# MATLAB Battery Facility Correlation Analysis Package

When battery performance data, on a single design exposed to identical test protocols and conditions, are available from several different measuring facilities, an inevitable question arises: how much of the observed variation in the data is due to natural variation in the battery performance characteristics versus differences in instrumentation and practices at the various facilities? For example, test laboratories may employ various testers, possess dissimilar temperature control capabilities and so on. These small effects may sum to produce significant differences but, in truth, are just artifacts of the measurement variation observed among the test laboratories employed. In the extreme case, different conclusions may be drawn from ostensibly identical test data taken at multiple facilities on the same cells. Further, in large collaborative studies, testing may be split across several laboratories each utilising the same test protocol. In this scenario how can we be sure the same measurements from multiple laboratories are equivalent? Equally, given there is evidence facilities are not equal, is there a means to correct data from outlying laboratories until the source(s) of the observed variation can be identified and subsequently resolved?

For example, the Faraday Institute Multi-Scale Modelling (MSM) consortium [[[1]](#endnote-1)] is one such large study whose ultimate objective is to provide accurate simulations to battery suppliers to assist in the design of advanced batteries, without incurring the costs of creating many expensive prototypes to test each possibility. To validate the efficacy of models developed by in the MSM study, various confirmatory experiments are being conducted across several university test laboratories by consortium members. The success of the simulation effort is proven through validation testing and hence it is important to confirm the equivalence of data from any of the test laboratories involved.

To this end, a MATLAB analysis package has been designed to:

* Implement the proper statistical design of experiment (*DoE*) procedures. A design of experiment tool included within the package automatically generates an appropriate design for an arbitrary number of test facilities and other experimental factors.
* Provide data import and pre-processing capabilities. Since no data channel naming and unit standard exists among facilities, this tool converts data channels and units to a consistent form employed by the analysis tool. In addition, the tool exports the formatted data to an Excel file assumed as the input for the subsequent analysis.
* Provides an inferential mechanism.
  + Creates and automatically identifies a suitable statistical model, consistent with the design and data collection procedure.
  + Supports, relevant hypothesis testing procedures. ***This provides a formal statistical test procedure for assessing whether the same data from different facilities is ostensibly the same.***

# Table of Contents

[MATLAB Battery Facility Correlation Analysis Package 1](#_Toc99618704)

[Table of Contents 1](#_Toc99618705)

[1 Software Package Architectures and Relevant Object Orientated Design Patterns 2](#_Toc99618706)

[1.1 Abstract Interfaces 4](#_Toc99618707)

[1.2 Inheritance 4](#_Toc99618708)

[1.3 Composition and Aggregation 4](#_Toc99618709)

[1.4 Relevant Object-Oriented Behavioural Patterns 5](#_Toc99618710)

[1.4.1 The Template Method 5](#_Toc99618711)

[1.4.2 The Strategy Pattern 6](#_Toc99618712)

[2 Installation Instructions 7](#_Toc99618713)

[2.1 Installing the data importer package 7](#_Toc99618714)

[2.2 Installing the analysis package 8](#_Toc99618715)

[3 Data Import Package (*correlationDataImporter*) User Notes 8](#_Toc99618716)

[3.1 Common concrete parent class properties and methods 9](#_Toc99618717)

[3.1.1 The *channelPresent* Method 10](#_Toc99618718)

[3.1.2 The *extractData* method 10](#_Toc99618719)

[3.1.3 The *export2excel* method 11](#_Toc99618720)

[3.2 Data import example 11](#_Toc99618721)

[3.3 Calculations 13](#_Toc99618722)

[3.3.1 The pulse test (internal resistance) data 13](#_Toc99618723)

[3.3.2 The rate test (discharge capacity) data 15](#_Toc99618724)

[3.4 Export data formats 16](#_Toc99618725)

[3.4.1 Writing a new data import class 17](#_Toc99618726)

[3.4.2 The Discharge Capacity (Rate) Test Data Format 17](#_Toc99618727)

[3.4.3 The Internal Resistance (Pulse) Test Data Format 18](#_Toc99618728)

[4 The facility correlation analysis package 18](#_Toc99618729)

[5 References 19](#_Toc99618730)

# Software Package Architectures and Relevant Object Orientated Design Patterns

The overall software tool is comprised of two packages:

1. A data import tool. This converts all data from each institution to a common format including signal names and units. All data can be aggregated into an Excel workbook ready for subsequent analysis. It is important to understand the common data format as a separate data tool must be written for each new facility.
2. The analysis tools. These perform the necessary DoE, identify an appropriate statistical model and permit the necessary hypotheses tests to be conducted.

