rcr Documentation

Release 2.4.7

Nick Konz

Jul 09, 2020

User Guide

RCR, or Robust Chauvenet Rejection, is advanced, and easy to use, outlier rejection. Originally published in Maples et al. 2018, this site will show you how to use RCR in Python. RCR can be applied to weighted data and used for model-fitting, and we have incorporated rejecting outliers in bulk to have the best of both computational efficiency and accuracy (see the *Installation Guide*).

RCR has been carefully calibrated, and extensively simulated. It can be applied to samples with both large contaminants and large contaminant fractions (sometimes in excess of 90% contaminated). Finally, because RCR runs on a C++ backend, it is quite fast.

CHAPTER 1

Introduction

The simplest form of outlier rejection is sigma clipping, where measurements that are more than a specified number of standard deviations from the mean are rejected from the sample. This number of standard deviations should not be chosen arbitrarily, but is a function of your sample's size. A simple prescription for this was introduced by William Chauvenet in 1863. Sigma clipping plus this prescription, applied iteratively, is what we call traditional Chauvenet rejection.

However, both sigma clipping and traditional Chauvenet rejection make use of non-robust quantities: the mean and the standard deviation are both sensitive to the very outliers that they are being used to reject. This limits such techniques to samples with small contaminants or small contamination fractions.

Robust Chauvenet Rejection (RCR) instead first makes use of robust replacements for the mean, such as the median and the half-sample mode, and similar robust replacements that we have developed for the standard deviation.

CHAPTER 2

Basic Usage Example

Here's a quick example of RCR in action: we have a dataset of $N = 1000$ measurements, 85% of which are contaminants. The contaminants are sampled from one side of a Gaussian/normal distribution with standard deviation $\sigma = 10$, while the uncontaminated points are from a regular, symmetric Gaussian with $\sigma = 1$. Both distributions are centered at $\mu = 0$.

The question is, how can we recover the μ and σ of the underlying distribution, in the face of such heavy contamination? The example below shows how to do it with RCR.

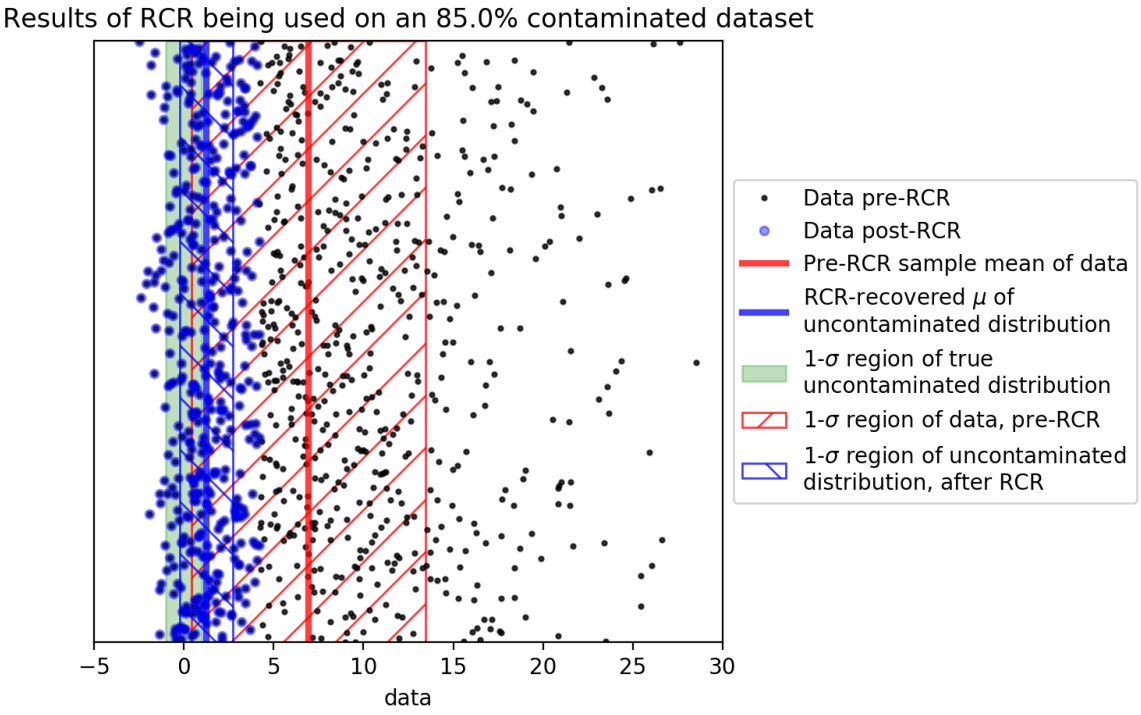
```
import numpy as np
import rcr

np.random.seed(18318) # get consistent random results

N = 1000 # total measurement count
frac_contaminated = 0.85 # fraction of sample that will be contaminated

# symmetric, uncontaminated distribution
mu = 0
sigma_uncontaminated = 1
uncontaminated_samples = np.random.normal(mu, sigma_uncontaminated,
    int(N * (1 - frac_contaminated)))

# one-sided contaminants
sigma_contaminated = 10
contaminated_samples = np.abs(np.random.normal(mu, sigma_contaminated,
    int(N * frac_contaminated)))

# create whole dataset
data = np.concatenate((uncontaminated_samples, contaminated_samples))
np.random.shuffle(data)

# perform RCR
# initialize RCR with rejection technique:
# (chosen from shape of uncontaminated + contaminated distribution)
```

(continues on next page)

(continued from previous page)

```
r = rcr.RCR(rcr.LS_MODE_68)
r.performBulkRejection(data) # perform outlier rejection

# View results
cleaned_data = r.result.cleanY
cleaned_mu = r.result.mu
cleaned_sigma = r.result.stDev


# plot data
import matplotlib.pyplot as plt

ydata = np.random.uniform(0, 1, N) # project randomly into 2D for better visualization
plt.figure(figsize=(8,5))
ax = plt.subplot(111)
ax.plot(data, ydata, "k.", label="Data pre-RCR", alpha=0.75, ms=4)
ax.plot(cleaned_data, ydata[r.result.indices], "bo",
    label="Data post-RCR", alpha=0.4, ms=4)

# plot results
cont_mean = np.mean(data)
cont_sigma = np.std(data)

ax.axvspan(mu - sigma_uncontaminated, mu + sigma_uncontaminated, color='g',
    alpha=0.25, label="1-$\sigma$ region of true\nuncontaminated distribution")
ax.axvline(x=cont_mean, c='r', lw=3, ls="-", alpha=0.75,
    label="Pre-RCR sample mean of data")
ax.axvspan(cont_mean - cont_sigma, cont_mean + cont_sigma,  color='r',
    fill=False, alpha=0.75, hatch="/", label="1-$\sigma$ region of data, pre-RCR")

ax.axvline(x=cleaned_mu, c='b', lw=3, ls="-", alpha=0.75,
    label="RCR-recovered $\mu$ of\nuncontaminated distribution")
ax.axvspan(cleaned_mu - cleaned_sigma, cleaned_mu + cleaned_sigma, color='b',
    fill=False, alpha=0.75, hatch= '\\',
    label="1-$\sigma$ region of uncontaminated\ndistribution, after RCR")

plt.xlim(-5, 30)
plt.ylim(0, 1)
plt.xlabel("data")
plt.title("Results of RCR being used on an" +
    " {}% contaminated dataset".format(frac_contaminated*100))
plt.yticks([])

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])

ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```

Output:



For a more in-depth explanation of using RCR for this type of one-dimensional outlier rejection, see *Rejecting 1D Outliers*.

CHAPTER 3

License and Attribution

3.1 The Paper

The original paper for RCR can be found here. If you use RCR, please cite it as:

```
@article{maples2018robust,
    title={Robust Chauvenet Outlier Rejection},
    author={{Maples}, M.P. and {Reichart}, D.E. and {Konz}, N.C. and {Berger}, T.A.
↪and {Trotter}, A.S. and {Martin}, J.R. and {Dutton}, D.A. and {Paggen}, M.L. and
↪{Joyner}, R.E. and {Salemi}, C.P.},
    journal={The Astrophysical Journal Supplement Series},
    volume={238},
    number={1},
    pages={2},
    year={2018},
    publisher={IOP Publishing}
}
```

and use it according to the license (the essential point here is just to contact us if you want to use RCR for commercial purposes).

3.1.1 Installation Guide

Python

Linux and macOS

The easiest way to install RCR into Python is with `pip`:

```
python3 -m pip3 install rcr
```

That's it; you're good to go from here!

Windows

Before installing, you'll need to have **Microsoft Visual C++ 14.0**, found under the Microsoft Visual C++ Build Tools. If that doesn't work, you may need the latest Windows SDK. (Both can be installed through the Visual Studio Installer.)

From here, install via `pip` from the terminal as usual:

```
python3 -m pip3 install rcr
```

(If you tried to install via `pip` without first installing the above requirements, `pip` would probably tell you to do so.)

C++

Because the RCR Python library uses pybind11 to wrap the original C++ source code seamlessly into Python, all of the speed of C++ is available through the Python library. However, if the C++ source code is desired, it can be found at the Github repository in the `src` directory (`RCR_python.cpp` is only used for wrapping the C++ code into the rcr Python library, so it can be ignored). Documentation specific to the C++ codebase can be found within the directory `docs/cpp_docs` of this repository.

3.1.2 Frequently Asked Questions

What is Bulk Rejection?

> The RCR algorithm is iterative; without bulk rejection, only one outlier is rejected per iteration. *With* bulk rejection, multiple outliers can be rejected per iteration, with hardly any negative effect on final results. This creates a huge speed-up, especially for large datasets. See Section 5 of *The Paper* for more info.

3.1.3 API Reference

RCR (Robust Chauvenet Outlier Rejection) Package API Details.

class rcr.**FunctionalForm**

class. Class used to initialize functional form/model-fitting RCR (see *Rejecting Outliers While Model Fitting*).

Constructor arguments:

Parameters

- **f** (*function*) – Model function $y(\vec{x}|\vec{\theta})$ to fit data to while performing outlier rejection, where $\vec{x}$ is an n-dimensional list/array_like (or float, for 1D models) of independent variables and $\vec{\theta}$ is an M-dimensional list/array_like of model parameters. Arguments for `f` must follow this prototype:

  param x Independent variable(s) of model

  type x float or 1D list/array_like

  param params Parameters of model

  type params list/array_like, 1D

  returns **y** – Model evaluated at the corresponding values of `x` and `params`.

  rtype float

- **xdata** (*list/array_like, 1D or 2D*) – n-dimensional independent variable data to fit model to. For 1D models ($n = 1$), this will be a 1D list/array_like, while for n-D models, this will be a 2D list/array_like where each entry is a list/array_like of length n.
- **ydata** (*list/array_like, 1D*) – Dependent variable (model function evaluation) data to fit model to.
- **model_partials** (*list of functions*) – A list of functions that return the partial derivatives of the model function `f` with respect to each, ordered, model parameter $\vec{\theta}$ (See *Rejecting Outliers While Model Fitting* for an example). Arguments for each one of these functions must follow this prototype (same as for the model function `f`):

  param x Independent variable(s) of model

  type x float or 1D list/array_like

  param params Parameters of model

  type params list/array_like, 1D

  returns **y** – Derivative of model (with respect to given model parameter), evaluated at the corresponding values of `x` and `params`.

  rtype float
- **guess** (*list/array_like, 1D*) – Guess for best fit values of model parameters $\vec{\theta}$ (for the fitting algorithm).
- **weights** (*list/array_like, optional, 1D*) – Optional weights to be applied to dataset (see *Weighting Data*).
- **error_y** (*list/array_like, optional, 1D*) – Optional error bars/y-uncertainties to be applied to dataset (see *Data with Uncertainties and/or Weights*).
- **tol** (*float, optional*) – Default: `1e-6`. Convergence tolerance of modified Gauss-Newton fitting algorithm.
- **has_priors** (*bool, optional*) – Default: `False`. Set to `True` if you're going to apply statistical priors to your model parameters (see *Applying Prior Knowledge to Model Parameters (Advanced)*; you'll also need to create an instance of *rcr.Priors* and set the `priors` attribute of this instance of `FunctionalForm` equal to it).
- **pivot_function** (*function, optional*) – Default: `None`. Function that returns the pivot point of some linearized model (see *Automatically Minimizing Correlation between Linear Model Parameters (Advanced)*). Must be of the form/prototype of:

  param xdata n-dimensional independent variable data to fit model to; same as above‘‘xdata‘‘.

  type xdata list/array_like, 1D or 2D

  param weights Optional weights to be applied to dataset (see *Weighting Data*).

  type weights list/array_like, optional, 1D

  param f Model function; same as above `f`.

  type f function

  param params Parameters of model

  type params list/array_like, 1D

  returns

- **pivot** (*float or 1D list/array_like*) – Pivot point(s) of the model; (`float` if you're using a one-dimensional model/independent variable, `list/array_like` if n-dimensional.)
- *However, note that all arguments need to be actually used for the pivot point computation. For example,*
- a simple linear model $y(x|b,m) = b + m(x - x_p)$ has a pivot point found by $x_p = \sum_i w_i x_i / \sum_i w_i$, where
- w_i are the weights of the datapoints.

- **pivot_guess** (*float or 1D list/array_like, optional*) – Initial guess for the pivot point(s) of the model (`float` if you're using a one-dimensional model/independent variable, `list/array_like` if n-dimensional; see *Automatically Minimizing Correlation between Linear Model Parameters (Advanced)*).

pivot_function
Function used to evaluate pivot point(s) (see `pivot_function` optional argument of *rcr.FunctionalForm* model constructor).

priors
`rcr.Priors` *object*. Object describing parameter prior probability distribution(s) applied to *rcr.FunctionalForm* model (see *rcr.Priors*).

To use priors on model parameters for some *rcr.FunctionalForm* model, this attribute of the model needs to be initialized as some instance of *rcr.Priors* (see *Applying Prior Knowledge to Model Parameters (Advanced)*).

result
`rcr.FunctionalFormResults` *object*. Access various results unique to Functional Form RCR with this (see *rcr.FunctionalFormResults*).

class rcr.**FunctionalFormResults**
Results from (and unique to) functional form/model-fitting RCR.

parameter_uncertainties
list of floats. Best-fit model parameter uncertainties, post-outlier rejection.

For example, if you're fitting to some linear model $y(x|b,m) = b + mx$, and you obtain a best fit of $b = 1.0 \pm 0.5$ and $m = 2 \pm 1$, then `parameter_uncertainties = [0.5, 1]`.

Note that in order for parameter uncertainties to be computed, either/both weights and data error bars/uncertainties must have been provided when constructing the *rcr.FunctionalForm* model.

parameters
list of floats. Best-fit model parameters, post-outlier rejection.

For example, if you're fitting to some linear model $y(x|b,m) = b + mx$, and you obtain a best fit of $b = 1$ and $m = 2$, then `parameters = [1, 2]`.

pivot
float. Recovered optimal "pivot" point for model that should minimize correlation between the slope and intercept parameters of the linearized model (1D independent variable case).

See *Automatically Minimizing Correlation between Linear Model Parameters (Advanced)*. For example, the pivot point for the model $y(x|b,m) = b + m(x - x_p)$ is x_p.

pivot_ND
float Recovered optimal n-dimensional "pivot" point for model that should minimize correlation between the slope and intercept parameters of the linearized model (n-D independent variable case).

See *Automatically Minimizing Correlation between Linear Model Parameters (Advanced)*. For example, the pivot point for the n-dimensional model $y(\vec{x}|\vec{b},\vec{m}) = \vec{b} + \vec{m}^T(\vec{x} - \vec{x}_p)$ is $\vec{x}_p$.

class rcr.**Priors**

class. Class that encapsulates probabalistic priors to be applied to model parameters when using model-fitting/functional form RCR (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).

Constructor arguments:

Parameters

- **priorType** (*rcr.priorsTypes*) – The type of priors that you're applying to your model (see *rcr.priorsTypes* and *Types of Model Parameter Priors in RCR*).
- **p** (*function, optional*) – Custom priors function; takes in a vector of model parameters and returns a vector of the prior probability density for each value (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).
- **gaussianParams** (*2D list/array_like, optional 2nd argument*) – A list that contains lists of mu and sigma for the Gaussian prior of each param. If no prior, then just use NaNs (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).
- **paramBounds** (2D list/array_like, optional 2nd argument (or 3rd, for the case of rcr.MIXED_PRIORS)) – A list that contains lists of the lower and upper hard bounds of each param. If not bounded, use NaNs, and if there's only one bound, use NaN for the other bound (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).

gaussianParams

2D list/array_like. A list that contains lists of mu and sigma for the Gaussian prior of each param. If no prior, then just use NaNs (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).

p

function. Custom priors function; takes in a vector of model parameters and returns a vector of the prior probability density for each value (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).

paramBounds

A list that contains lists of the lower and upper hard bounds of each param. If not bounded, use NaNs, and if there's only one bound, use NaN for the other bound (see *Applying Prior Knowledge to Model Parameters (Advanced)* for an example).

priorType

rcr.priorsTypes *object*. The type of priors that you're applying to your model (see *rcr.priorsTypes* and *Types of Model Parameter Priors in RCR*).

class rcr.**RCR**

Master class used to initialize and run RCR outlier rejection procedures.

performBulkRejection(**args*, ***kwargs*)

Overloaded function.

1. performBulkRejection(self: rcr.RCR, data: List[float]) -> None

   Perform outlier rejection WITH the speed-up of bulk pre-rejection (see *What is Bulk Rejection?*).

   Parameters:

   data [list/array_like, 1D] Dataset to perform outlier rejection (RCR) on. Access results via the result attribute (*rcr.RCRResults*) of your instance of *rcr.RCR*.

2. performBulkRejection(self: rcr.RCR, weights: List[float], data: List[float]) -> None

