# **RasCAL Algorithms Toolbox (RAT) – overview.**

***1. Overview of the toolbox structure.***

The RAT toolbox is a Matlab toolbox for analysing neutron reflectivity data. Whilst it is written in Matlab, deployment is after ‘transpiling’ to C++ using [matlab coder](https://uk.mathworks.com/products/matlab-coder.html). Presently only a Matlab API into the toolbox exists, but a Python API is planned. Currently, conversion to C++ has been completed for the ‘reflectivity calculation’ onwards, as shown in the diagram below. Ultimately, everything up to the ‘main start’ section will be compiled, leaving only the input classes in Matlab.

A summary diagram of the basic structure of the toolbox is shown below:

Graphical user interface

Description automatically generated

Some of the toolbox has been documented with [’m2html’](https://www.artefact.tk/software/matlab/m2html/#:~:text=M2HTML%20is%20a%20powerful%20tool,customization%20of%20the%20output%20layout.). Hyperlinks to the relevant documentation are given from this document.

***2. Using the toolbox***

To initialise the toolbox from Matlab, edit addRatPaths.m in the /RAT folder. Change the path of ‘ratRoot’ on the first line. Run addRatPaths and this then sets the necessary paths for that matlab session.

There is a user manual in /Docs/User Manual. The examples discussed in the manual are in the /Docs/User Manual/Examples folder. There are further examples in the tests folder.

The toolbox can be viewed in terms of a series of layers, as suggested in the figure. Moving from the top level user interaction (Input Classes), to the lowest level calculations (‘common functions’). These can be summarised as follows…..

***1. Input Classes (Matlab API)***

These are what the user uses to interact with the toolbox and are described fully in the [user manual](User%20Manual/Chapter%201%20-%20Introduction.docx). The inputs are Matlab classes, with the attributes of the classes defining the model and data analysis actions, and these are set using class methods. There are always two input classes into the RAT toolbox:

*Project Class* – This class contains the details of the model and data to be analysed. This information is contained in the attributes of the class, and they are set using the class methods. Usage of the class is described in detail in Chapters 1 and 2 of the user manual.

*Controls Class* – Once the model has been defined, the controls class tells the toolbox what to do with the model. So for example, the choice of minimisation algorithm is done by selecting it in the controls class.

The input classes can be found in the ‘API’ folder, with the controls class in the ‘controlsDef\_class’ folder, and the model definition class in the ‘projectClass’ folder.

These are passed to the toolbox by calling RAT.m –

>> [outProblem,results] = RAT(problem,controls);

The outputs are a new projectClass updated with the results of the toolbox actions (e.g. fitted values for parameters), and a Results array, which contains the calculation results (e.g. simulated reflectivity and SLD profiles.

The m2html generated documentation for the input classes may be found [here](APIDocs/index.html)

***2. Main start classes.***

The first function called is RAT. This is the interface between the input classes, and the underlying toolbox (eventually in C). It’s main function is to strip out the relevant information into variable types that C can understand (because you can’t just pass a Mtalab class into C).

The main function that does this is:-

[problemDef,problemDef\_cells,problemDef\_limits,priors,controls] = RatParseClassToStructs\_new(problemDefInput,controls);

The two inputs are the two input classes. The outputs are a mixture of structs and ‘arrays of array’s. The reason for splitting them is that Coder will not allow variable sized arrays of variable sized arrays as fields in structs. It will allow variable sized (cell) arrays of variable sized arrays when not in a field in a struct. So, everything that can be pulled out into a struct is separated out into the ‘problemDef’ output. The arrays are put into the other outputs, as follows…

ProblemDef:

problemDef =

struct with fields:

contrastBacks: [2 1 2 1]

contrastBacksType: [1 1 1 1]

TF: 'standardTF'

resample: [0 0 0 0]

dataPresent: [1 1 1 1]

numberOfContrasts: 4

geometry: 'substrate/liquid'

contrastShifts: [1 1 1 1]

contrastScales: [1 1 1 1]

contrastNbas: [1 1 1 1]

contrastNbss: [1 2 1 2]

contrastRes: [1 1 1 1]

backs: [5.5000e-06 2.8000e-06]

shifts: 0

sf: 0.2300

nba: 0

nbs: [6.3500e-06 0]

res: 0.0300

params: [3 20 11 5 7 6.0000e-06 -4.0000e-07 3.0000e-06 1.4000e-06 10]

numberOfLayers: 4

modelType: 'layers'

contrastCustomFiles: [NaN NaN NaN NaN]

fitpars: []

otherpars: []

fitconstr: []

otherconstr: []

Note that many of these fields contain variable sized arrays of doubles. This is allowed - it’s arrays of arrays that are forbidden in fields.

problemDef\_cells:

All the variable sized cell arrays. This is always a [1 x 14] cell array:

problemDef\_cells =

1×14 cell array

Columns 1 through 9

{1×4 cell} {1×4 cell} {1×4 cell} {1×4 cell} {1×4 cell} {4×1 cell} {1×10 cell} {1×2 cell} {1×1 cell}

Columns 10 through 14

{1×1 cell} {1×1 cell} {1×2 cell} {1×1 cell} {1×1 cell}

These are always as follows:

%

% {1} - inputProblemDef.contrastRepeatSLDs

% {1 x nContrasts} array of cells

% Each cell is {1 x 2 double}.