Figure 1 presents the architecture for the data import package. This is based on the so-called strategy pattern and is designed to provide a common user interface, regardless of test facility, and to be easily extendable to incorporate data from new facilities added to an existing study. The architecture is common to both the discharge capacity (“rate”) and the internal resistance (“pulse”) test data. The abstract parent class provides the common user interface, but also includes some concrete properties and methods, such as a common data export method to an Excel workbook.

Figure 2 presents the software architecture employed for the analysis package. Data is imported in the common format, naming and signal convention imposed by the data import package. Note, the naming and unit convention is hard coded in the analysis software. Regardless, both packages employ several modern software design concepts which are discussed in the following subsections.

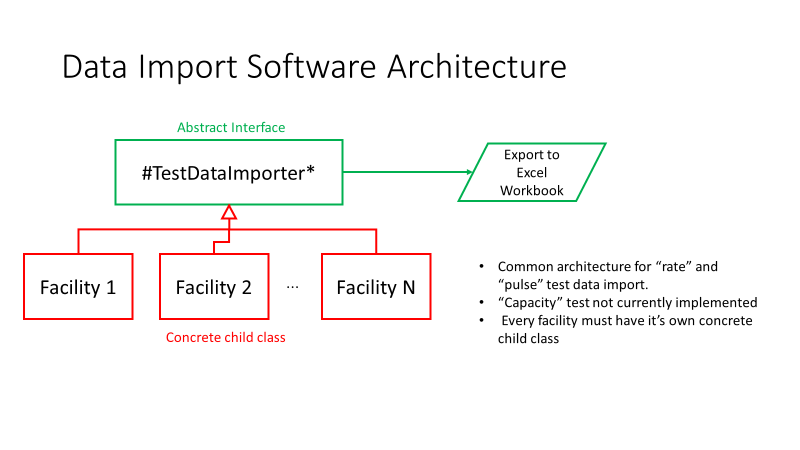


Figure 1: Correlation data import package architecture.

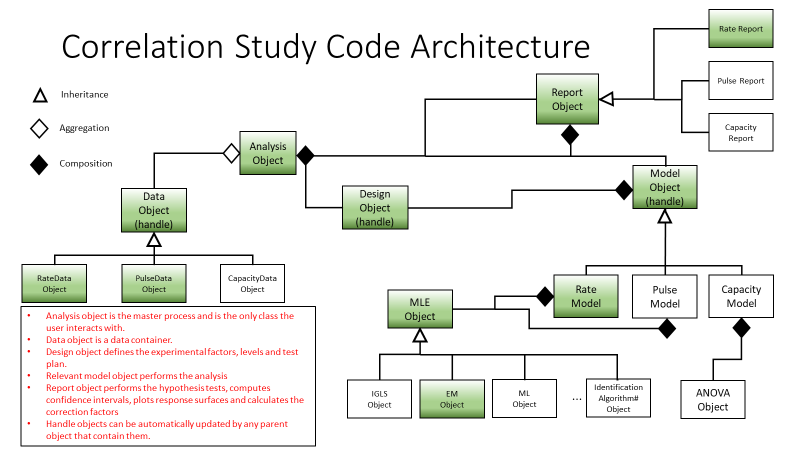


Figure : Facility Correlation Analysis Package Code Architecture

## Abstract Interfaces

Abstract classes are extremely useful for describing functionality common to a group of subclasses but requires unique implementations within each subclass. An abstract class cannot be *instantiated*, *i.e.* you cannot create an abstract class object in the workspace. Instead, an abstract class defines the components used by its subclasses. The terminology *abstract member* is used to refer to properties or methods declared in the abstract parent but implemented in a child subclass.

