Documentation/Starter Guide for the Robust Chauvenet Outlier Rejection Algorithm

Nick Konz

February 18, 2019

For RCR version 2.3.2

Prerequisites

1. A basic knowledge (at least) of C++ is recommended.
2. Make sure that the following files are included in your project (besides your file with `int.main()`): (available from the RCR webpage)
   Source files
   (a) `RCR.cpp`
   (b) `NonParametric.cpp`
   (c) `FunctionalForm.cpp`
   (d) `MiscFunctions.cpp`

   Header files
   (a) `RCR.h`
   (b) `NonParametric.h`
   (c) `FunctionalForm.h`
   (d) `MiscFunctions.h`
3. Other useful files are included, such as `Demo.h` and `Demo.cpp`, which include example model functions for functional RCR, and `RCRstarterfile.cpp`, which has ready-to-run code utilizing RCR.

1 Single-Value RCR

The simplest RCR algorithm to use; the basic steps for this are as follows (code example at the bottom):

1. Instantiate an object of the `RCR` class. It's recommended, but optional to specify the rejection technique from the following (see figure 19 in the RCR paper) as an argument of the constructor:

   (a) `SS_MEDIAN_DL`
   (symmetric uncontaminated distribution with two-sided contaminants)

   (b) `LS_MODE_68`
   (symmetric uncontaminated distribution with one-sided contaminants)

   (c) `LS_MODE_DL`
   (symmetric uncontaminated distribution with a mix of the above types of contaminants)

   (d) `ES_MODE_DL`
   (Mildly asymmetric uncontaminated distribution and/or very low number of data points)

   If no technique is specified (the constructor argument is held blank) then `LS_MODE_DL` will be used by default.

2. Depending on whether you want to perform regular RCR or Bulk Rejection RCR, and/or if your data is weighted or not, call the `RCR` class method `.performRejection()` or `.performBulkRejection()` as follows: if non-weighted, the sole argument will be an `std::vector <double>` of your data points, `y` (or whatever you'd like to name it). If weighted, the two arguments will be in order, `w` and `y`, where `w` is a vector of the weights of the `y`, in the same order as that of `y`.

   Two basic examples: (within `int.main()`)

```
std::vector <double> y, w;

RCR rcr = RCR(SS_MEDIAN_DL);
rcr.performBulkRejection(w, y);

//or

RCR rcr = RCR();
rcr.performRejection(y);
```

1.1 Accessing Results of RCR

The vector that contains the "flags" on your data, i.e. "true" for nonrejected and "false" for rejected, is `rcr.result.flags`. The nonrejected data is `rcr.result.cleanY`, while the rejected is `rcr.result.rejectedY`. Other results are also members of `rcr.result`.

2 Functional Form/Parametric RCR

The most complex RCR algorithm to use, it is fairly involved, and there are a number of things to specify. The basic steps are as follows (multiple code examples at the bottom). Following that section, there are sections with further explanation on how to implement the usage of prior distributions/information on the parameters, as well as custom functions for the $\bar{x}$, $\overline{\log x}$ type of variables often seen in proper model functions.

1. First, you must define your model function of the form $y = f(\vec{x}, \vec{\theta})$, where $\vec{x}$ is either a single independent variable x, or a vector of independent variables $\vec{x} = (x_1, x_2, \cdots x_i)$; $\vec{\theta}$ is a vector of the model parameters of the function (to be determined by RCR). For example, with the function $y = be^{m(x-0.5)}$, we have that $\vec{x} \equiv x$ and $\vec{\theta} = (b, m)$ (or however else you want to order the parameters). **You'll need to define your function somewhere in your codebase so that it returns a double; it'll have different arguments depending on whether x is 1-dimensional, or an n-dimensional vector.** In this 1D case, your function will need to have two arguments: a double x and a vector of doubles corresponding to $\vec{\theta}$, the parameters (even if there is only a single parameter). An example of this, using the same function, is as follows:

```
double func(double x, std::vector <double> params)
{
    double b = params[0];
    double m = params[1];

    return b * std::exp(m *(x - 0.5));
}
```

   To show the ND case, define $y = a_0 + a_1x_1 + a_2x_2$, where now, $\vec{x} = (x_1, x_2)$ and $\vec{\theta} = (a_0, a_1, a_2)$. Now, x is a vector, so in the code, you'd have

```
double NDfunc(std::vector <double> x, std::vector <double> params)
{
    double a0 = params[0];
    double a1 = params[1];
    double a2 = params[2];

    double x1 = x[0];
    double x2 = x[1];

return a0 + a1*x1 + a2*x2;
}
```

