded set is accessible as an N x 1 structure in `sim_batch.sim_configs`. The example below shows a sample config string where a variety of tags are used. This string is then loaded into the batch which can then be viewed

```
>> config_string = 'VEH:vehicle_FWD_midsize_car + ENG:engine_2013_Chevrolet_Ecotec_LCV_
↪2L5_Reg_E10 +
TRANS:transmission_6AT_FWD_midsize_car + ELEC:electric_starter_alternator_param +␣
↪ACC:accessory_HPS_param +
CYC:{''EPA_FTP_NOSOAK'',''EPA_HWFET''} + CON:CVM_controls_param_midsize_car + TRGA_
↪LBS:30.62 +
TRGB_LBS:-0.0199 + TRGC_LBS:0.01954';

>> sim_batch.load_config_strings(config_strings); % parse config strings
>> sim_batch.config_set{1}

    struct with fields:

    aggregation_keys: {}
         drive_cycle: {{1×2 cell}}
             vehicle: {'vehicle_FWD_midsize_car'}
        target_A_lbs: 30.6200
        target_B_lbs: -0.0199
        target_C_lbs: 0.0195
              engine: {'engine_2013_Chevrolet_Ecotec_LCV_2L5_Reg_E10'}
        transmission: {'transmission_6AT_FWD_midsize_car'}
            electric: {'electric_starter_alternator_param'}
           accessory: {'accessory_HPS_param'}
            controls: {'CVM_controls_param_midsize_car'}
```

3.2.1 Config Keys & Tags

As mentioned previously the config keys are stored in `sim_batch.config_keys` and can be viewed via the `sim_batch.show_keys` method, an example of this is shown below. This will display a list of potential `sim_config` fieldnames in the 'Key' column, the key tags, for use in the config string, in the 'Tag' column, optional default values, an optional description and the name of the script which defined the key in the 'Provided by' column.

```
>> sim_batch.show_keys
```

```
      Key                                  |     Tag                |     Default␣
↪Value                  |     Provided by                |     Description
    ------------------------------------------------------------------------------
↪-------------------------------------------------------------
    aggregation_keys                       |                        |               ␣
↪                   |  class_REVS_sim_batch          |
    test_data                              |  DATA                  |               ␣
↪                   |  REVS_config_external_data     |
    test_data_index                        |  DATA_INDEX            |  1            ␣
↪                   |  REVS_config_external_data     |
```

(continues on next page)

(continued from previous page)

```
    external_drive_cycle                        |  XCYC                  |  0                ␣
↪                    |  REVS_config_external_data     |
    external_trans_temp                         |  XTTMP                 |  false            ␣
↪                    |  REVS_config_external_data     |
    external_shift                              |  XSHFT                 |  false            ␣
↪                    |  REVS_config_external_data     |
    external_lockup                             |  XLOCK                 |  false            ␣
↪                    |  REVS_config_external_data     |
    external_accessory_elec                     |  XEACC                 |                   ␣
↪                    |  REVS_config_external_data     |
    external_accessory_mech                     |  XMACC                 |                   ␣
↪                    |  REVS_config_external_data     |
    external_cyl_deac                           |  XDEAC                 |  false            ␣
↪                    |  REVS_config_external_data     |
    ambient                                     |  AMB                   |  {ambient=class_
↪REVS_ambient;}      |  REVS_config_ambient           |
    package                                     |  PKG                   |                   ␣
↪                    |  REVS_config_vehicle           |
    drive_cycle                                 |  CYC                   |                   ␣
↪                    |  REVS_config_vehicle           |
    vehicle                                     |  VEH                   |                   ␣
↪                    |  REVS_config_vehicle           |
    driver                                      |  DRV                   |  {driver=class_
↪REVS_driver;}       |  REVS_config_vehicle           |
    vehicle_lbs                                 |  VEH_LBS               |                   ␣
↪                    |  REVS_config_vehicle           |
    vehicle_kg                                  |  VEH_KG                |                   ␣
↪                    |  REVS_config_vehicle           |
    performance_mass_penalty_kg                 |  PERF_KG               |  0                ␣
↪                    |  REVS_config_vehicle           |
    ETW_lbs                                     |  ETW_LBS               |                   ␣
↪                    |  REVS_config_vehicle           |
    ETW_kg                                      |  ETW_KG                |                   ␣
↪                    |  REVS_config_vehicle           |
    ETW_multiplier                              |  ETW_MLT               |  1                ␣
↪                    |  REVS_config_vehicle           |
    target_A_lbs                                |  TRGA_LBS              |                   ␣
↪                    |  REVS_config_vehicle           |
    target_B_lbs                                |  TRGB_LBS              |                   ␣
↪                    |  REVS_config_vehicle           |
    target_C_lbs                                |  TRGC_LBS              |                   ␣
↪                    |  REVS_config_vehicle           |
...
```

`sim_config` is a struct variable created automatically by `class_REVS_sim_batch` and is made available to the simulation workspace prior to simulation. The `sim_config` fieldnames give at least a preliminary understanding of what a tag means and can be further examined by taking a look at the default pre- and post-processing scripts.

As mentioned previousy config keys are generally defined with their processing scripts within a pacakge constructed from `class_REVS_sim_config_options` where each key is an instance of a `class_REVS_sim_config_key`. For example:

```
package = class_REVS_sim_config_options();

package.keys = [ ...
    class_REVS_sim_config_key('drive_cycle',         'tag', 'CYC',    'eval', false);
    class_REVS_sim_config_key('ETW_lbs',             'tag', 'ETW_LBS');
    class_REVS_sim_config_key('roadload_multiplier', 'tag', 'RL_MLT', 'default', 1.0);
    ...
    ]
```

The arguments to the `class_REVS_sim_config_key` constructor are the property name, followed by optional name value pairs of 'tag' for the tag used in config strings, 'eval' for the tag evaluation type, 'default' for the default value to use if not provided in the config set, and 'description' to provide a plaintext description of the key's purpose.

3.2.2 Literal vs Eval Config Tags

When defining simulations via config strings the contents of some tags (keys) need to be evaluated while in other situations it may be preferred the value is retained in its string form. In the above example `ETW_lbs` key is an 'eval' tag which means its value will be automatically evaluated when loading the config strings. If the eval tag is created with a default value, that value will be used if the tag is not specified by the user. Eval tags are generally numeric, and must be an evaluatable expression. An eval tag may evaluate to a single value or a vector of multiple values to perform variable sweeps. For example, the following would all be valid eval tags within a config string:

```
ETW_LBS:3625
ETW_LBS:[3000:500:5000]
ETW_LBS:4454*[0.8,1,1.2]
```

The first case evaluates to a single number, 3625. The second case evaluates to a vector, [3000 3500 4000 4500 5000] as does the last case which becomes [3563.2 4454 5344.8]. Any valid Matlab syntax may be used in an eval tag including mathematical operations such as multiply, divide, etc. If addition is used, there must not be any spaces surrounding the + sign because ' + ' (space, plus-sign, space) is the separator used to build composite config strings and will result in an erroneously split string.

In the previously referenced example above, the `drive_cycle` property holds a non-evaluated tag, which means the part of the string associated with that tag will not automatically be evaluated (turned into a numeric or other value, but rather taken as a string literal). Typically this would be used for something like file names or other strings. Literal tags may be evaluated in user scripts. For example, if the literal tag was the name of a script, then that script may be called in the user pre- or post-processing scripts at the appropriate time to perform whatever its function is. Literal tags can be used to hold a single value or, when combined with delayed evaluation (in a user script, instead of during config string parsing) may hold multiple values. For example, within a config string, these are possible uses of the CYC: tag:

```
CYC:EPA_IM240
CYC:{''EPA_FTP_NOSOAK'',''EPA_HWFET'',''EPA_US06''}
```

In the first example, the CYC: tag refers to a single drive cycle file, `EPA_IM240.mat` which will be used for the simulation. In the second case, the CYC: tag is used to store a string representation of a Matlab cell array of drive cycle strings. In this case, `sim_config.drive_cycle` would be:

```
'{''EPA_FTP_NOSOAK'',''EPA_HWFET'',''EPA_US06''}'
```

which would evaluate (using the Matlab `eval()` or `evalin()` command) the cell array of strings:

```
{'EPA_FTP_NOSOAK','EPA_HWFET','EPA_US06'}
```

Drive cycle loading of a single cycle or the combining of multiple cycles into a single cycle is automatically handled in `class_REVS_sim_case.load_drive_cycles()` but the same concept can apply to user-defined literal tags initiated by user scripts. Drive cycle creation and handling will be discussed in further detail later.

3.2.3 Building Config Set Directly

One workflow option is to build the config set by directly setting the `sim_batch.config_set` property. This property must be either a structure or cell array of structures. The latter allows a batch consisting of multiple groups of simulations to be constructed from different config keys. An example of a batch config set configured directly can be seen below:

```
>> sim_batch.config_set.drive_cycle = {{'EPA_FTP_NOSOAK','EPA_HWFET'}}
>> sim_batch.config_set.vehicle = {'vehicle_FWD_midsize_car'};
>> sim_batch.config_set.engine = {'engine_2013_Chevrolet_Ecotec_LCV_2L5_Reg_E10'};
>> sim_batch.config_set.transmission = {'transmission_6AT_FWD_midsize_car'};
>> sim_batch.config_set.electric = {'electric_starter_alternator_param'};
>> sim_batch.config_set.accessory = {'accessory_EPS_param'};
>> sim_batch.config_set.controls = {'CVM_controls_param_midsize_car'};
>> sim_batch.config_set.ETW_lbs = [3000:1000:5000];
>> sim_batch.config_set.start_stop = [false, true];
```

In this example many of the config keys are set directly. Notice that the various string based keys are stored as cell arrays of strings. The reason for this will be discussed in the next section. It should also be noted that not all config keys need to be specified, and those not specified will use the default value established when that config key was defined.

3.2.4 Config Set Expansion

Individual config set entries are expanded full factorial to create multiple sim configs which become the cases in `sim_batch.sim_case` when the batch is executed. In the example above this single config set will yield 6 simulations, three different ETW values multiplied by two options for start stop. Note that while drive cycle may appear to contain multiple entries it is contained within an outer cell array and thus is a single entry. The expanded config set is accessible via `sim_batch.sim_configs` and each index represents a planned simulation. As shown below the the sim configs contain entries for all defined config keys, not just those specified in the config set.

```
>> sim_batch.sim_configs

ans =

6×1 struct array with fields:

    test_data
    test_data_index
    external_drive_cycle
    external_trans_temp
    external_shift
    external_lockup
    external_accessory_elec
    external_accessory_mech
    external_cyl_deac
    ambient
    package
    drive_cycle
```

(continues on next page)

(continued from previous page)

```
    vehicle
    driver
    vehicle_lbs
    vehicle_kg
    performance_mass_penalty_kg
    ETW_lbs
    ETW_kg
    ETW_multiplier
    target_A_lbs
    target_B_lbs
    target_C_lbs
    dyno_set_A_lbs
    dyno_set_B_lbs
    dyno_set_C_lbs
    calc_ABC_adjustment
    target_A_N
    target_B_N
    target_C_N
    dyno_set_A_N
    dyno_set_B_N
    dyno_set_C_N
    adjust_A_lbs
    adjust_B_lbs
    adjust_C_lbs
    adjust_A_N
    adjust_B_N
    adjust_C_N
    roadload_multiplier
    NV_ratio
    FDR
    FDR_efficiency_norm
    powertrain_type
    vehicle_type
    vehicle_manufacturer
    vehicle_model
    vehicle_description
    tire_radius_mm
    engine
    fuel
    engine_vintage
    engine_modifiers
    engine_scale_pct
    engine_scale_kW
    engine_scale_hp
    engine_scale_Nm
    engine_scale_ftlbs
    engine_scale_L
    engine_scale_adjust_BSFC
    engine_scale_num_cylinders
    engine_deac_type
    engine_deac_num_cylinders
    engine_deac_scale_pct
```

(continues on next page)

(continued from previous page)

```
    engine_deac_max_reduction_pct
    engine_deac_reduction_curve
    engine_deac_activation_delay_secs
    engine_DCP
    engine_CCP
    engine_GDI
    engine_transient_fuel_penalty
    engine_fuel_octane_compensation
    transmission
    transmission_vintage
    TC_K_factor
    TC_stall_rpm
    TC_torque_ratio
    TC_lockup_efficiency_pct
    transmission_autoscale
    electric
    propulsion_battery
    accessory_battery
    propulsion_battery_initial_soc_norm
    propulsion_battery_reference_soc_norm
    accessory_battery_initial_soc_norm
    propulsion_battery_cells_in_series
    propulsion_battery_cells_in_parallel
    propulsion_battery_cell_capacity_Ah
    MG1
    MG2
    P0_MG
    P2_MG
    MOT
    MG1_max_power_kW
    MG2_max_power_kW
    P0MG_max_power_kW
    P2MG_max_power_kW
    MOT_max_power_kW
    MG1_max_torque_Nm
    MG2_max_torque_Nm
    P0MG_max_torque_Nm
    P2MG_max_torque_Nm
    MOT_max_torque_Nm
    accessory
    controls
    start_stop
    base_hash
    aggregation_hash
    simulation_hash
```

A deeper look into the `sim_batch.sim_configs` structure array shows how some of the keys supplied vary across the cases providing full factorial coverage.

```
>> [sim_batch.sim_configs.vehicle]

ans =
```

(continues on next page)

(continued from previous page)

```
    'vehicle_FWD_midsize_carvehicle_FWD_midsize_carvehicle_FWD_midsize_carvehicle_FWD_
↪midsize_carvehicle_FWD_midsize_carvehicle_FWD_midsize_car'

>> {sim_batch.sim_configs.vehicle}

ans =

1×6 cell array

    {'vehicle_FWD_midsize_car'}    {'vehicle_FWD_midsize_car'}    {'vehicle_FWD_midsize_
↪car'}    {'vehicle_FWD_midsize_car'}    {'vehicle_FWD_midsize_car'}    {'vehicle_FWD_
↪midsize_car'}

>> {sim_batch.sim_configs.ETW_lbs}

ans =

1×6 cell array

    {[3000]}    {[4000]}    {[5000]}    {[3000]}    {[4000]}    {[5000]}

>> {sim_batch.sim_configs.start_stop}

ans =

1×6 cell array

    {[0]}    {[0]}    {[0]}    {[1]}    {[1]}    {[1]}
```

One note regarding config set expansion is that only the horizontal dimension of a matrix or cell array is considered. Thus a column vector would not be expanded and the entire vector would be passed to each configuration. Similarly, if a 4 x 5 matrix was passed into a config set it would yield 5 different cases each passed a 4 x 1 vector.

