2018-09-01

Linear Least Squares Curve Fitting

R. Steven Turley
Brigham Young University, turley@byu.edu

Linear Least Squares

Version 1.0

R. Steven Turley

August 31, 2018

Contents

1. Introduction

This article describes how to find the best fit to a function of the form

$$f(x;\vec{b}) = \sum_{k=1}^{p} b_k g_k(x) \tag{1}$$

to a set of data points (x_i, y_i). It is linear in the sense that f is linear in the parameters b_k. It is called "least squares" because by "best fit" I mean the function which finds

the set of parameters b_k which minimizes χ^2, the sum of the squares of the differences between $f(x_i;\vec{b})$ and y_i.

$$\chi^2 = \sum_{i=1}^{n} [f(x_i;\vec{b}) - y_i)]^2 \tag{2}$$

If the data is heteroscedactic (i.e. if the extent of the deviations of y_i from $f(x_i,\vec{b})$ varies across the range of x_i, the appropriate function to minimize is a weighted sum of the squares. This can be written in terms of the weights w_i or in terms of the relative uncertainties at each data point σ_i.

$$\chi^2 = \sum_i w_i [f(x_i;\vec{b}) - y_i)]^2 \tag{3}$$

$$\chi^2 = \sum_i \left[\frac{f(x_i;\vec{b}) - y_i)}{\sigma_i} \right]^2 \tag{4}$$

$$w_i = \sigma_i^{-2} \tag{5}$$

A good, but somewhat dated reference for the derivations here is the current version of Bevington[1], the book I first learned statics from. Some other helpful references which helped me with these derivations are the article in Wikipedia[2] and the article in MathWorld[3].

A special case of linear least squares fitting in polynomial fitting which will be considered separately in another article[4].

2. Finding Parameters

The function χ^2 can be minimized by finding the point where it's gradient is zero. This is true if all of the partial derivatives[1] of χ^2 with respect to the parameters b_i are equal to zero. In the case of unweighted least squares in Eq. 2

$$\frac{\partial \chi^2}{\partial b_j} = \frac{\partial}{\partial b_j} \sum_i [f(x_i;\vec{b}) - y_i)]^2 \tag{6}$$

$$= 2 \sum_i [f(x_i;\vec{b}) - y_i)] \left(\frac{\partial f(x_i;\vec{b})}{\partial b_j} \right). \tag{7}$$

$$= 0 \tag{8}$$

The partial derivative is simple when f has the linear form in Eq. 1.

$$\partial f(x;\vec{b}) \partial b_j = g_j(x) \tag{9}$$

[1] A partial derivative ∂ / b_i is the derivative with respect to b_i holding all other quantities constant.

Substituting Eq. 1 and Eq. 9 into Eq. 7,

$$0 = \sum_i [f(x_i; \vec{b}) - y_i)] g_j(x_i) \tag{10}$$

$$= \sum_i \left[\sum_k b_k g_k(x_i) - y_i \right] g_j(x_i) \tag{11}$$

$$\sum_{i,k} g_k(x_i) g_j(x_i) b_k. = \sum_i g_j(x_i) y_i. \tag{12}$$

If there are p parameters b_k, Eq. 12 represents p linear equations with p unknowns which can be written in matrix form.

$$\begin{pmatrix} \sum_i g_1(x_i)^2 & \cdots & \sum_i g_1(x_i) g_p(x_i) \\ \vdots & \vdots & \vdots \\ \sum_i g_n(x_i) g_1(x_i) & \cdots & \sum_i g_n(x_i) g_p(x_i) \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} \sum_i g_1(x_i) y_i \\ \vdots \\ \sum_i g_p(x_i) y_i \end{pmatrix} \tag{13}$$

Eq. 13 can be solved for the parameters b_k using standard matrix techniques.

The weighted fits in Eq. 3 can similar be solved.

$$\begin{pmatrix} \sum_i w_i g_1(x_i)^2 & \cdots & \sum_i w_i g_1(x_i) g_p(x_i) \\ \vdots & \vdots & \vdots \\ \sum_i w_i g_n(x_i) g_1(x_i) & \cdots & \sum_i w_i g_n(x_i) g_p(x_i) \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} \sum_i w_i g_1(x_i) y_i \\ \vdots \\ \sum_i w_i g_p(x_i) y_i \end{pmatrix} \tag{14}$$

Substituting Eq. 5 into Eq. 14 gives the equation for weighted linear least squares in terms of the uncertainties σ_i.

$$\begin{pmatrix} \sum_i g_1(x_i)^2/\sigma_i^2 & \cdots & \sum_i g_1(x_i) g_p(x_i)/\sigma_i^2 \\ \vdots & \vdots & \vdots \\ \sum_i g_n(x_i) g_1(x_i)/\sigma_i^2 & \cdots & \sum_i g_n(x_i) g_p(x_i)/\sigma_i^2 \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} \sum_i g_1(x_i) y_i/\sigma_i^2 \\ \vdots \\ \sum_i g_p(x_i) y_i/\sigma_i^2 \end{pmatrix} \tag{15}$$

If the g_k are monomials with $g_k(x_i) = x_i^{k-1}$, then Eq. 13 takes on a particularly simple form.

$$\begin{pmatrix} n & \cdots & \sum_i x_i^{p-1} \\ \vdots & \vdots & \vdots \\ \sum_i x_i^{n-1} & \cdots & \sum_i x_i^{n+p-2} \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix} = \begin{pmatrix} \sum_i y_i \\ \vdots \\ \sum_i x_i^{p-1} y_i \end{pmatrix} \tag{16}$$

Fitting polynomials is discussed further in my polynomial fitting article[4].