In contrast, a *concrete class* can be instantiated. A concrete class has no abstract members. The terminology *concrete member* is applied to properties or methods fully implemented within a class. Note an abstract class may contain concrete as well as abstract members. In this scenario, the concrete elements realised in the abstract class would be required by all subclasses. An abstract class predominantly forms an *interface*, describing functionality common to a group of subclasses. The abstract class defines the interface of every subclass without specifying the concrete implementation, which is contained in the subclass. Any concrete subclass must implement all inherited abstract members to become a concrete class. Note, methods with *private* access cannot be abstract as they cannot be inherited by the child subclass.

## Inheritance

Inheritance is one of the key aspects in object-orientated programming for building upon existing classes, and so in reusing code. As the name implies, with inheritance the child class has access to all the parent properties in methods. In contrast the parent has no access of the child properties or methods. Thus, instead of writing the same code multiple times for several similar applications, the child can simply inherit the attributes of the parent, which can now be reused many times.

Inheritance supports the implementation of an “***is-a***” model. For example, a *car*, *bus* or *motor* *bike* are all types of a broader *vehicle* category – a bus ***is a*** vehicle, a car ***is a*** vehicle and so on. Here the vehicle category is analogous to the parent class. Conversely, car, bus and bike correspond to specific child objects. In this case, the parent might contain general information such as the vehicle registration number or manufacturer name. All vehicles have a registration number. However, the maximum number of passengers or whether the vehicle is a hybrid, fully electric and so on is specific to the child.

## Composition and Aggregation

Composition is another of the fundamental concepts in object-oriented programming. It describes a class that references one or more objects of other classes in instance variables. Essentially an object of another class (child) is stored as a property of the parent. This allows you to model a “***has-a***” association between objects. Such relationships occur quite naturally in the real world. For example, a car, has an engine and modern coffee machines may have an integrated grinder and a brewing unit.

Given its broad use in the real world, it is not surprising that composition is routinely used in carefully designed software components. The advantages of composition are:

1. Code re-use. The child class requires no modification.
2. Implementing clean interfaces.
3. Change the implementation of a class used in a composition without adapting any external clients

Figure 2 illustrates composition schematically. In the diagram, both child classes are available as properties to the parent. The solid blue diamond is the symbol denoting composite classes. A very similar concept is aggregation. Again, with aggregation the child class is stored as a property of the parent. The difference between composition and aggregation occurs when the parent object is deleted. With composited classes, destruction of the parent object leads to destruction of the child. Conversely, with aggregation when the parent object is deleted the child object persists. Aggregation makes sense with data objects for example, where it is necessary to preserve the child for subsequent use with alternative analysis packages.

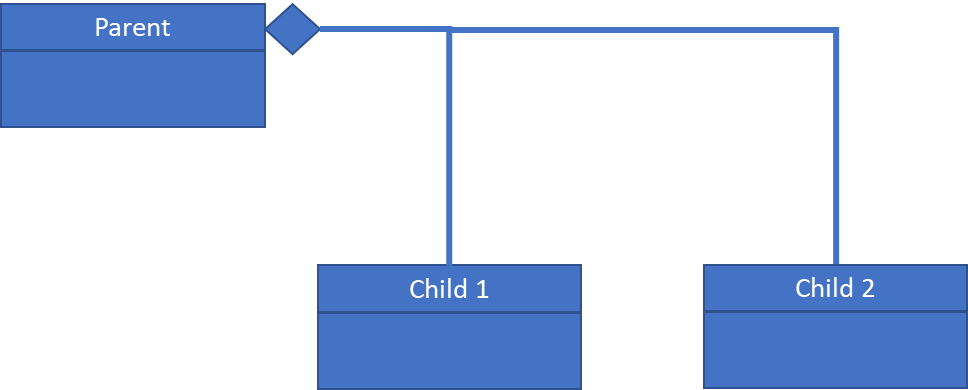


Figure 3: Class diagram depicting composition. Both child classes are stored as properties of the parent. The solid blue diamond is the symbol for composition.

