# Regularised Iterative Least Squares Package () User Notes

The MATLAB package has been written to implement a regularised iterative least squares algorithm, with optimal hyper-parameter selection. Optimal hyper-parameter estimation is based on information theoretic measures, and , and requires fixed-point iteration to update the regularisation parameter at each step of the iterative nonlinear least squares optimisation process. The code also supports heteroscedastic errors as required. This document details how to install and use the package. Although the package was originally conceived for identifying models for battery state-of-health prediction, the algorithm implemented is completely general. Hence, from the outset, the package was designed to be easily extended by allowing the user to incorporate their own models. Consequently, we provide detailed user instructions and instructions on how to incorporate custom models into the package. This requires some knowledge of the code architecture and a comprehensive description of it is given.

Contents

[1 Preamble 2](#_Toc39498994)

[2 Note on Abstract Classes and Composition 5](#_Toc39498995)

[2.1 Abstract Interfaces 5](#_Toc39498996)

[2.2 Composition 6](#_Toc39498997)

[3 Class Architecture and Relevant Object-Oriented Behavioural Patterns 7](#_Toc39498998)

[3.1 The Template Method 8](#_Toc39498999)

[3.2 The Strategy Pattern 9](#_Toc39499000)

[4 Installation Instructions 10](#_Toc39499001)

[5 Notes on Individual Classes 11](#_Toc39499002)

[5.1 Optimal hyper-parameter estimation 11](#_Toc39499003)

[5.2 Regularised fit model parameter estimation classes 14](#_Toc39499004)

[5.2.1 ***fitModel*** Abstract Members and Generating New Subclasses 15](#_Toc39499005)

[*5.2.2* Updating the enumeration class ***fitModelType*** 16](#_Toc39499006)

[5.2.3 The Class 17](#_Toc39499007)

[5.2.4 The Class 18](#_Toc39499008)

[5.2.5 The Class 19](#_Toc39499009)

[5.3 Covariance model parameter estimation 19](#_Toc39499010)

[6 Data Coding 21](#_Toc39499011)

[7 A brief note on MATLAB Packages 23](#_Toc39499012)

[8 The ***RegFit.regNonlinIGLS()*** class in detail 24](#_Toc39499013)

[8.1 The predictions and jaocbean methods 26](#_Toc39499014)

[8.2 Parameter estimation, the regOLSestimates, regIGLS and diagnosticPlot methods 27](#_Toc39499015)

[8.3 Estimating parameter precision, the stdErrors, confInt and bootStrapSamples methods 27](#_Toc39499016)

[8.3.1 The ***stdErrors*** and ***confInt*** Methods 27](#_Toc39499017)

[8.3.2 Bootstrapping vis the ***bootStrapSamples*** Method 29](#_Toc39499018)

[9 Worked Example 30](#_Toc39499019)

[10 Future Enhancements 33](#_Toc39499020)

[11 References 34](#_Toc39499021)

# Preamble

Often phenomenological models, formulated from physical reasoning, possess structures that have partially or completely confounded parameters to be estimated from the data. For example, consider the following model for battery state of health prediction which attempts to predict relative capacity, :

(1)

The parameters, , are to be estimated from the data. It is the multiplicative nature of the model structure which calls into question the apparent *identifiability* of the model. A model is ***identifiable*** if it is theoretically possible to learn the true values of this model's underlying parameters after obtaining an infinite number of observations from it. Identifiability is closely linked with sensitivity analysis and so to the properties of the model’s Jacobian matrix, . Specifically, if the columns of are linearly independent, then the model is identifiable, and the parameters can be uniquely estimated. This is equivalent to stating that the matrix is invertible.

In situations where the columns of are nearly colinear, the conditioning of the Jacobian is very poor, resulting in very imprecise parameter estimates. In addition, the parameter estimates are very sensitive to the peculiarities of the training data, implying that a small perturbation in the data can lead to a large change in the parameter estimates. One way of converting a problem from ill-posed to well-posed is to introduce a regularisation method. The basic idea is to confine the problem to a restricted set of parameters using a regularisation functional . Typically, this is achieved by introducing a constraint of the form to the usual log likelihood function, , such that:

(2)

The regularisation parameter , or hyper-parameter, implicitly defines a structure on the possible models by constraining the model. Roughly speaking, low values of in (4) correspond to high values of ; *i.e.* a weak constraint. Alternatively, a large value for the hyper-parameter, corresponding to a low value for , implies greater importance to in the minimisation of the cost function. Thus, a large value for represents a more severe constraint. Hence, we see the regularised cost minimisation problem represents a trade-off between model fidelity, minimising , and constraining the model to be a member of a small compact subset, . Furthermore, the balance between fitting the data and satisfying the constraint is controlled by the hyper-parameter.

In a recent paper to be published, we showed how the information theoretic measures and could yield recursive formulae, which can be iterated to yield an optimal value for at each step of the optimisation procedure used to minimise . The appropriate formulae are:

(3)

(4)

Where, . The term, represents the *effective degrees of freedom* for the model. These equations can be embedded in a full regularised iterative generalised least squares algorithm as follows:

***Algorithm 1:***

1. Obtain the initial estimates, , by minimising the following nonlinear functional with respect to . Re-estimate *λ* at each iteration, by iterating equation (20) or (21) to convergence as required.



1. Using , generate the residuals, . Using there, minimise the following functional with respect to . Denote these estimates as .
2. Using construct the matrix . Update estimates for by minimising the nonlinear function:

* Re-estimate *λ* at each iteration, by iterating equation (20) or (21) to convergence as required.
* Denote the values at convergence by .

1. Compute . If , then treating as return to step 2, else stop.

* Compute .

The algorithm has been implemented in MATLAB and utilises the ***fmincon*** function from the MATLAB *optimisation toolbox*. To work as intended, it is necessary to supply analytical gradients to the ***fmincon*** routine, so that the hyper-parameter is calculated only once per iteration, as opposed to -times if finite differences are utilised. Using standard results on matrix derivatives, it can be shown that:

(5)

A MATLAB package, ***RegFit***, has been written to implement the algorithm, which makes use of the *strategy* and *template* object oriented behavioural patterns. The purpose of this document is to:

* Describe how to install the package.
* Describe the package architecture, including a brief description of the relevant object-oriented programming behavioural patterns.
* Provide a high-level description of the package classes.
* Provide a worked example of how to utilise the package to fit a model.
* Provide instructions for entering user-defined models into the package.

# Note on Abstract Classes and Composition

The code architecture makes considerable use of *abstract interfaces* and *composition*. Consequently, we include a brief introduction to these concepts in the following two sub-sections.