2. The second important item(s) that must be provided somewhere in your code are the partial derivatives of your aforementioned function with respect to each of the parameters (for use in the Jacobian within the modified Gauss-Newton algorithm used in the code). Using the same exponential example, you'd have that $\frac{\partial f}{\partial b} = e^{m(x-0.5)}$ and $\frac{\partial f}{\partial m} = (x-0.5)be^{m(x-0.5)}$. **In your code, these need to be defined as seperate functions, taking arguments and returning values in the exact same fashion as how the corresponding functions should be defined** (again dependent on whether there is > 1 independent variables or not). As an example, the above partials could appear in the code as follows:

```
double partial1(double x, std::vector <double> params)
{
   double b = params[0];
   double m = params[1];

   return std::exp(m *(x - 0.5));
}

double partial2(double x, std::vector <double> params)
{
   double b = params[0];
   double m = params[1];

   return (x-0.5) * b * std::exp(m *(x - 0.5));
}
```

Be careful that you maintain the same order of the parameters within the parameter vectors for all of your definitions; if they aren't consistent, the code won't work properly.

3. Next, you'll need to create a vector of your partial derivative functions. The syntax for this is a bit unusual, and will again differ on whether your x is 1D, or multi-dimensional. Make sure to order the partials (corresponding to what they are differentiating with respect to) in the same order as your chosen definition for the parameter vector as used in the rest of your own code. For the 1-dimensional example, you'd have something similar to

   ```
   std::vector <double(*)(double, std::vector <double>)> partialsvector
    =  partial1, partial2 ;
   ```

   and for the ND case (such as the example used above) you'd have something like

   ```
   std::vector <double(*)(std::vector <double>, std::vector <double>)>
   NDpartialsvec =  NDpartial1, NDpartial2, NDpartial3 ;
   ```

4. Finally, you'll need to create two more things:

   (a) A chosen `double` tolerance for the convergence of the Gauss-Newton algorithm (the lower it is, the longer the process may take, but the more precise it will be. 0.01 is a good, balanced number to start with.

   (b) An initial guess vector for the true values of the parameters in the model parameters vector, also defined as an `std:vector <double>` . Dependent on the situation, if your guess is too far off of the true values, you may not get the algorithm to always converge, which could mess up your results, so try to determine the best guess possible.

5. Next, on to the data: you'll need your `std:vector <double>` vectors of your $\vec{y}$ data. If there is a single independent variable x, the x data vector will also be a `std:vector <double>`. In the ND case, your x data will instead be a vector of these vectors (`std:vector < std::vector < <double> >`. Be sure that your inner x vectors within the big vector have their data ordered in the same manner as you defined your function and your partials. If you desire to have unequal weights on your data, you'll to create some weight vector w (also a `std:vector <double>`). You can also, instead of using weights, put error bars/uncertainties on your y data, which need to be contained in some `std:vector <double>` σ_y. If desired, you run weighted RCR with these error bar values by converting them to weights with the relation $w_i \propto 1/\sigma_{y,i}$. Also, be sure that your y, optional σ_y, x and optional weights w (also a `std:vector <double>`) are in the correct order with respect to each other.

6. Finally, you are ready to run FunctionalForm RCR. First, instantiate an object of the `FunctionalForm` class with the following arguments if unweighted:

```
std::vector <double> x, guess;
double tolerance;

FunctionalForm model = FunctionalForm(func, x, y, partialsvector,
 tolerance, guess);
```

or if weighted,

```
FunctionalForm model = FunctionalForm(func, x, y, partialsvector,
 tolerance, guess, weights);
```

or, if with error bars,

```
FunctionalForm model = FunctionalForm(func, x, y, sigma_y,
 partialsvector, tolerance, guess);
```

7. Next, instantiate an object of the RCR class with your chosen (or non-chosen) rejection technique.
8. Next call the method `.setParametricModel()` on your RCR object, with the argument of the method being your FunctionalForm object.
9. Finally, perform rejection (or bulk rejection) the same as the other cases, by calling `performRejection` or `performBulkRejection` on the RCR object, either with or without weights. To summarize, here are two examples:

```
FunctionalForm model = FunctionalForm(func, x, y, partialsvector,
 tolerance, guess, weights);
RCR rcr = RCR(SS_MEDIAN_DL);
rcr.setParametricModel(model);
rcr.performBulkRejection(weights, y);

//or

FunctionalForm model = FunctionalForm(func, x, y, sigma\_y,
 partialsvector, tolerance, guess);
RCR rcr = RCR(LS_MODE_68);
rcr.setParametricModel(model);
rcr.performRejection(y);
```