3.2.5 Config Set Aggregation

When conducting a large number of simulations it may be desirable to examine or aggregate the results over different subsets of the full collection of sim configs. In the above example it can be noted that there are three hashes computed in relation to the sim configs. `base_hash` corresponds to the original (unexpanded) config set entry that created the resulting sim config. `simulation_hash` corresponds to the specific sim config or sim_case to be run. `aggregation_hash` is supplied to allow the user to specify groups by which they may want to aggregate the results. The `sim_batch.config_set` object by default includes a special member `aggregation_keys` where the string for each key the user wants to aggregate over can be included. Each unique set of values for the keys not specified in `aggregation_keys` will end up with the same `aggregation_hash`, which can then be used to the batch post processing the generate the desired outputs.

3.2.6 Building Config Set via Config Strings

Config strings offer the ability to construct a simulation or set of simulations via a one line string. As seen above it can be tedious to set a large number of config keys individually. A config string is constructed via tag-value pairs separated by : and joined by the + symbol. Within each element spaces cannot be used. The config string representation of the above config set would look like:

```
>> config_string = 'VEH:vehicle_FWD_midsize_car + ENG:engine_2013_Chevrolet_Ecotec_LCV_
↪2L5_Reg_E10 +
TRANS:transmission_6AT_FWD_midsize_car + ELEC:electric_starter_alternator_param +␣
↪ACC:accessory_EPS_param +
CYC:{''EPA_FTP_NOSOAK'',''EPA_HWFET''} + CON:CVM_controls_param_midsize_car + ETW_
↪LBS:[3000:1000:5000] + SS:[0,1];
```

As mentioned previously the `sim_batch.load_config_strings` method is used to load these strings and would set the `sim_batch.config_set` matching the prior example and would also result in matching `sim_bat.sim_configs` output.

If multiple config strings are desired they can be provided as a cell array. This would be analogous to config set being a cell array as well.

The aggregation of sim configs / sim cases is implemented in config strings via the || operator. All tags are expanded, but only those to the left of the || are used to generate the aggregation hash meaning all combinations to the right of the || can used to compute each aggregate result. Again, it is good to note that how this aggregation is handled depends on the batch postprocessing and by default no processing is conducted. As shown below this example generates the same six simulation cases, but only two aggregation cases are generated. In this exammple one would correspond to `SS:0` and the other to `SS:1`.

```
>> config_string = 'VEH:vehicle_FWD_midsize_car + ENG:engine_2013_Chevrolet_Ecotec_LCV_
↪2L5_Reg_E10 +
TRANS:transmission_6AT_FWD_midsize_car + ELEC:electric_starter_alternator_param +␣
↪ACC:accessory_EPS_param +
CYC:{''EPA_FTP_NOSOAK'',''EPA_HWFET''} + CON:CVM_controls_param_midsize_car + SS:[0,1]␣
↪|| ETW_LBS:[3000:1000:5000]';

>> sim_batch.load_config_strings(config_string);

>> {sim_batch.sim_configs.aggregation_hash}'

ans =

6×1 cell array

    {'dfd9bb5cce637383ef2e7d668d2fd9649f0acf72'}
    {'dfd9bb5cce637383ef2e7d668d2fd9649f0acf72'}
    {'dfd9bb5cce637383ef2e7d668d2fd9649f0acf72'}
    {'97b691b8a096dffec2b5ac6fc85d436ab5142ef2'}
    {'97b691b8a096dffec2b5ac6fc85d436ab5142ef2'}
    {'97b691b8a096dffec2b5ac6fc85d436ab5142ef2'}
```

3.2.7 Creating New Config Keys or Config Options

The many config option packages included with ALPHA (stored in `REVS_Common\config_packages`) define quite a few useful keys and tags that should cover many modeling applications but new ones are easy to add. There are two different approaches for adding new keys and associated processing functions. One approach is to create a new config option package, this is discussed further in *Constructing Config Options*. The remainder of this section shows how to add custom keys and associated processing for a single batch. A demo that uses this feature can be found in `run_ALPHA_demo.m`.

The first step is adding the key to the batch. This is done using the `sim_batch.add_key` method. Similar to the the `class_REVS_sim_config_key` constructor the first argument is a string containing the key name. The other options listed below can be used can configure how the key is processed:

Parameter	Usage
tag	Tag for use with config strings
eval	Evaluate tag value used in config strings
default	Default value to use if none provided
description	Description to display in show keys

With the config key defined any scripting to provide the desired function can be specified using one of the following methods `sim_batch.add_case_preprocess_script` for scripts to run before simulation, `sim_batch.add_case_postprocess_script` for scripts run after a simulation, and `sim_batch.add_batch_postprocess_script` for scripts to run after all cases have been run and post processed. Scripts specified in the case processing methods will be run in the simulation workspace. The data specified in the config keys will be accessible via the `sim_config` variable. Batch post process scripts are run in the workspace of the `sim_batch.run_sim_cases` method.

3.2.8 Constructing Config Options

The process for creating a custom config option package is very similar to what is shown in the previous section and is shown in the example below.

```
function package = REVS_config_transmission()
%REVS_CONFIG_AMBIENT REVS configuration keys for ambient conditions


package = class_REVS_sim_config_options();

package.keys = [ ...
    class_REVS_sim_config_key('transmission',                  'tag',  'TRANS',    'eval
↪',   false);

    class_REVS_sim_config_key('transmission_vintage',          'tag',  'TRX_VTG');

    class_REVS_sim_config_key('TC_K_factor',                   'tag',  'TCK');
    class_REVS_sim_config_key('TC_stall_rpm',                  'tag',  'TCSTALL');

    class_REVS_sim_config_key('TC_torque_ratio',               'tag',  'TCTR');

    class_REVS_sim_config_key('TC_lockup_efficiency_pct',      'tag',  'TCLUEFF_PCT');

    class_REVS_sim_config_key('transmission_autoscale',        'tag',  'NTRT',
```

(continues on next page)

(continued from previous page)

```
↪'default', 0);

];
```

package.case_preprocess_scripts = [...
class_REVS_sim_config_script('REVS_config_transmission_load', 1) class_REVS_sim_config_script('REVS_config_transmissi
5)];

Config option packages are functions that return a package of class `class_REVS_sim_config_options`, and the first step in the function is to create that object. Next, the keys are defined as an array of `class_REVS_sim_config_key` elements where the optional arguments for tag, eval and default value discussed previously are specified and stored into the `keys` property. Finally the scripts are specified in the appropriate properties. `case_preprocess_scripts` are run before simulation `case_postprocess_scripts` are run after simulation and `batch_postprocess_scripts` are run after all simulations are complete. Each of these properties must be set as an array os `class_REVS_sim_config_script``objects. The constructor for ``class_REVS_sim_config_scripts` takes two arguments. The first is the name of the script, the second defines when the scripts will be run.

3.3 Controlling Datalogging and Auditing

This section describes how to control the datalogging and auditing features of ALPHA. It may be helpful to understand the different data objects generated, which can be found in *Workspace Outputs*.

3.3.1 Controlling Datalogging

Datalogging and auditing are controlled by the settings stored in the `logging_config` property of the `class_REVS_sim_batch` object. `logging_config` is an object of class `class_REVS_logging_config`. The `add_log` method of `class_REVS_sim_batch` is used to add logging packages that define signals to log within the ALPHA model. Many predefined log lists are contained in the `REVS_Common\log_packages` folder including meta-packages that are intended to provide an easy bundle of packages. These packages will control what data is avaialble in the `datalog`, `result` and `model_data` output variables.

The following are typical examples of creating a sim batch and setting up the datalogging:

```
sim_batch = class_REVS_sim_batch();
sim_batch.add_log(REVS_log_default);
```

`REVS_log_default` logs only the bare minimum required to calculate fuel economy and GHG emissions, this runs the fastest

```
sim_batch = class_REVS_sim_batch();
sim_batch.add_log(REVS_log_all);
```

`REVS_log_all` logs every available signal, this runs the slowest

Log packages can also be combined to tailor the output to a projects needs:

```
sim_batch = class_REVS_sim_batch();
sim_batch.add_log(REVS_log_default);
sim_batch.logging_config.add_log(REVS_log_engine_all);
sim_batch.logging_config.add_log(REVS_log_transmission);
```

Logs the minimum required signals and adds all the engine signals and many common transmission datalogs.

3.3.2 Constructing Log Packages

Log packages are built as functions that return a `class_REVS_log_package` object or an array of objects. The package functions generally consist of three parts. The first is the list of signals to log stored into the `log_list` property. These are the signals specified in the logging blocks of the model, and wildcards can be used to select multiple items. Note that as mentioned in *Workspace Outputs* additional signals may be available if they can be calculated from the logged output. Next, stored in the `package_list` property is the name of any contained packages. The list of packages is available for the post processing to determine is necessary signals are available to complete a given calculation. Generally, the name of the log package function is used. The final item, stored in the `postprocess_list` property is a list of scripts to run after simulation, which can be used to calculate or adjust and outputs. Below the `REVS_log_all` package is shown which demonstrates selecting signals via wildcard, using mfilename for package naming and uses an array of postprocessing scripts.

```
function [log_package] = REVS_log_all()

log_package = class_REVS_log_package;

log_package.log_list = {
    'result.*'
    'datalog.*'
    };

log_package.package_list = {mfilename};

log_package.postprocess_list = {'REVS_postprocess_accessory_battery_log',
                                'REVS_postprocess_alternator_log',
                                'REVS_postprocess_DCDC_log',
                                'REVS_postprocess_drive_motor_log',
                                'REVS_postprocess_engine_basics_log',
                                'REVS_postprocess_engine_idle_log',
                                'REVS_postprocess_mech_accessories_log',
                                'REVS_postprocess_propulsion_battery_log',
                                'REVS_postprocess_transmission_log',
                                'REVS_postprocess_vehicle_basics_log',
                                'REVS_postprocess_vehicle_performance_log',
                                };

end
```

3.3.3 Auditing

Auditing of the energy flows within the model is another feature of ALPHA that can be controlled by the audit flags of the `logging_config` propertry of `class_REVS_sim_batch` and their usage is shown below.

```
sim_batch.logging_config.audit_total = true;
```

Audits the total energy flow for the entire drive cycle.

Or:

```
sim_batch.logging_config.audit_phase = true;
```

Audits the total energy flow for the entire drive cycle and also audits each drive cycle phase individually.

By default both flags are set to false, only one flag or the other needs to be set. To print the audit to the console, use the `print` method of the `audit` variable that is generated in the workspace:

```
audit.print
```

This should return something like the following for a conventional vehicle:

```
   EPA_UDDS audit: -----------------

         ---- Energy Audit Report ----

Gross Energy Provided            = 28874.34 kJ
    Fuel Energy                  = 28868.08 kJ     99.98%
    Stored Energy                =     6.26 kJ      0.02%
    Kinetic Energy               =     0.00 kJ      0.00%
    Potential Energy             =     0.00 kJ      0.00%

Net Energy Provided              =  7641.47 kJ
    Engine Energy                =  7637.05 kJ   99.94%
         Engine Efficiency       =    26.46 %
    Stored Energy                =     4.41 kJ    0.06%
    Kinetic Energy               =     0.00 kJ    0.00%
    Potential Energy             =     0.00 kJ    0.00%

Energy Consumed by ABC roadload  =  3007.20 kJ     39.35%
Energy Consumed by Gradient      =     0.00 kJ      0.00%
Energy Consumed by Accessories   =   823.48 kJ     10.78%
    Starter                      =     0.40 kJ      0.01%
    Alternator                   =   286.81 kJ      3.75%
    Battery Stored Charge        =     0.00 kJ      0.00%
    Engine Fan                   =     0.00 kJ      0.00%
         Electrical              =     0.00 kJ      0.00%
         Mechanical              =     0.00 kJ      0.00%
    Power Steering               =     0.00 kJ      0.00%
         Electrical              =     0.00 kJ      0.00%
         Mechanical              =     0.00 kJ      0.00%
    Air Conditioning             =     0.00 kJ      0.00%
         Electrical              =     0.00 kJ      0.00%
         Mechanical              =     0.00 kJ      0.00%
    Generic Loss                 =   536.27 kJ      7.02%
         Electrical              =   536.27 kJ      7.02%
         Mechanical              =     0.00 kJ      0.00%
    Total Electrical Accessories =   536.27 kJ      7.02%
    Total Mechanical Accessories =     0.00 kJ      0.00%
Energy Consumed by Driveline     =  3811.03 kJ     49.87%
     Engine                      =     0.00 kJ      0.00%
     Launch Device               =   541.63 kJ      7.09%
     Gearbox                     =  1572.46 kJ     20.58%
         Pump Loss               =   874.74 kJ     11.45%
         Spin Loss               =   382.50 kJ      5.01%
         Gear Loss               =   256.71 kJ      3.36%
         Inertia Loss            =    58.51 kJ      0.77%
     Final Drive                 =     0.00 kJ      0.00%
     Friction Brakes             =  1669.65 kJ     21.85%
```

(continues on next page)

(continued from previous page)

```
     Tire Slip                    =    27.30 kJ      0.36%
System Kinetic Energy Gain        =     0.44 kJ      0.01%
                                    ------------
Total Loss Energy                 =  7642.15 kJ
Simulation Error                  =    -0.68 kJ
Energy Conservation               =  100.009 %
```

3.4 How to Save and Restore Simulation Workspaces

There are several methods available to save and restore simulation workspaces. Generally, only one approach will be used at a time, but it is possible to combine approaches if desired.