%

% {2} - inputProblemDef.allData

% {1 x nContrasts} array of cells

% Each cell is {Inf x 3 double}

%

% {3} - inputProblemDef.dataLimits

% {1 x nContrasts} array of cells

% Each cell is {1 x 2 double}

%

% {4} - inputProblemDef.simLimits

% {1 x nContrasts} array of cells

% Each cell is {1 x 2 double}

%

% {5} - inputProblemDef.contrastLayers

% {1 x nContrasts} array of cells

% Each cell is {1 x Inf double}

%

% {6} - inputProblemDef.layersDetails

% {n x 1} array of cells

% Each cell is (1 x 5 double}

%

% {7} - inputProblemDef.paramNames

% {1 x nParams} array of cells

% Each cell is {1 x Inf char}

%

% {8} - inputProblemDef.backgroundsNames

% {1 x nBackgrounds} array of cells

% Each cell is {1 x Inf char}

%

% {9} - inputProblemDef.scalefactorNames

% {1 x nScales} array of cells

% Each cell is {1 x Inf char}

%

% {10}- inputProblemDef.qzshiftNames

% {1 x nShifts} array of cells

% Each cell is {1 x Inf char}

%

% {11}- inputProblemDef.nbairNames

% {1 x nNba} array of cells

% Each cell is {1 x Inf char}

%

% {12}- inputProblemDef.nbsrNames

% {1 x nNba} array of cells

% Each cell is {1 x Inf char}

%

% {13}- inputProblemDef.resolNames

% {1 x nNba} array of cells

% Each cell is {1 x Inf char}

%

% {14} - inputProblemDef.customFiles

% array of cells

% Each cell is {fName, lang, path} char

problemDef\_limits:

These are the parameter limits for all the parameters. Separated out for concenience. Again this is a struct with each field an array of doubles, in this case [n x 2]

problemDef\_limits =

struct with fields:

params: [10×2 double]

backs: [2×2 double]

scales: [0.0200 0.2500]

shifts: [-1.0000e-04 1.0000e-04]

nba: [0 0]

nbs: [2×2 double]

res: [0.0100 0.0500]

priors:

Bayesian priors for all the parameters in a struct. Even though the fields are cell arrays, each individual cell array is a consistent {char, string, double, double}, so coder allows this.

priors =

struct with fields:

paramPriors: {10×4 cell}

backsPriors: {2×4 cell}

resolPriors: {'Resolution par 1' ["uniform"] [0] [Inf]}

nbaPriors: {'SLD Air' ["uniform"] [0] [Inf]}

nbsPriors: {2×4 cell}

shiftPriors: {'Qz shift 1' ["uniform"] [0] [Inf]}

scalesPriors: {'Scalefactor 1' ["uniform"] [0] [Inf]}

Controls:

This simply is the relevant attributes of the controls class put into a struct:

controls =

struct with fields:

para: 'single'

proc: 'calculate'

display: 'iter'

tolX: 1.0000e-06

tolFun: 1.0000e-06

maxFunEvals: 10000

maxIter: 1000

populationSize: 20

F\_weight: 0.5000

F\_CR: 0.8000

VTR: 1

numGen: 500

strategy: 4

Nlive: 50

nmcmc: 0

propScale: 0.1000

nsTolerance: 1

calcSld: 0

repeats: 1

nsimu: 1000

burnin: 100

resamPars: [0.4000 50]

checks: [1×1 struct]

The exception in the ‘checks’ field, which is the logical true or false for each parameter declaring whether it is fittable. It is convenient downstream to have this here rather than in problemDef.

Following this, there is a declaration of an output class. **This is not currently being used.** But the function of this class is to throw an event, that will be monitored by a listener in an eventual GUI. The intention is to throw and event saying something like ‘update the graphs’, along with some data, and have the GUI respond appropriately, for example. Currently, all outputs from the toolbox are simple ‘fprintf’ statements to the command line.

Then, execution passes to RAT\_main. This will eventually be the entry point to the compiled C++ (i.e. Rat\_main\_mex). However, as shown in the figure we haven’t compiled to this point yet.