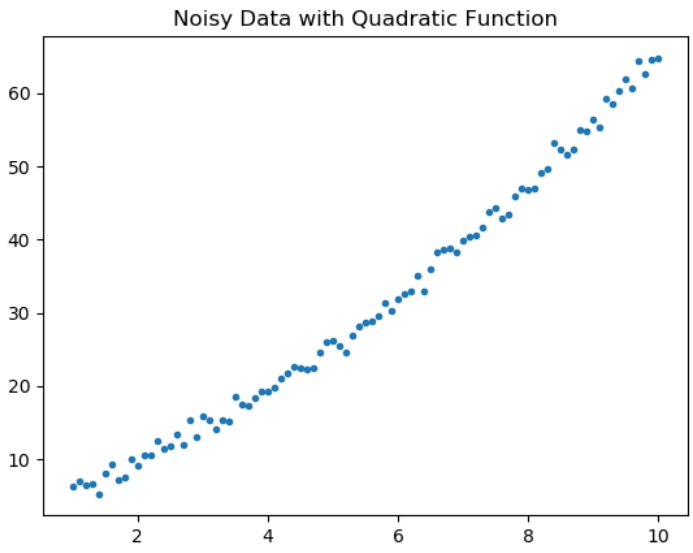


Figure 1: Quadratic function with random noise added.

3. Residuals

A measure of the quality of the fit are the residuals r_i, which are the differences between the data points y_i and the fit function f.

$$r_i = y_i - f(x_i; \vec{b}) \tag{17}$$

It is always a good idea to plot the residual or the fit and the data when you're fitting to make sure it looks reasonable. Two common problems which are easy to spot graphically are an inaccurate or incomplete set of functions g_k and incorrect weights w_i. Fig. 1 is some data from a quadratic function with noise. I fit the data with a linear function $y = mx + b$ and plotted the residual in Fig. 2. Notice how the residual jumps around but is characteristically below zero for parts of the and characteristically above zero for other parts. This is characteristic of using a wrong or incomplete function top fit the data. If I fit the data to a quadratic curve, I get the residual in Fig. 3. Notice how the data is randomly on either side of the origin in this case.

The need for a weighted fit can be illustrated by fitting the data in Fig. 4. Fig. 5 shows the data and the fit when I use equal weights at each point (i.e. an unweighted fit). Notice that the fit is considerable closer to the data points in the region where the function is large compared to the region where the function is small. Fig. 6 is a fit to the same data using the same fit function but with weights proportional to the signal.

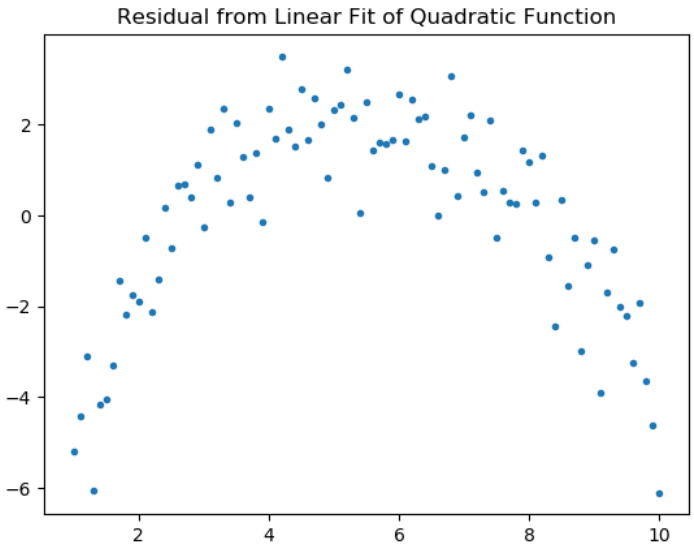


Figure 2: Residual from the fit of the data in Fig. 1 to a straight line.

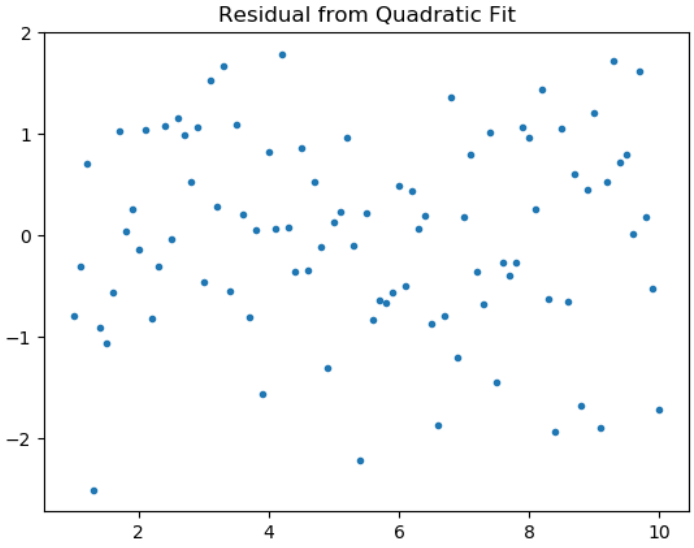


Figure 3: Residual from the fit of the data in Fig. 1 to a quadratic polynomial.

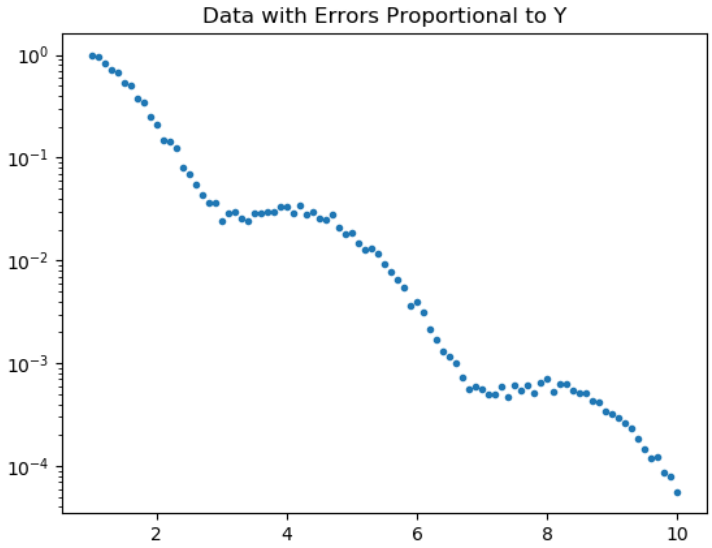


Figure 4: Data with large differences in magnitude and an error which is proportional to the signal.

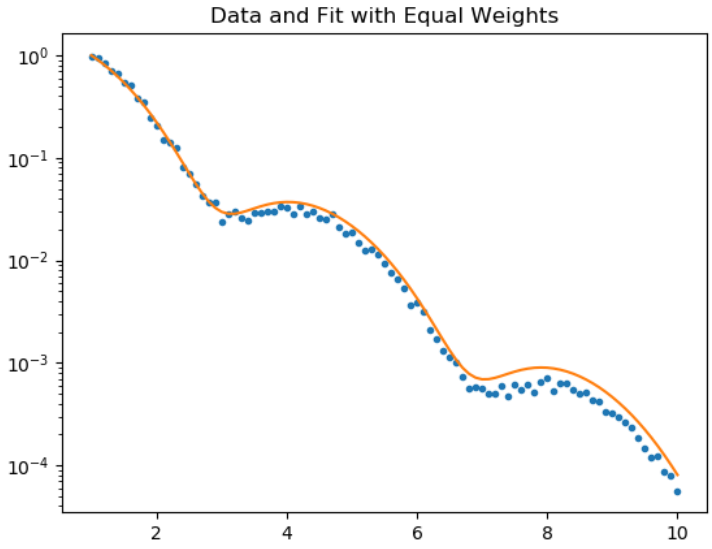


Figure 5: Data and fit for the data in Fig. 4 using equal weights.

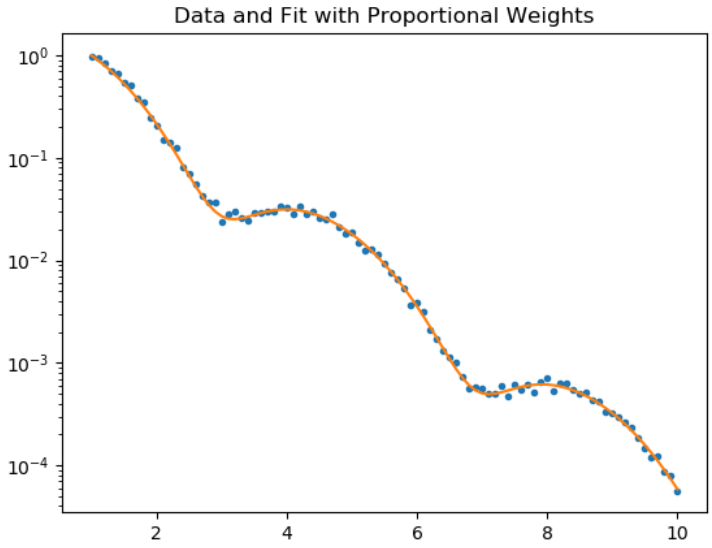


Figure 6: Data and fit for the data in Fig. 4 using proportional weights.