   Perform outlier rejection WITH the speed-up of bulk pre-rejection (see *What is Bulk Rejection?*).

   Parameters:

   weights [list/array_like, 1D] Weights for dataset to perform outlier rejection (RCR) on.

   data [list/array_like, 1D] Dataset to perform outlier rejection (RCR) on. Access results via the `result` attribute (*`rcr.RCRResults`*) of your instance of *`rcr.RCR`*.

performRejection (**args*, ***kwargs*)
Overloaded function.

1. performRejection(self: rcr.RCR, data: List[float]) -> None

   Perform outlier rejection WITHOUT the speed-up of bulk pre-rejection (slower; see *What is Bulk Rejection?*).

   Parameters:

   data [list/array_like, 1D] Dataset to perform outlier rejection (RCR) on. Access results via the `result` attribute (*`rcr.RCRResults`*) of your instance of *`rcr.RCR`*.

2. performRejection(self: rcr.RCR, weights: List[float], data: List[float]) -> None

   Perform outlier rejection WITHOUT the speed-up of bulk pre-rejection (slower; see *What is Bulk Rejection?*).

   Parameters:

   weights [list/array_like, 1D] Weights for dataset to perform outlier rejection (RCR) on.

   data [list/array_like, 1D] Dataset to perform outlier rejection (RCR) on. Access results via the `result` attribute (*`rcr.RCRResults`*) of your instance of *`rcr.RCR`*.

result
`rcr.RCRResults` *object*. Access various results of RCR with this (see *`rcr.RCRResults`*).

setParametricModel (*self: rcr.RCR*, *model: FunctionalForm*) → None
Initialize parametric/functional form model to be used with RCR (see *Rejecting Outliers While Model Fitting* for a tutorial).

Parameters **model** (*`rcr.FunctionalForm`*) – n-dimensional model to fit data to while performing outlier rejection.

setRejectionTech (*self: rcr.RCR*, *rejection_technique: rcr.RejectionTechniques*) → None
Modify/set outlier rejection technique to be used with RCR.

See *Table of Rejection Techniques* for an explanation of each rejection technique, and when to use it.

Parameters **rejection_technique** (*`rcr.RejectionTechniques`*) – The rejection technique to be used with your instance of *`rcr.RCR`*.

class `rcr.`**RCRResults**
Various results from performing outlier rejection with RCR.

cleanW
list of floats. The user-provided datapoint weights that correspond to NON-outliers in the original dataset.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` with weights `w = [1, 1.1, 0.9, 1.2, 0.8, 0.2, 0.95, 2]` was provided, and only the `37` and `-100` were found to be outliers, then `cleanW = [1, 1.1, 0.9, 1.2, 0.8, 0.95]`.

cleanY
list of floats. After performing RCR on some original dataset, these are the datapoints that were NOT found to be outliers.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` was provided and only the `37` and `-100` were found to be outliers, then `cleanY = [0, 1, -2, 1, 2, 0.5]`.

flags
list of bools. Ordered flags describing outlier status of each inputted datapoint (True if datapoint is NOT an outlier).

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` was provided and only the `37` and `-100` were found to be outliers, then `flags = [True, True, True, True, True, False, True, False]`.

indices
list of ints. A list of indices of datapoints from original inputted dataset that are NOT outliers.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` was provided and only the `37` and `-100` were found to be outliers, then `indices = [0, 1, 2, 3, 4, 6]`.

mu
float. Mean/median/mode (central value) of uncontaminated data distribution.

The central value of the uncontaminated part of the provided dataset, recovered from performing RCR.

originalW
list of floats. The user-provided datapoint weights, pre-RCR.

For example, if a dataset with weights `w = [1, 1.1, 0.9, 1.2, 0.8, 0.2, 0.95, 2]` was provided, then `originalW = [1, 1.1, 0.9, 1.2, 0.8, 0.2, 0.95, 2]`.

originalY
list of floats. The user-provided dataset, pre-RCR.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` was provided, then `originalY = [0, 1, -2, 1, 2, 37, 0.5, -100]`.

rejectedW
list of floats. The user-provided datapoint weights that correspond to outliers in the original dataset.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` with weights `w = [1, 1.1, 0.9, 1.2, 0.8, 0.2, 0.95, 2]` was provided, and only the `37` and `-100` were found to be outliers, then `rejectedW = [0.2, 2]`.

rejectedY
list of floats. After performing RCR on some original dataset, these are the datapoints that WERE found to be outliers.

For example, if a dataset of `y = [0, 1, -2, 1, 2, 37, 0.5, -100]` was provided and only the `37` and `-100` were found to be outliers, then `rejectedY = [37, -100]`.

sigma
float. Recovered robust 68.3-percentile deviation of uncontaminated data distribution.

A more robust (less sensitive to outliers) version of the standard deviation/width σ of the uncontaminated part of the provided dataset (see Section 2.1 of *The Paper*), recovered from performing RCR. For the case of a symmetric uncontaminated data distribution.

sigmaAbove
float. Recovered robust 68.3-percentile deviation abpve mu (mean/median/mode) of uncontaminated data distribution.

A more robust (less sensitive to outliers) version of the standard deviation/width σ_+ of the positive side of a mildly asymmetric uncontaminated data distribution (see Section 2.1 of *The Paper*), recovered from performing RCR. (For the symmetric case, $\sigma_+ = \sigma_- \equiv \sigma$).

sigmaBelow
float. Recovered robust 68.3-percentile deviation below mu (mean/median/mode) of uncontaminated data distribution.

A more robust (less sensitive to outliers) version of the standard deviation/width σ_- of the negative side of a mildly asymmetric uncontaminated data distribution (see Section 2.1 of *The Paper*), recovered from performing RCR. (For the symmetric case, $\sigma_- = \sigma_+ \equiv \sigma$).

stDev
float. Standard deviation of uncontaminated data distribution.

The standard deviation/width σ of the uncontaminated part of the provided dataset, recovered from performing RCR. For the case of a symmetric uncontaminated data distribution.

stDevAbove
float. Standard deviation above mu (mean/median/mode) of uncontaminated (asymmetric) data distribution.

The *asymmetric* standard deviation/width σ_+ of the positive side of a mildly asymmetric uncontaminated data distribution, recovered from RCR (for the symmetric case, $\sigma_+ = \sigma_- \equiv \sigma$).

stDevBelow
float. Standard deviation below mu (mean/median/mode) of uncontaminated (asymmetric) data distribution.

The *asymmetric* standard deviation/width σ_- of the negative side of a mildly asymmetric uncontaminated data distribution, recovered from RCR (for the symmetric case, $\sigma_- = \sigma_+ \equiv \sigma$).

stDevTotal
float. Combined standard deviation both above and below mu (mean/median/mode) of uncontaminated (asymmetric) data distribution.

A combination of the *asymmetric* standard deviation/width σ_+ of the positive side of a mildly asymmetric uncontaminated data distribution and the width σ_- of the negative side of the distribution, recovered from RCR. Can be used to approximate a mildly asymmetric data distribution as symmetric.

class rcr.**RejectionTechniques**
RCR Standard Rejection Techniques.

Members:

SS_MEDIAN_DL : Rejection technique for a symmetric uncontaminated distribution with two-sided contaminants.

LS_MODE_68 : Rejection technique for a symmetric uncontaminated distribution with one-sided contaminants.

LS_MODE_DL : Rejection technique for a symmetric uncontaminated distribution with a mixture of one-sided and two-sided contaminants.

ES_MODE_DL : Rejection technique for a mildly asymmetric uncontaminated distribution and/or a very low number of data points.

name
handle) -> str

Type (self

class rcr.**priorsTypes**

Types of prior probability density functions that can be applied to model parameters.

Members:

CUSTOM_PRIORS : Custom, function-defined prior probability density functions(s).

GAUSSIAN_PRIORS : Gaussian (normal) prior probability density function(s).

CONSTRAINED_PRIORS : Bounded/hard-constrained prior probability density function(s).

MIXED_PRIORS : A mixture of gaussian (normal), hard-constrained, and uninformative (uniform/flat) prior probability density functions.

name

handle) -> str

Type (self

3.1.4 Rejecting 1D Outliers

This page gives a full tutorial for using RCR to detect and reject outliers within one-dimensional datasets. Although this page avoids unnecessary statistical technicalities (see *The Paper*), a more bare-bones example is given on the main page, *rcr*.

To begin, consider some dataset of N measurements, made up of 1) samples from some contaminant distribution (outliers) and 2) samples from some underlying "true" uncontaminated distribution. RCR has various outlier rejection techniques that have each been chosen to work best for different shapes of these distributions. The table below illustrates this.

Table of Rejection Techniques

Best Rejection Technique	Uncontaminated ("true") Distribution	Contaminant Distribution
`SS_MEDIAN_DL`	Symmetric	Two-Sided/Symmetric
`LS_MODE_68`	Symmetric	One-Sided
`LS_MODE_DL`	Symmetric	In-Between One- and Two-Sided
`ES_MODE_DL`	Mildly Asymmetric/Very low N	(Any)

(Note that an uncontaminated distribution labeled as "symmetric" means approximately Gaussian/normal, mildly peaked, or mildly flat-topped, meaning an exponential power distribution/generalized normal distribution *with positive and negative kurtosis, respectively.)*

For this tutorial, let's consider the case of both the uncontaminated and contaminated distributions being Gaussian/normal (so then, symmetric), both centered at $\mu = 0$. Being outliers, we'll give the contaminated distribution a standard deviation of $\sigma = 5$, and the uncontaminated distribution $\sigma = 1$. Referring to the table above, this means that the rejection technique that we'll need to use is `SS_MEDIAN_DL`, which we will show how to do shortly. Let's arbitrarily choose $N = 500$ datapoints total, with a fraction of 50% contaminated. We can create the dataset in Python as follows:

```
import numpy as np

np.random.seed(18318) # get consistent random results

N = 500                  # total measurement count
frac_contaminated = 0.5  # fraction of sample that will be contaminated
```

(continues on next page)

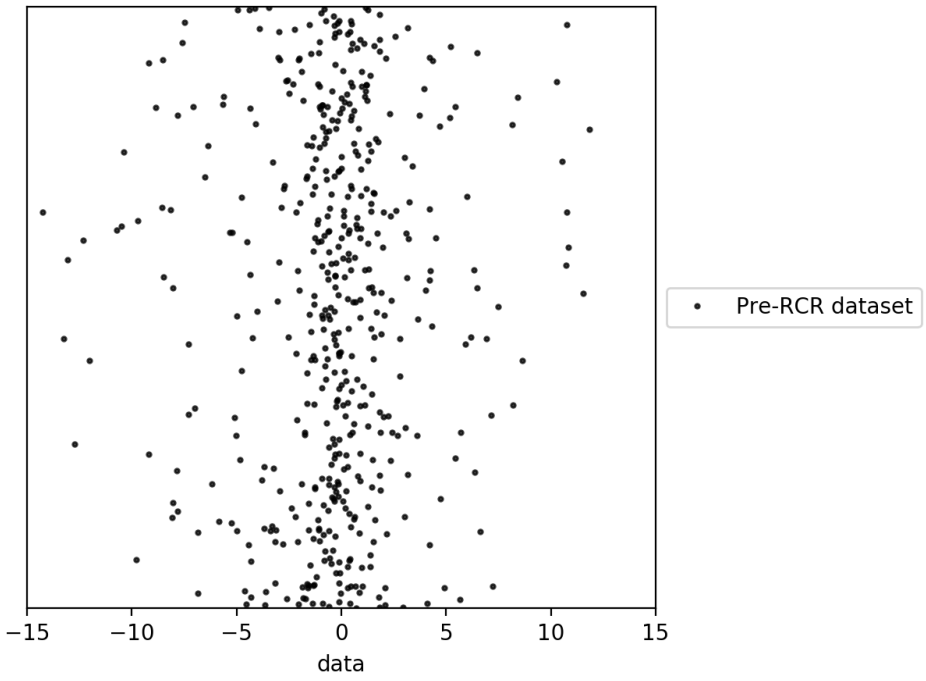
(continued from previous page)

```
# symmetric, uncontaminated distribution
mu = 0
sigma_uncontaminated = 1
uncontaminated_samples = np.random.normal(mu, sigma_uncontaminated,
    int(N * (1 - frac_contaminated)))

# symmetric, contaminated distribution
sigma_contaminated = 5
contaminated_samples = np.random.normal(mu, sigma_contaminated,
     int(N * frac_contaminated))

# combine to create overall dataset
data = np.concatenate((uncontaminated_samples, contaminated_samples))
np.random.shuffle(data)
```

To see what this dataset looks like, we'll plot it below (projected randomly along the y-axis for added visibility).

```
plot data
import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
ax = plt.subplot(111)

ydata = np.random.uniform(0, 1, N) # project randomly into 2D for better visualization

ax.plot(contaminated_samples, ydata[:int(N * frac_contaminated)], "k.",
   label="Pre-RCR dataset", alpha=0.75, ms=4)
ax.plot(uncontaminated_samples, ydata[int(N * frac_contaminated):], "k.",
   alpha=0.75, ms=4)

plt.xlim(-15, 15)
plt.ylim(0, 1)
plt.xlabel("data")
plt.yticks([])

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```

Output:



Now, what do we do if we to estimate the μ and σ of the underlying uncontaminated distribution? Without RCR, we get:

```
# get results pre-RCR
contaminated_mu = np.mean(data)
contaminated_sigma = np.std(data)
print(contaminated_mu, contaminated_sigma)
```

Output:

```
-0.3168378799621606 3.792535849537549
```

Unsurprisingly, the contaminants don't have a great effect on μ, as both the contaminants and the true distribution have the same $\mu = 0$. However, σ is grossly overestimated due to the contaminants, compared to the expected $\sigma = 1$.

So, how can we use RCR? After importing `rcr` (see *Installation Guide*), we initialize the `RCR` object with the desired rejection technique; in our case `SS_MEDIAN_DL`. Next, we perform the outlier rejection (the, recommended, bulk rejection variant; see *What is Bulk Rejection?*) using the *performBulkRejection()* method and the data (as well as optional weights for the data; see *Weighting Data*), as follows:

```
# perform RCR
import rcr

# initialize RCR with rejection technique:
# (chosen from shape of uncontaminated + contaminated distribution)
r = rcr.RCR(rcr.SS_MEDIAN_DL)
r.performBulkRejection(data) # perform outlier rejection
```

Next, we can obtain the results of RCR with the `result` member of the `RCR` class. In our case, we're interested in the RCR-recovered values for μ and σ of the underlying uncontaminated distribution:

```
# View results post-RCR
cleaned_mu = r.result.mu
cleaned_sigma = r.result.stDev
print(cleaned_mu, cleaned_sigma)
```

Output:

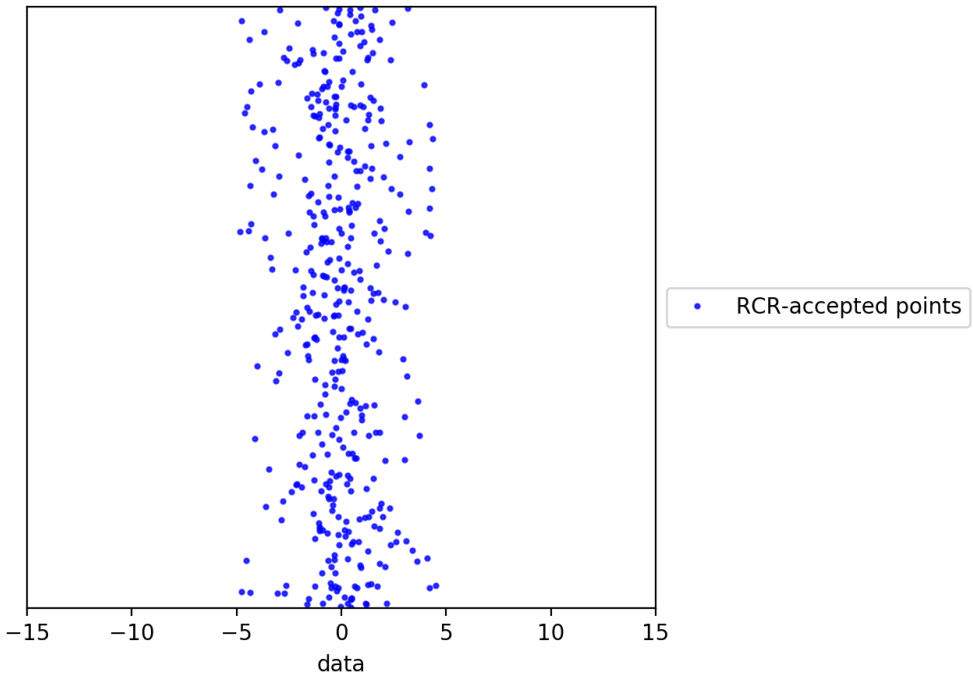
```
-0.1584668560834893 1.8260572902969874
```

Successfully, RCR managed to recover both a μ and σ that are significantly closer to the true values of 0 and 1, respectively, both by a factor of about 2.

We can also access the subsets of rejected and nonrejected datapoints of the dataset, as well as the corresponding indices and flags thereof, from `RCR.result`. For example, we can plot the post-rejection dataset with:

```
# plot rejections
cleaned_data = r.result.cleanY

flags = r.result.flags
# list of booleans corresponding to the original dataset,
# true if the corresponding datapoint is not an outlier.

cleaned_data_indices = r.result.indices
# indices of data in original dataset that are not outliers

plt.figure(figsize=(8,5))
ax = plt.subplot(111)
ax.plot(data[cleaned_data_indices], ydata[cleaned_data_indices], "b.",
     label="RCR-accepted points", alpha=0.75, ms=4)

plt.xlim(-15, 15)
plt.ylim(0, 1)
plt.xlabel("data")
plt.yticks([])

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```

Output:



In the next section, we'll explore how we can apply weights to datapoints to use with RCR.

Weighting Data

For both single-value/one-dimensional RCR, and the n-dimensional model-fitting/functional variant (see *Rejecting Outliers While Model Fitting*), numerical, non-negative weights can be optionally provided for each of the datapoints. However, what does it really mean to weight datapoints? If you have some datapoint y_n, giving it a weight of $w_n = 2$ is simply analogous to counting it twice. Now, what's an example of where weighting can be useful?

Lets say that we'd like to perform RCR on the same dataset as above, except now we somehow know *a priori* that the true, uncontaminated datapoints should be normally/Gaussian-distributed (again with $\mu = 0$ and $\sigma = 1$). We can use this prior knowledge to perform a sort of Bayesian outlier rejection, by giving the datapoints weights that are proportional to the value of the known normal probability density function. In Python, we can do this simply as:

```
from scipy.stats import norm

# function to weight each datapoint according to the prior knowledge
def weight_data(datapoint):
   return norm.pdf(datapoint, loc=mu, scale=sigma_uncontaminated)

# create weights
weights = weight_data(data)
```

Next we can perform RCR and view the results as usual, only now providing the weights as the first argument of `performBulkRejection()`:

```
# perform RCR; same rejection technique
r = rcr.RCR(rcr.SS_MEDIAN_DL)
r.performBulkRejection(weights, data) # perform outlier rejection, now with weights
```

(continues on next page)

(continued from previous page)