## 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 members* 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 members* 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 in order to become a concrete class. For example, the ***reEstLamda*** interface contains one abstract property (***Measure***) and three abstract methods (***calculateLamda(), getMeasure()*** and ***calcNewLam()***). The abstract property is defined with the ***setAccess*** attribute set to *protected*. Consequently, the concrete implementation must also implement the property ***Measure*** with identical attributes; i*.e*. ( ***setAccess*** = *protected* ). Similar comments relate to abstract methods. Note, private methods cannot be abstract as they cannot be inherited by the child subclass.

## Composition

Composition is one 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 1 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.

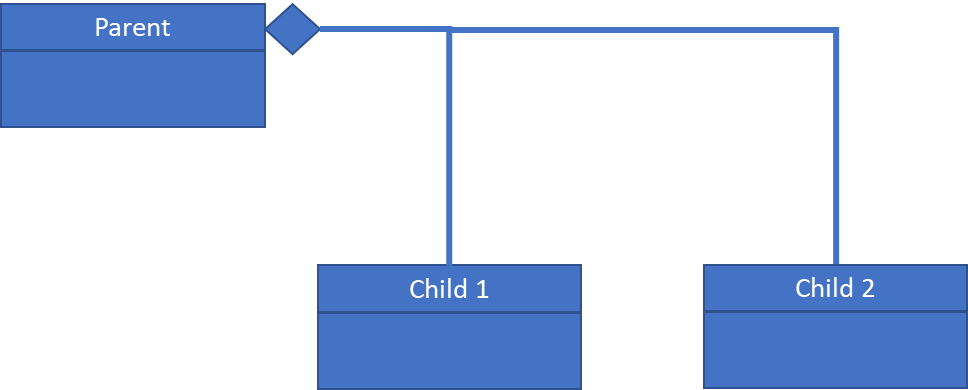


Figure 1: Class diagram depicting composition. Both child classes are stored as properties of the parent. The solid blue diamond is the symbol for composition.

# Class Architecture and Relevant Object-Oriented Behavioural Patterns

A schematic depicting the high-level package architecture is shown in Figure 1. Although originally conceived for state of health estimation for lithium-ion battery, the algorithms implemented can be applied to any non-linear function. The architecture makes use of two object-oriented programming (OOP) behavioural patterns: the *template method* and the *strategy pattern*.

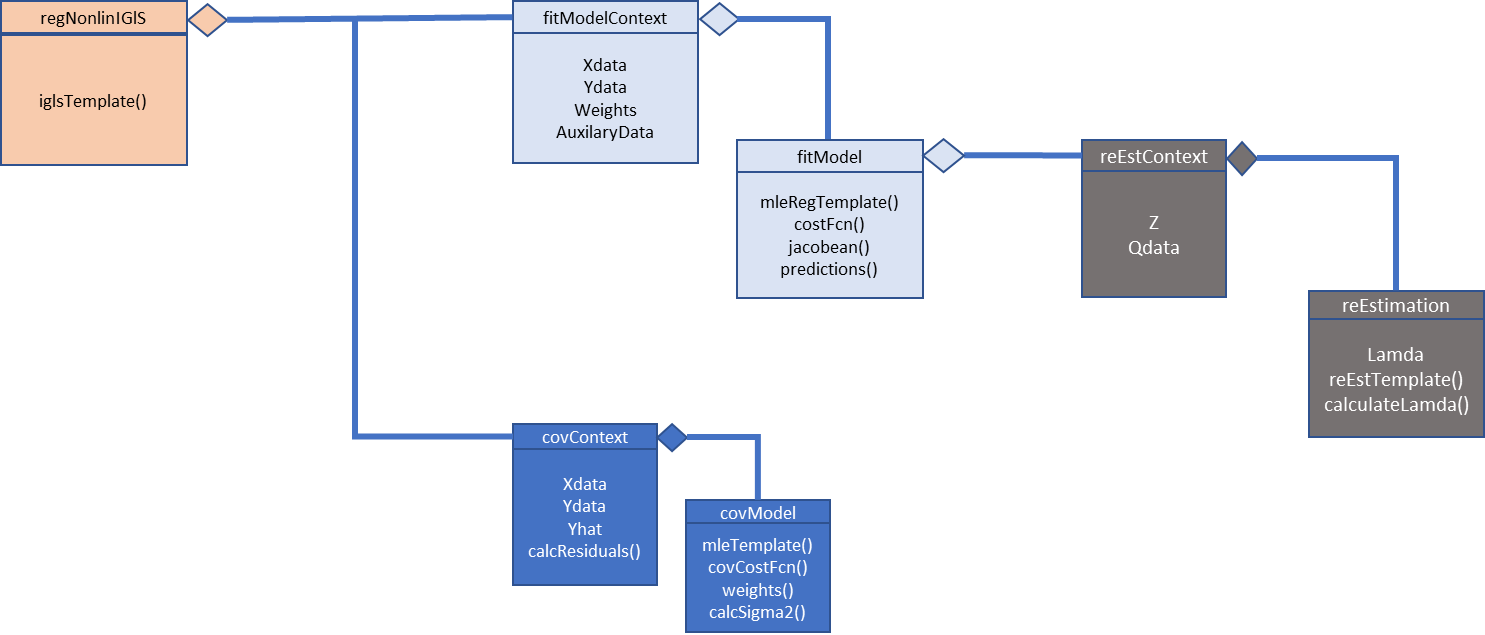


Figure 2: High-Level Package Architecture Depicting the Relationship Among Classes

## 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 2 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 signatures, as defined by input and output arguments, is the same.

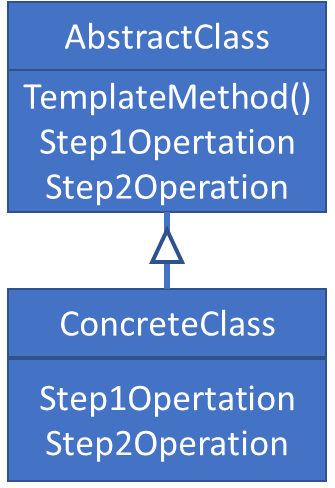


Figure 3: Template Method Class Architecture

## 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 3 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.

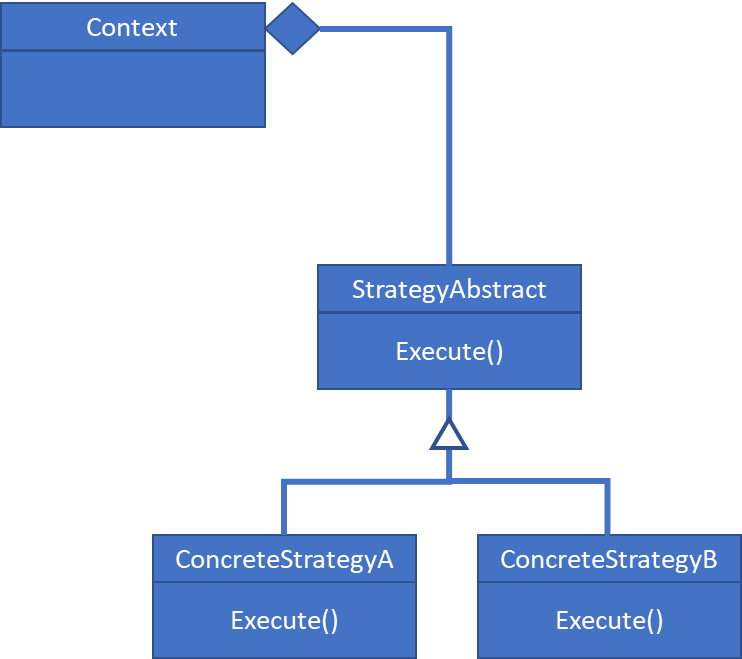


Figure 4: Schematic for the strategy pattern. The desired algorithm is selected at run-time from a family of related algorithms.

# Installation Instructions

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

Enter GITHUB project details here….

1. Create a directory called ***+RegFit*** in any directory on the current MATLAB path. Note, as the directory name begins with a “+” this identifies the contents as a MATLAB package.
2. Copy the file ***RegFit.zip*** to the ***+RegFit*** directory created in step 1. Unzip the contents to this directory.
3. At the command line type ***which RegFit.regNonlinIGlS.*** This should generate a message similar to: ***C:\Users\...\CLASSES\+RegFit\regNonlinIGLS.m.***

To list the package contents type ***help RegFit*** at the command line. This will return a list of package contents and component definitions.

# Notes on Individual Classes

As suggested by Figure 1, the entire package is essentially comprised of 4 *strategy-pattern* class groups which perform the following tasks: optimal hyper-parameter estimation, model regression coefficient determination, identification of the covariance model parameters and process control. As will become apparent in section 8, the user only interfaces with the ***regNonlinIGLS*** class. Consequently, for completeness sake, In the next three sections we provide a brief description of each of these strategy-pattern class groups. These notes also serve to provide the user with insight as to how to embed custom classes into the general package architecture. In this way, any nonlinear function may be fitted using algorithm 1 utilising any preferred covariance model, with the hyper-parameter estimated using any favoured information theoretic metric.

## Optimal hyper-parameter estimation

Optimal hyper-parameter estimation is implemented via the ***reEstLamdaContext*** and ***reEstLamda*** classes. The class relationships are illustrated in Figure 4. The ***reEstLamda*** class is an abstract interface containing a template method. The algorithm details are:

***Algorithm 2:***

1. Randomly select an initial value of the regularisation parameter from the interval . Denote this by . Set .
2. Increment .
3. Calculate using (3) or (4) as desired.
4. If . Else treating as return to step 2.

The specific formulae to be iterated, equations (3) and (4), are implemented in the concrete classes ***aicReEst*** and ***bicReEst***. Any preferred formula, based on an alternate information criterion, can be implemented by adding a new concrete class. To accomplish this, the concrete child class must implement the abstract property ***Measure***, and in addition, it must also define the methods ***getMeasure()*** and ***calcLamda()***, ***calcNewLam()***. The corresponding attributes are defined in Table 1.

The ***aicReEst*** and ***bicReEst*** classes can form a template to implement your own prefer measures,

for example . Notice ***calcNewLam()*** is called from within the iterative (while) loop implemented in method ***calcLamda()***.

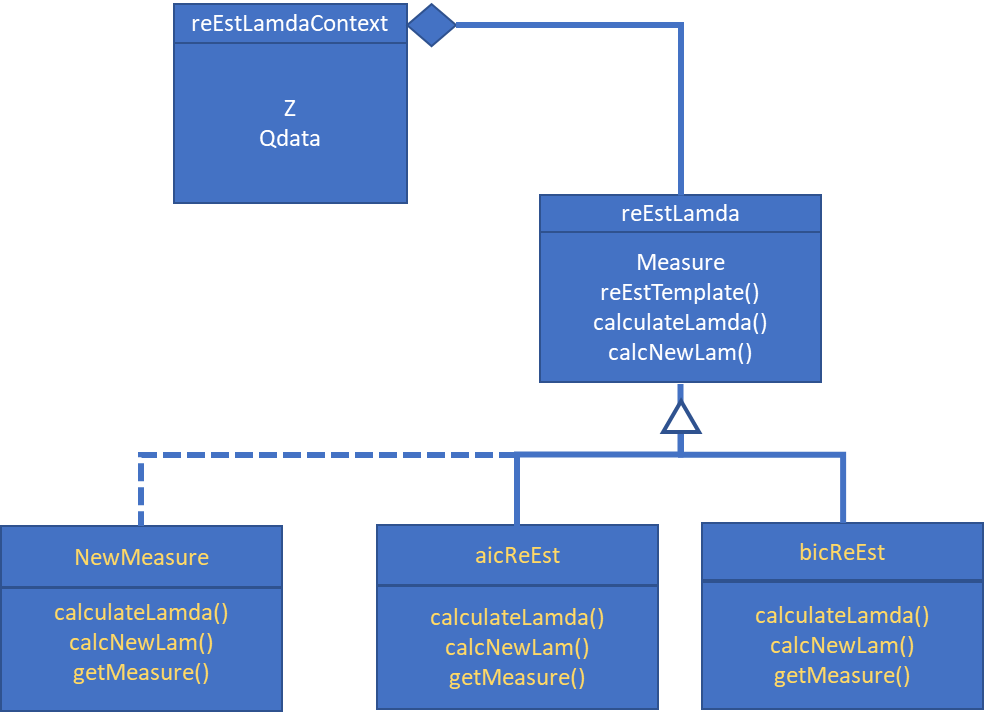


Figure 5: Optimal hyper-parameter estimation classes schematic

Table : Abstract members for implementation in concrete hyper-parameter estimation classes

|  |  |  |
| --- | --- | --- |
| Abstract Member | Abstract Member Type | Attributes |
| ***Measure*** | Property | setAccess = protected |
| ***getMeasure()*** | Method | Access = public |
| ***calcLamda()*** | Method | Access = protected |
| ***calcNewLam()*** | Method | Access = protected |

## Regularised fit model parameter estimation classes

The regularised weighted least squares fit is implemented in classes ***fitModelContext*** and ***fitModel*** classes. These classes implement steps 1 and 3 of ***Algorithm 1***. Figure 5 presents a schematic of the relevant class architecture. Again, the class ***fitModel*** is abstract and requires child classes to define methods to calculate the model predictions and associated Jacobian. These are required for calculation of the regularised cost function and associated analytical gradients with respect to the fit parameters. At each iteration, the hyper-parameter is updated from within the method ***costFcn()***, which returns the regularised cost and analytical gradients directly to the optimisation toolbox ***fmincon*** function.