4. Parameter Uncertainties

As shown in my general uncertainties article[5], the uncertainties in the fit parameters can be estimated from covariance matrix C which can be computed from the Jacobian J.

$$J = \begin{pmatrix} \partial f(x_1;\vec{b})/\partial b_1 & \cdots & \partial f(x_1;\vec{b})/\partial b_p \\ \vdots & \vdots & \vdots \\ \partial f(x_n;\vec{b})/\partial b_1 & \cdots & \partial f(x_n;\vec{b})/\partial b_p \end{pmatrix}. \tag{18}$$

From Eq. 1, it is apparent that

$$\frac{\partial f(x_i;\vec{b})}{\partial b_k} = g_k(x_i) \tag{19}$$

so that

$$J = \begin{pmatrix} g_1(x_1) & g_2(x_1) & \cdots & g_{p-1}(x_1) & g_p(x_1) \\ g_1(x_2) & g_2(x_2) & \cdots & g_{p-1}(x_2) & g_p(x_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_1(x_{n-1}) & g_2(x_{n-1}) & \cdots & g_{p-1}(x_{n-1}) & g_p(x_{n-1}) \\ g_1(x_n) & g_2(x_nj) & \cdots & g_{p-1}(x_n) & g_p(x_n) \end{pmatrix}. \tag{20}$$

The covariance matrix C can be computed from J.

$$C = s_y^2(J^T J)^- 1 \tag{21}$$

The uncertainty in y, s_y can be estimated from the residuals

$$r_i = y - i - f(x_i; \vec{b}) \tag{22}$$

where $\vec{b}$ has the best values from the fit.

$$s_y \approx \frac{1}{n-p} \sum r_i^2. \tag{23}$$

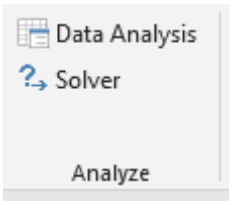
The uncertainty in the fit parameter b_i is

$$s_i = \sqrt{C_{ii}}\ . \tag{24}$$

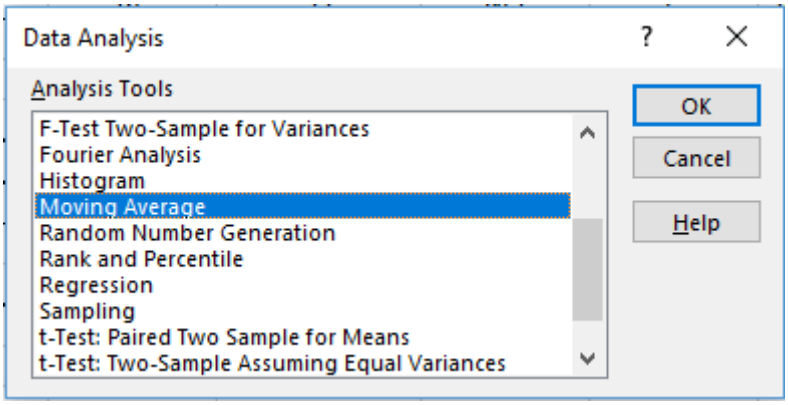
A. Computer Codes

A.1. Excel

The original motivation for this work was to understand the output of a regression analysis in Excel. After installing the Data Analysis Add-in, I saw the following section in the Data tab.



I clicked on the Data Analysis item and it brought up the following menu.



I picked the Random Number Generation item to create a normally distributed 5 random numbers in Column A with mean 0 and standard deviation 1.

	A	B	C
1	0.712705	1	18.71271
2	1.037488	2	34.03749
3	0.292159	3	48.29216
4	-0.71953	4	62.28047
5	-1.18729	5	76.81271
6			
7	m	15	
8	b	3	

I filled column B with the x values I wanted to fit and set column C using the following formula for example for cell C2: `=A2+`$B7$`*B2+`$B8$. Then I went back to the Data Analysis menu and selection regression. The y input range was the data in Column C. The x input range was the data in column B. The output range was the cell E1. The data I was most interested in from the regression was the second table under the ANOVA heading.

	Coefficients	*Standard Error*	*t Stat*	*P-value*
Intercept	4.694206958	0.430462388	10.90503396	0.001650427
X Variable 1	14.44430033	0.129789293	111.2903842	1.59945E-06

Lower 95%	*Upper 95%*
3.324283522	6.064130394
14.03125287	14.85734779

This is the data I wanted to understand by comparing it with the output of other programs. I determined that the coefficients were what one gets doing a linear least squares fit as outlined in this article. It agreed with the other programs I tried. Likewise, the Standard Error column is the estimated standard deviation in the fit parameters as computed by other programs. The t-stat column is the ratio of the coefficient to the standard deviation. The P-value is (correctly calculated as following from the integral of the Student t distribution $s(t, \nu)$ where t is the t statistic described earlier and ν is the number of degrees of freedom. It is equal to 3 in this case, the number of data points (5) minus the number of fit parameters (2).

$$p = 2 * \left(1 - \int_{\tau=0}^{t} s(\tau, \nu)\right) \tag{25}$$

A.2. MATLAB

I duplicated the Excel calculation in Matlab using the polyfit routine and their prescription for calculating the standard errors using the information in the S structure.

```
% Polynomial fit coefficient uncertainties
noise=[0.712705059,
1.037487891,
```

```
0.292159257,
-0.71952627,
-1.187286216]';
m=15;
b=3;
x=1:5;
y=noise + m*x + b;
[p,S] = polyfit(x,y,1);
p
ste = sqrt(diag(inv(S.R)*inv(S.R')).*S.normr.^2./S.df)
```

You'ree note that these values agree with the Excel calculations.