## Relevant Object-Oriented Behavioural Patterns

The architecture makes use of two object-oriented programming (OOP) behavioural patterns: the *template method* and the *strategy pattern*. The template algorithm is useful for encoding algorithms, while the strategy pattern permits one of several analyses, selected at runtime, to be accessed through a common interface. These are discussed in detail in the following sections:

### The Template Method

The concept behind the template method is to define the skeleton of an algorithm, as an abstract class, deferring the specific details of the implementation to subclasses. This permits the subclasses to redefine certain steps of the algorithm without changing the algorithm’s structure. Figure 3 provides a schematic of the typical implementation of the template method. The template is implemented in an abstract class and defines the order in which the individual steps of the *algorithm* are implemented. However, none of the detailed specifics are defined. Rather, these are implemented as methods in a concrete class, which inherits from its abstract parent.

An example might be an optimisation algorithm. The template would define the order in which the specific operations associated with the optimisation process would be carried out. Perhaps, step 1 would be to evaluate the cost function for the current decision variable estimates. Step 2 might calculate the gradients of the cost function with respect to the decision variables are calculated, step 3 calculate the Hessian, step 4 determine the slack in the constraints and so on.

The child (concrete) class does not define the algorithm, rather it provides the implementation details. For example, it would provide methods for calculating the appropriate cost function, the necessary first and second derivative vector and matrices and the appropriate constraints. Whatever implementation details are specified by the template. Now consider we write two concrete classes, both inheriting from the same abstract parent and therefore possessing the same template method.

In the first implementation, the analyst defines a method to calculate both the necessary first and second derivatives of the cost function with respect to the decision variables numerically, using a quadrature method for example. Conversely, in the second implementation, the analyst implements analytical formulae for the required derivative information for the same cost function. The parent template method still calls the appropriate methods as defined in the order specified but has absolutely no knowledge of how each step is performed by either of the child classes. All that is required is the abstract interface defined by the template method, defined in the abstract parent, is respected by both child classes. In practice this means the method signature, as defined by input and output arguments, is the same.

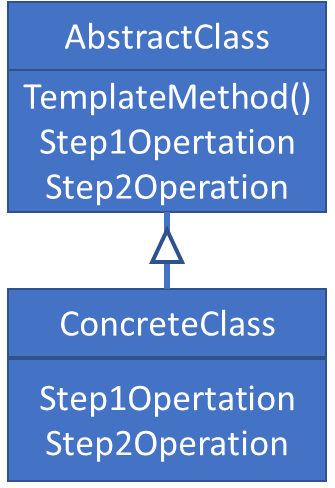


Figure 4: Template Method Class Architecture

### The Strategy Pattern

Often, an analyst may have a range of algorithmic options available to them to solve a specific problem. However, which one to use can only be determined at run time. The *strategy* is an OOP behavioural pattern that enables runtime selection of an algorithm. Thus, instead of implementing a single algorithm directly, code receives run-time instructions as to which, in a family of related algorithms, to use. This permits the algorithm implemented to vary independently from clients that use it.

Figure 4 is a schematic depicting the implementation of the *strategy* pattern. Note the *context* class, or *client*, does not implement the strategy directly. Instead, it refers to the strategy interface (***StrategyAbstract***). This makes the context independent of how the algorithm is implemented. The respective concrete strategy classes implement the desired algorithm. The inclusion of new algorithms is very straightforward as it simply requires a new concrete strategy class, consistent with the abstract strategy interface.

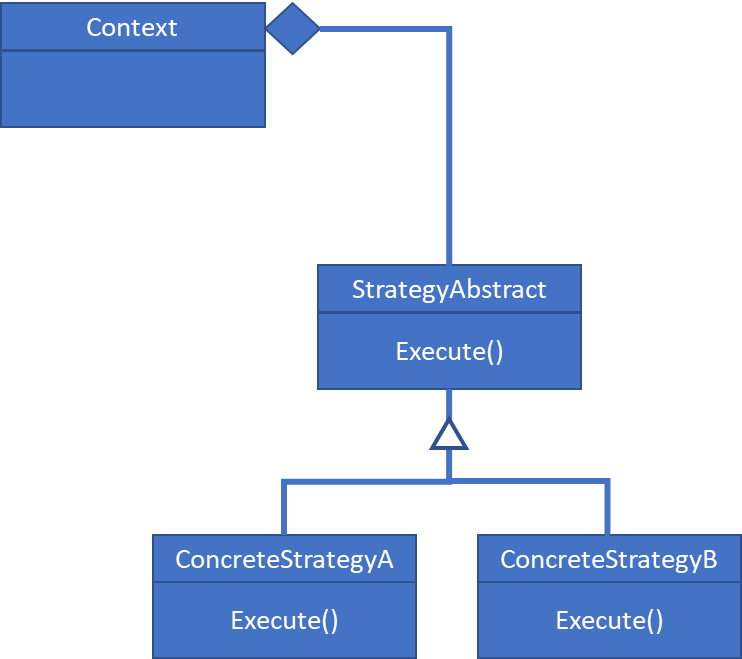


Figure 5: Schematic for the strategy pattern. The desired algorithm is selected at run-time from a family of related algorithms.