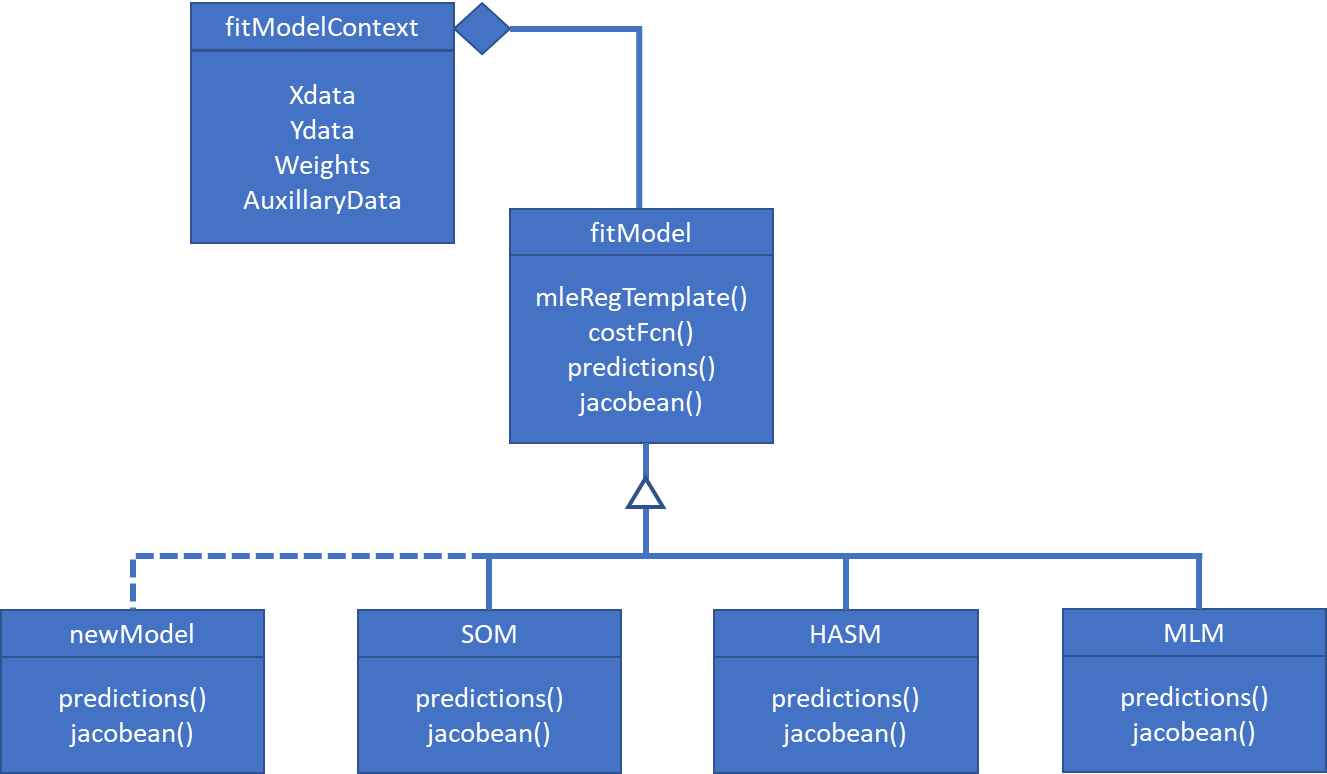


Figure 6: Regularised fit parameter estimation class diagram. Although originally conceived for battery state-of-health prediction, any nonlinear function can be implemented in a new concrete subclass.

A template method, ***mleRegTemplate()***, implements the regularised iterative generalised least squares algorithm. Observed data, their associated weights, to form the necessary covariance matrices, and any auxiliary data are supplied from the context class. In this way, the ***fitModel*** class does not require to make copies of these – simplifying the code and reducing storage requirements.

### ***fitModel*** Abstract Members and Generating New Subclasses

Any new model subclass must define the abstract properties and methods defined in Table 2. The classes ***mlm***, ***som*** and ***hasm*** serve as templates. The underlying principle is that the subclass implements tasks and defines properties specific to the model and required by the algorithm. For example, the parent abstract class has no knowledge of how to calculate a Jacobean matrix for a given model. Instead the parent only knows when such calculations must be carried out.

Note, when defining abstract members, it is essential to define them with the appropriate attributes specified in the table. For example, the abstract property ***LB*** is defined as a constant in the parent abstract ***fitModel*** class. Any concrete child subclass implementing this property must implement ***LB*** as a constant. Similarly, the ***assignPars()*** method must be implemented as a static method. You will not be able to instantiate a new subclass object until all abstract members are implemented correctly. Consequently, we strongly recommend using the ***mlm***, ***som*** and ***hasm*** classes as templates to minimise the chances of errors.

Table : Abstract members for implementation in concrete child nonlinear fit models

|  |  |  |  |
| --- | --- | --- | --- |
| Abstract Member | Member Type  Variable type | Description | Attributes |
| ***Theta*** | Property  double(p,1) | Regression coefficient vector | Access = public |
| ***LB*** | Property  double(p,1) | Lower bound constraint for regression coefficient vector terms | Constant |
| ***UB*** | Property  double(p,1) | Upper bound constraint for regression coefficient vector terms | Constant |
| ***ModelName*** | Property  string | Model name. | Constant |
| ***ParNames*** | Property  string(1,p) | Regression vector parameter names. Used during plotting. | Constant |
| ***startingValues()*** | Method | Method for calculating initial values for the regression coefficients. | Access = public |
| ***predictions()*** | Method | Calculates the predicted output from the model. | Access = public |
| ***jacobean()*** | Method | Returns the model Jacobian matrix | Access = public |
| ***assignPars()*** | Method | Assigns the contents of the regression coefficient vector (***obj.Theta***) to individual named parameters. | Static |
| ***parseInputs()*** | Method | A method to remove any data points where the model yields a zero output. This prevents the code producing infinite weights and crashing the optimisation process. | Static |

In addition, many models may require the definition of auxiliary data or constants. For example, the ***som*** class defines a value for the universal gas constant. It also requires the average cell temperature, normalised average current for the cycle and average state of charge. Whereas any necessary additional constants may be defined in an appropriate property block, it is strongly recommended that auxiliary data properties be defined wit SetAccess = immutable. An immutable property is one that can only be defined as a class object is instantiated. This prevents the user changing it from then on. Obviously, changing auxiliary data at a later stage would have serious potential consequences for any analysis. To illustrate this point, we now briefly discuss the pre-defined fit model subclasses supplied with the package. All of these were originally conceived for battery state of health estimation.