3.4.1 Retain Workspaces in Memory

The simplest approach, for a relatively small number of simulations, is to retain the workspace in memory. Set the sim batch `retain_output_workspace` property to true. For example:

```
sim_batch.retain_output_workspace = true;
```

The workspace will be contained in the sim batch `sim_case` property which holds one or more `class_REVS_sim_case` objects. To pull the workspace into the top-level workspace, use the sim case's `extract_workspace()` method:

```
sim_batch.sim_case(1).extract_workspace;
```

The workspace is contained in the sim case workspace property but extracting the workspace to the top-level makes it easier to work with.

3.4.2 Saving the Input Workspace

The simulation workspace may be saved prior to simulation by setting the sim batch `save_input_workspace` property to true:

```
sim_batch.save_input_workspace = true;
```

This will create a timestamped `.mat` file in the sim batch output folder's `sim_input` directory. The filename also includes the index of the sim case. For example, the input workspace for the first simulation (`sim_1`) in a batch:

```
output\sim_input\2019_02_11_16_46_37_sim_1_input_workspace.mat
```

The workspace is saved after all pre-processing scripts have been run so the workspace contains everything required to replicate the simulation at a later time. This can be useful when running too many simulations to retain the workspaces in memory while also providing the ability to run individual cases later without having to set up a custom sim batch. The workspace may be loaded by using the load command, or double-clicking the filename in the Matlab Current Folder file browser.

3.4.3 Saving the Output Workspace

The simulation workspace may be saved after simulation by setting the sim batch `save_output_workspace` property to true:

```
sim_batch.save_output_workspace = true;
```

This will create a timestamped `.mat` file in the sim batch output folder. The filename also includes the index of the sim case. For example, the output workspace for the first simulation (`sim_1`) in a batch:

```
output\2019_02_11_16_52_39_sim_1_output_workspace.mat
```

The workspace is saved after all post-processing scripts have been run so the workspace contains everything required to replicate the simulation at a later time and also all of the datalogs, audits, etc. The simulation may be run again or the outputs examined directly without the need for running the simulation. Keep in mind that output workspaces will always be bigger than input workspaces and also take longer to save. The workspace may be loaded by using the load command or double-clicking the filename in the Matlab Current Folder file browser. Also note that the resulting mat file will contain variables constructed from various REVS classes and will require `REVS_common` to be on the MATLAB path to operate property.

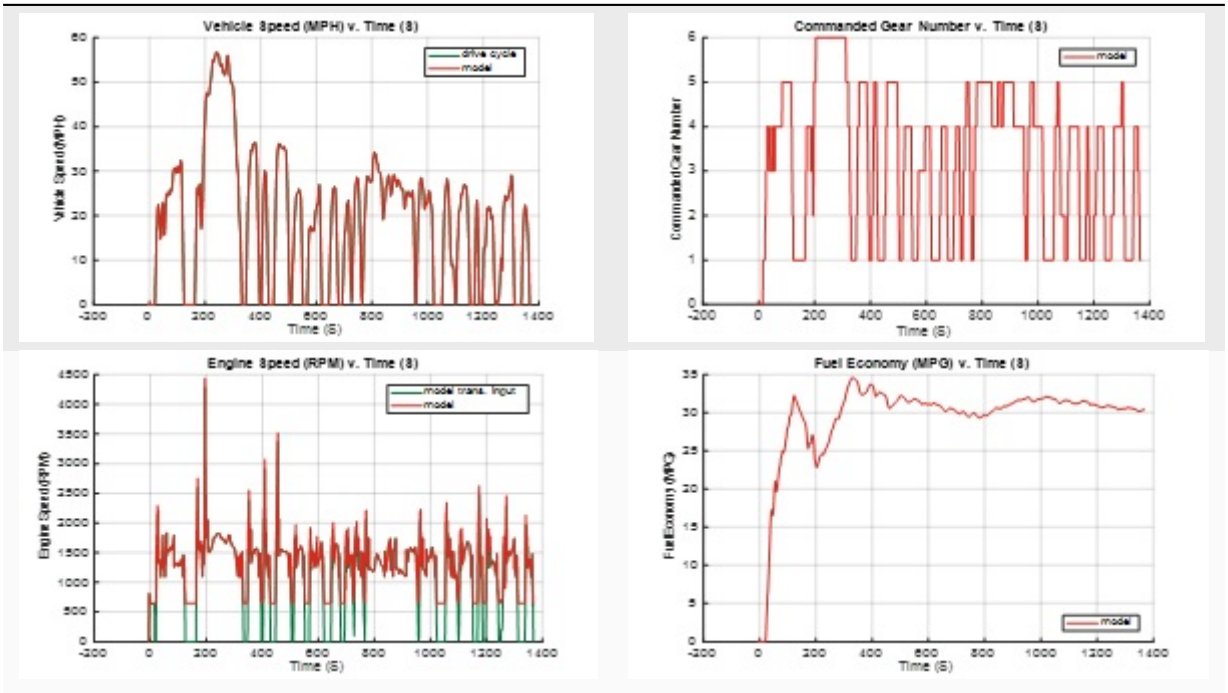
3.5 Post-Simulation Data Analysis

As mentioned, a `model_data` object is created in the output workspace and may contain various model outputs. One of the easiest ways to take a look at simulation data is to run a Data Observation Report (DOR) on the model data. There are DORs for conventional (CVM), hybrid (HVM) and electric vehicles (EVM). To run the default conventional vehicle model DOR, use the `REVS_DOR_CVM()` function:

```
REVS_DOR_CVM({}, model_data);
```

The first parameter (unused, in this case) allows the model outputs to be compared with one or more sets of test data in the form of `class_test_data` objects. If there are multiple sets of test data, the first input would be a cell array of `class_test_data` objects. The default DOR generates a number of plots representing some of the most commonly observed outputs such as vehicle speed, engine speed, transmission gear number, etc. For example:

Table 3.1: Sample Figures from `REVS_DOR_CVM()`



The various DORs support several optional arguments, known as varargs in Matlab. Optional arguments are passed in after the `model_data` and consist of strings and/or string-value pairs. For example:

```
REVS_DOR_CVM({}, model_data, 'name of some vararg', vararg_value_if_required);
```

The top-level DOR calls sub-DORs that are grouped by component, for example `REVS_DOR_CVM()` calls `REVS_DOR_vehicle()`, `REVS_DOR_engine()`, etc. Each component DOR may have its own unique varargs in addition to supporting some common varargs. Varargs passed to the top-level DOR are automatically passed to the component DORs. Available varargs are listed in (Table 3.2).

Table 3.2: List of Available DOR Varargs

Vararg Target	Vararg Name	Value	Description
Common	'descriptor'	string	A string description of the data being presented, for example 'ALPHA Quickstart'
	'time_range'	numeric vector	A two-element vector representing the desired start and end time in seconds for plots and analysis. For example, [505 1375]
REVS_DOR_Vehicle()	'vehicle_speed_units'	string	Vehicle speed units will be miles per hour (by default or if ?mph? is provided) else units will be meters per second
REVS_DOR_Engine()	'engine'	class_REVS_engine object	If provided, an engine map will be plotted, showing areas of operation during the simulation, limited to the ?time_range?, if provided
	'engine_speed_units'	string	Engine speed units will be RPM (by default or if ?rpm? is provided) else units will be radians per second
REVS_DOR_Fuel()	'fuel_plots'	none	Enables fuel plots, if provided, otherwise fuel plots are disabled
REVS_DOR_Transmission()	'analyze_ratios'	none	BROKEN
REVS_DOR_Accessories()	'accessory_plots'	none	Enables accessory plots such as alternator and battery current, voltage, etc, if provided

3.6 Understanding Datalogging

This section will provide details on how to control and understand the datalogging process in ALPHA.

3.6.1 Logging Overview

Logging model internal signals is probably one of the most important things the model does, it is also one of the things that has the biggest impact on model run time. Simulink seems to incur quite a bit of overhead related to logging data to the workspace. As a result, ALPHA implements a flexible system to control how much or how little data is logged from the model. In this way, the user can trade off run time speed and the logging of signals of interest.

The `REVS_Common\log_packages` folder contains functions to define pre-made 'packages' of signals for datalogging, and also scripts for post-processing the data if required.

`class_REVS_log_package` defines the data structure used to define datalogs. Each package has three properties:

- `log_list` - a list of `datalog` or `result` signals to enable. Signal names can include * wildcards. For example, `result.engine.crankshaft*` would log all result signals that start contain `engine.crankshaft` such as `result.phase.engine.crankshaft_tot_kWh` or `result.phase.engine.crankshaft_pos_kJ`. Result signals are a unique form of datalog that record final values for each phase of the drive cycle. So for each phase of the drive cycle a `result` will contain a scalar value for each signal. The result may be a sum or an average or other statistical data such as a minimum or maximum. See the `logging_lib` for more details.

- `package_list` - a package may contain other packages, however in practice, each package lists itself in the `package_list` and the total package list is the unique set of all the individual packages. So, each `REVS_log_XXX.m` will contain `log_package.package_list = {mfilename};`. Metapackages are formed by creating a list of packages, such as `REVS_log_CVM_metapackage` which creates the metapackage of conventional vehicle model (CVM) datalogs:

```
function [log_package] = REVS_log_CVM_metapackage()

log_package = [
               REVS_log_vehicle_basics
               REVS_log_engine_basics
               REVS_log_transmission
               REVS_log_alternator
               REVS_log_accessory_battery
               REVS_log_mech_accessories
              ];

end
```

- `postprocess_list` - contains a list of one or more post-processing scripts to run after the workspace has been populated with data. For example, `REVS_log_engine_basics` lists `REVS_postprocess_engine_basics_log` to post-process data from raw simulation signals into the `model_data` structure for more universal use in post-processing scripts such as plotting simulation data versus real-world test data as in a `DOR`.

CHAPTER

FOUR

COMMON USE CASES

This chapter will present a few common use cases of ALPHA as an aid to getting started.

4.1 Running a Batch with Various Engines

The typical method of running several engines is simply to define the engine names as strings in the workspace then set up simulation cases for each one. For example:

```
GDI_ENGINE              = 'ENG:engine_2013_Chevrolet_Ecotec_LCV_2L5_Reg_E10';
TDS12_ENGINE            = 'ENG:engine_2016_Honda_L15B7_1L5_Tier2';
TDS12_ENGINE2           = 'ENG:engine_2016_Honda_Turbo_1L5_paper_image';
TDS21_ENGINE            = 'ENG:engine_future_Ricardo_EGRB_1L0_Tier2';
TDS11_ENGINE            = 'ENG:engine_2015_Ford_EcoBoost_2L7_Tier2';
TDS11_ENGINE2           = 'ENG:engine_2013_Ford_EcoBoost_1L6_Tier2';
ATK2p0_ENGINE           = 'ENG:engine_2014_Mazda_Skyactiv_2L0_Tier2';
ATK2p0_CEGR_ENGINE      = 'ENG:engine_future_atkinson_CEGR_2L0_tier2';
TNGA_ENGINE             = 'ES_CYL:6 + ENG:engine_2016_toyota_TNGA_2L5_paper_image';
ATK2p5_ENGINE           = 'ENG:engine_2016_Mazda_Skyactiv_Turbo_2L5_Tier2';
ATK2p0_X_ENGINE         = 'ENG:engine_future_Mazda_Skyactiv_X_2L0_paper_image';

config_strings = {
    ['PKG:1a + ' base_config TDS11_ENGINE2      ...]
    ['PKG:1b + ' base_config TDS11_ENGINE       ...]
    ['PKG:1c + ' base_config TDS12_ENGINE2      ...]
    ['PKG:1d + ' base_config TDS12_ENGINE       ...]
    ['PKG:1e + ' base_config TDS21_ENGINE       ...]
    ['PKG:1f + ' base_config ATK2p0_ENGINE      ...]
    ['PKG:1g + ' base_config TNGA_ENGINE        ...]
    ['PKG:1h + ' base_config ATK2p5_ENGINE      ...]
    ['PKG:1i + ' base_config ATK2p0_X_ENGINE    ...]
    ['PKG:1j + ' base_config ATK2p0_CEGR_ENGINE ...]
...
};

sim_batch.load_config_strings(config_strings);
```

In this abbreviated example, `base_config` refers to a workspace variable that holds a string of the config tags that all the cases have in common, for example roadload settings, drive cycle selection, fuel type, etc. Grouping the common settings into a single variable makes it easier to change the setup and improves readability. Matlab string concatenation does the rest (the use of brackets, [ ], tells Matlab to combine all the separate strings into one). Another advantage of using workspace variables to hold engine definition strings is illustrated in the `TNGA_ENGINE` workspace variable which

not only defines the engine but also uses the `ES_CYL:` tag to tell the simulation to run as a six-cylinder regardless of any engine resizing that may take place. Breaking a config string down into smaller substrings and workspace variables is a good technique for managing complexity in larger batches.

The use of the `PKG:` defines a quick reference name for each case.

4.2 ALPHA Roadloads and Test Weight

Vehicle weight / inertia is specified by setting the ETW (Equivalent Test Weight, which includes vehicle curb weight and a ballast of 300 pounds and is effectively tested with a 1.5% axle inertia penalty) or by setting the vehicle mass and inertias directly.