```
# View results post-RCR
cleaned_mu = r.result.mu
cleaned_sigma = r.result.stDev
print(cleaned_mu, cleaned_sigma)
```

Output:

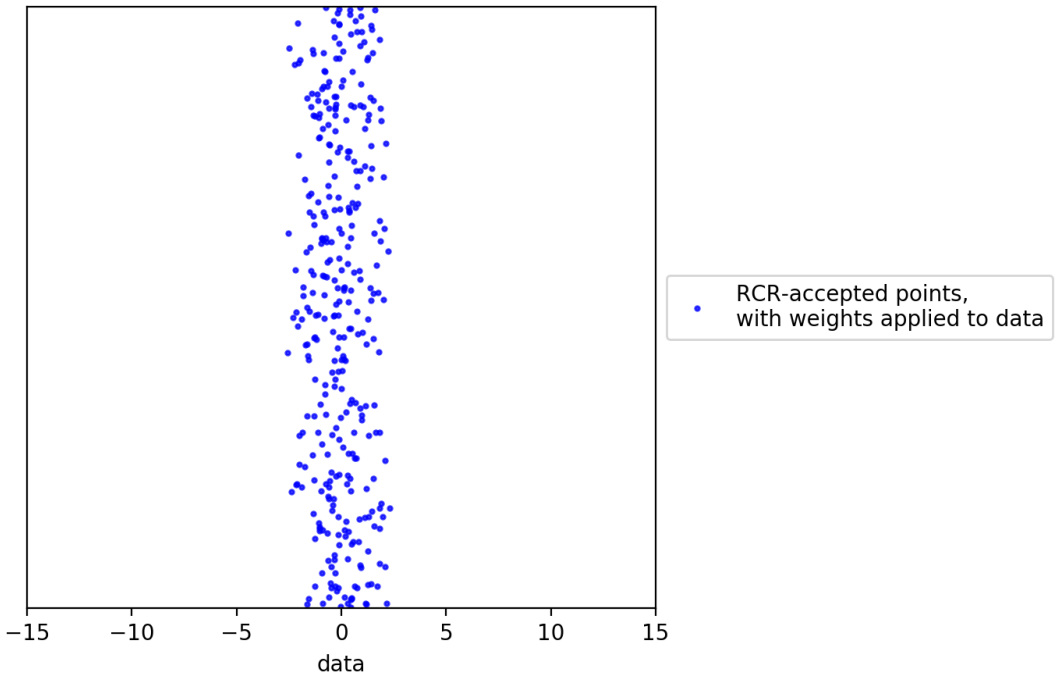
```
-0.05519770432617514 0.7825197746126461
```

This is much closer to the expected values of $\mu = 0$ and $\sigma = 1$ than what we got with the unweighted/equally-weighted dataset above (this time actually, σ was slightly *under*-estimated).

We can then plot the cleaned dataset/non-rejected data as usual:

```
# plot rejections
cleaned_data = r.result.cleanY
cleaned_data_indices = r.result.indices

plt.figure(figsize=(8,5))
ax = plt.subplot(111)
ax.plot(data[cleaned_data_indices], ydata[cleaned_data_indices], "b.",
   label="RCR-accepted points,\nwith weights applied to data", alpha=0.75, ms=4)

plt.xlim(-15, 15)
plt.ylim(0, 1)
plt.xlabel("data")
plt.yticks([])

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```

Output:



As expected, the width of the cleaned dataset is noticeably smaller after applying weights.

3.1.5 Rejecting Outliers While Model Fitting

Introduction

In its most simple form, RCR is an excellent tool for detecting and rejecting outliers within heavily contaminated one-dimensional datasets, as shown in *Rejecting 1D Outliers*. However, this only scratches the surface of RCR. In its more generalized form, RCR can also be used to reject outliers within some n-dimensional dataset while also *simultaneously* fitting a model to that dataset. This section will explain how this can be done fairly easily in practice, while avoiding going into unnessarily technicalities. We recommend reading *Rejecting 1D Outliers* before tackling this section, as the following is essentially a generalization of that section.

For the case of one-dimensional data (see *Rejecting 1D Outliers*), RCR can be thought of as being used to reject outliers from some dataset $\{y_i\}_{i=1}^{N}$, distributed about a single, parameterized "true" value y. In this case, we often wish to get a best estimate of y, in order to properly characterize the underlying measurement distribution; y is just a one-dimensional *model* that we want to fit to the data, characterized by location and scale parameters like μ and σ.

If we generalize the dimensionality of this, we can imagine using RCR on measurements distributed about some n-dimensional model *function* $y(\vec{x}|\vec{\theta})$, where $\vec{x}$ is an n-dimensional vector of the model's independent variable(s), and $\vec{\theta}$ is an M-dimensional vector of the model's parameters. In this case, we say that $y(\vec{x}|\vec{\theta})$ is an n-dimensional, M-parameter model. For a more concrete example of this, consider a simple linear model $y = b + mx$. In this case, $n = 1$, and our parameter vector is just $\vec{\theta} = (b, m)$.

For this more-general case, what does our dataset look like? Each datapoint will be some value of $y(\vec{x}|\vec{\theta})$ associated with a value for $\vec{x}$. As such, if we have N datapoints in total, indexing each by i, our dataset can be written compactly as $\{(\vec{x}_i, y_i)\}_{i=1}^{N}$ (be sure to avoid getting N confused with n here; the former is the number of datapoints that we're fitting the model to, while the latter is the dimensionality of the dataset/model).

Last but not least, before we get into the code, it's important to point out that in order for RCR to fit any arbitrary model function to a dataset, (partial) derivatives of the model function with respect to each model parameter must be supplied (due to the specific algorithm that is used for fitting). For example, consider a one-dimensional exponential model of the form $y(\vec{x}|\vec{\theta}) = be^{mx}$. If we choose to order the model parameters as $\vec{\theta} = (b, m)$, then our model parameter derivatives are

$$\frac{\partial y(\vec{x}|\vec{\theta})}{\partial b} = e^{mx} \qquad \text{and} \qquad \frac{\partial y(\vec{x}|\vec{\theta})}{\partial m} = xbe^{mx}.$$

Implementation

Finally, we have everything that we need to use RCR for outlier rejection during model fitting. Although `rcr` supports any arbitrary n-dimensional nonlinear model function (as long as the model parameter derivatives are well-defined), for simplicity let's consider a simple linear model $y(\vec{x}|\vec{\theta}) = b + mx$. The parameter partial derivatives are then simply

$$\frac{\partial y(\vec{x}|\vec{\theta})}{\partial b} = 1 \qquad \text{and} \qquad \frac{\partial y(\vec{x}|\vec{\theta})}{\partial m} = x.$$

Before we start coding, it's important to consider the following:

Note: Within `rcr`, model functions and their derivatives must be defined exactly with arguments 1) `x` and 2) `params`, where `x` is the n-dimensional list or numpy array (or `float`, in the case of $n = 1$) of independent variable(s), and `params` is the M-dimensional list/array of model parameters. *Make sure to maintain consistent ordering of the model parameters vector throughout your code.*

Now, onto the code; let's start by defining our model function and its parameter derivatives:

```
def linear(x, params): # model function
    return params[0] + x * params[1]

def d_linear_1(x, params): # first model parameter derivative
    return 1

def d_linear_2(x, params): # second model parameter derivative
    return x
```

Next, let's start creating our dataset. We'll have $N = 200$ points total, with 85% of the datapoints being outliers. Our "true" model that the datapoints will be generated about will have parameters of $b = 0$ and $m = 1$. In code, this is simply:

```
import numpy as np

N = 200 # number of datapoints
f = 0.85 # fraction of datapoints that are outliers

params_true = [0, 1] # parameters of "true" model
```

We'll generate our datapoints in a certain range of x values about the "true" model line. For this example, we'll make uncontaminated datapoints that are Gaussian/normally distributed, with standard deviation $\sigma = 1$, about the true model. In order to highlight the power of RCR with dealing with especially difficult outliers, we'll generate one-sided outliers/contaminants, sampled from the positive side of a Gaussian with $\sigma = 10$. In code, this will take the form of:

```
sigma_uncontaminated = 1 # standard deviations used to generate datapoints
sigma_contaminated = 10
```

(continues on next page)

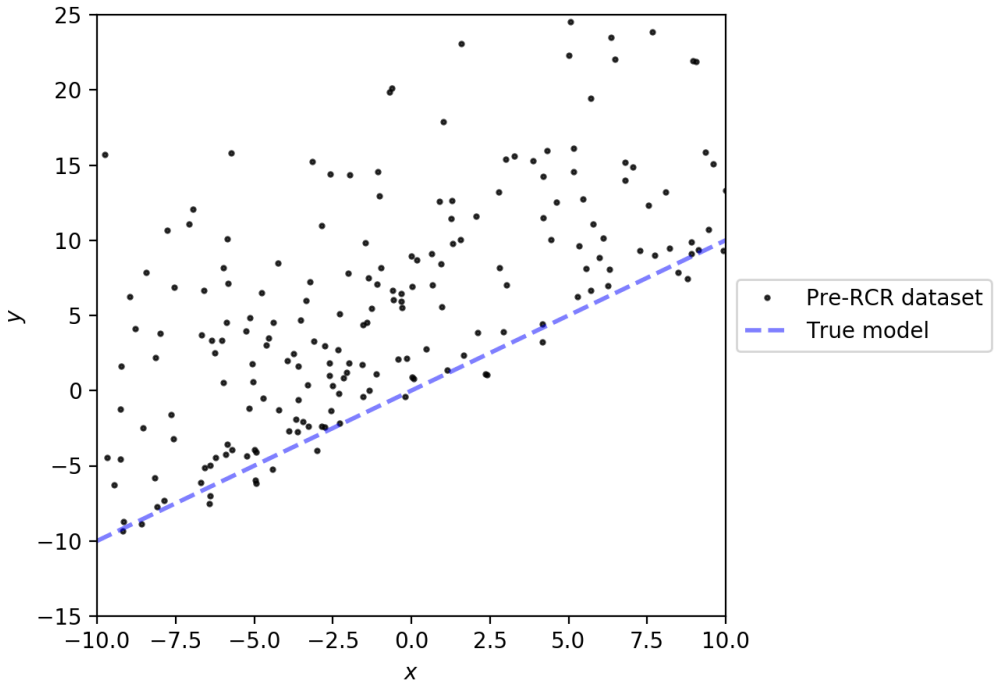
(continued from previous page)

```
# generate x-datapoints randomly in an interval
x_range = (-10, 10)
xdata_uncontaminated = np.random.uniform(
    x_range[0], x_range[1], int(N * (1 - f)))
xdata_contaminated = np.random.uniform(
    x_range[0], x_range[1], int(N * f))


# generate y-datapoints about the true model:
# symmetric uncontaminated distribution
ydata_uncontaminated = np.random.normal(
    loc=linear(xdata_uncontaminated, params_true),
    scale=sigma_uncontaminated
    )

# one-sided contaminated distribution
ydata_contaminated = linear(xdata_contaminated, params_true) + np.abs(
    np.random.normal(0, sigma_contaminated, int(N * f)))


# combine dataset
xdata = np.concatenate((xdata_contaminated, xdata_uncontaminated))
ydata = np.concatenate((ydata_contaminated, ydata_uncontaminated))
```