### Updating the enumeration class ***fitModelType***

Associated with the ***fitModel*** abstract class and associated child classes is an enumeration class, ***fitModelType***. Enumerations allow assign fixed names to a single type of value. The named value, or enumeration, can then be used in multiple places without redefining it for each class. Currently, the ***fitModelType*** class is comprised of the following statements:

classdef fitModelType < int8

% Enumerations for supported fit models. You need to edit this if you add your own classes.

enumeration

SOM (0)

MLM (1)

HASM (2)

end

end

Now imagine you have created a new fit model subclass implementing a three-parameter sigmoid function called ***tps***. To inform the package this is a new supported model, modify the ***fitModelType*** class to read:

classdef fitModelType < int8

% Enumerations for supported fit models. You need to edit this if you add your own classes.

enumeration

SOM (0)

MLM (1)

HASM (2)

TPS (3)

end

end

### The Class

The Martinez-Laserna *et al* [[[1]](#endnote-1)] model is defined by equation (1) and predicts capacitive fade as a function of cumulative throughput or time. The parameters are to be estimated from the data. Like many such physically motivated models for this application, the model is comprised of a gain, an Arrhenius like rate constant and a power law component. The rate constant term is an exponential function of the average cell temperature and the state of charge of the battery. The auxiliary data and must be defined when creating a class object, *i.e.* are *immutable* properties. The syntax for instantiating an ***mlm*** class object is:

***m = RegFit.mlm( ReEst, SOC, T );***

Where ***ReEst*** is a ***RegFit.reEstLamdaContext*** object, which implements the hyper-parameter re-estimation scheme. The child subclass implements the model structure and consequence can define such as items as the parameter names. These are defined in the concrete subclass implementation in the constant property ***ParNames***. The concrete implementation of the ***assignPars()*** method, decodes the parameter vector, assigning the vector components to named variables making subsequent code easier to read. The method signature is:

***[ Omega, Beta1, Beta2, Z] = obj.assignPars( Theta );***

If the fit function returns zero, the corresponding data weight is infinity – crashing the ***fmincon*** routine. Consequently, the method ***parseInputs()*** is implemented to automatically remove any data corresponding to this condition. As this is a function of the input data only, this need only be called once, and minimisation of the regularised cost function carried out in the reduced data set.

### The Class

The Suri and Onori [[[2]](#endnote-2)] model is another physically motivated phenomenological model which attempts to predict capacitive fade versus time or cumulative throughput. The model is defined as:

(6)

(7)

The parameters are to be estimated from the data. The similarity with the Martinez-Laserna equation is quite marked, with the model again being comprised of a gain, an Arrhenius like rate constant and a power law. The auxiliary data and average cycle c-rate, , are immutable properties which must be defined when instantiating a class object. The command syntax is:

***obj = RegFit.som( ReEst, SOC, Ic, T );***

Where ***ReEst*** is a ***RegFit.reEstLamdaContext*** object, which implements the hyper-parameter re-estimation scheme. The child subclass implements the model structure and consequence can define such as items as the parameter names. These are defined in the concrete subclass implementation in the constant property ***ParNames***. The concrete implementation of the ***assignPars()*** method, decodes the parameter vector, assigning the vector components to named variables making subsequent code easier to read. The method signature is:

***[ Alpha, Beta, Eta, Z] = obj.assignPars( Theta );***

If the fit function returns zero, the corresponding data weight is infinity – crashing the ***fmincon*** routine. Consequently, the method ***parseInputs()*** is implemented to automatically remove any data corresponding to this condition. As this is a function of the input data only, this need only be called once, and minimisation of the regularised cost function carried out in the reduced data set.

### The Class

The hypothesised active site model [[[3]](#endnote-3)] is a more complex empirical model containing a variety of terms related to specific mechanisms for active site loss. The model contains components associated with accelerated binder failure at high temperature, bulk intercalation strain, bulk thermal strain and intercalation gradient strain accelerated by low temperature. Specifically, the model is:



The parameter vector must be estimated from the data.

## Covariance model parameter estimation

The covariance models implemented account for either identically independently distributed or heteroscedastic data, where the variance depends on the magnitude of the predictions. Currently, four models are implemented. These are:

1. data: **.**
2. The power model:
3. The exponential model:
4. The two-components of variance model:

Figure 7 presents a schematic of the class architecture. Again, the ***covModel*** strategy interface is composited in the associated context class. The specific implementation, or model, is selected at run-time. Notice the necessary data is supplied from the context. In this way the concrete subclasses do not need to maintain copies of the data, simplifying the interface. This is particularly germane as the predictions will vary each iteration of the overall algorithm.