***3. Algorithm Selection Layer.***

In RAT\_main, a switch case chooses which algorithm is applied to fit the data in various ways. Details of the available algorithms can be found [here](MinimisersDocs/index.html). The relevant algorithm is set using ‘procedure’ in the controls input block. e.g.

>> controls.procedure = 'simplex'

controls =

controlsDef with properties:

parallel: 'single'

procedure: 'simplex'

calcSldDuringFit: 'no'

display: 'iter'

tolX: 1.0000e-06

tolFun: 1.0000e-06

maxFunEvals: 10000

maxIter: 1000

The various options in controls are:

‘calculate’ – No fit. Do a single calculation of the simulated reflectivity only.

‘simplex’ – Fit the data using a Nelder-Mead simplex. This used the Matlab function ‘fminsearch’.

‘DE’ – Differential Evolution genetic Algorithm. 3rd party function as described in the comments of ‘RAT\_deopt’.

‘Bayes’ – Delayed Rejection Adaptive Metropolis Bayesian inference. Slightly added from ‘mcmcstat’, described [here.](https://mjlaine.github.io/mcmcstat/)

‘NS” – Nested sampling algorithm for Bayesian model selection. Slightly adapted version of ‘matlab multinest’ described [here.](https://github.com/mattpitkin/matlabmultinest)

Each of the algorithms uses the Target Functions from the next layer to calculate the reflectivity and goodness of fit.

***4. Target Function– ‘Reflectivity Calculation’***

Each algorithm calculates the reflectivity and SLD profiles, and goodness of fit using ‘reflectivity\_calculation’. The toolbox is compiled from this point onwards, but before ‘reflectivity\_calculation’ itself, ‘reflectivity\_calculation\_wrapper’ is called which contains a switch/case allowing for either the mex or the matlab to be chosen for debugging purposes.

Documentation for the reflectivity calculation can be found [here.](RefCalcDocs/index.html)

The first layer of the calculation is a switch/case deciding between 4 different calculation types, which we call ‘Target Functions’ (TF’s). Only 1 TF type is available in phase 1. These directly represent the 4 main types of reflectivity experiment that are done at ISIS. The options will be (green are not phase 1):

standardTF\_reflectivityCalculation – Non-polarised, non-absorbing neutron reflectivity. This covers the bulk of ISIS reflectivity experiments

standardTFAbs\_reflectivityCalculation – Same as the first option, but with absorption terms. Most useful for x-rays.

oilWaterTF\_reflectivityCalculation – wit Experiments carried out at th eiol-water interface.

domainsTF\_reflectivityCalculation – eExperiments where the samples consist of domains which are larger than the transverse neutron coherence length.

of reflectivity. All the TF types will use pretty much the same basic calculatin functions, grouped in different ways. These can be found in the ‘common’ folder.

***Calcualtion types within the target functions.***

Having chosen the experiment type, there are always three options for the types of model.

Standard Layers – A simple method of describing an interface as stratified layers (see Abeles matrix formalism [here](https://en.wikipedia.org/wiki/Transfer-matrix_method_(optics))). This is all hard coded, and the actual layers used are defined in the input classes.

Custom Layers – Also a layers calculation, but this time the layers are defined by a used supplied ‘custom layers’ function. This gives a lot of flexibility in how the model can be defined.

Custom XY – In this case, the user provides a function which builds an arbitrary XY profile for describing the interface.

The ability to define custom models, either as layers or XY profiles is a unique and very powerful feature of RasCAL / RAT. It gives complete flexibility to the user as to a modelling strategy, and doesn’t limit the available approaches to anything dictated by the developer. See the user manual for further details.

***Parallelisation.***

Neutron reflection is almost always a simultaneous fit of multiple ‘contrasts’ (datafiles). Each datafile is obviously a number of points. So for each calculation, there is a choice of whether to parallelise over the outer loop (contrasts), or the inner loop (points). This is selected in the controls class by the user. In the code, we do this by having three versions of the same calculation routines, rather than trying to make the parallelisation on the loops. This may or may not be best practice.

In other words, there a 9 possible routes through for the standardTF reflectivity calculation, depending on calculation type and parallelisation, but each will use the same basic set of common functions underneath.

standardTF\_reflectivityCalculation

standardTF\_stanlay reflectivityCalculation

standardTF\_custlay reflectivityCalculation

standardTF\_custXY reflectivityCalculation

stanlay\_single

stanLay paraPoints

stanLay paraContrasts

custlay\_single

custlay paraPoints

custlay paraContrasts

custXY\_single

custXY paraPoints

custXY paraContrasts

Common Calculation Functions