Documentation/Starter Guide for the TRK Regression Algorithm

Nick Konz

March 19, 2020

For TRK version 0.11.1 or higher.

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) `TRK.cpp`

   Header files
   (a) `TRK.h`
3. Other useful files are included, such as `exampleModels.h` and `exampleModels.cpp`, which include example model functions, and `TRKstarterfile.cpp`, which has ready-to-run code utilizing TRK.

1 Basic TRK Regression

1.1 Defining the Model

To perform any version of TRK Regression, it's needed to provide a model function, as well as its first two derivatives with respect to the independent, or "x" variable. Syntactically, these functions need to be defined to return a `double`, and take arguments of, in order, the "x" variable (a `double`) and a vector of the model parameters $\vec{a} = (a_0, a_1, \cdots)$ (an `std::vector <double>`), in order to work with TRK. For example, given some

$$f(x|\vec{a}) = a_0 + a_1 x + a_2 x^2, \tag{1}$$

we have that then,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = a_1 + 2a_2 x \quad \text{and} \quad \frac{\mathrm{d}^2 f}{\mathrm{d}x^2} = 2a_2. \tag{2}$$

Implemented into the C++ code, these would be written as

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

   return a0 + a1*x + a2*std::pow(x, 2.0);
}

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

   return a1 + 2.0*a2*x;
}

double ddf(double x, std::vector <double> params) {
   double a2 = params[2];

   return 2.0*a2;
}
```

Note that even if some or all the parameters are not needed to actually evaluate the function, it is still needed to maintain the same arguments in the function itself.

Finally, note that the TRK suite contains six example models that can be used within `exampleModels.cpp`; these will be listed near the end of this documentation.

1.2 Guessing Model Parameters and Slop Parameters

At it's core, the algorithm relies on an iterative method to determine the best fit model parameters, as well as the best fit for the values of the extrinsic scatter/ sample variance along both x and y, σ_x and σ_y of the data set (colloquially known as *slop*). So, for this to work, some guess for the model parameter vector $\vec{a}$, as well as σ_x and σ_y. In order to guess the slop values, think of them as being proportional to the scatter of the dataset that is not fully caused by the error bars (intrinsic scatter) of the datapoints. For example, if your data set is akin to being a cloud of datapoints, with relatively small error bars, it probably has large slop.

We've found that for most models, the range of $[0, 1]$ is where most good guesses for slop values lie, but in any case, the algorithm is powerful enough to find the best fit values of the slops even if the guesses are way off. As such, the algorithm should still work well even if these guesses are very far off/approximate, and the results are fairly independent of these guesses; there just needs to be some sort of starting point for the iteration.

1.3 Initializing an Instance of TRK

From here, we can set up for the usage of TRK by creating an instance of the `TRK` class, with data either weighted or unweighted. (You also can define prior distributions for the model parameters at this step, which we be covered in a later section). Given vectors of your data $\vec{x}$ and $\vec{y}$, associated error bar/(intrinsic) uncertainty data $\vec{\sigma}_x$ $\vec{\sigma}_y$ (not to be confused with extrinsic scatter/slop parameters σ_x and σ_y), and optional unequal data weights $\vec{w}$ (where all of these vectors are implemented as (`std::vector <double>`'s), the `TRK` class is instantiated as

```
  std::vector <double> x, y, sx, sy, w;
  std::vector <double> params_guess;
  double slopx_guess, slopy_guess;

  // UNWEIGHTED:
  TRK TRKtest = TRK(f, df, ddf, x, y, sx, sy, params_guess, slopx_guess,
 slopy_guess);

  // WEIGHTED:
  TRK TRKtest = TRK(f, df, ddf, x, y, w, sx, sy, params_guess, slopx_guess
 , slopy_guess);
```

, where I'm using the same model function and derivatives `f`, `df` and `ddf`, and `sx`, `sy` are the error bar vectors. **Note that the model parameter guess vector `params_guess` must be ordered the same way as the parameter vector argument in the model function definition.**

1.4 Optimizing Fitting Scale

As shown in the paper, the best fit results of TRK fitting can change if the y-datapoints are multiplied by some scaling factor s. Although going into the details of it is beyond the scope of this documentation, note that for TRK fitting, there is a minimum scale $s = a$ where the best fit extrinsic scatter/slop $\sigma_x \to 0$, a maximum scale $s = b$ where $\sigma_y \to 0$, and an *optimum scale* s_0 where the TRK likelihood function is maximized, i.e. the global best fit is found. As such, the results of running a TRK fit at the optimum scale s_0 will usually be better than the results of running the fit on other scales.

1.5 Running a Basic Fit

The TRK library includes an easy-to-use algorithm to obtain s_0 if it's not known *a priori*. However, if a fit is run at the default $s_0 = 1$ if desired, in many cases the results will probably be just fine. Finally, if this s_0 is known in advance for the model and dataset, it can be provided for the fitting algorithm. These three cases are implemented as (using the same `TRK` instantiation `TRKtest`)

```
    // Perform scale optimization and then run TRK fit:
    TRKtest.performSimpleTRKFit();

    // Skip scale optimization and run TRK fit with some provided scale s0
    double s0;
    TRKtest.performSimpleTRKFit(s0);

    //Skip scale optimization and run TRK fit at the default scale of 1.0:
    TRKtest.performSimpleTRKFit(1.0);
```

These cases do not determine model parameter uncertainties; this will be covered in the following section. Accessing the results of a fit is simple, and will be covered in the next subsection.

1.6 Accessing Fit Results

The various results of TRK fits can be found in the `results` attribute of the `TRK` class, including

```
    // std::vector <double> of best fit model parameters:
    TRKtest.results.bestFitParams;

    // best fit slop parameters (doubles):
    TRKtest.results.slop_x;
    TRKtest.results.slop_y;

    // optimum fit scale s0, minimum scale a, and maximum scale b found by
   scale optimization algorithm:
    TRKtest.results.optimumScale;
    TRKtest.results.minimumScale;
    TRKtest.results.maximumScale;
```

.

2 Model Parameter Uncertainty Estimation

Another core part of the TRK regression suite is the ability to generate the actual probability distributions of the model parameters, which are in turn used to estimate parameter uncertainties. To do this, the user has two options (again using the same instance of the `TRK` class, `TRKtest`:

```
    // Perform scale optimization, run TRK fit, and calculate parameter
   uncertainties/distributions:
    TRKtest.performTRKFit();

    // Skip scale optimization and run TRK fit with some provided scale s0,
   and calculate parameter uncertainties/distributions:
    double s0;
    TRKtest.performTRKFit(s0);
```

.

2.1 Accessing Parameter Uncertainty Results

The results of this uncertainty calculation can be called from the `results` attribute of the `TRK` instance. The parameter distributions, generated with an adaptive Monte Carlo Metropolis-Hastings sampler, are assumed to be Gaussian, potentially asymmetric—i.e. the width σ_+ on the positive side of the mean could be different from the width on the negative side σ_-—when analyzed. Each of the first three confidence intervals $\sigma_{\pm 1}$, $\sigma_{\pm 2}$ and $\sigma_{\pm 3}$ (analogous to the first three standard deviations) are found numerically, such that for each model parameter, the algorithm determines $(\sigma_{-1}, \sigma_{-2}, \sigma_{-3})$ and $(\sigma_{+1}, \sigma_{+2}, \sigma_{+3})$. Note that by default, the Metropolis-Hastings sampler generates $R = 100,000$ samples; this number can be changed, as shown in §6.1.

In the code, these values are found with (still using the same example of the instance of `TRK`) the `std::vector < std::vector < std::vector <double> > >` variable

```
    TRKtest.results.bestFit_123Sigmas
```

.

Each element of `bestFit\_123Sigmas` corresponds to a model parameter, given the vector of model parameters $\vec{a} = (a_1, a_2, \cdots)$. Schematically, `bestFit\_123Sigmas` is organized as

$$\begin{Bmatrix} [(\sigma_{-1}^{a_1}, \sigma_{-2}^{a_1}, \sigma_{-3}^{a_1}) \,,\, (\sigma_{+1}^{a_1}, \sigma_{+2}^{a_1}, \sigma_{+3}^{a_1})] \\ [(\sigma_{-1}^{a_2}, \sigma_{-2}^{a_2}, \sigma_{-3}^{a_2}) \,,\, (\sigma_{+1}^{a_2}, \sigma_{+2}^{a_2}, \sigma_{+3}^{a_2})] \\ \vdots \end{Bmatrix}, \tag{3}$$

where, for example, $\sigma_{-3}^{a_2}$, is the -3-sigma width for the second model parameter a_2.

Given that the distributions of the slop parameters σ_x and σ_y themselves are actually generated while fitting, the algorithm also computes the same type of uncertainties for these parameters, found in the form of `std::vector < std::vector < std::vector <double> >`'s as

```
    TRKtest.results.slopX_123Sigmas;
    TRKtest.results.slopY_123Sigmas;
```

.

These are organized exactly the same way as any given element of `bestFit\_123Sigmas`, e.g.

$$\texttt{TRKtest.results.slopX\_123Sigmas} = \left\{ \left(\sigma_{-1}^{\sigma_x}, \sigma_{-2}^{\sigma_x}, \sigma_{-3}^{\sigma_x} \right), \left(\sigma_{+1}^{\sigma_x}, \sigma_{+2}^{\sigma_x}, \sigma_{+3}^{\sigma_x} \right) \right\} \tag{4}$$

2.2 Accessing Parameter Distributions

If desired, the generated model parameter distributions themselves can be accessed, in the form of histograms, using the `std::vector < std::vector < std::vector <double> > >` variable

```
  TRKtest.results.paramDistributionHistograms;
```

.

This variable is organized schematically as

$$\begin{Bmatrix} \left(\vec{h}_1, \, \vec{e}_1 \right) \\ \left(\vec{h}_2, \, \vec{e}_2 \right) \\ \left(\vec{h}_3, \, \vec{e}_3 \right) \\ \vdots \end{Bmatrix}, \tag{5}$$

where for some model parameter a_i, the vector $\vec{h}_i$ contains the count values for each histogram bin in order, in the direction from negative a_i to positive a_i. The vector $\vec{e}_i$ contains the a_i-values for the vertical edges of these histogram bins, ordered along the same direction. So, the "left-most" bin of a parameter's histogram would have a left boundary at the a_i-value of the first element of $\vec{e}_i$, a right boundary at the a_i-value of the second element of $\vec{e}_i$, and a height/count value of the first element of $\vec{h}_i$.

2.3 Outputting Parameter Distributions to a File

If desired, the sampled parameter distribution can also be outputted in the form of a text file. To do this, set `outputDistributionToFile` to `true` for the chosen instance of `TRK`, and define the output path by setting the `outputPath` attribute of the TRK instance to be a directory path string, e.g.

```
  TRKtest.outputPath = "/Users/username/outputfolder";
```

.

3 Advanced: Using Prior Probability Distributions for Model Parameters

3.1 Instantiating Priors

If you have some prior knowledge of the distribution(s) of the model function parameter(s) (before necessarily having any data), this can be implemented into TRK Regression. (Both with the fitting routine, and the parameter distribution sampling). 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 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). To do so, 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_1 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);
```

3.1.2 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);
```

3.1.3 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);
```

3.1.4 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);
```

3.2 Using Your Priors with TRK

Now that you've instantiated your priors object, using it with TRK is easy; it's just an additional argument at the end of the TRK constructor, as shown below.

```
// WEIGHTED EXAMPLE:
TRK TRKtest = TRK(f, df, ddf, x, y, w, sx, sy, params_guess, slopx_guess,
    slopy_guess, testPriors);
```

As usual, following this instantiation of your TRK object, you can perform any of the TRK regression techniques.

4 Advanced: Finding Pivot Points to Remove Model Parameter Correlation

For certain linear/linearized models, it is possible to minimize correlation between model parameters by determining what is known as the correct "pivot point" of the model. (This minimized correlation corresponds to a non-tilted confidence ellipse between the two parameters in parameter space). For example, given a model of the classic linear form $y = mx + b$, it's typical that when fitting to this model, the best-fit m will be correlated to the best-fit b. This is because in choosing this model, we've (accidentally) placed the intercept of the line at $x = 0$, which can introduce a kind of "lever arm" effect whereby fitted m depends on b and vice versa. Such correlation can be minimized by choosing a particular "pivot point", x_p, somewhere in the middle of the data range, so that the model becomes $y(x) = m(x-x_p)+b_p$, where b_p is the "new" intercept defined by this changed model.

4.1 Defining Your Model's Pivot Point

The TRK suite can determine the optimal pivot point for any such linear, or *linearized* model, using an iterative method. While the details of this are beyond the scope of this documentation, the important part is that in order to find the pivot point of your model, TRK needs

1. the model to be in the form $y(x) = m(x - x_p) + b$,
2. a way to compute the slope and intercept parameters m and b given a vector of model parameters $\vec{a}$, and

3. the initial (x, y) dataset to be converted to (x, y) data that corresponds to the form of the new, linearized model.

For a simple linear model $y(x) = a_0 + a_1(x - x_p)$, the model function is already in this linear form, $m = a_1$ and $b = a_0$, and the dataset $(x, y) \to (x, y)$. To see an example of linearizing a model, consider the power law model

$$y(x) = a_0 \left(\frac{x}{10^{x_p}} \right)^{a_1} . \tag{6}$$

To linearize this, begin by taking the log of both sides as

$$\begin{aligned} \log_{10} y(x) &= \log_{10} \left[a_0 \left(\frac{x}{10^{x_p}} \right)^{a_1} \right] = \log_{10} a_0 + \log_{10} \left(\frac{x}{10^{x_p}} \right)^{a_1} = \log_{10} a_0 + a_1 \log_{10} \left(\frac{x}{10^{x_p}} \right) \\ &= \log_{10} a_0 + a_1 \left[\log_{10} x - \log_{10} 10^{x_p} \right] = \log_{10} a_0 + a_1 \left(\log_{10} x - x_p \right) . \end{aligned} \tag{7}$$

So, the linearized version of the power law has intercept $\log_{10} a_0 \to b$, slope $a_1 \to m$, and the data transforms as $\log_{10} y \to y$ and $\log_{10} x \to x$.

For this to work with the TRK code, you need to define functions that can take a vector of model parameters, and return the new, transformed intercept and slope variables. Continuing with this power law example, these would be implemented as

```
// returns the intercept for the linearized power law model given a vector
   of model parameters:
double powerlawIntercept(std::vector <double> params){
    return std::log10(params[0]);
}

// returns the slope for the linearized power law model given a vector of
   model parameters:
double powerlawSlope(std::vector <double> params){
    return params[1];
}
```

.

For the models included within `exampleModels.cpp` that can be linearized in this manner, we also include intercept and slope transformation functions within `exampleModels.cpp`, listed in the following section.

4.2 Running the Pivot Point Finder in TRK

In order to run the pivot point finding routine within a TRK fit, it's needed to

1. call the constructor for your instance of `TRK` with a model (and its derivatives) that uses `TRK::pivot` for the pivot point in your model equation

2. tell your instance of TRK to find the pivot points
3. provide your instance of `TRK` with the functions that define the intercept and slope of your linearized model.

To show how this is done in practice, continuing with the power law example, you'd define your model and its first two derivatives as

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

   return a0 * std::pow((x / std::pow(10.0, TRK::pivot)), a1);
}

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

   return std::pow(10.0, -1.0*TRK::pivot) * a0 * a1 * std::pow(std::pow
   (10.0, -TRK::pivot) * x, a1 - 1.0);
}

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

   return std::pow(10.0, -2.0*TRK::pivot) * a0 * (a1 - 1.0) * a1 * std::pow(
   std::pow(10.0, -TRK::pivot) * x, a1 - 2.0);
}
```

.

Then, when creating an instance of `TRK`, you'd do the usual (this example with unweighted data, without priors on the model parameters):

```
TRK TRKtest = TRK(powerlaw, dPowerlaw, ddPowerlaw, x, y, sx, sy,
   params_guess, slopx_guess, slopy_guess);
```

Finally, before running any fits, you need to both tell the algorithm to find the pivot point, and to define the intercept and slope transformation functions of the linearized model for the instance of `TRK` with the `linearizedIntercept` and `linearizedSlope` attributes of the `TRK` class, as follows (again using the power law example with the previously defined `powerlawIntercept` and `powerlawSlope`):

```
 // tell your instance of TRK to find the pivot point: this is false by
default.
 TRKtest.findPivotPoints = true;

 // define the intercept and slope transformation functions of the
linearized model:
 TRKtest.linearizedIntercept = powerlawIntercept;
 TRKtest.linearizedSlope = powerlawSlope;
```

Note: *The initial guess that is provided for model parameters should be for a model that has a pivot point,* ***`TRK::pivot`****, of zero*, because the pivot point-finder initializes with a default pivot point of zero. From here, any TRK fit can be run as usual. Following the fit, the final pivot point can be called with

```
 TRKtest.results.pivot;
```

, and the best fit model parameters and their distributions (with minimized correlation) can be called in the same way as defined earlier in this documentation.