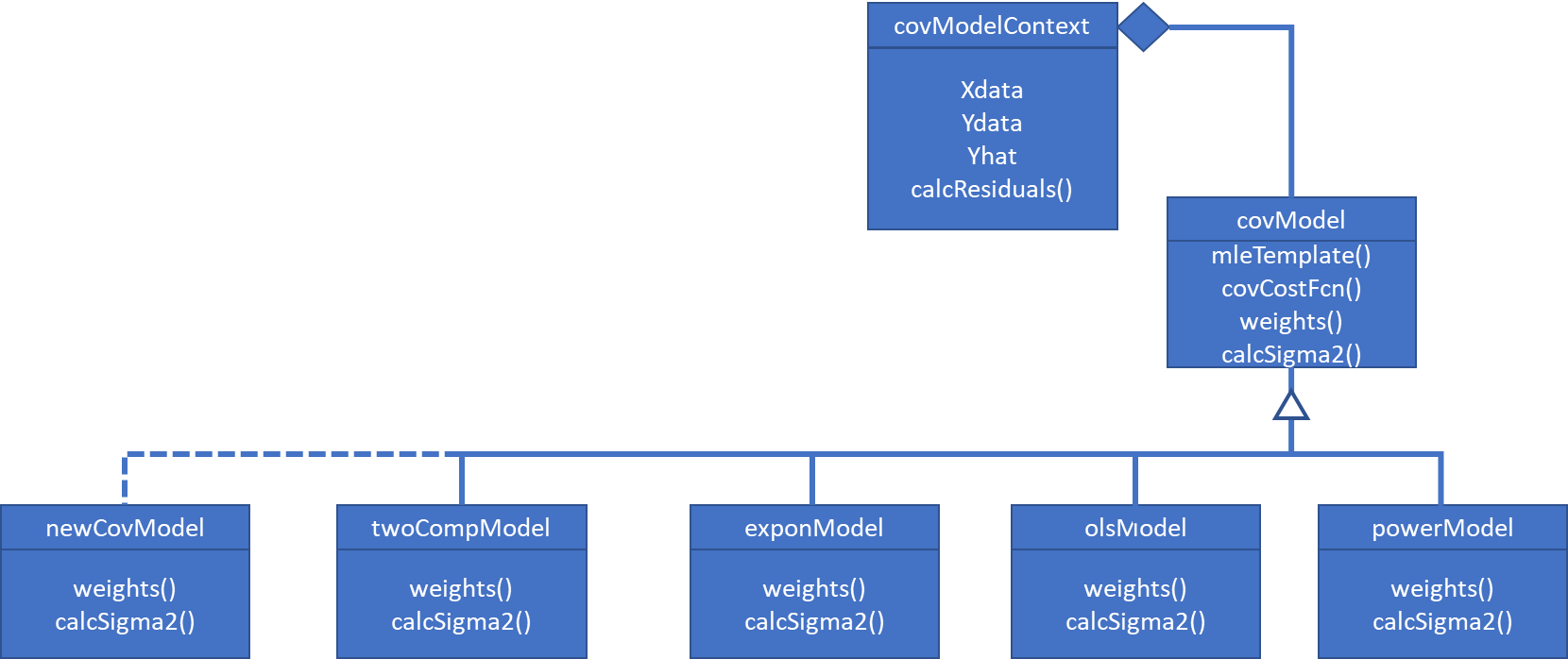


Figure 7: Covariance model parameter estimation class diagram. This currently supports both homogeneous and heterogeneous variance. Models for serially correlated data, e.g. ARMAX, are not currently supported.

As Figure 7 suggests, it is possible to implement additional custom covariance models, which essentially plug into the ***covModel*** abstract interface. Table 3 defines the abstract members that any concrete child class must implement. The classes ***twoCompModel***, ***powerModel*** and ***exponModel*** can be treated as templates for any custom addition.

Table : Abstract members for implementation in concrete child covariance model subclasses

|  |  |  |  |
| --- | --- | --- | --- |
| Abstract Member | Member Type  Variable type | Description | Attributes |
| ***Delta*** | Property  double(q,1) | Covariance model parameter vector | Access = public |
| ***LB*** | Property  double(q,1) | Lower bound constraint for regression coefficient vector terms | Constant |
| ***UB*** | Property  double(q,1) | Upper bound constraint for regression coefficient vector terms | Constant |
| ***calcWeights()*** | Method | Return the corresponding data weights. Required for the RIGLS algorithm implementation. | Access = public |

Like the ***fitModel*** abstract class, the ***covModel*** interface utilises an enumeration class, ***covModelType*** to identify supported models. This class must be updated if you add in your own covariance models. For example, imagine you implement an AR(1) serial correlation model in the custom class ***ar1Model***. To identify this new model to the ***covModel*** interface, edit ***covModelType*** as follows:

classdef covModelType < int8

% Enumeration class for supported covariance model types. You need to edit this if you add additional covariance models.

enumeration

OLS (0)

Power (1)

Exponential (2)

TwoComponents (3)

AR1 (4)

end

end

# Data Coding

Inputs with vastly different unit scales can give rise to ill-conditioned Jacobian matrices, which tend to over inflate parameter estimates and seriously degrade precision. The issue is easily demonstrated through example. Consider the simple linear model:

(8)

Assume the available data is 1000, 2000, …, 6000. The corresponding regression matrix in natural units is:

However, the corresponding condition number foris a massive , implying the matrix is appalling ill-conditioned. Consequently, we might expect significant numerical rounding issues when attempting to solve the corresponding normal equations to estimate the regression coefficients. Now consider the linear mapping: . Let denote the regression matrix in coded units. That is:

However, after this change of scale the condition number of is just . This reveals the poor conditioning of is a function of the unit scale and is not an inherent property of the design. In nonlinear regression problems we can apply the same thinking to the conditioning of the Jacobian matrix. Recall in nonlinear problems it is necessary to use iterative search procedures (gradient optimisation algorithms) to determine the optimal values for the model coefficients. If the problem scaling yields a very poorly conditioned Jacobian matrix, then the search direction will be almost parallel to the likelihood function contours and convergence may be slow or, in severe cases, not possible.

For this reason, the package offers automated scaling onto any user defined interval for both the regression and response variables. These are controlled by the settings for the properties ***XLB***, ***XUB***, ***YLB*** and ***YUB***. Consider scaling the model input. Say we want to implement the linear mapping . Then the corresponding settings for XLB and XUB are:

***obj.XLB = [10, 0];***

***obj.XUB = [100, 1];***

Similar remarks apply to scaling the response variable. Once defined, coding and decoding of the inputs or output is handled invisibly by the code and the user interacts with the class objects in natural units.

# A brief note on MATLAB Packages

Packages are special folders that can contain class folders, function, and class definition files, and other packages. The names of classes and functions are *scoped* to the package folder. A package is a namespace within which names must be unique. Function and class names must be unique only within the package. Using a package provides a means to organize classes and functions. Packages also enable you to reuse the names of classes and functions in different packages. If required, this permits classes with the same name to be independently modified within different packages to perform slightly different implementations. Equally if a class or function is modified within a given package, those changes are visible only to the current package contents. This is true even if the same class or function name appears in other packages. Think of this as a crude means of implementing version control.