Roadloads in ALPHA can be specified either by "ABC" (or "F0, F1, F2") coastdown curve fit coefficients or by directly specifying the coefficients of rolling resistance and aerodynamic drag along with the vehicle's frontal area.

A convenient source of ABC coefficients and test weights is the EPA's own test car data, such as at https://www.epa.gov/compliance-and-fuel-economy-data/data-cars-used-testing-fuel-economy.

4.2.1 Setting Vehicle Weight and Inertia

The test car list format varies somewhat over time, but the vehicle ETW is listed in the `Equivalent Test Weight (lbs.)` column in the 2020 test car data.

The ETW in the test car list is determined by vehicle curb weight (with a full tank of gas, all fluids, accessories, etc) plus a 300 lb ballast penalty. ETW is binned in fixed increments for compliance purposes (a throwback to old water-brake dynos with discrete inertia weights). Larger test weights are in larger bins. The bins are defined in 40 CFR § 1066.805.

So ETW is fairly straightforward. Where it gets more interesting is when the axle inertias are factored into the dyno settings. As an engineering rule of thumb, the inertia of each axle (including wheels, tires, brakes, etc) acts as an effective 1.5% weight penalty.

As a matter of EPA test procedure for a two-wheel-drive test, the dyno simulated inertia is set to the ETW (See 40 CFR § 1066.410). As a result of the spinning front (or rear) axle, the effective total inertia is ETW * 1.015.

For a 4WD (dual roll) test, if the dyno inertia were set to the ETW, the approximate total inertia would be ETW * 1.03, accounting for both axles spinning. As a result, for a 4WD test, the dyno inertia is set to ETW * 0.985, since ETW * 1.03 * 0.985 is approximately ETW * 1.015, thereby maintaining the total test inertia when compared with a 2WD certification test.

For consistency with certification testing, setting the ETW results in the simulated inertia being set to ETW * 1.015. If the intention is to model a vehicle with actual weights and component inertias then the ETW property should not be used, the individual masses and inertias should be set directly instead, as discussed below.

Within ALPHA there are several parameters that determine the vehicle's weight and equivalent weight considering axle inertia. It is possible to set the mass and inertias directly and independently. It is also possible to set the ETW and allow for the standard inertia adjustment as described above.

To set ETW, either use the `ETW_LBS:` or `ETW_KG:` config string tags or set the vehicle's `ETW_lbs` or `ETW_kg` property in a param file, for example:

```
vehicle = class_REVS_vehicle;
...
vehicle.ETW_lbs = 3500;
```

Vehicle Mass Properties

The `class_REVS_vehicle` properties related to mass are:

```
ETW_kg
mass_static_kg
mass_dynamic_kg
mass_curb_kg
mass_ballast_kg

ETW_lbs
mass_curb_lbs
mass_ballast_lbs
mass_static_lbs
mass_dynamic_lbs
```

Conversion between pounds and kilograms is automatic, so there is no need for the user to manually convert between SAE and SI units, just set the simulation settings based on the source data used.

The `curb` and `ballast` masses are the vehicle curb mass and ballast mass as discussed above. The `static` mass is the curb mass plus the ballast mass and the `dynamic` mass is the static mass plus weight-equivalent axle inertias, if desired. The `dynamic` mass is used to calculate vehicle acceleration in the model. The `static` mass is used to calculate roadload forces due to road grade and rolling resistance (if ABC coefficients are not used, see below), so both must be set correctly if the drive cycle grade is non-zero or if rolling resistance drag coefficients are used.

Because the mass terms are interrelated, `class_REVS_vehicle` provides methods to try to keep them synchronized, such that a change in curb weight will result in a change in the dynamic weight, etc. Setting the ETW sets the static mass to ETW * 0.985, dynamic mass to ETW * 1.015, ballast mass to 300 lbs and curb weight to static mass minus ballast. In practice, the various terms can get out of sync depending on the order in which they are set, so it's best to just use the ETW property or set the individual non-ETW terms separately.

Using Component Inertias

If the goal is to simulate known inertias and actual vehicle weights then it is necessary to set the individual component inertias and masses directly. For example:

```
vehicle = class_REVS_vehicle;
...
vehicle.mass_static_kg = 1000
vehicle.mass_dynamic_kg = 1000 % no default adjustment, actual inertias defined below

vehicle.drive_axle1.tire.inertia_kgm2 = 0.9 * 4 % for a single-axle-equivalent model
vehicle.drive_axle1.final_drive.inertia_kgm2 = 0.1
... etc
```

Setting `mass_static_kg` defaults the dynamic mass to 1.03 * mass_static, so it needs to also be set manually.

4.2.2 ABC Coefficients

The test car list format varies somewhat over time, but the ABC coefficients for the 2020 test car data are in the following columns:

```
Target Coef A (lbf)
Target Coef B (lbf/mph)
Target Coef C (lbf/mph**2)

Set Coef A (lbf)
Set Coef B (lbf/mph)
Set Coef C (lbf/mph**2)
```

The `Target` coefficients represent the observed drag forces acting on the vehicle during coastdown, treating the vehicle as a point mass. The `Set` coefficients are determined by coasting the vehicle down on a vehicle dynamometer and adjusting the set coefficients in the target coastdown is achieved, within a tolerance.

It should be noted that the target ABC coefficients represent internal **and** external losses that act on the vehicle during coastdown. As such, a portion of the ABC coefficients may represent driveline drag that may also be present in the transmission component model, for example. Using unmodified ABC coefficients will generally result in an over-estimation of the fuel consumption of a vehicle, by a few percent.

`class_REVS_vehicle` contains a `calc_roadload_adjust` method to approximate the driveline loss double-count given a set of target and dyno-set ABC coefficients, based on vehicles in the 2019 test car list. For more information, see SAE 2020-01-1064. As these losses may vary over time as the fleet evolves, it is the responsibility of the user to determine if the adjustments are appropriate for newer or older vehicles.

ABC coefficients can be specified in SAE or SI units, via the `class_REVS_vehicle` properties:

```
coastdown_target_A_N;                    % coastdown target "A" term, SI units, Newtons
coastdown_target_B_Npms;                 % coastdown target "B" term, SI units, Newtons /␣
↪(meter / second)
coastdown_target_C_Npms2;                % coastdown target "C" term, SI units, Newtons /␣
↪(meter / second)^2

dyno_set_A_N;                            % dyno set "A" term, SI units, Newtons
dyno_set_B_Npms;                         % dyno set "B" term, SI units, Newtons / (meter /
↪ second)
dyno_set_C_Npms2;                        % dyno set "C" term, SI units, Newtons / (meter /
↪ second)^2

and

coastdown_target_A_lbf;                 % coastdown target "A" term, SAE units, pounds␣
↪force
coastdown_target_B_lbfpmph;             % coastdown target "B" term, SAE units, pounds␣
↪force / mph
coastdown_target_C_lbfpmph2;            % coastdown target "C" term, SAE units, pounds␣
↪force / mph^2

dyno_set_A_lbf;                          % dyno set "A" term, SAE units, pounds force
dyno_set_B_lbfpmph;                      % dyno set "B" term, SAE units, pounds force /␣
↪mph
dyno_set_C_lbfpmph2;                     % dyno set "C" term, SAE units, pounds force /␣
↪mph^2
```

Units provided in SAE units are automatically converted to SI units, and vice versa, so there is no need for the user to manually convert values.

Roadload adjustments, if desired, are stored in:

```
coastdown_adjust_A_N = 0;               % coastdown adjustment for double counting "A"␣
↪term, SI units, Newtons
coastdown_adjust_B_Npms = 0;            % coastdown adjustment for double counting "B"␣
↪term, SI units, Newtons / (meter / second)
coastdown_adjust_C_Npms2 = 0;           % coastdown adjustment for double counting "C"␣
↪term, SI units, Newtons / (meter / second)^2

and

coastdown_adjust_A_lbf;                 % coastdown adjustment for double counting "A"␣
↪term, SAE units, pounds force
coastdown_adjust_B_lbfpmph;             % coastdown adjustment for double counting "B"␣
↪term, SAE units, pounds force / mph
coastdown_adjust_C_lbfpmph2;            % coastdown adjustment for double counting "C"␣
↪term, SAE units, pounds force / mph^2
```

Adjust values are added to the target values, so they should be negative to decrease roadload.

To enable the use of ABC coefficients, the `use_abc_roadload` property should be set to `true`. A typical example param file snippet:

```
vehicle = class_REVS_vehicle;
...
vehicle.use_abc_roadload = true;

vehicle.coastdown_target_A_lbf          = 32.27;
vehicle.coastdown_target_B_lbfpmph      = 0.0754;
vehicle.coastdown_target_C_lbfpmph2     = 0.01993;
```

ABC coefficients can also be set using config tags. Sample output from `sim_batch.show_tags` is shown below (keys are defined in `REVS_config_vehicle`, as seen in 'Provided by'):

```
Target and dyno-set tags:

   Key                                  |     Tag                |     Default Value  ␣
↪                |     Provided by                |     Description
-------------------------------------------------------------------------------------
↪-------------------------------------------------------------

target_A_lbs                            |  TRGA_LBS              |                    ␣
↪                |  REVS_config_vehicle           |
target_B_lbs                            |  TRGB_LBS              |                    ␣
↪                |  REVS_config_vehicle           |
target_C_lbs                            |  TRGC_LBS              |                    ␣
↪                |  REVS_config_vehicle           |
dyno_set_A_lbs                          |  DYNA_LBS              |                    ␣
↪                |  REVS_config_vehicle           |
dyno_set_B_lbs                          |  DYNB_LBS              |                    ␣
↪                |  REVS_config_vehicle           |
dyno_set_C_lbs                          |  DYNC_LBS              |                    ␣
```

(continues on next page)

(continued from previous page)

```
↪               |  REVS_config_vehicle           |

target_A_N                                  |  TRGA_N               |                 ␣
↪               |  REVS_config_vehicle           |
target_B_N                                  |  TRGB_N               |                 ␣
↪               |  REVS_config_vehicle           |
target_C_N                                  |  TRGC_N               |                 ␣
↪               |  REVS_config_vehicle           |
dyno_set_A_N                                |  DYNA_N               |                 ␣
↪               |  REVS_config_vehicle           |
dyno_set_B_N                                |  DYNB_N               |                 ␣
↪               |  REVS_config_vehicle           |
dyno_set_C_N                                |  DYNC_N               |                 ␣
↪               |  REVS_config_vehicle           |

Adjustment tags:

adjust_A_lbs                                |  ADJA_LBS             |                 ␣
↪               |  REVS_config_vehicle           |
adjust_B_lbs                                |  ADJB_LBS             |                 ␣
↪               |  REVS_config_vehicle           |
adjust_C_lbs                                |  ADJC_LBS             |                 ␣
↪               |  REVS_config_vehicle           |

adjust_A_N                                  |  ADJA_N               |                 ␣
↪               |  REVS_config_vehicle           |
adjust_B_N                                  |  ADJB_N               |                 ␣
↪               |  REVS_config_vehicle           |
adjust_C_N                                  |  ADJC_N               |                 ␣
↪               |  REVS_config_vehicle           |
```

Automatic calculation of the roadload adjustments discussed above can be performed, using the `CALC_ABC_ADJ:` tag, as in:

```
'... + CALC_ABC_ADJ:1 + ...'
```

4.2.3 Drag Coefficients

Drag coefficients can be set by the using the following `class_REVS_vehicle` properties:

```
frontal_area_m2;                    % vehicle frontal area, square meters
aerodynamic_drag_coeff;             % vehicle aerodynamic drag coefficient
```

and the `rolling_resistance_coefficient` of the drive axle tire component as well as the `vehicle_weight_norm` property which says what proportion of the vehicle's weight is applied to the given axle.

The `vehicle.use_abc_roadload` must also be set to `false`.

Example param file snippet:

```
vehicle = class_REVS_vehicle;
...
```

(continues on next page)

(continued from previous page)

```
vehicle.use_abc_roadload = false;
vehicle.frontal_area_m2 = 2.0;
vehicle.aerodynamic_drag_coeff = 0.33;
...
vehicle.drive_axle1.tire.rolling_resistance_coefficient = 0.010;
vehicle.drive_axle1.tire.vehicle_weight_norm = 1.0
```

If a vehicle has multiple drive axles then the rolling resistance coefficient and `vehicle_weight_norm` must be set for each axle. The sum of the axle `vehicle_weight_norm` terms must add up to 1.0. For the default vehicle, a single axle configuration is used and `vehicle_weight_norm` defaults to 1.0.

At the time of this writing there are no config tags for setting drag coefficients so they must be specified in the vehicle param file as seen in the snippet above.

4.3 Drive Cycles

ALPHA comes with a set of drive cycles, in the `REVS_Common\drive_cycles` folder. The general naming convention is SOURCE_CYCLE, as in `EPA_HWFET` for the EPA highway cycle or `UNECE_WHVC` for the European World Harmonized Vehicle Cycle. Each `.mat` file contains a single object of type `class_REVS_drive_cycle` named `drive_cycle`.

