XGFIT's Curve Fitting Algorithm with GSL

Overview

XGFIT is a GTK+ application that takes as input a set of (x, y) points, produces a line graph, and performs a gaussian fit of the data. XGFIT uses the GNU Scientific Library (GSL) to perform the curve fit. GSL may be downloaded freely via RedHat at http://sources.redhat.com/gsl.

This document describes the gaussian equation, the use of GSL to perform the curve fit, and documents the mathematical work done to develop the code.

Curve Fitting Implementation

As mentioned above, the curve fitting is now implemented using GSL. The function we are fitting is

$$f(x) = b + p \cdot e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2},$$

where b, p, c and w represent the base, peak, center and width of f, respectively. The actual task performed by the curve fitting algorithm is to find values for the 4 parameters. The technique used to implement the fit is the *Levenberg-Marquardt Method*.

In order to perform the fit, GSL requires that the programmer supply three procedures:

1. A function that will calculate and store $f(x)$ for each given value of x.
2. A function that will calculate and store $f'(x)$ for each given value of x. Note that because we are dealing with a function of 4 parameters, f is a vector function and $f'(x)$ is the vector derivative of f.
3. A function that will invoke $f(x)$ and $f'(x)$ with the appropriate parameters.

Procedures 1 and 3 are relatively easy to understand. However, procedure 2 involves advanced calculus and is explained in detail presently.

Finding the derivative of *f*

The derivative of a vector v is calculated by taking partial derivatives with respect to each component in v. In our case, this involves 4 partial derivatives, one each for b, p, c, and w. If we have n data points, we construct an n x 4 matrix and fill each row with the corresponding partial derivatives of f_i. This n x 4 matrix is the Jacobian matrix of f.

The partial derivatives are represented as

$$\frac{\partial}{\partial b} f, \frac{\partial}{\partial p} f, \frac{\partial}{\partial c} f, \text{ and } \frac{\partial}{\partial w} f.$$

The first partial derivative above is read "the partial derivative of *f* with respect to *b*". The remaining three partial derivatives are read in a similar fashion.

Calculating a partial derivative is not much different from calculating a normal derivative. Each parameter, except the one being differentiated against, is treated like a constant. Normal differentiation rules then apply.

The partial derivative of *f* with respect to *b* is shown below. Note that since we treat *b* as the only variable parameter, the complicated second term falls off completely.

$$\frac{\partial}{\partial b} f = \frac{\partial}{\partial b}\left[b + p \cdot e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2} \right] = 1$$

Next, we see the partial derivative of *f* with respect to *p*. This calculation is only slightly more complicated than the previous one. Because it is not related to *p*, the *b* term disappears completely. Also, if we let

$$r = e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2},$$

we are left with $r \cdot p$, which differentiates to *r*.

$$\frac{\partial}{\partial p} f = \frac{\partial}{\partial p}\left[b + p \cdot e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2} \right] = e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2}$$

Now things begin to get a bit more complicated. Finding the derivative with respect to *c* involves using the chain rule. Recall that the chain rule gives us a way of calculating the derivative of a composite function, *F*, such that

$$F'(x) = (f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

Here, we let

$$F(c) = e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2},$$

which can be broken into the components

$$f(x) = e^{c} \text{ and } g(c) = -\frac{1}{2}\left(\frac{x-c}{w}\right)^{2}.$$

Notice that

$$F(c) = e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^{2}} = f(g(c)).$$

Now, we calculate *f'(c)* and *g'(c)* in order to apply the chain rule. For *f*, we may begin differentiating immediately.

$$f'(c) = e^{c}dc$$

Before differentiating *g*, however, let's make it look more like a polynomial.

$$g(c) = -\frac{1}{2}\left(\frac{x-c}{w}\right)^{2}$$
$$= -\frac{1}{2w^{2}}\left(x^{2} - 2cx + c^{2}\right)$$

Now, we can see that

$$g'(x) = -\frac{1}{2w^{2}}(-2x + 2c)$$

$$= \frac{x-c}{w^{2}}dc.$$

Now that we have both *f'* and *g'*, we can apply the chain rule.

$$f'(g(c)) \cdot g'(c)$$
$$= f'\left(-\frac{1}{2}\left(\frac{x-c}{2}\right)^{2}\right) \cdot \frac{x-c}{w^{2}}$$
$$= e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^{2}} \cdot \left(\frac{x-c}{w^{2}}\right)$$

At this point, we have the derivative of *F*. Next, we must remember to multiply this derivative by *p* in order to find the partial derivative of our original *f* with respect to *c*. Thus,

$$\frac{\partial}{\partial c} f = p \cdot e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2} \cdot \left(\frac{x-c}{w^2}\right)$$