MATLAB package folders always begin with a “+” character. Hence, the ***RegFit*** package is stored in folder ***+RegFit***. The contents of the package can be listed using the ***help*** or ***what*** commands. For example, the command ***what RegFit*** returns:

*MATLAB Code files in folder C:\Users\markc\OneDrive\MATLAB\CLASSES\+RegFit*

*aicReEst covModelType fitModelType powerModel som*

*bicReEst exponModel hasm reEstLamda twoCompModel*

*covModel fitModel mlm reEstLamdaContext*

*covModelContext fitModelContext olsModel regNonlinIGLS*

*MAT-files in folder C:\Users\markc\OneDrive\MATLAB\CLASSES\+RegFit*

*mlmTestData*

Any reference to packages, functions and classes in the package must use prefix. For example, to create a ***powerModel*** object, use the syntax:

***pwr = RegFit.powerModel();***

Once a package object is instantiated it is not necessary to employ the package prefix. For example, to obtain the lower bound(s) for the variance model parameter vector type ***pwr.LB*** at the command line.

You cannot add package folders to the MATLAB path. However, you must add the package parent folder to the MATLAB path. Package members are not accessible if the package parent folder is not on the MATLAB path, even if the package folder is the current folder. That is, making the package folder the current folder is not sufficient to add the package parent folder to the path. In short, package members remain scoped to the package. Always refer to the package members using the package name.

# The ***RegFit.regNonlinIGLS()*** class in detail

The user is granted access to the entire regularised iterative generalised least squares algorithm, with optimal hyper-parameter estimation, via the ***RegFit.regNonlinIGLS()*** class. Consequently, this is the only class in the package requiring detailed description. This is provided in this section.

Table 4: RegFit.regNonlinIGLS class property definitions and attributes

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Description** | ***Attributes*** |
| ***X*** | Independent or regressor variable | *SetAccess = immutable* |
| ***Y*** | Dependent or response variable | *SetAccess = immutable* |
| ***XLB*** | Lower bound for coding independent data | *Access = Public* |
| ***XUB*** | Upper bound for coding independent data | *Access = public* |
| ***YLB*** | Lower bound for coding dependent data | *Access = public* |
| ***YUB*** | Upper bound for coding dependent data | *Access = public* |
| ***FitModelContextObj*** | ***RegFit.fitModelContext*** object | *SetAccess = protected* |
| ***CovModelContextObj*** | ***RegFit.covModelContext*** object | *SetAccess = protected* |
| ***Lamda*** | Optimal regularisation or hyper-parameter | *Dependent = true* |
| ***DoF*** | Effective number of regression parameters | *Dependent = true* |
| ***Algorithm*** | Re-estimation algorithm name | *Dependent = true* |
| ***ModelName*** | Fit Model Name | *Dependent = true* |
| ***N*** | Number of data points | *Dependent = true* |
| ***ParNames*** | Fit parameter names | *Dependent = true* |
| ***Theta*** | Regression parameter estimates | *Dependent = true* |
| ***Xc*** | Coded independent data | *Dependent = true* |
| ***Yc*** | Coded dependent data | *Dependent = true* |
| ***W*** | Data weights | *Dependent = true* |
| ***Delta*** | Covariance model parameter vector | *Dependent = true* |
| ***Sigma2*** | Variance scale parameter | *Dependent = true* |
| ***Sigma*** | Dispersion scale parameter | *Dependent = true* |
| ***NumCovPar*** | Number of covariance model parameters, | *Dependent = true* |
| ***TotalNumPar*** | Total number of model parameters | *Dependent = true* |
| ***Measure*** | Model selection criterion, either or | *Dependent = true* |

Table 4 lists the available class properties. Access is provided to all pertinent information and the user does not need to understand the specific implementations in composite classes that provide these data. The specific implementations are effectively hidden, with any necessary operations required to be executed by composite classes called automatically as required. This is classic *polymorphism*, one of the fundamental principles of OOP. Consequently, the user only requires to be able to manipulate a ***RegFit.regNonlinIGLS()*** object to perform the complete analysis.

To create a ***RegFit.regNonlinIGLS()*** object in the workspace use the command syntax:

***reg = RegFit.regNonlinIGLS( X, Y, fitModelObj, covModelObj );***

Where ***X*** and ***Y*** are the respective independent and response data vectors, ***fitModelObj*** is a ***RegFit.fitModelContext*** object and ***covModelObj*** is ***a RegFit.covModelContext*** object. The context objects are defined by the use a priori, and thus the relevant fit and covariance models, and also the re-estimation metric, are decided upon at run-time. This is the underlying principle of the strategy OOP behavioural pattern. This is demonstrated in section 9. The syntax for examining any assigned property value is ***obj.PropertyName***. For example, to display the independent training data, use: ***reg.X***.

## The predictions and jaocbean methods

As the name suggests, the predictions method provides estimated response values at the specified input condition. The method signature is:

***Yhat = reg.predictions( X ); % Predictions at the locations specified***

***Yhat = reg.predictions(); % Predictions for the training data***

Any data coding applied to the independent and dependent variables is handled automatically by the class. Regardless ***X***-values are always in natural units. Similarly, predictions are also in engineering units.

The Jacobean matrix, essential to the algorithm, is calculated via the jacobean() method. Note, as required by the algorithm the derivative is with respect to the coded data and is consequently unitless. The method signature is:

***J = reg.jacobean( X ); % Jacobean at the locations specified***

***J = reg.jacobean(); % Jacobean for the training data***

## Parameter estimation, the regOLSestimates, regIGLS and diagnosticPlot methods

## Estimating parameter precision, the stdErrors, confInt and bootStrapSamples methods

The precision of the estimates can be evaluated in one of two ways using the class methods: by calculating approximate standard errors using a first order Taylor series representation of the fit function or from bootstrapping.

### The ***stdErrors*** and ***confInt*** Methods

For ridge regression the approximate variance-covariance matrix for the regression coefficients are given by:

(9)

Where, and

Hence, the vector of standard errors, , is given by:

(10)

Confidence intervals for the parameter estimates, , can be obtained using:

(11)

Where, . That is the effective number of parameters, plus the number of covariance model parameters and an additional degree of freedom for estimating . Note that will not be an integer and thus contains partial degrees of freedom. This is a function of the regularisation mechanism. The ***stdErrors*** and ***confInt*** methods implement equations (10) and (11).

The command syntax for obtaining standard errors is:

***SE = reg.strErrors();***

To obtain confidence intervals for the parameters use:

***[ LCI, UCI ] = reg.confInt( p );***

Where LCI is the lower bound and UCI the upper bound, such that: ***LCI reg.Theta UCI***.