# Installation Instructions

The complete software suite is comprised of two packages:

* The data importer package.
* The correlation analysis package.

## Installing the data importer package

The data import package can be obtained from GITHUB, in the following project:

<https://github.com/MarkCaryLboro/CorrelationDataImporter>

1. Create a directory called ***CorrelationDataImporter*** in any directory on the current MATLAB path.
2. Copy the file ***CorrelationDataImporter.zip*** to the ***CorrelationDataImporter*** directory created in step 1. Unzip the contents to this directory.
3. At the command line type ***which lancasterRateTestData.*** This should generate a message similar to: ***C:\Users\...\CLASSES\CorrelationDataImporter\lancasterRateTestData.m.***

## Installing the analysis package

The data import package can be obtained from GITHUB, in the following project:

<https://github.com/MarkCaryLboro/CorrelationAnalysis>

1. Create a directory called ***CorrelationAnalysis*** in any directory on the current MATLAB path.
2. Copy the file ***CorrelationAnalysis.zip*** to the ***CorrelationAnalysis*** directory created in step 1. Unzip the contents to this directory.
3. At the command line type ***which rateTestAnalysis.*** This should generate a message similar to: ***C:\Users\...\GitHub\CorrelationAnalysis\rateTestAnalysis.m.***

# Data Import Package (*correlationDataImporter*) User Notes

The objectives of the ***correlationDataImporter*** package are:

* To import raw data from a given facility and convert channel and unit data to a common format.
* To provide relevant data pre-processing operations such as calculating internal resistances or discharge or charge capacities and so on. This ensures calculations are implemented in a consistent manner among facilities.
* To export the pre-processed and reformatted data to a single file to seed the subsequent analysis.

The data import package consists of a family of classes indexed by facility and test type. For example, the ***birminghamPulseTestData*** class handles data from the Birmingham battery test facility and calculates the internal resistance data from high frequency pulses applied to a cell under prescribed conditions. Like all classes in the package, it makes use of the *strategy pattern* to ensure a consistent interface is presented to the user. For example, the ***birminghamPulseTestData*** class inherits from the abstract parent class ***pulseTestDataImporter***. The list of implemented classes per correlation test procedure is given in #:

Table 1: Data import package classes

|  |  |  |
| --- | --- | --- |
| ***Class or File Name*** | ***Abstract*** | ***Purpose*** |
| *README.md* | N/A | Text file describing package purpose and contents |
| *pulseTestDataImporter* | Yes | Abstract parent using interface for supporting data import for pulse test data for any facility. |
| *birminghamPulseTestData* | No | Concrete class for importing pulse test (internal resistance) data from the Birmingham battery test facility. |
| *imperialPulseTestData* | No | Concrete class for importing pulse test (internal resistance) data from the Imperial College battery test facility. |
| *lancasterPulseTestData* | No | Concrete class for importing pulse test (internal resistance) data from the Lancaster *BatLab* facility. |
| *oxfordPulseTestData* | No | Concrete class for importing pulse test (internal resistance) data from the Oxford battery test facility. |
| *warwickPulsetestdata* | No | Concrete class for importing pulse test (internal resistance) data from the Warwick battery test facility. |
| *rateTestDataImporter* | Yes | Abstract parent using interface for supporting data import for rate test data for any facility. |
| *birminghamPulseTestData* | No | Concrete class for importing rate test (discharge capacity) data from the Birmingham battery test facility. |
| *imperialPulseTestData* | No | Concrete class for importing rate test (discharge capacity) data from the Imperial College battery test facility. |
| *lancasterPulseTestData* | No | Concrete class for importing rate test (discharge capacity) data from the Lancaster *BatLab* facility. |
| *oxfordPulseTestData* | No | Concrete class for importing rate test (discharge capacity) data from the Oxford battery test facility. |
| *warwickPulsetestdata* | No | Concrete class for importing rate test (discharge capacity) data from the Warwick battery test facility. |
| *states* | No | Enumeration class for automatically assigning facility names to numeric coding. |