A.3. Mathematica

Here is the Mathematica code I used to reproduce the Excel calculation.

```
noise = {0.712705059,
  1.037487891,
  0.292159257,
  -0.71952627,
  -1.187286216};
x = Range[1, 5];
m = 15.;
b = 3.;
y = noise + m*x + b
data = Transpose[{x, y}];
lm = LinearModelFit[data, w, w];
lm["ParameterTable"]
```

It produced the following output, which agreed with Excel.

	Estimate	Standard Error	t-Statistic	P-Value
1	4.69421	0.430462	10.905	0.00165043
w	14.4443	0.129789	111.29	1.59945×10^{-6}

I also did a distribution check to see if average estimated variance in the fit parameters matched the actual variance in those parameters with 10,000 different random noise samples. Here is the Module to compute the fit parameters and parameter error estimates.

```
myFit[] := Module[{noise, m = 15., b = 3., x, y, data, lm, w},
  x = Range[1, 5];
  noise = RandomReal[NormalDistribution[], 5];
  y = noise + m*x + b;
  data = Transpose[{x, y}];
  lm = LinearModelFit[data, w, w];
```

```
  Join[lm["BestFitParameters"], lm["ParameterErrors"]]]
```

Here is the code to do the 10,000 trials.

```
trials = 10000;
pars = Table[myFit[], {i, trials}];
spars = Total[pars]/trials
ssqpars = Total[pars*pars]/trials;
Sqrt[ssqpars - spars^2]
```

The results were as follows.

```
{2.98924, 15.003, 0.962881, 0.29032}
{1.05124, 0.314441, 0.408231, 0.123086}
```

These don't agree with each other as well as my Julia results and I'm not sure why. However, they are close enough (within 10%) that I'm not too concerned.

A.4. Python

The following python code was intended to calculate the same thing as in the other languages.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Aug 31 12:00:20 2018

@author: rstur_000
"""
import numpy as np
# import matplotlib.pyplot as plt

noise=[0.712705059,
     1.037487891,
     0.292159257,
     -0.71952627,
     -1.187286216]
m=15
b=3
x=np.linspace(1,5,5)
y=noise+m*x+b

p, cov = np.polyfit(x,y,1,full=False, cov=True)
pfit = np.poly1d(p)
yfit = pfit(x)
res = y-yfit
mse = sum(res*res)/3
sigma = np.sqrt(mse)
```

```
print("fit standard deviation in y = {:.4f}".format(sigma))
std = np.sqrt(np.diag(cov))
print("slope     = {:8.5f} +/- {:.5f}".format(p[0],std[0]))
print("intercept = {:8.5f} +/- {:.5f}".format(p[1],std[1]))
print("adjusted slope error is {:.4f}".format(std[0]/sigma))
print("adjusted intercept error is {:.4f}".format(
      std[1]/sigma))

# Try something else
p, cov = np.polyfit(x, y, 1, cov=True)

# Test these variances, they disagree with everyone else
sump = np.zeros(2)
sumsq = np.zeros(2)
sumstd = np.zeros(2)
trials=100
for i in range(100):
    noise = np.random.normal(size=5)
    y = noise + m*x + b
    p, cov = np.polyfit(x,y,1,full=False, cov=True)
    sump += p
    sumsq += p**2
    sig = np.sqrt(np.diag(cov))
    sumstd += sig
meanp = sump/trials
meansq = sumsq/trials
meanstd = sumstd/trials
calcstd = np.sqrt(meansq-meanp**2)
print("calculted std = {:.5f}, {:.5f}".format(calcstd[0],
      calcstd[1]))
print("mean std: ",meanstd)
print("meanstd/calcstd = ", meanstd/calcstd)
```

The output disagrees with the other programs. It gets the same fit parameters, but different calculated uncertainties. The distribution of fit parameters were similar to the other programs.

```
fit standard deviation in y = 0.4104
slope     = 14.44430 +/- 0.22480
intercept =  4.69421 +/- 0.74558
adjusted slope error is 0.5477
adjusted intercept error is 1.8166
calculted std = 0.29739, 1.04233
mean std:  [0.49968945 1.65728242]
meanstd/calcstd =  [1.68022641 1.58997506]
```

A.5. Julia

Most of the lengthy calculations and checks for the derivations in this article were done with Julia. In the following section I will list code snippets from the `LsqFit` module which I used for the core of these calculations. That is followed by a section with code used to produce the figures in this report and test the results.

A.5.1. LsqFit.jl

I rewrote the library routine `LsqFit.jl` to fix some problems in the routines and to work with Julia 1.0. I also added some linear and polynomial curve fitting routines and used the BFGS fitting routine instead of the Levenberg-Marguardt routine it used (and which I regard as inferior in most cases). This section has snippets of the code form the LsqFit module and comments.

The module uses the CurveFitResult concrete type to store the results of both linear and nonlinear fits.