### Bootstrapping vis the ***bootStrapSamples*** Method

Bootstrapping is a resample with replacement procedure for empirically estimating the sampling population of the parameters [[[4]](#endnote-4)]. Necessarily, the bootstrap involves many repeated nonlinear regularised fits to the bootstrap sample data. The command syntax is:

***[P, M, S] = reg.bootStrapSamples( Nboot );***

Where ***Nboot*** is the number of bootstrap samples. The default is 1000. The output arguments are the matrix of bootstrap parameter estimates and the regularisation hyper-parameter, ***P***, the mean bootstrap sample parameter vector, ***M***, and the vector of standard deviations of the bootstrap parameter estimates, ***S***. In addition, the method creates histograms for the bootstrap parameter estimates and the regularisation hyper-parameters. Examples are presented in Figure 8 and Figure 9.



Figure 8: Example bootstrap parameter histograms for the model of equation (1).



Figure 9: Example bootstrap histogram for the regularisation hyper-parameter.

# Worked Example

The package includes an example data file, ***mlmTestData.mat***, with the data corresponding to the model defined by equation (1). The data exhibits heteroscedastic behaviour, according to the variance function , where again takes the form of equation (1). To demonstrate the nature of the heteroscedasticity, we begin by making an ordinary least squares fit to the data. This is accomplished using the following commands:

1. Switch to the ***+RegFit*** directory generated in step 1 of section 4.
2. Load the file ***mlmTestData.mat*** into the workspace by typing the following command at the command line.
   1. ***load(‘mlmTestData.mat’)***
3. Create an ***aicReEst*** object using the syntax:
   1. ***aic = RegFit.aicReEst;***
4. Define a ***reEstLamdaContext*** object via the command:
   1. ***reEst = RegFit.reEstLamdaContext( aic );***
5. Create on ***olsModel*** object using:
   1. ols = RegFit.olsModel;
6. Make a ***mlm*** object using the command:
   1. ***m = RegFit.mlm( reEst, SOC/100, T);***
7. Create a ***regNonlinIGLS*** object by executing the command:
   1. ***reg = RegFit.regNonlinIGLS( Ah, Qloss, m, ols );***
8. Set the coding for the Ah and Qloss data by defining the ranges:
   1. : ***reg.XLB = [0, 0]; reg.XUB = [50, 1];***
   2. ***: reg.YLB = [0, 0]; reg.YUB = [35, 1];***
9. Execute the nonlinear fit by executing the command:
   1. ***reg = reg.regOLSestimates());***
10. Generate fit diagnostic plots. The command syntax is:
    1. ***reg.diagnosticPlots();***

Diagnostic plots are presented in Figure 8. The residual versus predicted plot, top right, demonstrates heteroscedastic behaviour with the magnitude of the variance increasing proportionally to the predicted quantity. This suggests the necessity to incorporate a covariance model to mitigate this concern.



Figure 10: Diagnostic plots for the OLS fit. The residual versus predicted plot demonstrates the heteroscedastic nature of the data as variation clearly increases with increasing predicted value. Regardless the normal probability plot suggests the presence of at least one outlying observation (circled).

To incorporate a covariance model of the form execute the following command line instructions:

1. Create a ***powerModel*** object using:
   1. ***pwr = RegFit.powerModel;***
2. Create a second ***regNonlinIGLS*** object by executing the command:
   1. ***reg2 = RegFit.regNonlinIGLS( Ah, Qloss, m, pwr );***
3. Set the coding for the Ah and Qloss data by defining the ranges:
   1. : ***reg2.XLB = [0, 0]; reg.XUB = [50, 1];***
   2. ***: reg2.YLB = [0, 0]; reg.YUB = [35, 1];***
4. Execute the nonlinear fit by executing the command:
   1. ***reg2 = reg2.regIGLS( 4 );***
5. Generate fit diagnostic plots. The command syntax is:
   1. ***reg.diagnosticPlots();***

Figure 9 presents the model diagnostic plots for the fit. The appearance of the weighted residual versus predicted plot, top right, is greatly improved, thus demonstrating the effectiveness of the applied covariance model. At convergence, the optimal hyper-parameter value is . This corresponds to for the ordinary least squares case. Regardless, despite the increased bias (larger hyper-parameter values) associated with the fit, this affords no practical significance and the fit appears to be of high quality.



Figure : Diagnostic plots for the weighted RIGLS fit. The appearance of the weighted residual versus prediction plot is much improved, indicating the efficacy of the covariance model. The normal probability plot is not ideal but considered adequate, given the fit quality.

# Future Enhancements

The main obvious short coming of the package is the current version cannot account for serially correlated data. Including linear noise models, *e.g.* AR, MA or ARMA, would be useful in the analysis of time series data. In this scenario, the fit model can be considered to detrend the data, leaving the noise model to approximate the (hopefully) stationary nature of the error structure. If non-stationary behaviour is indicated, then ARIMA models could be implemented.

Dealing with both heteroscedastic and serially correlated noise is easily accomplished by using the following general factorisation for the covariance matrix, :

Where denotes the heteroscedastic model parameters and the serial correlation model parameters. Clearly, . The matrix is the correlation matrix and . Implementing serially correlated data models is important for analysing time series with small sample intervals, where successive errors are likely to be positively correlated. Modelling the correlation applies the appropriate weight to each data point, improving the quality of the estimates. The presence of serial correlation implies that *block bootstrapping* is required. This is not currently implemented in the package.

Alternatively, the package could be extended to support higher order derivative regularisation algorithms. For example, the average second derivative is proportional to the inverse average radius of curvature and therefore, directly related to *smoothness*. Mathematically speaking, in principle this extension is quite straightforward, but requires adding additional methods to subclasses to support the necessary derivative calculations.

# References

1. E. Martinez-Laserna, V. I. Herrera, I. Gandiaga, A. Milo, E. Sarasketa-Zabala, H. Gaztanaga, Li\_ion Battery Lifetime Model’s Influence on the Economic Assessment of a Hybrid Electric Bus’s Operation, *World Electric Vehicle Journal*, 2018, 9, 28. [↑](#endnote-ref-1)
2. Suri, G., Onori, S., A control-oriented cycle-life model for hybrid-electric vehicle lithium-ion batteries, *Energy*, 2016, 96, 644-653. [↑](#endnote-ref-2)
3. K. Smith, J. Neubauer, E. Wood, M. Jun, A. Pesaran, Models for Battery Reliability and Lifetime, Applications in Design and Health Management, Advanced Vehicles and Fuels Research, NREL/PR-5400-58550, Battery Congress, Ann Arbor, MI, Apr 2013. [↑](#endnote-ref-3)
4. K. P. Burnham, D. R. Anderson, Model Selection and Multimodel Inference, 2002, 2nd edition, *Springer* [↑](#endnote-ref-4)