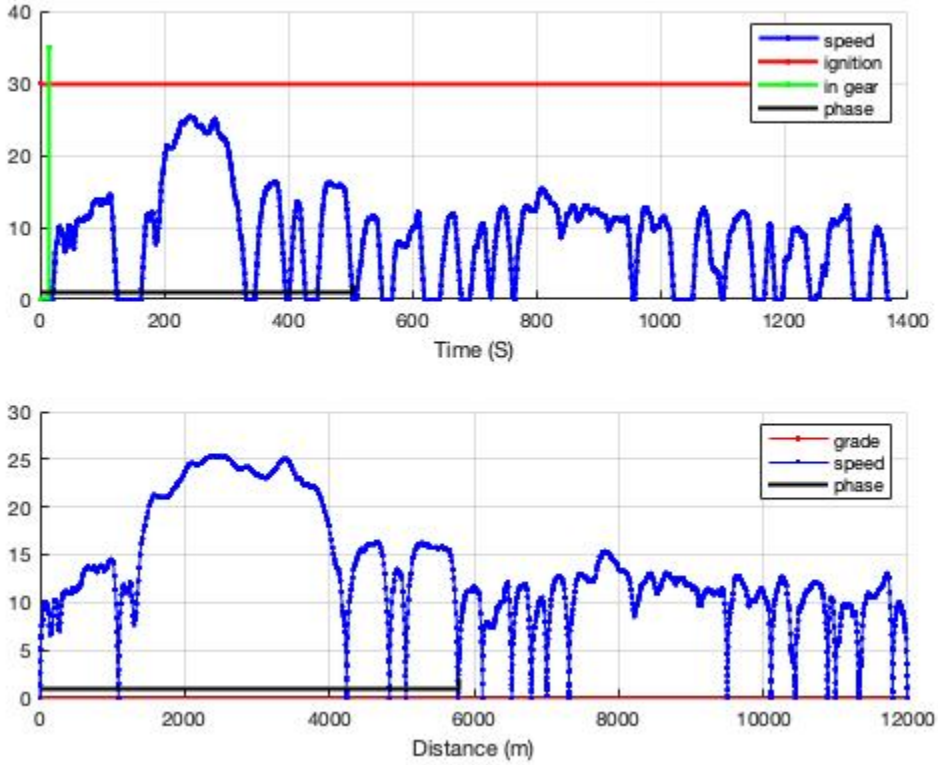
The drive cycle may be plotted using `REVS_plot_drive_cycle` as follows:

```
>>  REVS_plot_drive_cycle % if `drive_cycle` is in the workspace

or

>>  REVS_plot_drive_cycle(drive_cycle_object)
```

Which results in the following for `EPA_UDDS`:



The data structure for the same cycle looks like:

```
>> drive_cycle

drive_cycle =
  class_REVS_drive_cycle with properties:

                  name: 'EPA_UDDS'
    sample_start_enable: 1
             phase_name: ["1"    "2"]
                  phase: [1 2]
             phase_time: [0 505]
             cycle_time: [1370×1 double]
        cycle_speed_mps: [1370×1 double]
                in_gear: [0 1]
           in_gear_time: [0 15]
               ignition: [1 1]
          ignition_time: [0 1369]
           grade_dist_m: [0 11990.238656]
              grade_pct: [0 0]
      cfr_max_speed_mps: [1370×1 double]
      cfr_min_speed_mps: [1370×1 double]
```

Where

- `name` is the name of the drive cycle
- If `sample_start_enable` is true then datalogging begins immediately, otherwise datalogging drive cycle phase

results doesn't start until time 0. Simulation start time is set in the simulation workspace variable `REVS.sim_start_time_secs`. The default value is -5.

- `phase_name` contains the names of the drive cycle phases as strings.
- `phase` and `phase_time` define the phase numbers and start times of the drive cycle phases. Drive cycle phase results are only logged for non-zero phase numbers.
- `cycle_time` and `cycle_speed_mps` define the speed trace (in meters per second) versus time.
- `in_gear` and `in_gear_time` determines when the vehicle driveline is engaged and active (as in the case of normal driving) or disengaged and deactivated (as in the case of a coastdown)
- `ignition` and `ignition_time` define when the vehicle is meant to be running and the engine started (as for conventional vehicles).
- `grade_dist_m` and `grade_pct` define the road grade as a function of distance in meters. Grade is defined by distance and not time to cover the case where heavy vehicles may not be able to maintain the desired speed on high grades. It is recommended to run the driver model (aka "cyberdriver") in distance compensated mode when running grade cycles by setting `driver.distance_compensate_enable` to `true` in the appropriate driver param file. Distance compensation extends the drive cycle time when the vehicle falls behind the target speed and contracts it when the vehicle speed exceeds the target speed. `distance_compensate_enable` defaults to `false`, which is appropriate for zero-grade drive cycles.
- `cfr_max_speed_mps` and `cfr_min_speed_mps` are calculated values that represent the allowable minimum and maximum speeds (the speed tolerance) at each point in the drive cycle, as defined in 40 CFR § 86.115-78

4.3.1 Turnkey Drive Cycles

The following drive cycles are provided with ALPHA, as well as others.

- `EPA_FTP_NOSOAK` defines a three-phase EPA "city" cycle, with no soak time between phases 2 and 3.
- `EPA_HWFET` defines the EPA "highway" cycle
- `EPA_US06` defines the EPA US06 cycle
- `REVS_Performance_cruise75mph` defines a performance cycle meant to allow for measuring 0-60, 30-50 and 50-70 passing times followed by a 75 mph cruise that can be used to calculate top gear gradability.
- `EPA_FTP_2HWFET_PERF` defines a combined cycle - a three phase FTP followed by a highway prep, the full warmed up highway and a performance drive cycle.

4.3.2 Making Custom Drive Cycles

There are two ways to make new drive cycles: create one from scratch, filling in the drive cycle properties as outline above, or combine existing drive cycles into a new combined cycle.

To combine drive cycles, use the `REVS_combine_drive_cycles` function, as in:

```
>> drive_cycle = REVS_combine_drive_cycles({'EPA_HWFET', 'EPA_US06'})

drive_cycle =
  class_REVS_drive_cycle with properties:

                   name: 'EPA_HWFET & EPA_US06'
    sample_start_enable: 0
             phase_name: ["EPA_HWFET"    "EPA_US06_1"    "EPA_US06_2"]
```

(continues on next page)

(continued from previous page)

```
                phase: [5×1 double]
           phase_time: [5×1 double]
           cycle_time: [1365×1 double]
      cycle_speed_mps: [1365×1 double]
              in_gear: [3×1 double]
         in_gear_time: [3×1 double]
             ignition: [3×1 double]
       ignition_time: [3×1 double]
        grade_dist_m: [4×1 double]
           grade_pct: [4×1 double]
    cfr_max_speed_mps: [1365×1 double]
    cfr_min_speed_mps: [1365×1 double]
```

The `drive_cycle` variable can be saved to a new `.mat` file in the `drive_cycles` folder:

```
>> save('my_new_cycle.mat', 'drive_cycle')
```

CHAPTER

FIVE

MODEL INPUTS

Every model requires inputs, and ALPHA is no exception. In order to run the model it is necessary to populate the Matlab workspace with the data structures required to run the model.

Model inputs can be provided from `.m` files, referred to as param files, or a previously saved workspace may be loaded from a `.mat` file. For information on saving simulation workspaces, see *Saving the Input Workspace* and *Saving the Output Workspace*

5.1 Param Files

Param files are simply Matlab scripts that instantiate required data structures and objects. It's possible to create a single script that creates every variable required by the model, however it's best practice (enforced by the batch process) to separate param files by type.

The batch process canonical expected param file types are:

- an engine param file, using the `ENG:` tag, for non battery-electric vehicles
- a transmission file, using the `TRANS:` tag
- a vehicle file, using the `VEH:` tag, that defines vehicle characterstics such as test weight, roadload, tire radius, etc
- a param file to define the electrical system and/or accessory loads, using the `ELEC:` and/or `ACC:` tags. The electrical and accessory files may be separate or are sometimes combined
- a controls param file, using the `CON:` tag, that defines the overall vehicle behavior such as engine start-top, etc
- an optional driver param file, using the `DRV:` tag, that tunes the response of the "cyberdriver" drive cycle trace follower. If none is provided then the batch process will load the default parameters
- an optional ambient param file, using the `AMB:` tag, that defines the ambient test conditions. If none is provided then the batch process will load the default parameters

The param files are loaded by `class_REVS_sim_params` which is called from the `class_REVS_sim_case` `preprocess_workspace` method as called by the `preprocess_and_load_workspace` method.

Also required, and loaded by `class_REVS_sim_case` from the `preprocess_workspace` method:

- one or more drive cycle file names, using the `CYC:` tag, to define the target vehicle speeds versus time, etc. If multiple drive cycle file names are provided they combined using `REVS_combine_drive_cycles` as outlined in *Making Custom Drive Cycles*

As part of the batch process, each of the above tags may load multiple param files by providing a cell array of strings of the names of the param files, for example:

```
'... + CYC:{''EPA_UDDS'',''EPA_HWFET'',''EPA_US06'',''CARB_LA92''} + ...'
```

Note the use of double single-quotes, '', as opposed to single double-quotes, ", this is required in order for the batch process to evaluate the cell array properly.

If multiple files are provided, they are loaded in left-to-right order, so param files that have dependencies should be listed to the right of the files they depend on.

5.2 Workspace Data Structures

The following variables must be present in the workspace prior to running the model, and are saved by `class_REVS_sim_case` in the `preprocess_and_load_workspace` mehtod if the `save_input_workspace` property of the `sim_batch` is set to `true`.

5.2.1 REVS

An instance of `class_REVS_setup` that stores the top-level settings that control the simulation. `REVS` is created by the `preprocess_workspace` method of `class_REVS_sim_case`.

```
>> REVS

REVS =

  class_REVS_setup with properties:

          current_model: 'REVS_VM'
                verbose: 1
     global_decimation: 1
            output_fid: 1
    sim_step_time_secs: 0.01
   sim_start_time_secs: -5
    sim_stop_time_secs: Inf
        logging_config: [1×1 class_REVS_logging_config]
```

5.2.2 ambient

An instance of `class_REVS_ambient` which defines the ambient environmental conditions of the simulation. The atmospheric properties come into play when using Cd / frontal area drag coefficients, as opposed to ABC roadload coefficients. See *ABC Coefficients* and *Drag Coefficients* for more information on vehicle roadload calculations.

```
>> ambient

ambient =

  class_REVS_ambient with properties:

             variant: 'default ambient'
    temperature_degC: 20
         pressure_Pa: 98210
   air_density_kgpm3: 1.16771071212578
          Rgas_JpkgK: 286.9
        gravity_mps2: 9.80665
```

5.2.3 driver

An instance of `class_REVS_driver` that defines the response of the drive cycle trace-following driver model. These settings typically won't need adjustment, with the exception of `distance_compensate_enable` which should be set `true` for drive cycles with road grade. See *Drive Cycles* for more information on drive cycles, including grade.

```
>> driver

driver =

  class_REVS_driver with properties:

                               variant: 'default driver'
                                    Kp: 1
                                    Ki: 3
                                    Kd: 0
            proportional_fade_in_secs: 1
   proportional_fade_in_min_speed_mps: 2
                       lookahead_secs: 0.25
              launch_anticipate_secs: 0.5
          dynamic_gain_lookahead_secs: 3.75
           distance_compensate_enable: 0
                          late_braking: 0
                     human_mode_enable: 0
                       brake_gain_norm: 0.13
       accel_pedal_response_speed_mps: [0 5 20 70]
            accel_pedal_response_norm: [0.125 0.2 1 1]
```

5.2.4 drive_cycle

An instance of `class_REVS_drive_cycle` that defines the simulation drive cycle. For more information on drive cycles, see *Drive Cycles*.

```
>> drive_cycle

drive_cycle =
  class_REVS_drive_cycle with properties:

                    name: 'EPA_UDDS'
   sample_start_enable: 1
            phase_name: ["1"    "2"]
                 phase: [1 2]
            phase_time: [0 505]
            cycle_time: [1370×1 double]
       cycle_speed_mps: [1370×1 double]
               in_gear: [0 1]
          in_gear_time: [0 15]
              ignition: [1 1]
         ignition_time: [0 1369]
          grade_dist_m: [0 11990.238656]
             grade_pct: [0 0]
     cfr_max_speed_mps: [1370×1 double]
     cfr_min_speed_mps: [1370×1 double]
```

5.2.5 accessories

An instance of `class_REVS_ALPHA_accessories` that defines electrical and mechanical accessory loads.

```
>> accessories

accessories =

  class_REVS_ALPHA_accessories with properties:

              name: 'accessory_EPS_param'
      generic_loss: [1×1 class_REVS_accessory_load]
               fan: [1×1 class_REVS_accessory_load]
    power_steering: [1×1 class_REVS_accessory_load]
   air_conditioner: [1×1 class_REVS_accessory_load]
```

5.2.6 electric

An instance of `class_REVS_electric` that defines the vehicle's electrical system, for both hybrid and conventional vehicles. Not all properties are populated, depending on the powertrain.

```
>> electric

electric =

  class_REVS_electric with properties:

                 name: 'electric_EPS_midsize_car'
       matrix_vintage: present
              starter: [1×1 class_REVS_starter]
           alternator: [1×1 class_REVS_alternator]
     low_voltage_DCDC: [1×1 class_REVS_DCDC_converter]
    high_voltage_DCDC: [1×1 class_REVS_DCDC_converter]
              battery: [1×1 class_REVS_battery]
    accessory_battery: [0×0 class_REVS_battery]
   propulsion_battery: [0×0 class_REVS_battery]
                P0_MG: [0×0 class_REVS_emachine_geared]
          drive_motor: [0×0 class_REVS_emachine]
                  MG1: [0×0 class_REVS_emachine_geared]
                  MG2: [0×0 class_REVS_emachine_geared]
```

5.2.7 controls

A data structure that defines control system parameters. The example below is for a conventional vehicle, that may or may not enable engine start-stop.

```
>> controls

controls =
  class_REVS_CVM_control with properties:
```

(continues on next page)

(continued from previous page)

```
                start_stop_enable: 0
         start_stop_off_delay_secs: 0
       start_stop_warmup_condition: '(@cycle_pos_secs >= 100) && (@cycle_pos_secs <=␣
↪3406)'
   hot_soak_warmup_start_condition: '@cycle_pos_secs > 0 '
                  pedal_map_type: max_engine_power
        pedal_map_engine_torque_Nm: [1×1 class_REVS_dynamic_lookup]
            shift_inertia_est_kgm2: 0.187389225679636
                           variant: ''
```