Let's plot the dataset over the true, underlying model:

```
# plot dataset
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
ax = plt.subplot(111)

ax.plot(xdata_contaminated, ydata_contaminated, "k.",
    label="Pre-RCR dataset", alpha=0.75, ms=4)
ax.plot(xdata_uncontaminated, ydata_uncontaminated, "k.",
    alpha=0.75, ms=4)


# plot model
x_model = np.linspace(x_range[0], x_range[1], 1000)
ax.plot(x_model, linear(x_model, params_true),
    "b--", label="True model", alpha=0.5, lw=2)

plt.xlim(-10, 10)
plt.ylim(-15, 25)
plt.xlabel("$x$")
plt.ylabel("$y$")

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```

Output:



Clearly, these outliers are pretty nasty. This looks like a job for RCR. First, we need to supply an initial guess for the model parameters, to give the fitting engine within RCR a starting place. Approaching this dataset with no knowledge of what is or isn't an outlier, it would be hard to tell what the true best fit should be; as such, we'll use an initial guess that naively should work with the data, but is pretty far off of the true values of $b = 0$ and $m = 1$; let's try $b = 5$ and $m = 1.5$:

```
guess = [5, 1.5]
```

Next, we'll need to initialize the model, as an instance of the `rcr.FunctionalForm` class. The required arguments (in order) to construct an instance of this class are 1) the model function, 2) the (n-dimensional) x-data, 3) the y-data, 4) a list of the model parameter derivative functions, in order and 5) the guess for the parameters. This is implemented as:

```
model = rcr.FunctionalForm(linear,
    xdata,
    ydata,
    [d_linear_1, d_linear_2],
    guess
)
```

Now, we're finally ready to run RCR on the dataset/model. Our uncontaminated distribution of data is symmetric, while our contaminated distribution is one-sided/completely asymmetric. Therefore, following the *Table of Rejection Techniques*, the rejection technique that will perform best on this dataset is `LS_MODE_68`. Given this, we'll perform RCR as usual, except now, we need to tell our instance of the `RCR` class that we're fitting to our specific parametric model:

```
r = rcr.RCR(rcr.LS_MODE_68) # setting up for RCR with this rejection technique

r.setParametricModel(model)
# tell RCR that we are model fitting, and give it the model of choice
```

(continues on next page)

(continued from previous page)

```
r.performBulkRejection(ydata) # perform RCR
```

That was only a few lines of code, but what actually happened here? Essentially, (see *The Paper* for more details), RCR can iteratively reject outliers and fit the model to the data at the same time. As such, we can access the same outlier-rejection results from `r.result` as in *Rejecting 1D Outliers*, while also having model-fitting results from our model, with the member `model.result`:

```
best_fit_parameters = model.result.parameters # best fit parameters

rejected_data = r.result.rejectedY # rejected and non-rejected data
nonrejected_data = r.result.cleanY
nonrejected_indices = r.result.indices
# indices from original dataset of nonrejected data

print(best_fit_parameters)
```

Output:

```
[1.2367288755077883, 1.004037971689524]
```

Before we discuss this result, it's teaching to compare it to the traditional method of ordinary least-squares fitting; we'll summarize this in a plot, as follows:

```
# plot results

plt.figure(figsize=(8, 5))
ax = plt.subplot(111)

ax.plot(xdata_contaminated, ydata_contaminated, "k.",
    label="Pre-RCR dataset", alpha=0.75, ms=4)
ax.plot(xdata_uncontaminated, ydata_uncontaminated, "k.",
    alpha=0.75, ms=4)

ax.plot(xdata[nonrejected_indices], ydata[nonrejected_indices], "bo",
    label="Post-RCR dataset", alpha=0.4, ms=4)

# plot true model
ax.plot(x_model, linear(x_model, params_true),
    "b--", label="True model", alpha=0.5, lw=2)

# plot regular linear least squares best fit
from scipy.stats import linregress

slope_lsq, intercept_lsq, _, _, _ = linregress(xdata, ydata)

ax.plot(x_model, linear(x_model, [intercept_lsq, slope_lsq]),
    "r-", label="Least-squares best fit", alpha=0.5, lw=2)

# plot RCR-fitted model
ax.plot(x_model, linear(x_model, best_fit_parameters),
    "g-", label="RCR best fit", alpha=0.5, lw=2)


plt.xlim(-10, 10)
plt.ylim(-15, 25)
```

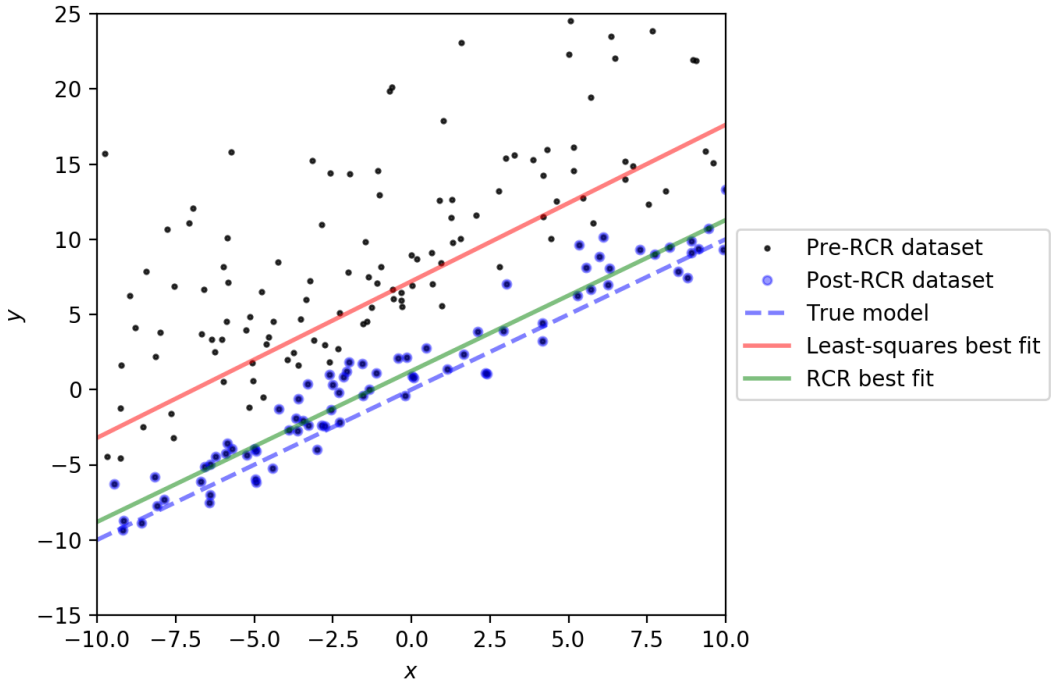
(continues on next page)

(continued from previous page)

```
plt.xlabel("$x$")
plt.ylabel("$y$")

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

print("Least-squares fit results:", intercept_lsq, slope_lsq)

plt.show()
```

Output:

```
Least-squares fit results:
7.202089027278407 1.0412871059773106
```



RCR gave us a best fit values of $b = 1.237$ and $m = 1.004$, while traditional linear least squares gave $b = 7.202$ and $m = 1.041$. The slope (true value of $m = 1$) was recovered very well in both cases, but this isn't super surprising, given that both the contaminated and uncontaminated measurement distributions were generated without any scatter along the x-axis. However, due to the heavy scatter/contamination along the y-axis, the least-squares result for the intercept b is, expectly, heavily biased by the outliers, very far off of the true value of $b = 1$. However, RCR was able to successfully reject many of the outliers, while maintaining almost all of the uncontaminated distribution (shown in blue circles), giving a best fit $b = 1.237$ that is significantly closer to the true value of $b = 1$ than the least-squares result.

Overall, the RCR fit (green line) is clearly a much better fit (true best fit in blue dashed line) than the least squares best fit (red line).

Data with Uncertainties and/or Weights

Realistically, many datasets will have measurements that have uncertainties, or *error bars*, as practically all physical measurements cannot truly be made with exact precision. In most model-fitting scenarios, only uncertainties in the *dependent* variable (y) are considered, with any uncertainties in the independent variable(s) $\vec{x}$ considered to be negligible (for a more generalized treatment, that includes such $\vec{x}$-uncertainties, as well as uncertainty in the dataset that cannot solely be attributed to the data error bars, see e.g. Konz 2020). In this case, which we take for RCR, our dataset becomes $\{(\vec{x}_i, y_i \pm \sigma_{y,i})\}_{i=1}^{N}$, i.e. our measurement error bars/uncertainties are $\{\sigma_{y,i}\}_{i=1}^{N}$.

Just as in one-dimensional RCR, weights w_i can also be applied to model-fitting datasets (e.g. *Weighting Data*). We note that the inclusion of error bars as described in the previous paragraph is not mutual exclusive with such weighting; both weights and error bars can be used in practice.

To use a dataset with error bars and/or weights with model-fitting RCR, simply use the optional arguments `error_y` and `weights` of the `rcr.FunctionalForm()` constructor, where the former is an ordered vector/list of measurement uncertainties $\{\sigma_{y,i}\}_{i=1}^{N}$, and the latter is an ordered vector/list of measurement weights $\{w_i\}_{i=1}^{N}$. An example of this is given in the following section.

Model Parameter Uncertainties/Error Bars

In many cases, we often want not just best fit parameters for a model and dataset, but also *uncertainties*, or "error bars" for these parameters. This is easily available in `rcr`, again via the `model.result` object, as `model.result.parameter_uncertainties`. However, before we go into a worked code example, note the following:

Note: In `rcr`, best fit model parameter uncertainties can only be calculated if error bars/uncertainties *and/or* weights were given for the dataset before fitting.

Now, let's try adding error bars to our linear dataset, same as above. First, we'll initialize the error bars, randomly, giving higher error, on average, to the contaminants:

```
error_y_uncontaminated = np.random.uniform(low=0.1, high=1, size=int(N * (1 - f)))
error_y_contaminated = np.random.uniform(low=1, high=2, size=int(N * f))

error_y = np.concatenate((error_y_contaminated, error_y_uncontaminated))
```

Next, let's initailize the model as before, except now using the optional keyword argument `error_y` to input the error bars. We then can perform RCR as usual.

```
# instantiate model
model = rcr.FunctionalForm(linear,
    xdata,
    ydata,
    [d_linear_1, d_linear_2],
    guess,
    error_y=error_y
)

# initialize and perform RCR as usual
r = rcr.RCR(rcr.LS_MODE_68) # setting up for RCR with this rejection technique
r.setParametricModel(model) # tell RCR that we are model fitting
r.performBulkRejection(ydata) # perform RCR
```

Let's check out the results:

```
# view results
best_fit_parameters = model.result.parameters # best fit parameters
best_fit_parameter_errors = model.result.parameter_uncertainties # and their␣
↪uncertainties

rejected_data = r.result.rejectedY # rejected and non-rejected data
nonrejected_data = r.result.cleanY
nonrejected_indices = r.result.indices

print(best_fit_parameters)
print(best_fit_parameter_errors)
```

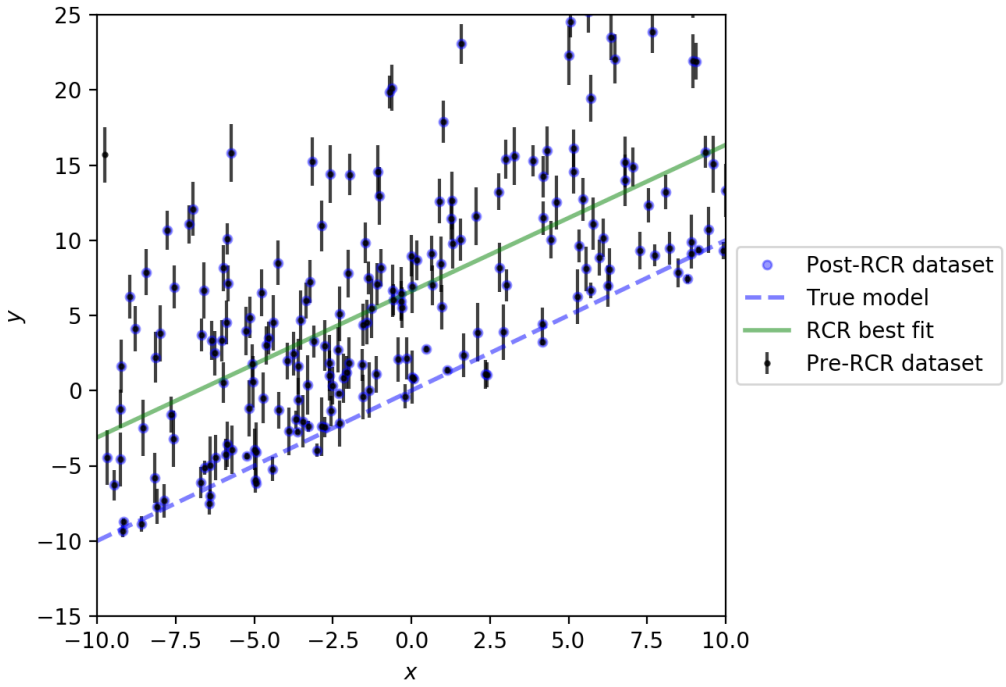
Output:

```
[6.612942587028933, 0.9732622673909074]
[1.6299290812536242, 0.3258511725157285]
```

So, our RCR-recovered best fit is $b = 6.61 \pm 1.63$ and $m = 0.973 \pm 0.326$. Unfortunately, this fit isn't nearly as good as when we didn't have measurement uncertainties. But why? To see, let's plot the dataset alongside the fit:

```
# plot results

plt.figure(figsize=(8, 5))
ax = plt.subplot(111)

ax.errorbar(xdata_contaminated, ydata_contaminated, yerr=error_y_contaminated,
    fmt="k.", label="Pre-RCR dataset", alpha=0.75, ms=4)
ax.errorbar(xdata_uncontaminated, ydata_uncontaminated, yerr=error_y_uncontaminated,
    fmt="k.", alpha=0.75, ms=4)

ax.plot(xdata[nonrejected_indices], ydata[nonrejected_indices], "bo",
    label="Post-RCR dataset", alpha=0.4, ms=4)

# plot true model
ax.plot(x_model, linear(x_model, params_true),
    "b--", label="True model", alpha=0.5, lw=2)

# plot RCR-fitted model
ax.plot(x_model, linear(x_model, best_fit_parameters),
    "g-", label="RCR best fit", alpha=0.5, lw=2)


plt.xlim(-10, 10)
plt.ylim(-15, 25)
plt.xlabel("$x$")
plt.ylabel("$y$")

box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.65, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```

Output:



Adding error bars, or *intrinsic* uncertainties, to the measurements in the dataset introduced even more overall uncertainty to the data, beyond just the *extrinsic* uncertainty, or scatter/sample variance of the datapoints themselves. That, combined with the extremely high contaminant fraction of 85%, made it so that RCR was unable to tell apart the contaminants from the non-outlier datapoints, under-rejecting the outliers, as shown in the plot. As such, the final dataset that the model was fit to included too many outliers, biasing the fitted line to have too high an intercept. RCR would've worked better if either/both 1) there were smaller error bars or 2) the fraction of contaminants was lower.

Applying Prior Knowledge to Model Parameters (Advanced)

Let's say that we want to fit some model to a dataset, and we know certain, *prior* information about one of the parameters of the model, a, in advance. From the point of view of Bayesian inference, this can be formalized by specifying the *prior probability distribution*, or *prior probability density function* (PDF) of that parameter $p(a)$. For example, let's say that for the linear dataset/model above, we know *a priori* that the intercept b should be $b = 0$, with uncertainty of 1, i.e. $b = 0 \pm 1$. This translates to a *prior probability distribution* of a Gaussian with mean $\mu = 0$ and standard deviation $\sigma = 1$, i.e.

$$p(b) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}b^2}.$$

However, let's say that we don't know anything in advance about the slope m. In this case, we say that the prior on m is *uninformative*, i.e. all values are equally likely (again, this is before we even consider any data), which manifests mathematically as

$$p(m) \propto 1.$$

In `rcr`, prior probability distributions can be specified for any or all of the parameters of a model, which will affect the rejection of outliers (essentially by modifying the rejection probability of certain measurements according to the prior probabilities of all of the model parameter solutions that these measurements can contribute to). For simplicity and ease-of-use, we've included two types of common priors within the library, as well as allowing for any sort of custom prior PDF. These options are described in the table below.

Types of Model Parameter Priors in RCR

Prior Type	Parameters Needed To Specify
`GAUSSIAN_PRIORS`	Means and standard deviations of some or all model parameters
`CONSTRAINED_PRIORS`	Lower and/or upper bounds on some or all model parameters
`MIXED_PRIORS`	Combination of some or all of the two above
`CUSTOM_PRIORS`	For some or all model parameters a_j, custom prior PDF $p(a_j)$

Now, how can we use these different types of priors in practice?

Gaussian Priors

Let's say that you want to apply Gaussian/normal prior probability distributions on some (or all) of your model parameters. To do so, you'll first need to create a list, where each element of the list corresponds to a model parameter, and is itself a list of 1) the mean of the Gaussian for that parameter's prior and 2) the standard deviation of the same. If no Gaussian prior is desired for a certain parameter, just give NaNs for those fields.