## Common concrete parent class properties and methods

All data import classes make use of the MATLAB *datastore* implementation. This is specifically designed to handle large quantities of data stored in multiple files.  A datastore is a repository for collections of data that may be too large to fit in memory. The datastore object is *composited* in the data importer class. For more information, type the following at the MATLAB command prompt:

**doc datastore**

Concrete properties associated with the parent classes are defined in Table 2. To view any of these properties at the command line use the command syntax:

**childObjectName.PropertyName**

For example, if **L** is a **lancasterRateTestData** object then to reveal the number of files contained in the associated datastore property type the following at the command line:

**L.NumFiles**

Table 2: Concrete parent class properties for the **pulseDataImporter** and **rateDataImporter** abstract parent classes

|  |  |  |
| --- | --- | --- |
| Property Name | Set (Get) Access | Definition |
| Ds | Protected (public) | Datastore object containing links to raw data files |
| Signals | Protected (public) | List of available data channels |
| Data | Protected (public) | Data table |
| Battery | Protected (public) | Name of the battery type under test |
| NumFiles | N/A (public) | Number of files in the datastore |

The corresponding common concrete parent class methods are described in the following subsections. We make no attempt to describe all methods, only those which are likely to be called by a user. All methods are commented and implement help messages.

### The *channelPresent* Method

This method returns a logical value indicating whether a specified signal is contained within the data table. The command syntax is:

**obj.channelPresent( SignalName )**

Where **SignalName** is a MATLAB string. For example, to confirm if the variable "Current (A)" is present use:

**obj.channelPresent( "Current (A)" )**

The output will be true (false) if the named signal is present (not present).

### The *extractData* method

As the name suggests, this method extracts the data from each file contained within the datastore, performs all necessary pre-processing calculations and converts all signal names and units to the common format required for subsequent analysis. The command syntax is simply:

**obj = obj.extractData();**

### The *export2excel* method

This method exports the pre-processed data to an Excel workbook in the necessary common naming and unit format. Data channels defined in the workbook are hard coded into the analysis package. The command syntax is:

**obj.export2excel( Fname, Sheet )**

Where **Fname** is the full path statement to an Excel File and **Sheet** is the sheet number to write to.

## Data import example

In this section, we provide an example of the data import process. The scenario considered is importing data for the discharge capacity test (rate test) from the Lancaster University BatLab facility. Data import for other facilities or for internal resistance (pulse test) data processing is essentially identical in nature.

1. Instantiate a ***lancasterRateTestData*** object using the following command: **L = lancasterRateTestData();** 
   1. This will cause the standard windows file dialog box to appear. Navigate to the folder containing the data and click on the “Select Folder” button.

Graphical user interface, application, Teams

Description automatically generated

Figure : Window file dialog box. Navigate to the folder containing the data and click on the Select Folder button

A message of the following form should appear in the command window:

**L =**

**lancasterRateTestData with properties:**

**Fileformat: ".csv"**

**Tester: "Novonix"**

**Facility: Lancaster**

**Current: "Current (A)"**

**Capacity: "Capacity (Ah)"**

**Ds: [1×1 matlab.io.datastore.TabularTextDatastore]**

**Signals: [1×15 string]**

**Data: [0×0 table]**

**Battery: "LGM50"**

**NumFiles: 18**

Note, at this stage, the **Data** property is not populated, even though the datastore is defined.

1. To extract the data from the datastore and process it into the required common format for subsequent analysis execute the following command: **L = L.extractData()**. A wait bar, as displayed in Figure 1 tracking the data extraction and pre-processing progress will appear.

Graphical user interface, application

Description automatically generated

Figure 7: Data import wait bar

Upon completion, the wait bar will close and a message like the following should be displayed. Note the **Data** property is now populated.