5.2.8 engine

For conventional or hybrid vehicles, an instance of `class_REVS_engine` that defines engine properties such as torque limits, fuel consumption rates as a function of speed and load, etc.

```
>> engine

engine =
  class_REVS_engine with properties:

                      full_throttle_speed_radps: [17×1 double]
                        full_throttle_torque_Nm: [17×1 double]
                    closed_throttle_speed_radps: [6×1 double]
                      closed_throttle_torque_Nm: [6×1 double]
                naturally_aspirated_speed_radps: [17×1 double]
                  naturally_aspirated_torque_Nm: [17×1 double]
                       power_time_constant_secs: 0.2
                       boost_time_constant_secs: 0.5
              boost_falling_time_constant_secs: 0.3
              run_state_activation_speed_radps: 1
               run_state_activation_delay_secs: 0.5
           run_state_deactivation_speed_radps: 0
                      idle_target_speed_radps: [1×1 class_REVS_dynamic_lookup]
                              idle_control_Kp: 25
                              idle_control_Ki: 65
                              idle_control_Kd: 1
                      idle_control_ramp_radps: 10.471975511966
                  idle_control_ramp_time_secs: 1.5
               idle_control_torque_reserve_Nm: 10
       idle_control_slow_est_time_constant_sec: 0.2
                          fuel_map_speed_radps: [18×1 double]
                            fuel_map_torque_Nm: [26×1 double]
                                  fuel_map_gps: [26×18 double]
                             deac_fuel_map_gps: [26×18 double]
                                 deac_strategy: [1×1 struct]
                            deac_num_cylinders: 0
             deac_transition_on_duration_secs: 0.99
            deac_transition_off_duration_secs: 0.11
          deac_transition_off_fuel_multiplier: [1 1]
   deac_transition_off_fuel_multiplier_time_secs: [0 0.1]
   deac_transition_off_fuel_multiplier_limit_gps: Inf
```

(continues on next page)

(continued from previous page)

```
                    fast_torque_fuel_adjust_norm: 0
                           DFCO_enable_condition: '@veh_spd_mps>5'
                          DFCO_min_duration_secs: 2.1
                          DFCO_refuel_multiplier: [1 1.3 1]
                DFCO_refuel_multiplier_time_secs: [0 0.1 1.1]
                DFCO_refuel_multiplier_limit_gps: Inf
                       transient_correction_mult: 1.4
                                            name: '2018 Toyota 2.5L A25A-FKS Engine␣
↪Tier 3 Fuel converted to 2.46L'
                                 source_filename: 'engine_2018_Toyota_A25AFKS_2L5_Tier3'
                                  matrix_vintage: present
                                         variant: 'basic engine'
                                 combustion_type: spark_ignition
                                  displacement_L: 2.46445235878698
                                         bore_mm: 87.2347659681488
                                       stroke_mm: 103.086569155504
                                    num_cylinders: 4
                                compression_ratio: 13
                                    configuration: []
                                     inertia_kgm2: 0.143041293924769
                                             fuel: [1×1 class_REVS_fuel]
                                        base_fuel: [0×0 class_REVS_fuel]
                         nominal_idle_speed_radps: 61.7846555205993
                                     max_torque_Nm: 247.20140331587
                       max_torque_min_speed_radps: 504.1
                       max_torque_max_speed_radps: 551
                       max_torque_avg_speed_radps: 526.233333333333
                                     min_torque_Nm: -50.7899054656206
                                       max_power_W: 149994.914232163
                        max_power_min_speed_radps: 660.6
                        max_power_max_speed_radps: 698.4
                        max_power_avg_speed_radps: 683.366666666667
                             max_test_speed_radps: 694.6
```

5.2.9 transmission

An instance of class_REVS_AT_transmission, class_REVS_CVT_transmission, class_REVS_DCT_transmission, class_REVS_AMT_transmission, etc, that supports the powertrain of the vehicle to be modeled. Below is an example for an automatic transmission.

```
>> transmission

 transmission =
  class_REVS_AT_transmission with properties:

                type: automatic
             variant: 'automatic transmission system'
      matrix_vintage: present
                name: 'transmission_6AT_FWD_midsize_car'
     rated_torque_Nm: 284.281613813251
                gear: [1×1 class_REVS_gearbox]
```

(continues on next page)

(continued from previous page)

```
    torque_converter: [1×1 class_REVS_torque_converter]
             control: [1×1 class_REVS_AT_control]
             thermal: [1×1 struct]
        pump_loss_Nm: [1×1 class_REVS_dynamic_lookup]
       gear_strategy: [1×1 class_REVS_ALPHAshift]
        tcc_strategy: [1×1 class_REVS_uber_dynamic_lockup]
```

5.2.10 vehicle

An instance of `class_REVS_vehicle` that defines vehicle properties such as roadload, test weight, axle definitions, etc. For more information on roadloads and test weight see *ALPHA Roadloads and Test Weight*.

```
>> vehicle

vehicle =
  class_REVS_vehicle with properties:

                          name: []
                         class: 'midsize_car'
                          fuel: [1×1 class_REVS_fuel]
                       variant: 'default vehicle'
             powertrain_variant: 'CVM / P0'
              driveline_variant: 'one axle drive'
               controls_variant: 'CVM control'
                powertrain_type: conventional
           delta_mass_static_kg: [1×1 class_REVS_dynamic_lookup]
          delta_mass_dynamic_kg: [1×1 class_REVS_dynamic_lookup]
              use_abc_roadload: 1
            coastdown_target_A_N: 133.44666
         coastdown_target_B_Npms: 0
        coastdown_target_C_Npms2: 0.445167735635058
                   dyno_set_A_N: []
                dyno_set_B_Npms: []
                dyno_set_C_Npms2: []
            coastdown_adjust_A_N: 0
         coastdown_adjust_B_Npms: 0
        coastdown_adjust_C_Npms2: 0
                 frontal_area_m2: 0
          aerodynamic_drag_coeff: 0
                     driveshaft: [1×1 class_REVS_driveshaft]
                  transfer_case: [1×1 class_REVS_transfer_case]
                     steer_axle: [1×1 class_REVS_drive_axle]
                    drive_axle1: [1×1 class_REVS_drive_axle]
                    drive_axle2: [1×1 class_REVS_drive_axle]
                  trailer_axle1: [1×1 class_REVS_drive_axle]
                  trailer_axle2: [1×1 class_REVS_drive_axle]
              max_brake_force_N: 26818.456903
          coastdown_target_A_lbf: 30
      coastdown_target_B_lbfpmph: 0
     coastdown_target_C_lbfpmph2: 0.02
                 dyno_set_A_lbf: []
```

(continues on next page)

(continued from previous page)

```
           dyno_set_B_lbfpmph: []
          dyno_set_C_lbfpmph2: []
       coastdown_adjust_A_lbf: 0
    coastdown_adjust_B_lbfpmph: 0
   coastdown_adjust_C_lbfpmph2: 0
                       ETW_kg: 1587.572
               mass_static_kg: 1563.75842
              mass_dynamic_kg: 1611.38558
                 mass_curb_kg: 1427.68082
              mass_ballast_kg: 136.0776
                      ETW_lbs: 3500
                 mass_curb_lbs: 3147.5
              mass_ballast_lbs: 300
               mass_static_lbs: 3447.5
              mass_dynamic_lbs: 3552.5
```

CHAPTER

SIX

MODEL OUTPUTS

When using the batch process, a standardized, customizable output file is created in the `output` folder. When running from a saved workspace, or running from a batch, outputs are always produced in the simulation workspace.

6.1 Workspace Outputs

From a batch, the simulation output workspace can be pulled up to the Matlab top-level workspace using the `extract_workspace` method of `class_REVS_sim_case`:

```
sim_batch.sim_case(sim_number).extract_workspace
```

where `sim_number` is a number >= 1 that represents the simulation to be investigated.

For the workspace to be extractable, it must be retained in memory by setting the `retain_output_workspace` property of the `class_REVS_sim_batch` to `true`. For more information see *Retain Workspaces in Memory*. See *Post-Simulation Data Analysis* and *Controlling Datalogging and Auditing* for more information on controlling and using workspace outputs.

There are four primary model output variables generated in the simulation workspace, `datalog`, `model_data`, `result` and `audit`.

6.1.1 The datalog Output

The `datalog` output variable contains continuous model outputs. It has hierarchical properties that somewhat mirror the model structure. The top level should look something like this:

```
datalog =
  class_REVS_datalog with properties:

     accessories: [1×1 class_REVS_logging_object]
        controls: [1×1 class_REVS_logging_object]
     drive_cycle: [1×1 class_REVS_logging_object]
          driver: [1×1 class_REVS_logging_object]
        electric: [1×1 class_REVS_logging_object]
          engine: [1×1 class_REVS_logging_object]
    transmission: [1×1 class_REVS_logging_object]
         vehicle: [1×1 class_REVS_logging_object]
            time: [137402×1 double]
```

For example, vehicle speed can be plotted versus time:

```
plot(datalog.time, datalog.vehicle.output_spd_mps);
```

`datalog` is constructed from a `class_REVS_logging_object` which calculates some values that have not been computed and output from the model itself. An example of this would be engine power, there is no logging block within the model for engine power, but if `datalog.engine.crankshaft_speed_radps` and `datalog.engine.crankshaft_torque_Nm` are logged a property will be generated for `datalog.engine.crankshaft_power_W` to compute the value when needed.

Controlling what signals are included in the datalog as well as the frequency of the output is discussed in *Controlling Datalogging and Auditing*.

6.1.2 The model_data Output

The datalog object is also associated with a `class_test_data` object called `model_data`. The primary difference between the two is that `model_data` represents a subset of the logged data and has a common, high-level namespace that can be used to compare model data with test data or data from multiple model runs or even data different models. For example, vehicle speed can be plotted versus time:

```
plot(model_data.time, model_data.vehicle.speed_mps);
```

6.1.3 The results Output

The `result` variable is a `class_REVS_result` object that contains summarized simulation results. The top level should look something like this:

```
result =

  class_REVS_result with properties:

         phase: [1×1 class_REVS_CVM_result]
      weighted: [1×4 class_REVS_CVM_result]
      map_fuel: [1×1 class_REVS_fuel]
    unadjusted: [1×1 class_REVS_result]
   performance: [1×1 class_REVS_logging_object]
   output_fuel: [1×1 class_REVS_fuel]
```

The `phase` property contains the result data from each simulation phase while `weighted` contains the data aggregated over the drive cycles. The intention of the `result` object is storing scalar values for each drive cycle phase and cycle. `class_REVS_result` contains methods to print the results to the console, an example of which is shown below:

```
>> result.print_weighted()
---############### Weighted Results ##############---
   EPA_FTP: --------------------------------
       Percent Time Missed by 2mph =   0.00 %
       Distance                    = 11.109 mi
       Fuel Consumption            = 1028.6233 grams
       Fuel Consumption            = 0.3590 gallons
       Fuel Economy (Volumetric)   = 30.399 mpg
       Fuel Economy (CAFE)         = 30.821 mpg
       Fuel Consumption            = 94.266 g/mile
       CO2 Emission                = 285.72 g/mile
```

(continues on next page)

(continued from previous page)

```
  EPA_HWFET: -------------------------------
      Percent Time Missed by 2mph =   0.00 %
      Distance                    = 10.269 mi
      Fuel Consumption            = 698.1742 grams
      Fuel Consumption            = 0.2436 gallons
      Fuel Economy (Volumetric)   = 42.146 mpg
      Fuel Economy (CAFE)         = 42.731 mpg
      Fuel Consumption            = 67.992 g/mile
      CO2 Emission                = 206.08 g/mile

  EPA_HWFET: -------------------------------
      Percent Time Missed by 2mph =   0.00 %
      Distance                    = 10.269 mi
      Fuel Consumption            = 698.1571 grams
      Fuel Consumption            = 0.2436 gallons
      Fuel Economy (Volumetric)   = 42.147 mpg
      Fuel Economy (CAFE)         = 42.732 mpg
      Fuel Consumption            = 67.990 g/mile
      CO2 Emission                = 206.08 g/mile

  REVS_Performance_cruise75mph: ------------
      Percent Time Missed by 2mph =  79.83 %
      Distance                    =  6.559 mi
      Fuel Consumption            = 2035.1302 grams
      Fuel Consumption            = 0.7102 gallons
      Fuel Economy (Volumetric)   =  9.235 mpg
      Fuel Economy (CAFE)         =  9.364 mpg
      Fuel Consumption            = 310.283 g/mile
      CO2 Emission                = 940.47 g/mile
```

The `result` object also contains other summary values from the model such as integrated fuel consumption or battery current and are controlled similarly to the `datalog` outputs, see *Controlling Datalogging and Auditing* for more details. An example of this is displaying transmission data for each phase is shown below:

```
>> result.phase.transmission

ans =

  class_REVS_logging_object with properties:

           output_pos_kJ: [2.9296e+03 2.7390e+03 2.9296e+03 7.2149e+03 7.2148e+03 1.
↪0335e+04 4.9111e+03 6.9897e+03]
          output_pos_kWh: [0.8138 0.7608 0.8138 2.0041 2.0041 2.8707 1.3642 1.9416]
          num_downshifts: [25 46 25 7 7 0 1 1]
              num_shifts: [51 92 51 15 15 5 5 4]
   num_target_downshifts: [25 46 25 7 7 0 5 1]
     num_target_upshifts: [26 46 26 8 8 5 4 3]
            num_upshifts: [26 46 26 8 8 5 4 3]
```

Note that as with `datalog` the `result` object is constructed from `class_REVS_logging_object`, so additional calculated properties are added based on what signals are logged directly in the model. This can be seen in the example above where `output_pos_kWh` is calculated from `output_pos_kJ`.