Calculating the partial derivative of *f* with respect to *w* is very similar, and after a bit of work we arrive at

$$\frac{\partial}{\partial w} f = p \cdot e^{-\frac{1}{2}\left(\frac{x-c}{w}\right)^2} \cdot \frac{(x-c)^2}{w^3}$$

Populating the Jacobian Matrix

Now that we have calculated all four partial derivatives, we will be able to populate our *n* x 4 Jacobian matrix.

$$\begin{bmatrix} \frac{\partial}{\partial b} f(x_1) & \frac{\partial}{\partial p} f(x_1) & \frac{\partial}{\partial c} f(x_1) & \frac{\partial}{\partial w} f(x_1) \\ \frac{\partial}{\partial b} f(x_2) & \frac{\partial}{\partial p} f(x_2) & \frac{\partial}{\partial c} f(x_2) & \frac{\partial}{\partial w} f(x_2) \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial}{\partial b} f(x_n) & \frac{\partial}{\partial p} f(x_n) & \frac{\partial}{\partial c} f(x_n) & \frac{\partial}{\partial w} f(x_n) \end{bmatrix}$$

In the actual implementation, we divide each element of the Jacobian matrix by a $sigma_i$ value. Initially, each value of sigma is set to 0.1.

Using FitGSL

FitGSL is a utility created to remove the complexity of GSL from the programmer. The following example demonstrates the use of FitGSL.

```
#include "fitgsl.h"

#define N_DATA 25

int main(int argc, char *argv[])
{
	fitgsl_data	*d;
	int		i, r;
	float		c[4];

	d	= fitgsl_alloc_data(N_DATA);

	for( i = 0; i < N_DATA / 2; i++ )
	{
		d->pt[i].x = i;
		d->pt[i].y = (float)(i + ((rand() % 5) - 2));
	}

	for( i = N_DATA / 2; i < N_DATA; i++ )
	{
		d->pt[i].x = i;
		d->pt[i].y = (float)((N_DATA - i) + ((rand() % 5) - 2));
	}

	r = fitgsl_lm(d, c, 0);

	fitgsl_free_data(d);

	printf("\n");
	printf(" b = %f, ", c[B_INDEX]);
	printf(" p = %f, ", c[P_INDEX]);
	printf(" c = %f, ", c[C_INDEX]);
	printf(" w = %f\n\n", c[W_INDEX]);

	return EXIT_SUCCESS;
}
```

First, we allocate an instance of a `fitgls_data` structure that will hold `N_DATA` data points. Next, we proceed to populate the structure with (x, y) values. Normally, these values would come from a file or some kind of meaningful calculation. Finally, we invoke the fit using the `fitgsl_lm()` function, and output the calculated values.

Note: For a reference of the functions and types defined by FitGSL, please see Appendix A.