```
struct CurveFitResult
    dof::Int
    param::Vector{Float64}
    resid::Vector{Float64}
    jacobian::Matrix{Float64}
    converged::Bool
    wt::Array{Float64}
    mse::Float64
end
```

The `dof` variable is the number of degrees of freedom in the fit. This is equal to the number of data points minus the number of fit parameters. The `param` vector are the fitted parameters. The `resid` is the residual (the difference between the fit and the data at each x point. The `jacobian` is the Jacobian of the fit function which is explained in Sec. 4. For nonlinear fits, `converged` is set equal to true if the fit converges. The parameters are still updated, even if the fit doesn't converge and may still be useful. The `wt` vector is the vector if weights passed to the fitting function. If no weights are supplied, this is set equal to a vector of ones. The `mse` field is the mean square error. Is is the sum of the squares of the residuals divided by the number of degrees of freedom.

The `curve_fit` function fits data to a model using a nonlinear least squares algorithm.

```
function curve_fit(model::Function,
                   xpts::AbstractArray, ypts::AbstractArray,
                   wt::AbstractArray, p0::Vector; kwargs...)
    # construct a weighted cost function, with a vector weight
    # for each ydata. For example, this might be
    # wt = 1/sigma where sigma is some error term
    f(p) = wt .* ( model(xpts, p) - ypts )
    ssq(p) = sum(abs2, f(p))
```

```
    results = optimize(ssq, p0, BFGS(),
        Optim.Options(x_tol = 1e-10,
            f_tol = 1e-10,
            iterations = 10_000))
    # Fill the result with useful data
    dof = length(xpts) - length(p0)
    p = Optim.minimizer(results)
    res = f(p)
    g = Calculus.jacobian(f)
    jac = g(p)
    conv = Optim.converged(results)
    mse = sum(abs2, res)/dof
    CurveFitResult(dof, p, res, jac, conv, wt, mse)
end
```

The routine takes five required arguments and permits (but ignores) optional arguments.

model: a function with two parameters x and p that computes the fit function with a given set of parameters at the vector of data x.

xpts: the x_i data points

ypts: the y_i data points

wt: an array of weights used to weight the residual. Note that it uses the weight differently than described in this article. The variable w_i in Eq.3 is the square of the elements in the passed `wt` array.

p0: the initial guess for the parameter values. Since nonlinear least squares is an iterative technique, it needs a starting point.

The routine defines the function `f(p)` to be the weighted residual at each step between the model and the data. The `ssq` function is the sum of squares of the weighted residuals and is what the optimizer minimizes. The routine uses the `optimize` function from the library `Optim` package to do the optimization. I've chosen to do use the BFGS (Broyden-Fletcher-Goldfarb-Shanno) method, which has worked well for me over the years. The options are set to allow convergence under most normal conditions using numerical derivatives in the gradient and hessian. After the fit is complete, the numerical jacobian is calculated using the `Calculus` package and the results are returned as a `CurveFitResult` type.

The `linear_fit` function does the unweighted linear least squares calculations which are the focus of this paper.

```
function linear_fit(basis::AbstractArray, xpts::AbstractArray,
        ypts::AbstractArray)
    order=length(basis)
    A=zeros(order,order)
```

```
    y=zeros(order)
    for i=1:order
        for j=1:order
            A[i,j] = sum(basis[i](xpts).*basis[j](xpts))
        end
        y[i] = sum(ypts.*basis[i](xpts))
    end
    b = A\y
    npts = length(xpts)
    dof = npts-order
    yf = zeros(npts)
    for i=1:order
        yf += b[i].*basis[i](xpts)
    end
    res = yf .- ypts
    jac=Matrix{Union{Missing, Float64}}(missing,order,npts)
    for i=1:order
        jac[i,:] = basis[i](xpts)
    end
    conv = true
    wt = ones(npts)
    mse = sum(abs2, res)/dof
    CurveFitResult(dof, b, res, jac', conv, wt, mse)
end
```

It has three calling parameters.

basis: an array of basis functions by which the fit parameters are multiplied

xptsan array of the independent variables

yptsan array of the dependent variables to be fit

No initial guesses are required in this case since the fit parameters `b` can be computed directly. The array `yf` is the fit function evaluated with the final fit parameters. As noted in Sec. 4, the Jacobian can be computed directly in this case. Note that this routine uses unweighted fitting and an unweighted Jacobian.

The `poly_fit` function is a special case of the `linear_fit` function which it calls directly.

```
function poly_fit(order::Int, xpts::AbstractArray,
                  ypts::AbstractArray)
    lf = [x->x.^i for i=0:order]
    linear_fit(lf, xpts, ypts)
end
```

The only difference between these two functions if that the `poly_fit` function takes the order of the polynomial to fit as its first argument. From that, it computes the array of basis functions needed by `linear_fit`.

The `standard_error` function estimates the uncertainties in the fit parameters as explained in Sec. 4.

