Documentation/Starter Guide for the Robust Chauvenet Outlier Rejection Algorithm

January 6, 2019

Prerequisites

1. At least basic knowledge of C++ 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) `MiscFunction.cpp`

   Header files
   (a) `RCR.h`
   (b) `NonParametric.h`
   (c) `FunctionalForm.h`
   (d) `MiscFunction.h`

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()`)

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

or

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

2 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);
```

3 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 is a further explanation on how to implement the usage of prior distributions/information on the parameters.

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 the parameters are ordered the same within the parameter vectors for all of your definitons; if they aren't consistent, the code won't work properly.

3. Next, you'll need to create a vector of your partial 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. $1e-3$ 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, and the corresponding vector of error bars $\vec{\sigma}_y$. 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. Also, be sure that your y, $sigma_y$, x and optional weights (also a `std:vector <double>`) are in the correct order with respect to each other.

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

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

   or if weighted,

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

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, sigma_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);
```

3.1 Using 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).

3.1.1 Gaussian Priors

To use priors that give some or all of the parameters Gaussian distributions,
For the parameters of which you don't wish to add priors,

3.1.2 Constrained Priors

examples: variance (can't be lower than 0), probability (must be [0,1])

3.1.3 Gaussian and/or Constrained Priors

3.1.4 Custom Priors

If you have custom priors, simply create a function that takes in a vector of parameters, and returns those parameters modified by the priors, be it some of them multiplied by zero if they are outside of their constraints, 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 must be greater than zero
2. the second parameter a_1 must be drawn from a Gaussian of $\mu = 1.0$, $\sigma = 2.0$.
3. the third parameter a_2 has no prior.

To implement this prior, you would create the following function: