Draft version April 25, 2024
Typeset using LaTeX **preprint** style in AASTeX631

An Alternate Method for Minimizing χ^2

Jennifer C. Yee and Andrew P. Gould

ABSTRACT

In this paper, we describe an algorithm and associated software package for maximizing the likelihood function of a set of parameters by minimizing χ^2. The key element of this method, is that it estimates the second derivative of the χ^2 function using first derivatives of the function to be fitted. These same derivatives can also be used to calculate the uncertainties in each parameter. We test this algorithm against several standard minimization algorithms in `SciPy.optimize.minimize()`. We show that it works almost as quickly as a gradient method, and so much faster than Markov Chain Monte Carlos.

1. THE DERIVATION

Our method builds upon the discussion in "χ^2 and Linear Fits" (**?**), which noted that expanding the approach to non-linear models went beyond the scope of those notes. Suppose that the function we want to minimize is a function $F(x)$ that is described by n parameters A_i (where we use A_i(instead of a_i) as a reminder that in the general case, they are non-linear). Considering the general (nonlinear) case, we can Taylor expand χ^2, in terms of the n parameters:

$$\chi^2 = \chi^2_0 + \sum_i \frac{\partial \chi^2}{\partial A_i} A_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 \chi^2}{\partial A_i \partial A_j} A_i A_j \tag{1}$$

$$= \chi^2_0 + \sum_i D_i * A_i + \sum_{i,j} B_{ij} * A_i A_j + ... \quad , \tag{2}$$

where

$$D_i \equiv \frac{\partial \chi^2}{\partial A_i} \tag{3}$$

$$B_{ij} \equiv (1/2) \frac{\partial^2 \chi^2}{\partial A_i \partial A_j} \quad . \tag{4}$$

Then,

$$\frac{\partial \chi^2}{\partial A_i} = -2 \sum_k \frac{(y_k - F(x_k))}{\sigma_k^2} \frac{\partial F(x_k)}{\partial A_i} \tag{5}$$

and

$$\frac{\partial^2 \chi^2}{\partial A_i \partial A_j} = -2 \sum_k \left[\frac{1}{\sigma_k^2} \frac{\partial F(x_k)}{\partial A_i} \frac{\partial F(x_k)}{\partial A_j} + \frac{(y_k - F(x_k))}{\sigma_k^2} \frac{\partial^2 F(x_k)}{\partial A_i \partial A_j} \right] \quad . \tag{6}$$

In the special case of a linear function, $F(x) = \sum_i a_i f_i(x)$ then

$$\frac{\partial F(x)}{\partial a_i} = f_i(x) \quad \text{and} \quad \frac{\partial^2 F(x)}{\partial a_i \partial a_j} = 0, \tag{7}$$

so the second term disappears, and we find that the solution (derived from second derivative of χ^2) can be expressed in terms of products of the FIRST derivatives of the general functional form. For the general case, we simply make the approximation that the second derivative term can be neglected; i.e.,

$$\frac{\partial^2 \chi^2}{\partial A_i \partial A_j} \approx -2 \sum_k \frac{1}{\sigma_k^2} \frac{\partial F(x_k)}{\partial A_i} \frac{\partial F(x_k)}{\partial A_j} \quad . \tag{8}$$

Hence, there are three ways to generalize Newton's method (actually discovered by Simpson) to multiple dimensions:

1. Use only first derivatives of the χ^2 function (which is what Simpson did in 1-D), the so-called gradient method.
2. Taylor expand χ^2 and truncate at second term, then solve this (very inexact equation) exactly by inversion of the matrix of second derivatives (Hessian).
3. First generalize Simpson's idea that a 1-D function is well described by its first derivative (which can easily be solved exactly) to several dimensions (i.e., assume the function is well described by a tangent plane) and solve this exactly, as is done here.

Because first derivatives are a lot more stable than second derivatives, this algorithm could potentially be a lot more stable for situations in which the derivatives are derived numerically, which we investigate in the next section.

2. IMPLEMENTATION

We have implemented the above algorithm in the `sfit_minimizer` package. The goal was to make the API similar to that of `SciPy.optimize.minimize()`. Hence, the basic calling sequence is

```
result = sfit_minimizer.minimize(my_func, x0=initial_guess)          (9)
```

where `my_func` is an object of the type `sfit_minimizer.SFitFunction()` in which the user defines either the model, $F(x_k)$, or the residual, $y_k - F(x_k)$, calculation (i.e., the method `my_func.model()` or `my_func.residuals()`) and the partial derivatives of the function to be minimized, $\partial F(x_k)/\partial A_i$ (i.e., the method `my_func.df()`). The package includes a simple example (`example_00_linear_fit.py`) for fitting a linear model to demonstrate this usage.

The `sfit_minimizer.SFitFunction()` class contains methods that use the partial derivative function to calculate the next step from the D_i and B_{ij} following the method in **?** for linear functions. That is, D_i and B_{ij} are calculated from Equations 3 and 4, respectively. Then, the step for each parameter, Δ_i, is

$$\Delta_i = \sum_j C_{ij} D_j \quad \text{where} \quad C_{ij} \equiv B_{ij}^{-1} \quad , \tag{10}$$

which is returned by `sfit_minimizer.SFitFunction.get_step()`. Hence, the new value of A_i is calculated by `sfit_minimizer.minimize()` to be

$$A_i = A_{i,0} + \epsilon \Delta_i \quad . \tag{11}$$

In `sfit_minimizer.minimize()`, the user has the option to specify the value of ϵ or to make use of an adaptive step size, which starts at $\epsilon = 0.001$ and becomes larger as the minimum is approached.

Ultimately, `sfit_minimizer.minimize()` returns an `sfit_minimizer.SFitResults()` object that contains attributes similar to the object returned by `SciPy.optimize.minimize()`. These include the best-fit values of the parameters, `x`, and their uncertainties `sigma` (i.e., $\sigma_i = \sqrt{C_{ii}}$). For the rest of this paper, we will refer to our algorithm as `SFit` for brevity.

3. PERFORMANCE TEST

To test the performance of `SFit`, we run the algorithm on a sample of microlensing events and compare the results to several algorithms as implemented in `SciPy.optimize.minimize()`. For the comparison, we select the `Nelder-Mead` (?), `Newton-CG` (?), and `BFGS` (?) algorithms in `SciPy.optimize.minimize()`. The `Nelder-Mead` algorithm is a simplex algorithm, so only relies on evaluating the χ^2. In contrast to our algorithm, the `Newton-CG` and `BFGS` algorithms use the jacobian of the likelihood function for the minimization. In all cases, we set `tol = 1e-5`.

We select our sample from microlensing events discovered in 2018 by the Korea Microlensing Telescope Network (KMTNet; ???). We use only "clear" microlensing events with reported fit parameters. We eliminate any events that were flagged as anomalous in the 2018 AnomalyFinder Search (although possible finite source or buried host events were left in the sample; ??). These cuts left 1823 events in the sample.

For this sample, we use the online, I-band, pySIS (?) data from the KMTNet website (https://kmtnet.kasi.re.kr/ulens/). KMTNet takes data from three different sites and has multiple, sometimes overlapping, fields of observations. We treat data from different sites and different fields as separate datasets. For each dataset, we calculate the mean sky background and standard deviation as well as the mean and standard deviation of the full-width-half-max for each observation. We eliminate points with sky background more than 1 standard deviation above the mean or full-width-half-max more than 3 standard deviations above the mean. This removes a large fraction of the outliers from the data.

We fit each event with a standard, point-lens model, which is parameterized by the three Paczyński parameters: t_0, u_0, and $t_{\rm E}$ (for the definitions of these parameters see, e.g., ?). In addition, there are two flux parameters used to scale each dataset, k, to the model, A: $f_{{\rm mod},k} = f_{{\rm S},k}A + f_{{\rm B},k}$. We use `MulensModel` (?) to calculate the model, its derivatives, and the jacobians. Before doing the fitting, we determined the starting values for t_0, u_0, and $t_{\rm E}$ using the following procedure. The KMTNet website reports an estimate for the values of these parameters. To ensure an optimum starting point for the fits, we tested a series of values of $u_{0,i} = [0.01, 0.3, 0.7, 1.0, 1.5]$ and calculated $t_{{\rm E},i} = t_{\rm E}u_0/u_{0,i}$. We performed a linear fit of the flux parameters $(f_{{\rm S},k}, f_{{\rm B},k})$ for each set of parameters i and chose the values of $u_{0,i}$ and $t_{{\rm E},i}$ that produced the smallest χ^2 to initialize the fits.