```
function standard_error(fit::CurveFitResult;
                        rtol::Real=NaN, atol::Real=0)
    # computes standard error of estimates from
    #   fit    : a CurveFitResult from a curve_fit()
    covar = estimate_covar(fit)
    # then the standard errors are given by the
    # sqrt of the diagonal
    vars = diag(covar)
    # Take the absolute value to be safe
    sqrt.(abs.(vars*fit.mse))
end
```

It uses the `estimate_covar` routine to estimate the covariance from the Jacobian and then calculates the uncertainties from the diagonals of the covariance matrix.

```
function estimate_covar(fit::CurveFitResult)
    # computes covariance matrix of fit parameters
    J = fit.jacobian
    inv(J'*J)
end
```

A.5.2. Code for Figures

Here is the code I used to generate Fig. 1.

```
xpts = collect(1:0.1:10);
ypts = 2.0 .+ 3.0.*xpts .+ xpts.^2 ./3;
yn = ypts + randn(length(xpts));
import PyPlot
const plt = PyPlot;
plt.plot(xpts,yn,".")
plt.title("Noisy Data with Quadratic Function")
```

The two residual plots in Fig. 2 and Fig. 3 were generated with the following code.

```
cft = poly_fit(1, xpts, yn);
plt.plot(xpts,cft.resid,".");
plt.title("Residual from Linear Fit of Quadratic Function");
cft = poly_fit(2, xpts, yn);
plt.plot(xpts,cft.resid,".");
plt.title("Residual from Quadratic Fit")
```

Because I haven't implemented a weighted linear least squares routine yet, I used the `curve_fit` function for Fig.4, Fig. 5, and Fig. 6.

```
f(x,p)=exp.(-p[1].*(x.-1)) .*
        (0.8.*(cos.(p[2].*(x.-1))).^2 .+ 0.2)
ypts=f(xpts,p0);
yn = ypts.*(1 .+ randn(length(xpts))./10);
wt = 0.1.*ones(length(xpts))
cft = curve_fit(f, xpts, yn, wt, p0);
yft = f(xpts, cft.param);
plt.semilogy(xpts,yn,".",xpts,yft,"-")
plt.title("Data and Fit with Equal Weights")
wt = 1 ./(yn./10);
cft = curve_fit(f, xpts, yn, wt, p0);
yft = f(xpts, cft.param);
plt.semilogy(xpts,yn,".",xpts,yft,"-")
plt.title("Data and Fit with Proportional Weights")
```

A.5.3. Code for Testing

I added unit testing code to LsqFit to make sure my routines were doing what I expected. This is the code to test the nonlinear fitting routine

```
@testset "Single Fit" begin
model(x,p) = exp.(-p[1].*x).*(1.1.+0.8.*cos.(p[2].*x))/2
p=[1,pi];
x=range(0,stop=10,length=200);
y=model(x,p)
fval = 0.2
ypts=model(x,p).*(1.0.+fval.*randn(length(x)));

w=1.0./(fval.*y);
mfit = curve_fit(model, x, ypts, w, p);

@test mfit.dof == 198
@test isapprox(mfit.param[1], 1.0, atol=1e-2)
@test isapprox(mfit.param[2], pi, atol=1e-2)
sigma = standard_error(mfit)
@test isapprox(sigma[1], 2.44e-3, atol = 2e-4)
@test isapprox(sigma[2], 3.64e-3, atol = 2e-4)
@test mfit.converged # doesn't always pass
@test isapprox(mfit.mse, 1.0, atol = 0.25)
end;
```

This function computes a nonlinear fit multiple times to see if the distribution of the fit parameters matches the uncertainty estimates.

```
function afit(trials::Int)
    model(x,p) = exp.(-p[1].*x).*(1.1.+0.8.*cos.(p[2].*x))/2
    p=[1,pi];
    x=range(0,stop=10,length=200);
    y=model(x,p)
    fval = 0.2
    psum=[0.0,0.0]
    p2sum=[0.0,0.0]
    vsum=[0.0,0.0]
    fvar = 0.2;
    for i = 1:trials
        ypts=model(x,p).*(1.0.+fvar.*randn(length(x)));
        w=1.0./(fvar.*y);
        fit = curve_fit(model, x, ypts, w, p);
        sigma = standard_error(fit)
        psum = psum + fit.param
        p2sum = p2sum + fit.param.^2
        vsum = vsum + sigma
    end
    pbar = psum/trials
    p2bar = p2sum/trials
    vbar = vsum/trials
    sigmap = sqrt.(p2bar-pbar.^2)
    (vbar, sigmap)
end
@testset "Covariance Check" begin
    vb, sb = afit(1000)
    @test isapprox(vb[1], sb[1], rtol = 8e-2)
    @test isapprox(vb[2], sb[2], rtol = 0.05)
end;
```

Here is a similar function for testing the error estimates on linear fitting.