This is pretty dense, so we'll show a specific instance of this usage. Following the example within the introduction to this section (*Applying Prior Knowledge to Model Parameters (Advanced)*), lets use the same linear model as before, and apply a Gaussian prior to the intercept b, with mean $\mu = 0$ and standard deviation $\sigma = 1$. We'll use no prior (uninformative) on the slope m. From here, our list of parameters (not model parameters) that describe the Gaussian priors will be:

```
gaussianParams = [
        [0,             1], # mu = 0, sigma = 1
        [float('nan'),  float('nan')]
        # no prior on the slope parameter, so just use NaNs
    ]
```

Now, to introduce these priors before performing any fitting/RCR, we'll need to create an instance of the `Priors` class from `rcr`, making sure to specify which type of prior we're implementing using the correct object from the above table (in this case `GAUSSIAN_PRIORS`). Here it is in code:

```
mypriors = rcr.Priors(rcr.GAUSSIAN_PRIORS, gaussianParams)
```

From here, RCR can be performed as usual, by 1) supplying the optional argument `has_priors=True` to the `FunctionalForm` constructor when initializing the model, and after that 2) initializing the `priors` attribute of your model with your `Priors` object:, e.g.:

```
model = rcr.FunctionalForm(linear,
    x,
    y,
    [linear_partial1, linear_partial2],
    guess,
    has_priors=True
)

model.priors = mypriors
```

From here RCR can be utilized with this model given the usual methods.

Constrained/Bounded Priors

Another very common type of prior is to give hard constraints/bounds on certain model parameters. Following the same linear example, let's say that we know that the slope m of our model should be nonnegative (this type of prior is often for some physical reason), but we don't know anything about the intercept b.

Similar to the usage of Gaussian priors, to implement this we'll need to create a list where each element corresponds to a model parameter, and is itself a list of 1) the lower bound and 2) the upper bounds that we want to give the corresponding parameter if we only want to supply one (or neither) of the bounds, just use a NaN instead. Following our chosen example, this list can be coded as

```
paramBounds = [
        [float('nan'),  float('nan')]
        [0,             float('nan')]  # constrain m > 0
    ]
```

Next, we need to instantiate an `rcr.Priors` object, in a similar manner to the case of Gaussian priors (except now being sure to specify `CONSTRAINED_PRIORS`):

```
mypriors = rcr.Priors(rcr.CONSTRAINED_PRIORS, paramBounds)
```

Finally, we'll need to initialize our model with the priors as in the end of the previous section (again with `has_priors=True`), and then we're good to go.

Both Gaussian and/or Constrained (Mixed) Priors

What if we want to apply Gaussian priors to some model parameters, constrained priors to others, or even a mix of both for certain parameters (e.g. force a parameter to be positive, while also making it Gaussian-distributed)? To do this, simply create the lists that specify these priors— `paramBounds` and `gaussianParams` following the previous examples—and supply them both to the constructor for your `Priors` object, making sure to specify the priors type as `MIXED_PRIORS`:

```
mypriors = rcr.Priors(rcr.MIXED_PRIORS, gaussianParams, paramBounds)
```

From here, RCR can be used as normal, after initializing our model (with `has_priors=True`) and supplying the model with the Priors object.

Custom Priors

In the most general case, RCR can work with any type of prior probability distributions/density functions. To implement this, you'll need a function $\vec{p}(\vec{\theta})$ that takes in a vector of model parameters $\vec{\theta}$, and returns a vector of each parameter's prior probability density function evaluated given the corresponding parameter's value.

As an example, let's consider that for our linear model, we'd like to 1) place an (unusual) prior on b:

$$p(b) = e^{-|b|} \left|\cos^2 b\right|,$$

and 2) constrain m to be within the interval $(0, 2]$. We can then implement $\vec{p}(\vec{\theta})$ as:

```
def prior_pdfs(model_parameters):
    pdfs = np.zeros(2) # vector of model parameter density function values
    b = model_parameters[0]
    pdfs[0] = np.exp(-np.abs(b)) * np.abs(np.cos(b)**2.)
```

(continues on next page)

(continued from previous page)

```
    b = model_parameters[0]
    pdfs[1] = 1 if 0 < m <= 2 else 0
    # p(m) = 0 if m is outside bounds of (0, 2]

    return pdfs
```

After such a $\vec{p}(\vec{\theta})$ is defined, we'll need to use it to instantiate an `rcr.Priors` object as usual, this time declaring our type of priors as `CUSTOM_PRIORS`:

```
mypriors = rcr.Priors(rcr.CUSTOM_PRIORS, prior_pdfs)
```

After creating our model (with `has_priors=True`) and supplying it with our Priors object `mypriors`, RCR can then be used as usual.

Automatically Minimizing Correlation between Linear Model Parameters (Advanced)

Let's again consider a linear model $y = b + mx$. Usually the fitted slope m will be correlated with the fitted intercept b. Why is this? Consider redefining this model as $y = b + m(x - x_p)$, with some *pivot point* x_p. Then, the intercept parameter is effectively $b - mx_p$. Therefore, given some fitted m, the fitted intercept will be impacted by m, with the degree of this depending on choice of x_p. As shown in Trotter (2011), there exists some optimal x_p that minimizes the correlation between b and m.

RCR has an algorithm for this, that works for any model that can be written as $y = b + m(x - x_p)$. So, for example, we could have some power-law model

$$y(x|a_0, a_1) = a_0 \left(\frac{x}{10^{x_p}}\right)^{a_1},$$

that can be actually be *linearized* (to be used with RCR's pivot point optimizing algorithm) as follows:

$$\begin{aligned}
y(x) &= a_0 \left(\frac{x}{10^{x_p}}\right)^{a_1} \\
\log_{10} y(x) &= \log_{10} \left[a_0 \left(\frac{x}{10^{x_p}}\right)^{a_1}\right] \\
&= \log_{10} a_0 + \log_{10} \left(\frac{x}{10^{x_p}}\right)^{a_1} \\
&= \log_{10} a_0 + a_1 \log_{10} \left(\frac{x}{10^{x_p}}\right) \\
&= \log_{10} a_0 + a_1 \left[\log_{10} x - \log_{10} 10^{x_p}\right] \\
\log_{10} y(x) &\equiv \log_{10} a_0 + a_1 \left[\log_{10} x - (\log_{10} x)_p\right]
\end{aligned}$$

So, the linearized version of this power law has intercept $\log_{10} a_0$, slope a_1, pivot point $(\log_{10} x)_p$, and the data transforms as $\log_{10} y \to y$ and $\log_{10} x \to x$. The formula for the pivot point is

$$(\log_{10} x)_p = \frac{\sum_i w_i (\log_{10} x_i) y^2(x_i)}{\sum_i w_i y^2(x_i)}$$

(Maples et al. 2018, Section 8.3.5); we'll need such a formula for the pivot point of any model that we'd like to apply this procedure to. Keeping that in mind, lets look at this in code.

To use the slope-intercept correlation minimization procedure with RCR, we'll need to define this pivot point function. However, first read the following note:

Note: Pivot point functions need to be defined with parameters of 1) *xdata*, a list or array of the x-data in the dataset, 2) *weights*, a list/array of the weights of the dataset, 3) f, the model function, and 4) $params$, a list/array of model parameters. (To be made easier in a future patch)

Keeping this in mind, let's define our pivot point function for this power law model:

```
def get_pivot_powerlaw(xdata, weights, f, params):
    topsum = np.sum(weights * np.log10(xdata) * np.power(f(xdata, params), 2.))
    bottomsum = np.sum(weights * np.power(f(xdata, params), 2.))

    return topsum / bottomsum
```

Now, we need to define our model, making sure to use the pivot point. In order to use pivot points within the function definition of a model (and its derivatives), we'll need to use the static attribute `pivot` of the `rcr.FunctionalForm` class (or `pivot_ND` for the n-dimensional case) within these definitions. So, our power-law model and its parameter-derivatives can be defined as:

```
def powerlaw(x, params):
    a0 = params[0]
    a1 = params[1]
    return a0 * np.power(x / np.power(10., rcr.FunctionalForm.pivot), a1)

def powerlaw_partial1(x, params):
    a1 = params[1]
    return np.power((x / np.power(10., rcr.FunctionalForm.pivot)), a1)

def powerlaw_partial2(x, params):
    a0 = params[0]
    a1 = params[1]
    piv = rcr.FunctionalForm.pivot # renamed for brevity

    return a0 * np.power((x / np.power(10., piv)), a1) * np.log(x / np.power(10.,␣
↪piv))
```

Next, we can use this with RCR as normal, except now supplying additional arguments of `pivot_function` and `pivot_guess` to the model constructor `rcr.FunctionalForm`, where `pivot_function` is the function that returns the pivot for model give `xdata, weights, f, params`, and `pivot_guess` is a guess for the optimal pivot point (for the iterative optimization algorithm). For example, if our initial guess for the pivot point is some `pivot_guess = 1.5`, we could initialize our model as:

```
model = rcr.FunctionalForm(powerlaw,
    xdata,
    ydata,
    [powerlaw_partial1, powerlaw_partial2],
    guess,
    pivot_function=get_pivot_powerlaw,
    pivot_guess=pivot_guess
)
```

From here, we can perform RCR as normal, and access the optimal value for the pivot points found by RCR with `model.result.pivot` (or `model.result.pivot_ND` for the *n*-dimensional model case).

Finally, note that the support for n-dimensional models (i.e. n independent variables) is still available when using this feature; in this case, your pivot point function should return a list/array of n pivot points.

Python Module Index

r

Index