5 Example Models (and pivot point transformation functions) Included in TRK

The TRK suite contains six example models that can be used within `exampleModels.cpp`. In the code, these functions are named:

1. Linear model $y(x) = a_0 + a_1(x - x_p)$ and it's first two derivatives: `linear`, `dLinear` and `ddLinear`
2. Quadratic model $y(x) = a_0 + a_1 x + a_2 x^2$ and it's first two derivatives: `quadratic`, `dQuadratic` and `ddQuadratic`
3. Cubic model $y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ and it's first two derivatives: `cubic`, `dCubic` and `ddCubic`
4. Power law model $y(x) = a_0 \left(\frac{x}{10^{x_p}}\right)^{a_1}$ and it's first two derivatives: `powerlaw`, `dPowerlaw` and `ddPowerlaw`
5. Exponential model $y(x) = a_0 10^{a_1(x-x_p)}$ and it's first two derivatives: `exponential`, `dExponential` and `ddExponential`
6. Logarithmic model $a_0 + a_1 \log_{10}(x/x_p)$ and it's first two derivatives: `logarithmic`, `dLogarithmic` and `ddLogarithmic`

The x_p variables are the "pivot points" of the models, which are covered in the previous section, which can be determined to minimize correlation between model parameters. They are simply set to zero if the the user does not desire to have the algorithm determine them. If the user does desire to determine these pivot points, then as discussed in the previous section, functions returning the intercept and slope of the linearized model given a vector of model parameters need to be provided. For the above example models, the functions for the intercept and slope of the linearized models are given in `exampleModels.cpp` as

1. Linear model: `linearIntercept` and `linearSlope`
2. Power law model: `powerlawIntercept` and `powerlawSlope`
3. Exponential model: `exponentialIntercept` and `exponentialSlope`
4. Logarithmic model: `logarithmicIntercept` and `logarithmicSlope`.

6 Other Options/Settings

In addition to the core material covered in the previous sections, there are more settings that can be configured for the TRK suite. The following section again uses `TRKtest` as the example instance of the `TRK` class.

6.1 Basic Settings

1. **Sample count for the generated model parameter distribution(s)**: To generate the model parameter probability distributions, an adaptive Monte Carlo Metropolis-Hastings sampler is run until some number, R of samples of the parameter space is produced. This samples are then binned into histograms, where are used as the probability distributions of the model parameters. If desired, R can be modified from the default $R = 100,000$, in order to modify the precision/smoothness of the generated distribution. This will of course also affect the time that it takes to run the sampler. This is done by modifying the `R` attribute of `TRK`, e.g. setting `TRKtest.R` equal to a new value.

2. **Printing Results and Showing Algorithm Progress**: The `printResults` attribute of the `TRK` class is `true` by default, which can be toggled; if `true`, results of running a TRK fit will be displayed, dependent on the specific fit/algorithms used. Similarly, the `TRK` class attribute `verbose` is `false` by default; if `true`, various steps of the algorithms will be printed out when reached during the fitting process. Attributes `verboseMCMC` and `verbosePivotPoints` can also be configured similarly; `verbose` being made `true` will turn both of these on.

6.2 Advanced Settings

These settings should only be modified if the user knows what they're doing.

1. **Tolerance for the fitting algorithm**: If desired, the tolerance for the downhill simplex algorithm that maximizes the TRK likelihood function can be changed, for higher or lower precision and/or accuracy of final best fit results, from the default value of `1e-5`. Of course, this will affect the time it takes to run the algorithm. This is done by modifying the `simplexTol` attribute of `TRK`, e.g. setting `TRKtest.simplexTol` equal to a new value.

2. **Burn-in count for the MCMC parameter distribution sampler**: To get a good sampling of the parameter distribution, the MCMC sampler discards the first b samples, in order to allow for the Markov chain to reach the "good" part of the distribution. This is known as giving the sampler a "burn-in" period, and is a very empirical method. We've found that for most models, even complicated nonlinear ones, the default $b = 10,000$ is suitable. If desired this can be altered by modifying the `burncount` attribute of `TRK`, e.g. setting `TRKtest.burncount` equal to a new value.

3. **Algorithm parallelization settings** Most of the TRK algorithm is not parallelizable; however, the core tangent-point finding routine (for the TRK likelihood function) and the routine for determining the minimum and maximum fitting scales *are* parallelizable. By default, the C++11 version of TRK uses the `std::async` library for this parallelization, and the C++17 version uses the new `std::for_each`, and `std::execution`. If desired, other parallelization options are available:

   (a) **Open MP parallelization**: set `TRKtest.openMPMultiThread = true`, `TRKtest.cpp17MultiThread = false` and `TRKtest.cpp11MultiThread = false`

   (b) **No parallelization**: set `TRKtest.openMPMultiThread = false`, `TRKtest.cpp17MultiThread = false` and `TRKtest.cpp11MultiThread = false`