The output from the preceding program should be

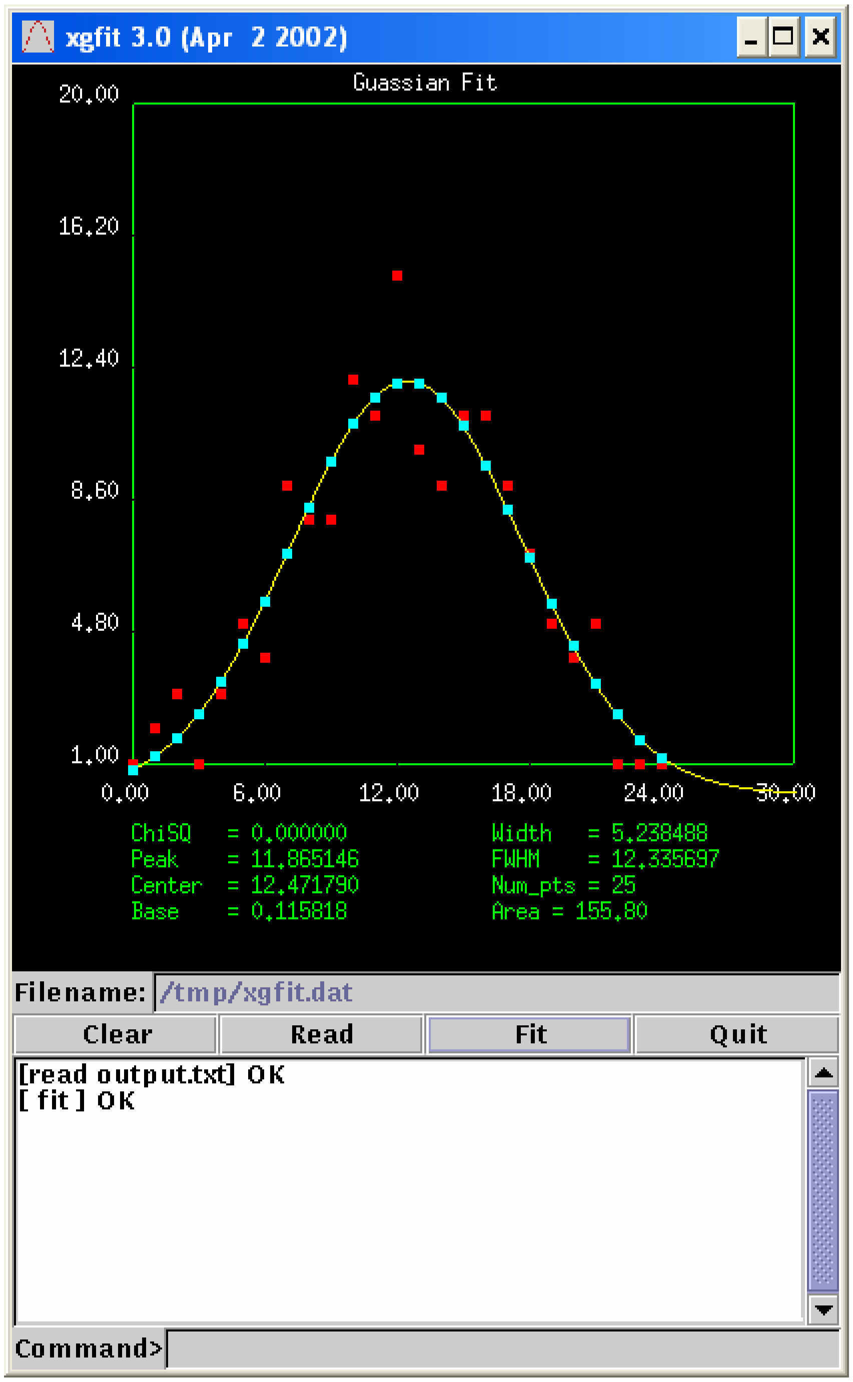
```
 b = 0.115818, p = 11.865146, c = 12.471790, w = 5.238488
```

If you would like to build the sample application, be sure to link with the following flags:

```
-lgsl -lgslcblas -lm
```

Also, be sure that GSL is installed on your system and that you have a copy of the FitGSL module.

An interesting point to note is that the curve fit in this case failed. However, running XGFIT with the same data shows that the calculated values *do* fit the curve relatively well. See *Figure 1* below.



```
ChiSQ   = 0.000000          Width   = 5.238488
Peak    = 11.865146         FWHM    = 12.335697
Center  = 12.471790         Num_pts = 25
Base    = 0.115818          Area = 155.80
```

Filename: /tmp/xgfit.dat

```
[read output.txt] OK
[ fit ] OK
```

Figure 1: A usable fit, even on failure

Of course, it's up to the user to decide whether or not to use the calculated values if the fit fails.

Appendix A: FitGSL Reference

Data Types

- **fitgsl_point2df** - A two-dimensional point on the Cartesian plain with floating-point components.

```
typedef struct _fitgsl_point2df
{
      float x;
      float y;
}fitgsl_point2df;
```

- **fitgsl_data** – Stores the data that will be fit by FitGSL.

```
typedef struct _fitgsl_data
{
      int            n;
      fitgsl_point2df   *pt;
}fitgsl_data;
```

See Also: `fitgsl_alloc_data(), fitgsl_free_data()`

Functions

int fitgsl_lm(const fitgsl_data *dat, float *results, int verbose);
Find *b, p, c* and *w* based on the data supplied in *dat*. The results are stored in *results*, which is a 4-element array of floats. Use the macros B_INDEX, P_INDEX, C_INDEX, and W_INDEX (defined in *fitgsl.h*) to get at the individual components. The *verbose* flag should be set to 0 to suppress verbose status output.

fitgsl_data *fitgsl_alloc_data(int n);
Allocates a fitgsl_data structure with sufficient storage for *n* data points.

void fitgsl_free_data(fitgsl_data *dat);
Frees a fitgsl_data structure.

float fitgsl_fx(float b, float p, float c, float w, float x);
Invoke *f(x)*, using the specified values for *b, p, c, w* and *x*. Useful for graphically comparing the curve fit to the input data points.