10. **A note about the ND case** (> 1 independent variable): you will need to define your `x` variable as an `std::vector < std::vector <double> >`, because each element in `y` will correspond to a specific vector of `x` values. For example, if you have 20 data points for a model function of 3 variables, `x` Will be a 2D vector of dimensions (20, 3).

2.1 Accessing Results of Functional RCR

Same with single-value RCR, the vector that contains the "flags" on your data, i.e. "true" for nonrejected and "false" for rejected, is `rcr.result.flags`. The nonrejected data is `rcr.result.cleanY`, while the rejected is `rcr.result.rejectedY`. Other results are also members of `rcr.result`.

In addition to these results, you can also access the final fitted model function parameters with `model.parameters` (continuing off of the example above).

2.2 Using Custom Prior Distributions for Model Parameters

2.2.1 Initializing Your Priors

If you have some prior knowledge of the distribution(s) of the function parameter(s) (before necessarily having any data), this can be implemented into functional RCR. The code has support for the following priors:

1. Gaussian/Normal distributions with different standard deviations and means for some or all parameters

2. Constraints/boundaries on some or all parameters

3. A mix of the above two

4. Custom priors (see below).

Bounded Priors

If you'd like to put a hard upper bound, lower bound, both, or neither on some or all of your model parameters, the first thing that needs to be done is creating a `Priors` object. The `Priors` object's constructor first requires an enum that can be `CONSTRAINED`, `GAUSSIAN`, `MIXED`, or `CUSTOM`, which specifies the type of priors that you're using. In the case of bounded/constrained priors, you also need to specify the upper and lower bound of each model parameter (if they have one or both). You must create a 2D vector, which contains a vector of doubles, size 2, for each of the parameters. In each one of these-2 vectors, you'll need the first element to be the lower bound of the parameter, with the second element being the upper bound. If one or both of the bounds is not needed (i.e. you only want the prior to be bounded above or below, or not bounded at all), put a `NAN` in for the bound. Make sure that the order in each parameter's vector is "lower bound, upper bound", and that the overall 2D vector containing each parameter's associated vector, is ordered with respect to the parameters the same way that you have been ordering the parameters within the rest of your code.

Here's a simple example: If you want to have your first parameter a_0 to have no bounds (flat prior), and you want your second parameter to be constrained to $[1, \infty)$, the following code should be used to create the `Priors` object:

```
std::vector < std::vector <double> > boundedParams = { {NAN, NAN}, {1.0, NAN
   } };  //params specifiying the bounds (priors) on the model function
   parameters

Priors testPriors = Priors(CONSTRAINED, boundedParams);
```

Gaussian Priors

You can also give some or all of the parameters Gaussian distributions, in a similar manner to creating bounded priors. Follow the same general method described in the above section for bounded priors, except for instead of creating vectors of size two that specify the lower and upper bounds, respectively, of the priors of each of the parameters, the vectors will contain the mean and the standard deviation of the each of the parameter's priors. Also, you'll use the `GAUSSIAN` enum to instantiate the priors object. In this case, if you don't want a Gaussian prior on all of your parameters, simply put `NAN` for that parameter's mean and standard deviation. This is more clear given an example:

If you want to have a Gaussian prior for your first parameter, with mean 1.0 and standard deviation 2.0, while keeping your second parameter free of any prior, you would do the following:

```
std::vector < std::vector <double> > gaussianParams = { {1.0, 2.0}, {NAN,
   NAN} };  //params specifiying the gaussian(s) on the model function
   parameters

Priors testPriors = Priors(GAUSSIAN, gaussianParams);
```

Gaussian AND Bounded Priors

This type of prior simply combines the last two, so that for any of your parameters, you can specify a Gaussian AND bound(s), just a Gaussian, just bound(s), or neither. This type of prior follows logically from the first two, where now, you'll use the enum `MIXED` to create your priors object, now including two 2D vectors: the first specifying the Gaussian prior(s), the second specifying the bounded prior(s).