```
# actual variations in the data.
function lfit(trials::Int)
    fl = [x->ones(length(x)), x->x, x->x.^2]
    Random.seed!(199382721)
    xpts = 1:10;
    lf = linear_fit(fl, xpts, ypts)
    b = lf.param
    std = standard_error(lf)
    psum=zeros(3)
    p2sum=zeros(3)
    vsum=zeros(3)
    fvar = 10.0;
```

```
    for i = 1:trials
        xnoise = randn(10)./fvar;
        ypts = xnoise .+ 5.0 .+ 3.0.*xpts .+ 0.1.*xpts.^2
        fit = linear_fit(fl, xpts, ypts);
        sigma = standard_error(fit)
        psum = psum + fit.param
        p2sum = p2sum + fit.param.^2
        vsum = vsum + sigma
    end
    pbar = psum/trials
    p2bar = p2sum/trials
    vbar = vsum/trials
    sigmap = sqrt.(p2bar-pbar.^2)
    (vbar, sigmap)
end
```

Here is the code that calls this function and does other tests for the linear fit unit testing.

```
function test_linear()
    @testset "linear_fit check" begin
        fl = [x->ones(length(x)), x->x, x->x.^2]
        Random.seed!(199382721)
        xnoise = randn(10)./10.0;
        xpts = 1:10;
        ypts = xnoise .+ 5.0 .+ 3.0.*xpts .+ 0.1.*xpts.^2
        lf = linear_fit(fl, xpts, ypts)
        b = lf.param
        @test length(b) == 3
        @test abs(b[1]-5.0) < 0.03
        @test abs(b[2]-3.0) < 0.02
        @test abs(b[3]-0.1) < 0.002
        @test lf.mse < 0.01
        @test lf.mse > 0.009
        vb, sb = afit(1000)
        @test isapprox(vb[1], sb[1], rtol = 0.04)
        @test isapprox(vb[2], sb[2], rtol = 0.02)
    end;
end
```

Finally, here is the code for testing the `poly_fit` routine.

```
function test_poly()
    @testset "poly_fit check" begin
        Random.seed!(199382721)
        xnoise = randn(10)./10.0;
        xpts = 1:10
```

```
        ypts = xnoise .+ 5.0 .+ 3.0.*xpts .+ 0.1.*xpts.^2
        lf = poly_fit(2, xpts, ypts)
        b = lf.param
        @test length(b) == 3
        @test abs(b[1]-5.0) < 0.03
        @test abs(b[2]-3.0) < 0.02
        @test abs(b[3]-0.1) < 0.002
        @test lf.mse < 0.01
        @test lf.mse > 0.009
        vb, sb = afit(1000)
        @test isapprox(vb[1], sb[1], rtol = 0.04)
        @test isapprox(vb[2], sb[2], rtol = 0.02)
    end;
end
```

A.5.4. T Tests and P Values

Here is the code I used to test the Student t Tests and P values reported in Excel using Julia. Compare these with Sec. A.1.

```
# Recreate excel fits
noise=[0.712705059,
    1.037487891,
    0.292159257,
    -0.71952627,
    -1.187286216]
xpts = 1:5
m = 15
b = 3
ypts = noise .+ m.*xpts .+ b
using LsqFit
pf = poly_fit(1,xpts,ypts)
sigma = standard_error(pf)
@printf("fit intercept = %.5f +/- %.5f\n",
    pf.param[1], sigma[1])
@printf("fit slope = %.5f +/- %.5f\n",
        pf.param[2], sigma[2])
```

This returned the following which exactly agreed with Excel.

```
fit intercept = 4.69421 +/- 0.43046
fit slope = 14.44430 +/- 0.12979
```

The following code checked the t statistics and p-values in Excel.

```
tstat = pf.param./sigma
@printf("fit tStats = %.2f, %.1f\n", tstat...)
```

```
df = 3
td = TDist(df)
tcdf = [cdf(td, tstat[i]) for i=1:2]
pval = 2 .* (1 .-tcdf)
@printf("P-value = %.2e %.2e\n", pval...)
```

These produced output which also agreed with Excel.

```
fit tStats = 10.91, 111.3
P-value = 1.65e-03 1.60e-06
```

I also checked the 95% confidence intervals in Excel using `tcdf` from above. I determined that a t statistic of 3.182 produced a p value of 0.05. Multiplying the above standard deviations by 3.182 reproduced the confidence intervals.

References

[1] Philip R. Bevington, D. Keith Robinson, "Data Reduction and Error Analysis for the Physical Sciences," Third Edition, McGraw Hill, 2003.

[2] Wikipedia, "Monotonic least squares," `https://en.wikipedia.org/wiki/Linear_least_squares` (accessed 31 Aug 2018).

[3] Eric W. Weisstein, "Least Squares Fitting," From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/LeastSquaresFitting.html` (accessed 31 Aug 2018).

[4] R. Steven Turley, "Polynomial Fitting," BYU, 2018.

[5] R. Steven Turley, "Fitting Parameter Uncertainties," BYU, 2018.