We calculated several metrics to evaluate the performance of each algorithm. First, for a given event, we compared the χ^2 of the best-fit reported by each algorithm to the best (minimum) value reported out of the four fits. We divide these comparisons by whether or not the algorithm reported that the fit was successful. The results are given in Table 1. `SFit` was successful in $1326/1823 = 73\%$ of cases and in all such cases found the best-fit solution. This may be contrasted with `Newton-CG`, which had a higher reported rate of success (89%), but was only within $\Delta\chi^2$ of the best-fit solution 40% of the time. By comparison, the `BFGS` algorithm reported success only 59% of the time, but was within $\Delta\chi^2$ of the minimum 99% of the time. Finally, the `Nelder-Mead` algorithm found the minimum 98% of the time with only a few rare instances of false success.

The second set of metrics we calculated was the number of χ^2 function evaluations (see Table 2). The `Nelder-Mead` algorithm required the most evaluations, while `Newton-CG` required the least (but, as pointed out above, frequently failed to find the true χ^2 minimum). `SFit` was more efficient than `Nelder-Mead` but in general required more function evaluations than `BFGS` to find the minimum.

4. SUMMARY

We have presented an alternative algorithm (and its Python implementation) for finding the best-fit parameters of a fitting function that builds upon the methodology presented in **?**. This algorithm uses partial derivatives to the fitted function to approximate the second derivatives of the likelihood (χ^2) function. in order to calculate step sizes. **?** showed how these same derivatives can be used to calculate the uncertainties in the fitted parameters.

By testing this algorithm on a sample of microlensing events, we show that it is more efficient at fitting Paczyński curves than the `Nelder-Mead` simplex algorithm. Consequently, it is also substantially more efficient than a Markov Chain Monte Carlo, which generally requires hundreds to thousands of function evaluations in order to estimate the value and uncertainties of parameters. In addition, we found that the `Newton-CG` algorithm did not perform reliably for microlensing events. So, before using this algorithm for microlensing events, further investigation should be carried to assess whether this could be remedied by adjusting the algorithm parameters. Finally, we note that the `BFGS` algorithm performed best out of all of the algorithms (including the `SFit` algorithm presented here), although it had the lowest reported rate of success.

The Python implementation of this algorithm, including its specific application to microlensing events, can be found on GitHub.

ACKNOWLEDGMENTS

We thank Radek Poleski and Keto Zhang for helpful discussions in the development of the code. J.C.Y. acknowledges support from U.S. NSF Grant No. AST-2108414. This research has made use of publicly available data (https://kmtnet.kasi.re.kr/ulens/) from the KMTNet system operated by the Korea Astronomy and Space Science Institute (KASI) at three host sites of CTIO in Chile, SAAO in South Africa, and SSO in Australia. Data transfer from the host site to KASI was supported by the Korea Research Environment Open NETwork (KREONET).

Table 1. Number of Fits with $\Delta\chi^2 \geq X$ of the Best-Fit

		$\Delta\chi^2 \geq$			
Algorithm	Total	0.1	1.0	10.0	100.0
All 1823 Events:					
Algorithm Reported Success:					
`SFit`	1326	0	0	0	0
`Nelder-Mead`	1557	4	3	2	2
`Newton-CG`	1631	1058	905	743	530
`BFGS`	1075	0	0	0	0
Algorithm Reported Failure:					
`SFit`	497	496	490	444	242
`Nelder-Mead`	266	111	41	19	10
`Newton-CG`	192	125	81	67	55
`BFGS`	748	23	21	19	18
1326 Events for which `SFit` Reported Success:					
Algorithm Reported Success:					
`Nelder-Mead`	1322	1	1	0	0
`Newton-CG`	1266	729	591	485	386
`BFGS`	865	0	0	0	0
Algorithm Reported Failure:					
`Nelder-Mead`	4	3	3	2	1
`Newton-CG`	60	32	32	31	26
`BFGS`	461	5	5	4	3
497 Events for which `SFit` Reported Failure:					
Algorithm Reported Success:					
`Nelder-Mead`	235	3	2	2	2
`Newton-CG`	365	329	314	258	144
`BFGS`	210	0	0	0	0
Algorithm Reported Failure:					
`Nelder-Mead`	262	108	38	17	9
`Newton-CG`	132	93	49	36	29
`BFGS`	287	18	16	15	15

Table 2. Number of χ^2 Function Evalutions

Algorithm	Mean	Median	StdDev	Max
`SFit`	142.6	125	121.2	999
`Nelder-Mead`	348.8	303	136.6	603
`Newton-CG`	44.5	21	80.2	633
`BFGS`	122.1	44	217.1	830