Let's show another example. Now, we have a model with three parameters. We'd like to bound the first parameter with $[0.0, \infty)$ (but with no Gaussian on it). The second parameter will have a Gaussian prior with mean of -1.4 and standard deviation 2.1, bounded to $[-5, 5]$, and our third and last parameter will be bounded with $[1, 10]$ (no Gaussian). This priors object would be initialized as follows:

```
std::vector < std::vector <double> > gaussianParams = { {NAN, NAN}, {-1.4,
    2.1}.{NAN, NAN}; //params specifiying the Gaussian priors on the model
    function parameters
std::vector < std::vector <double> > boundedParams = { {0.0, NAN}, {-5.0,
    5.0}, {1.0, 10.0 };  //params specifiying the bounds (priors) on the
    model function parameters

Priors testPriors = Priors(MIXED, gaussianParams, boundedParams);
```

Custom Priors

The last type of priors, which is the most general but the most complicated, is custom priors. The Gaussian and Bounded priors were simply special cases of this (where we created your prior functions for you). In the most general case, you can create your prior distribution(s). To do this, create a function that takes in a vector of model parameters (i.e. in the linear model case, the the slope intercept and the slope), a vector of the weights of those parameters, and returns those parameters weights modified by the priors, be it some of them multiplied by some function, bounded by within some range, multiplied by a custom Gaussian or other distribution for each parameter, etc. For example, here is a custom prior that stipulates that

1. the first parameter a_0 has a prior distribution of $e^{-|a_0|}\cos^2(a_0)$
2. the second parameter a_1 must be bounded by $(-\infty, 2.0)$
3. the third parameter a_2 has no prior.

In order to implement this prior, you would create the following function. Note that the weights of the parameters must be modified, not the values of the parameters themselves; this is effectively equivalent to modifying the probability of the parameters having whatever values they have, not modifying the values themselves.

```
std::vector <double> priorsFunction(std::vector <double> parameters, std::
    vector <double> parameter_weights)
{
   double a0 = parameters[0];
   double a1 = parameters[1];
   double a2 = parameters[2];

   double w_a0 = parameter_weights[0];
   double w_a1 = parameter_weights[1];
   double w_a2 = parameter_weights[2];

   std::vector <double> prioredWeights(parameter_weights.size(), 1.0);

   prioredWeights[0] = w_a0 * std::exp(-1.0 * std::abs(a0)) * std::pow(std::
    cos(a0), 2.0);
   if (a1 > 2.0){
      prioredWeights[1] = 0.0; //if the parameter is outside of it's bounds,
     modify it's weight to be zero (i.e. it's excluded from calculations)
   }

   prioredWeights[2] = w_a2; //no priors: unmodified weight

   return prioredWeights;
}

//within int main()
Priors testPriors = Priors(CUSTOM, priorsFunction);
```

2.2.2 Using Your Priors with Functional RCR

Now that you've instantiated your priors object, using it with RCR is easy; it's just an additional argument (which follows either `guess` (if unweighted data) or `w` (if weighted data) in the `FunctionalForm` constructor, as shown below.

```
FunctionalForm model = FunctionalForm(func, x, y, partialsvector, tolerance,
     guess, w, testPriors);
RCR rcr = RCR(SS_MEDIAN_DL); //chosen rejection technique
rcr.setParametricModel(model);
rcr.performBulkRejection(w, y); //weighted bulk rejection
```

2.3 Using Models With Custom $\bar{x}$, $\overline{\log x}$, or Similar Variable(s)

Many good statistical models include a properly linearized and weighted version of $\bar{x}$, $\overline{\log x}$, etc. that measures the mean of the independent variable data. For example, a proper linear model should be of the form $y(x) = a_0 + a_1(x - \bar{x})$, with $\bar{x} = \sum_i w_i x_i / \sum_i w_i$, where i indexes the data points. On the other hand, an exponential model should be of the form $y(x) = a_0 10^{a_1(x-\bar{x})}$, now with $\bar{x} = \sum_i w_i x_i y^2(x_i) / \sum_i w_i y^2(x_i)$ (for more detail on this discussion, see §8.3.5 in the RCR paper.) Yet another example of this is the power law model (all of these models and more can be found in `Demo.cpp` and `Demo.h`), of the form $y(x) = a_0 \left(x/10^{\overline{\log x}}\right)^{a_1}$ with $\overline{\log x} = \sum_i w_i (\log x_i) y^2(x_i) / \sum_i w_i y^2(x_i)$ (where log is base 10).