**L =**

**lancasterRateTestData with properties:**

**Fileformat: ".csv"**

**Tester: "Novonix"**

**Facility: Lancaster**

**Current: "Current (A)"**

**Capacity: "Capacity (Ah)"**

**Ds: [1×1 matlab.io.datastore.TabularTextDatastore]**

**Signals: [1×15 string]**

**Data: [90×7 table]**

**Battery: "LGM50"**

**NumFiles: 18**

1. To export the pre-processed and properly formatted data to sheet 1 in a suitably named excel workbook use the following command strings:

**Fname = “H:\FacilityCorrelation\...\RateTest\LGM50\analysisRate.xlsx”;**

**L.export2excel( Fname, 1 )**

Note if the data export workbook does not exist it is created and otherwise appended to.

## Calculations

In this section we review the necessary pre-processing calculation for evaluating the internal resistance and discharge capacity response data. Both rely on custom pulse detection algorithms implemented in the code. Details of this and subsequent relevant calculations are given in the following sections:

### The pulse test (internal resistance) data

The nature of the pulse test is to apply a short pulse, of typically ten seconds duration, to the battery. With reference to Figure 8, for either charging or discharging the battery internal resistance, , is calculated from:

(1)

Diagram

Description automatically generated

Figure 8: Calculation of internal resistance values for charge and discharge pulses

The cell current characteristic from a typical pulse test is presented in Figure 9. A [Ah] state of charge (SoC) reduction pulse proceeds a 10 second discharge pulse, immediately followed by a 10 second charge pulse. Note the first charge pulse is terminated prematurely to protect against over voltage conditions. This data is considered an outlier and is excluded from the analysis. The data is first divided into separate SoC reduction, charge and discharge pulse packets. This is accomplished using the following algorithm.

1. Calculate the reduction current: [A].
2. Establish a logical pointer to the reduction regions:
3. Discard any data outside the reduction regions:
4. Apply the signum function to the event channel data vector :

Where:

1. Discard any positive data
   1. , where is the indicator function. The indicator function is unity if the condition within the bracket is satisfied and elsewise zero.
2. Apply the difference operator to . The start of the pulse will be identified by the value -1 and the end of the pulse by a value of 1. The algorithm defines logical pointers and to the discharge events.
   1. – pad the vector with a leading zero for time alignment.

Chart

Description automatically generated

Figure : Current trace for a typical pulse (Internal Resistance) test

### The rate test (discharge capacity) data

Figure 8 presents the recorded data from a typical discharge capacity or rate test. Five discharge pulses are applied to the test cell, which are encircled on the plot. Between pulses the cell is charged at a constant rate to replenish the charge. The pulse detection algorithm in pseudocode is as follows:

1. Apply the signum function to the event channel data vector :

Where:

1. Discard any positive data
   1. , where is the indicator function. The indicator function is unity if the condition within the bracket is satisfied and elsewise zero.
2. Apply the difference operator to . The start of the pulse will be identified by the value -1 and the end of the pulse by a value of 1. The algorithm defines logical pointers and to the discharge events.
   1. – pad the vector with a leading zero for time alignment.

Diagram

Description automatically generated

Figure 10: Typical discharge capacity test data. The five discharge pulses are circled.

Having identified the discharge pulse locations, the corresponding discharge capacity, , is calculated as:

(2)

## Export data formats

If new facilities are to be supported, or the data format from an existing facility change, it will be necessary to write a new import data class. To this end, in the next three subsections we discuss the common data formats and also provide some hints about how to write a new data import class.

### Writing a new data import class

The data import class makes use of the OOP strategy pattern discussed in section 1.4.2.. As such there are two abstract interface classes, one for rate test data (***rateTestDataImporter***) and the other for pulse test data (***pulseTestDataImporter***). These classes define abstract properties and method signatures. For example, the ***pulseTestDataImporter*** class contains the code:

properties ( Abstract = true, SetAccess = protected )

Current string % Name of current channel

Voltage string % Name of voltage channel

Capacity string % Name of capacity channel

DischgCurrent double % Discharge current

PulseTime double % Required pulse time [s]

Time string % Name of time channel

CF double % Conversion factor from

% hours to seconds

end % Abstract & protected properties

properties ( Constant = true, Abstract = true )