6.1.4 The audit Output

The `audit` structure, like the `result` structure, contains scalar values for each phase, or total simulation.

For example:

```
>> audit.total.engine

    ans =

        class_REVS_logging_object with properties:

             crankshaft_delta_KE_kJ: 0.3309
            crankshaft_delta_KE_kWh: 9.1911e-05
                  crankshaft_neg_kJ: 604.0453
                 crankshaft_neg_kWh: 0.1678
                  crankshaft_pos_kJ: 7.4220e+03
                 crankshaft_pos_kWh: 2.0617
                  crankshaft_tot_kJ: 6.8180e+03
                 crankshaft_tot_kWh: 1.8939
                    fuel_consumed_g: 703.2932
                       gross_neg_kJ: 450.6905
                      gross_neg_kWh: 0.1252
                       gross_pos_kJ: 8.0877e+03
                      gross_pos_kWh: 2.2466
                       gross_tot_kJ: 7.6371e+03
                      gross_tot_kWh: 2.1214

>> audit.phase.engine

    ans =

      class_REVS_logging_object with properties:

         crankshaft_delta_KE_kJ: [0.3321 -0.0017]
        crankshaft_delta_KE_kWh: [9.2236e-05 -4.6631e-07]
              crankshaft_neg_kJ: [250.3882 353.6571]
             crankshaft_neg_kWh: [0.0696 0.0982]
              crankshaft_pos_kJ: [3.6640e+03 3.7581e+03]
             crankshaft_pos_kWh: [1.0178 1.0439]
              crankshaft_tot_kJ: [3.4136e+03 3.4044e+03]
             crankshaft_tot_kWh: [0.9482 0.9457]
                fuel_consumed_g: [319.6850 383.6047]
                   gross_neg_kJ: [192.0876 258.6029]
                  gross_neg_kWh: [0.0534 0.0718]
                   gross_pos_kJ: [3.9019e+03 4.1858e+03]
                  gross_pos_kWh: [1.0839 1.1627]
                   gross_tot_kJ: [3.7098e+03 3.9272e+03]
                  gross_tot_kWh: [1.0305 1.0909]
```

It should be noted here that the total and phase audits may appear to have discrepancies. In other words, the sum of the phase audit results may not add up to the total result for the same variable, such as `fuel_consumed_g`. This is because the phase audit results are only for phase numbers greater than zero. In the case of a drive cycle where the engine start is not sampled (not part of the phase results), the first five seconds may be phase zero. Also, it takes a couple of simulation time steps at the end of the drive cycle to shut down the model, and those are also phase zero.

Enabling the audits populates the workspace with audit data, via the `class_REVS_audit` class. `class_REVS_audit` is also responsible for calling the report generators for each unique powertrain type, as follows:

- `class_REVS_CVM_audit` - calculates and reports energy balances for Conventional Vehicle Models
- `class_REVS_EVM_audit` - calculates and reports energy balances for Electric Vehicle Models
- `class_REVS_HVM_audit` - calculates and reports energy balances for Hybrid Vehicle Models

There is no automatic method for the Simulink model itself to comprehend the correct sources and sinks of energy within the model, this is determined by the creator of the model and is based on the underlying physics of the powertrain components.

The audit classes for the various powertrains inherit methods and properties from a base class, `class_REVS_VM_audit`, which handles audit calculations common to all powertrains, i.e. brakes, tires, roadload losses, etc.

The audit energy logs (as seen above) are tallied according to whether they are sources of energy or sinks of energy in the `calc_audit` methods of the audit classes. If the model, audit logging and audit calculations are correct then the sum of the energy in the audit sinks will equal the sum of the energy in the audit sources. The sources and sinks are tallied in the `energy_balance` property of the audit class.

```
>> audit.total.energy_balance

ans =

  struct with fields:

                     source: [1×1 struct]
                       sink: [1×1 struct]
        simulation_error_kJ: -0.5840
    energy_conservation_pct: 100.0157

>> audit.total.energy_balance.source

ans =

  struct with fields:

          KE_kJ: 0
    gradient_kJ: 0
          gross: [1×1 struct]
            net: [1×1 struct]

>> audit.total.energy_balance.sink

ans =

  struct with fields:

        KE_kJ: 0.4379
      vehicle: [1×1 struct]
    accessory: [1×1 struct]
     total_kJ: 3.7313e+03
```

The audit sources consist of `gross` and `net` categories, where `gross` refers to fuel chemical energy and energy stored in batteries, for example. `net` refers to energy used to power the vehicle and/or run electrical accessories, for example.

```
>> audit.total.energy_balance.source.gross

ans =

  struct with fields:

      fuel_kJ: 1.3157e+04
    stored_kJ: 8.0583
     total_kJ: 1.3165e+04

>> audit.total.energy_balance.source.net

ans =

  struct with fields:

                engine_kJ: 3.7237e+03
    engine_efficiency_pct: 28.3017
                stored_kJ: 7.0347
                 total_kJ: 3.7307e+03
```

The difference between the net source energy and the total sink energy is the simulation error, which should be very small and is recorded as the energy balance `energy_conservation_pct` where 100% is the desired value.

```
>> audit.total.energy_balance.source.net.total_kJ

ans =

   3.7307e+03

>> audit.total.energy_balance.sink.total_kJ

ans =

   3.7313e+03

>> audit.total.energy_balance.energy_conservation_pct

ans =

  100.0157
```

Typical sources of simulation error are clutch / driveline re-engagements where the small modeled disparity in speeds at lockup causes a small gain or loss of kinetic energy. If the audit is off by a larger amount then either there is a problem with the model or a problem with the audit itself. Most of the time the audit is incorrect when there's a discrepancy. For example, a new component may have been added to the model but the `calc_audit` function has not been updated to include the energy as a source or sink, or perhaps the audit datalog has been placed on the wrong signal line or at the incorrect point in the model. One technique for sorting out whether an error is a just a simulation error due to approximation (like the slightly mismatched speeds) or due to an actual or accounting error is to run the model at a finer timestep. Generally, simulation errors should decrease as the step size decreases and audit or accounting errors should remain unchanged.

When creating an audit for a new component it's very important to understand that the topology of the blocks in the model in most cases is not the same as the topology of the sources and sinks of energy in the model. It's tempting to

place an audit datalog at the inputs and outputs of the blocks in the model, but if the block is not properly a source or sink of energy then the audit will likely fail. For example, torques and speeds may pass through several Simulink blocks, but each block is not necessarily a "source" of energy for the next block downstream.

In any case, it's important to track down audit issues if and when they occur.

6.1.5 Logging Details

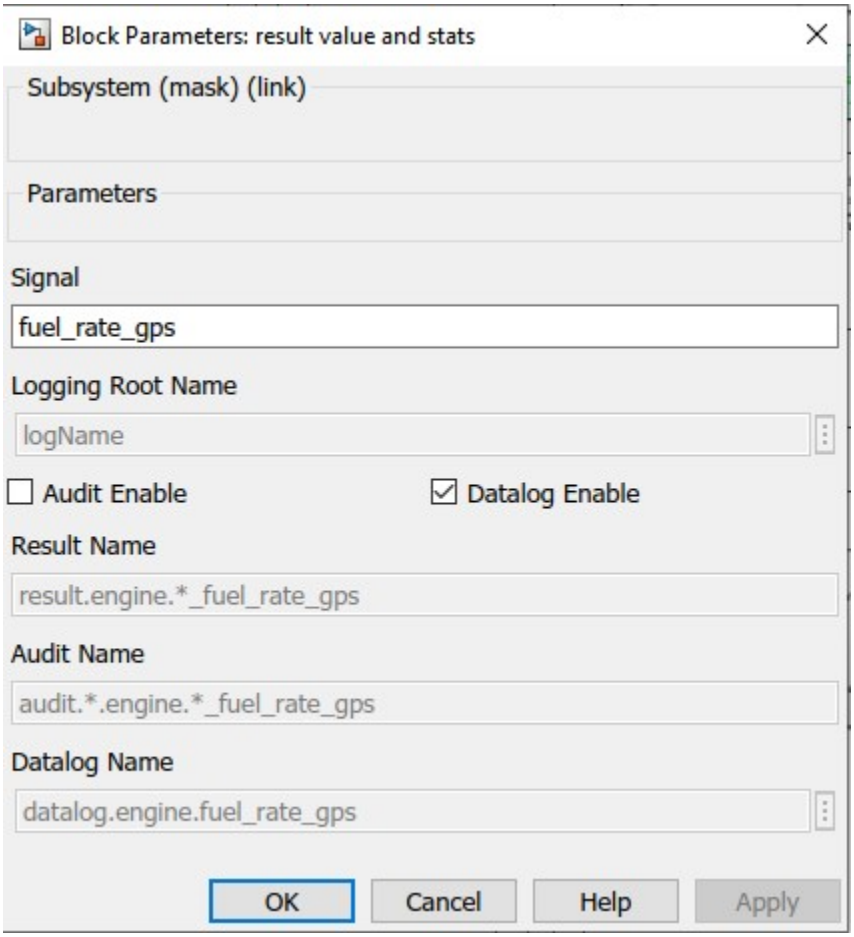
Since it's not possible for Simulink To Workspace blocks to directly create the output objects described above, there is a process for populating these data structures from individual logged workspace variables. This ia accomplished through employing a naming scheme for the logged signals that can then be loaded into the appropriate objects. For example, the raw post-simulation workspace will contain variables such as:

```
audit__accessories__air_conditioner__elec_neg_kJ
dl__engine__crankshaft_trq_Nm
rsltp__engine__fuel_consumed_g
```

The prefix determines the top-level data structure. `audit` maps to the `audit` data structure, `dl` maps to `datalog` and `rsltp` maps to the `phase` property of the `result` data structure, as in `result.phase`.

The double underscores, `__`, define the hierarchical structure. For example, `audit__accessories__air_conditioner__elec_neg_kJ` will become `audit.accessories.air_conditioner.elec_neg_kJ` in the final workspace. Single underscores are taken as part of the property name.

The construction of the raw workspace variable names is handled by the mask of the datalog blocks and can determined by the structure of the model. For example, datalogs in the `engine` block model will automatically be placed in the `datalog.engine` structure without having to be explicitly named as such. For example, the `datalog.engine.fuel_rate_gps` signal is set up as follows:



The only user-specified part of the name is `fuel_rate_gps`, the rest is automatic, and the final result is previewed in the `Datalog Name` text box.

6.2 File Outputs

By default, when a batch file runs, it produces several files in the simulation `output` folder.

The primary output file is the results file. The filename format is `YYYY_MM_DD_hh_mm_ss_BATCHNAME_results.csv` where `Y/M/D` represent the year, month and day, and `h/m/s` are hour, minute, and seconds respectively.

If `sim_batch.verbose` is > 0 then console outputs will also be produced in the `output` folder. The filename format is `YYYY_MM_DD_hh_mm_ss_BATCHNAME_N_console.txt`, as above, where `N` is the simulation number. The console outputs will include basic information on the drive cycle results as well as audit results if they are enabled. For more information on auditing, see *Auditing*.

The basic console outputs for a drive cycle phase look like:

```
1: ------------------------
Percent Time Missed by 2mph =   0.00 %
Distance                    =  3.592 mi
Fuel Consumption            = 320.5339 grams
Fuel Consumption            = 0.1119 gallons
Fuel Economy (Volumetric)   = 32.111 mpg
```

(continues on next page)

(continued from previous page)

```
Fuel Economy (CAFE)          = 32.557 mpg
Fuel Consumption             = 89.240 g/mile
CO2 Emission                 = 270.49 g/mile
```

Where the "1:" represents the drive cycle phase, which in this case is named "1".

SAEJ2951 drive quality metrics are available in `result.phase.drive_quality` as produced by the `REVS_SAEJ2951` function. See also https://www.sae.org/standards/content/j2951_201111/.

6.2.1 Post Processing Output File Scripts

The results output file is created within the `postprocess_sim_case` method of `class_REVS_sim_batch`. At this time there are three output scripts, depending on the type of vehicle powertrain: `REVS_setup_data_columns_CVM`, `REVS_setup_data_columns_HVM`, and `REVS_setup_data_columns_EVM` that are located in `REVS_Common\helper_scripts`. These output scripts call various sub-scripts for various output file column groups. For example, `REVS_setup_data_columns_CVM`:

```
%% define standard CVM output columns

REVS_setup_data_columns_VM;

REVS_setup_data_columns_transmission;

REVS_setup_data_columns_engine;

REVS_setup_data_columns_MPG;

REVS_setup_data_columns_vehicle_performance;

REVS_setup_data_columns_audit;

REVS_setup_data_columns_battery;

REVS_setup_data_columns_driveline_stats;
```

These scripts populate a variable called `data_columns`, a vector of `class_data_column` objects. Data column objects define the name and format of each output column. An example instance of `class_data_column`.

```
>> class_data_column({'Test Weight lbs','lbs'},'%f','vehicle.ETW_lbs',2)

ans =

  class_data_column with properties:

    header_cell_str: {'Test Weight lbs'  'lbs'}
         format_str: '%f'
           eval_str: 'vehicle.ETW_lbs'
            verbose: 2:
```

`class_data_column` objects have the following properties:

- `header_cell_str`, a cell array of strings. The first string is the column name, located in the first row of the output file. The second string is an optional string meant to represent the units of the variable or a supporting

description of the variable and occupies the second row of the output file.

- `format_str`, a standard Matlab `fprintf` `formatSpec` string.
- `eval_str` is a string that gets evaluated by the Matlab `evalin` function and should return a numeric or string value that can be printed. Any variable available in the simulation output workspace can be referenced in the `eval_str`.
- `verbose` is a numeric value that refers to the `class_REVS_sim_batch` `output_verbose` property. Output columns will be produced for columns where `verbose` is >= `output_verbose`. In this way the output file size and complexity can be controlled. The value of `verbose` is `0` unless overridden during the definition, as it was above. Columns with a `verbose` of `0` will always be output.