One useful feature of Functional RCR is that if your model has some type of “bar” variable akin to this, given some initial guess for it’s value, it will be iteratively updated as more data is rejected and RCR progresses, improving both the model fitting, and giving you a better estimate for it’s value. Furthermore, you can use any function to calculate the value of this “bar” variable(s), (multiple in the case of more than one independent model function “x” variable), such as the ones that can be found at the bottom of `MiscFunctions.cpp`, or any custom function that you create.

2.3.1 Creating and Using Your Custom “Bar” Function

Let’s say that you want to fit a 1D linear model $y(x) = a_0 + a_1(x - \bar{x})$ to weighted data with a custom function for $\bar{x}$, and prior distributions on the parameters. First, note that in order for any arbitrary model function to work with a custom “bar” variable, we declared a `double bar` in `FunctionalForm.h`, which will be the “bar” value that is iteratively updated (and reset upon the start of a new instance of Functional RCR) within your model. In the case of an ND model, the variable `std::vector <double> bar_ND` is declared within `FunctionalForm.h` for the same purpose. To see this in action, let’s create our model function and partials:

```
double func_linear(double x, std::vector <double> params) {
   double a0 = params[0];
   double a1 = params[1];

   return a0 + a1 * (x - bar);
}



double partial1_linear(double x, std::vector <double> params) {

   return 1.0;
}

```

```
double partial2_linear(double x, std::vector <double> params) {

   return x - bar;
}

std::vector <double(*)(double, std::vector <double>)> partialsvector_linear
   = { partial1_linear, partial2_linear };
```

As shown, the model utilizes the global `bar` variable described.

Now, create your function to define `bar`. **Note that the arguments and output of these functions must be the same for any custom "bar" value**, with the necessary `double`'s converted to `std::vector <double>`'s in the case of multiple independent variables/"bar" values for the model. The arguments, in order, need to be the x vector (1D in the case of one independent variable, 2D in the case of multiple), the weights vector (which will automatically default to just being equal weights in the unweighted case), the (function pointer to the) model function, and a vector of the values of the model parameters used to evaluate the model. In the 1D case, the output will be the calculated `bar`, a `double`, and in the multiple independent variable case, the output will be the calculated `bar_ND`, an `std::vector <double>`. While some "bar" variables may not use all of these arguments (such as the one below), they still need to be included within the declaration of the "bar" function.

```
double getAvg(std::vector<double> x, std::vector <double> w, double(*f)(
   double, std::vector <double>), std::vector<double> params) {
   double uppersum = 0.0;
   double lowersum = 0.0;

   for (int i = 0; i < x.size(); i++) {
      uppersum += (x[i] * w[i]);
      lowersum += w[i];
   }

   return (uppersum / lowersum);
}
```

From here, think of a initial guess for `bar` (or `bar_ND`), and you can run Functional RCR as follows, given the proceeding example. We've also included the usage of priors, to showcase the combination of the two.

```
double xb_guess;

FunctionalForm model = FunctionalForm(func_linear, x, y,
    partialsvector_linear, tolerance, guess, w, testPriors, getAvg, xb_guess
    );
RCR rcr = RCR(SS_MEDIAN_DL); //setting up for RCR with this rejection
    technique
rcr.setParametricModel(model);
rcr.performBulkRejection(w, y); //running Bulk rejection RCR
```

The final calculated value for `bar`/`bar_ND`, (following the example shown above), is simply `model.bar_result` or `model.bar_ND_result`, respectively.

3 Non-Parametric RCR

The basic steps for this are as follows (code example at the bottom):

1. In your `int.main()`, Instantiate an object of the `NonParametric` class.
2. Instantiate an object of the `RCR` class. Specify the rejection technique, if wanted.
3. Call the `RCR` class method `.setNonParametricModel()` on your `RCR` object, with your `NonParametric` object as the method's sole argument.
4. Depending on whether you want to perform regular RCR or Bulk Rejection RCR, and/or if your data is weighted or not, call the `RCR` class method `.performRejection()` or `.performBulkRejection()`.

Two basic examples: (within `int.main()`)

```
NonParametric model = NonParametric();
RCR rcr = RCR(SS_MEDIAN_DL);
rcr.setNonParametricModel(model);
rcr.performBulkRejection(w, y);

//or:

NonParametric model = NonParametric();
RCR rcr = RCR();
rcr.setNonParametricModel(model);
rcr.performRejection(y);
```