Fileformat string % Supported input formats

Tester string % Type of battery tester

Facility correlationFacility % Facility name

end % abstract & constant properties

The property attribute ( Abstract = true ) defines these properties as ***abstract***; that is they must be defined explicitly in any child class. Similarly, the parent class defines abstract methods, which again requires the concrete implementation to be defined in the child class.

methods ( Abstract = true )

obj = extractData( obj, varagin ) % Extract data from

% datastore

obj = setDischgCurrent( obj, Dc ) % Set the discharge

% current target value

end % Abstract methods signatures

The child class cannot be instantiated until all abstract properties and methods are defined. We strongly recommend copying, renaming and editing an existing child class when generating new data importer classes. More details can be found in the MATAL OOP user guide available from The Mathworks website.

### The Discharge Capacity (Rate) Test Data Format

The purpose of the data importer classes is to convert the custom signal naming convention, unit and even sign definitions to a consistent common format hard coded in the analysis package. In this section we discuss the required format for the discharge capacity test data, which is predicted in Table 3. This is a very straightforward format and easy to construct. Studying existing child class **extractData** methods is an excellent way of understanding techniques for converting custom facility to the common data format.

Table : Discharge capacity common data export format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BatteryName | SerialNumber | CRate | Cycle | Facility | Temperature | DischargeCapacity |
| NA | NA | [Ah] | [#] | NA | [Deg C] | Ah |
| LGM50 | 790 | 0.5 | 1 | Warwick | 25 | 4.91407 |
| LGM50 | 790 | 0.5 | 2 | Warwick | 25 | 4.908983333 |
| LGM50 | 790 | 0.5 | 3 | Warwick | 25 | 4.90329 |
| LGM50 | 790 | 0.5 | 4 | Warwick | 25 | 4.89719 |
| LGM50 | 790 | 0.5 | 5 | Warwick | 25 | 4.89188 |

### The Internal Resistance (Pulse) Test Data Format

Similarly, the purpose of the data importer classes is to convert the custom signal naming convention, unit and even sign definitions to a consistent common format hard coded in the analysis package. This format is slightly more complex and for brevity we do not present it here. Again, we strongly recommend interrogating existing child class **extractData** methods as an excellent way of understanding techniques for converting custom facility to the common data format.

# The facility correlation analysis package

Once the test data is properly formatted and incorporated into a single file, it can be analysed. To this end a custom MATLAB correlation data analysis package has been generated. The software architecture is depicted in Figure 2. The software is comprised of 5 main classes addressing:

1. Provide a user interface, permitting access to all required functionality through a single master process or class (***correlationAnalysis***).
2. Various data holder classes. The relevant class is selected at run time.
3. Design of experiment generation. The relevant algorithm is selected at runtime.
4. Model definition and identification.
5. A report class, which facilitates hypothesis testing and correction factor generation.

As the interface is (necessarily) quite complex two functions **rateTestAnalysis** and **pulseTestAnalysis** have been written to simplify the analysis. The next two sections describe these functions in detail. Both functions identify the appropriate model. Hypothesis testing procedures are discussed in section #.

## The rateTestAnalysis function

The function calling syntax is:

**obj = rateTestAnalysis( ModelType, Factors )**

To obtain help on the arguments type **help** **rateTestAnalysis** at the command line. The result is:

----------------------------------------------------------------------

Function to perform correlation analysis for the rate data.

obj = rateTestAnalysis( ModelType, Factors, Response, Xname );

Input Arguments:

ModelType --> Model type either: {"linear"},

"interaction", "quadratic" or "complete"

Factors --> (struct) multidimensional structure defining the

experimental factors. Each dimension defines an

individual factor, using fields:

Name: (string) Name of factor

Symbol: (string) Symbol denoting the factor.

Units: (string) Unit string

Levels: (cell) levels for factor

Cat: (logical) true for categorical factor

Values: (cell) possible numerical values for

categories

Cats: (string) vector of category names

Response --> (string) name of response variable

Xname --> (string) name of level-1 covariate

Output Arguments:

obj --> correlationAnalysis object

Ax --> Data plot axes handle array

ResAx --> Residual plot axes handle array

----------------------------------------------------------------------

The help explains the

# References

1. https://www.faraday.ac.uk/research/lithium-ion/battery-system-modelling [↑](#endnote-ref-1)