The `data_columns` vector is created by `REVS_setup_data_columns_VM` and appended with each data column object, as shown below:

```
data_columns(end+1) = class_data_column({'Test Weight lbs','lbs'},'%f','vehicle.ETW_lbs',
↪2);
```

The data_columns are evaluated one at a time by the `class_REVS_sim_batch` `postprocess_sim_case` method via the `write_column_row` function which is located in the `NVFEL_MATLAB_Tools\utilities\export` folder.

6.2.2 Custom Output Summary File Formats

There are at least a couple methods to modify the output file format: edit the various `setup_data_columns` scripts, or populate the `class_REVS_sim_batch` `setup_data_columns` property with the name of a custom output column definition script, which can be created using the default scripts as a guide. The custom script will be called after the default columns are created and therefore the custom columns will appear to the right of the previously defined columns in the output file.

For example, in the batch script:

```
sim_batch.setup_data_columns = 'setup_custom_data_columns';
```

In `setup_custom_data_columns.m`:

```
% setup up custom data columns

data_columns(end+1) = class_data_column({' ',' '}, separator, '0');
data_columns(end+1) = class_data_column({'ETW_HP','LB/HP'}, '%.3f', 'sim_config.pounds_
↪per_hp', 1);
data_columns(end+1) = class_data_column({'RLHP20','HP/LB'}, '%.3f', 'sim_config.roadload_
↪hp20plb', 1);
data_columns(end+1) = class_data_column({'RLHP60','HP/LB'}, '%.3f', 'sim_config.roadload_
↪hp60plb', 1);
data_columns(end+1) = class_data_column({'HP_ETW','HP/LB'}, '%.3f', '1/sim_config.pounds_
↪per_hp', 1);
data_columns(end+1) = class_data_column({'ETW','lbs'},'%f','vehicle.ETW_lbs',2);
```

CHAPTER

SEVEN

ALPHA DEVELOPMENT

This chapter will give some information on ALPHA development guidelines and more details on the Simulink model itself and review some of the critical data structures.

7.1 Conventions and Guidelines

There are a few guidelines that cover the use of variable names within the modeling environment and other conventions. Understanding and following the guidelines facilitates collaboration, ease of use and understanding of the modeling environment.

- Class definitions start with `class_`.
- Enumerated datatype definitions start with `enum_`.
- Physical unit conversions should be accomplished using the `unit_convert` class. For example `engine_max_torque_ftlbs = engine_max_torque_Nm * unit_convert.Nm2ftlbs`. Avoid hard-coded conversion constants.
- Any variable that has corresponding units should take the form `variable_units`, such as `vehicle_speed_kmh` or `shaft_torque_Nm`. SI units are preferred whenever possible unless superseded by convention (such as roadload ABCs). Units commonly use lowercase 'p' for 'per'. For example mps = meters per second, radps = radians per second. Readability outweighs consistency if convention and context allows, for example `vehicle_speed_mph` is understood to be vehicle speed in miles per hour, not meters per hour.
- English units are used by a class, but that class should also provide SI equivalents. REVS provides some framework and examples of automatic unit conversions that may be used.
- Variable names should be concise but abbreviations or acronyms are generally to be avoided unless superseded by convention. For example, `datalog.transmission.gearbox`, not `dl.trns.gbx`. Exceptions are bus signal names and the port names on Simulink blocks (long names reduce readability rather than enhancing it) - for example, torque may be trq and speed may be spd. Simulink block names may also receive abbreviated names to enhance readability.
- Underscores are preferred for workspace and data structure variable names, for example `selected_gear_num`. Camelcase is preferred for variables defined in Simulink masks and local block workspaces so they may be distinguished from ordinary workspace variables.
- Most functions that are specific to the REVS modeling platform start with `REVS_`.
- The 'goto' and 'from' flags are to be avoided in Simulink blocks as they significantly decrease the readability and understanding of block connections. Exceptions to this rule are the `REVS_VM top-level system_bus` component sub-buses, the `global_stop_flag` and the `REVS_audit_phase_flag` which must be made available throughout the model.

- Trivial Simulink blocks (such as multiplication, addition, etc) may have their block names hidden to enhance readability; non-trivial blocks should have names which concisely and accurately describe their function.
- Simulink blocks should have a white background and a black foreground. Exceptions are red foreground for blocks that are deprecated or orange foreground for blocks that may be unproven or experimental.
- Useful Simulink blocks should be added to the appropriate `REVS_Common` model library if they are likely to be reused.
- Simulink block names are lowercase unless superseded by convention and words are separated by spaces (as opposed to underscores).
- Simulink blocks that take in the system bus should have `system_bus` as input port 1.
- Simulink blocks that produce a signal bus should have `bus_out` as output port 1.
- Whenever possible, variant subsystem blocks should be controlled by a `variant` string property that matches the name of the block to be selected.

7.2 REVS_VM

This section will provide an overview of the Simulink model, `REVS_VM`. ALPHA represents various vehicle powertrains through the use of variant subsystems which are instantiated by the top-level model rather than by using separate models.

7.2.1 Overview

The top-level of `REVS_VM` consists of the following blocks:

- `ambient` - provides the ambient test conditions, logs the time signal and provides the drive cycle road grade as a function of distance travelled.
- `driver` - implements the trace-following driver model that produces the accelerator / brake pedal signals and other driver-related signals to the rest of the model. `driver` also contains the drive cycle lookups for target vehicle speed.
- `powertrain` - implements the various powertrains for conventional, hybrid or electric vehicles.
- `vehicle` - contains the vehicle roadload calculations (except for tire rolling resistance and losses, which are handled in the `powertrain` subsystems) and the vehicle speed integrator.

Each of the top-level blocks can be customized by the variant control properties `ambient.variant`, `driver.variant`, `vehicle.variant` and `vehicle.powertrain_variant`. These are string properties that contain the name of the desired variant subsystem to instantiate.

Also at the top level of the model is the system bus and the vehicle speed chart which shows target and achieved vehicle speeds.

7.2.2 Powertrain Variants

The `powertrain` variant subsystem is in many ways the heart of the ALPHA model. At this time there are two available powertrain variants:

- `CVM / P0` - implements CVM, the Conventional Vehicle Model and mild hybrid vehicles (BAS, Belt-Alternator-Starter and ISG, Integrated-Starter-Generator)
- `EVM` - implements EVM, the Electric Vehicle Model
- `PS Hybrid` - implements a Prius-type "powersplit" strong hybrid
- `P2 Hybrid` - implements a "P2" (drive motor upstream of transmission) strong hybrid

CVM / P0

The top level of the conventional vehicle model contains the following blocks:

- `controls` - handles engine start-stop logic and other control system algorithms. This variant subsystem is determined by the `vehicle.controls_variant` string property.
- `engine` - contains the engine model. This variant subsystem is determined by the `engine.variant` string property.
- `transmission` - contains the transmission model. This variant subsystem is determined by the `transmission.variant` string property.
- `driveline` - contains the axle models which contain the wheels, tires, final drive and driveshafts, etc. This variant subsystem is determined by the `vehicle.driveline_variant` string property.
- `electric` - implements the vehicle's electrical energy storage system. The engine starting and battery charging system is also implemented here. This block is not itself a variant subsystem but the `starting / charging` and `energy storage` subsystems are variants, determined by the `vehicle.powertrain_type` enumeration property.
- `mech & elec & accessories` - implements the vehicle's electrical accessories and mechanical engine accessory loads.

EVM

The top level of the electric vehicle model contains the following blocks:

- `controls` - handles control system algorithms such as acceleration and regeneration limits. This variant subsystem is determined by the `vehicle.controls_variant` string property.
- `drive_motor` - implements a single propulsion motor-generator model.
- `transmission` - contains the transmission model. This variant subsystem is determined by the `transmission.variant` string property.
- `driveline` - contains the axle models which contain the wheels, tires, final drive and driveshafts, etc. This variant subsystem is determined by the `vehicle.driveline_variant` string property.
- `xEV energy storage` - implements the vehicle's electrical energy storage system.
- `mech & elec & accessories` - implements the vehicle's electrical accessories and mechanical engine accessory loads.

PS Hybrid

The top level of the powersplit hybrid vehicle model contains the following blocks:

- `controls` - handles engine start-stop logic and other control system algorithms. This variant subsystem is determined by the `vehicle.controls_variant` string property.
- `engine` - contains the engine model. This variant subsystem is determined by the `engine.variant` string property.
- `hybrid transmission` - contains the planetary transmission model and "MG1" and "MG2" motor/generators.
- `driveline` - contains the axle models which contain the wheels, tires, final drive and driveshafts, etc. This variant subsystem is determined by the `vehicle.driveline_variant` string property.
- `electric` - implements the vehicle's electrical energy storage system. The engine starting and battery charging system is also implemented here. This block is not itself a variant subsystem but the `starting / charging` and `energy storage` subsystems are variants, determined by the `vehicle.powertrain_type` enumeration property.
- `mech & elec & accessories` - implements the vehicle's electrical accessories and mechanical engine accessory loads.

PS Hybrid

The top level of the powersplit hybrid vehicle model contains the following blocks:

- `controls` - handles engine start-stop logic and other control system algorithms. This variant subsystem is determined by the `vehicle.controls_variant` string property.
- `engine` - contains the engine model. This variant subsystem is determined by the `engine.variant` string property.
- `p2 hybrid transmission` - contains the transmission model, including the P2 motor and engagement clutch.
- `driveline` - contains the axle models which contain the wheels, tires, final drive and driveshafts, etc. This variant subsystem is determined by the `vehicle.driveline_variant` string property.
- `electric` - implements the vehicle's electrical energy storage system. The engine starting and battery charging system is also implemented here. This block is not itself a variant subsystem but the `starting / charging` and `energy storage` subsystems are variants, determined by the `vehicle.powertrain_type` enumeration property.
- `mech & elec & accessories` - implements the vehicle's electrical accessories and mechanical engine accessory loads.

7.3 Understanding the Simulink Libraries

This section provides an overview of the several Simulink libraries that hold the various component models and subsystem blocks.

7.3.1 accessory_lib

Contains blocks for describing mechanical and electrical accessory loads.

7.3.2 ambient_lib

The `ambient` variant block is the source of road grade (as a function of distance) and ambient temperature. The time `datalog` is also created here. Alternative ambient blocks can be created and selected using the `ambient.version` property

7.3.3 controls_lib

Contains the `controls` variant block and other controls-related blocks. The control blocks determine engine start-stop and control strategies for hybrid vehicles.

7.3.4 driver_lib

Contains the `driver` variant block, which determines the closed-loop drive cycle follower. The `driver` block produces the accelerator and brake pedal signals to the rest of the model as well as a few other signals such as the drive cycle speed, phase, and position in seconds. Alternative driver blocks can be created and selected using the `driver.version` property

7.3.5 electric_lib

Contains energy storage (battery) models and other electrical components such as starter, alternator, and e-machine (motor-generator) models.

7.3.6 engine_lib

Contains the `engine` variant block and engine and engine-related models, such as cylinder deactivation logic.

7.3.7 general_lib

Contains various utility blocks that may be used throughout the model, such as dynamic lookup tables, dynamic equations and other handy functions.

7.3.8 logging_lib

Contains the blocks that handle dynamic data logging within the model, including `audit` logging and drive cycle phase `result` values.

7.3.9 powertrain_lib

Contains the top-level `powertrain` variant block, and defines the available powertrains for conventional and hybrid vehicles.

7.3.10 transmission_lib

Contains transmission models for conventional and hybrid vehicles, and component models for things like clutches and torque converters.

7.3.11 vehicle_lib

Contains models of brakes, tires and other driveline components like axles, as well as the vehicle roadload calculations.

CHAPTER

EIGHT

CONTACT INFORMATION

8.1 ALPHA Technical Issues

Kevin Newman
United States Environmental Protection Agency
National Vehicle Fuel Emissions Laboratory (NVFEL)
2565 Plymouth Road
Ann Arbor, Michigan 48105
newman.kevin@epa.gov

Paul DeKraker
United States Environmental Protection Agency
National Vehicle Fuel Emissions Laboratory (NVFEL)
2565 Plymouth Road
Ann Arbor, Michigan 48105
dekraker.paul@epa.gov

8.2 ALPHA Rulemaking Usage

Jeff Cherry
cherry.jeff@epa.gov

CHAPTER

NINE

AGENCY INFORMATION

United States Environmental Protection Agency
National Vehicle Fuel Emissions Laboratory (NVFEL)
2565 Plymouth Road
Ann Arbor, Michigan 48105
Tel 734-214-4200
www.epa.gov

CHAPTER

TEN

PYTHON CODE

CHAPTER

ELEVEN

MATLAB CODE

model.**napolean_sample**(*first_param*, *second_param*, *varargin*)

NAPOLEAN_SAMPLE generates sample documentation in "Google Style"

For more information, see www.epa.gov

Parameters

- **first_param** (int) – the first parameter
- **second_param** (int) – the second parameter
- **varargin** (optional keyword and name-value arguments) –
  - **'foo':**
    turn foo on
  - **'bar', numeric:**
    set bar to the provided numeric value

Returns

Returns true

Attention: Rule number one: pay attention!!

Caution: Use caution, handle with care

Warning: This is a warning to you

Danger: Danger Will Robinson

Note: This is a note

Tip: Always leave 20%

Hint: Take a hint

Important: This is important!

See also:

legend

Example

This is an example code block:

```
foo = napolean_sample(1, 2, 'buckle_my_shoe');
```

CHAPTER

TWELVE

CODE INDEX

- py-modindex
- mat-modindex
- search

MATLAB MODULE INDEX

INDEX

M

N