Fitting a Gaussian Function to Binned Data

M.Lampton  UCB Space Sciences Lab
May 2002; revised March 2009
Motivation

• Commonly encountered situation in spectroscopy: an emission or absorption line on a continuum
  – Example: estimating redshift errors on faint targets
• Commonly encountered situation in astronomy: a star or galaxy atop a diffuse background
  – Example: estimating star tracker photon jitter
• Often need an estimate of the likely errors in the fitted intensity, position, breadth derived from a real observation or hypothetical observation
• Lots of standard fitting routines are out there!
• Here I apply one approach based on the minimum chisquared maximum likelihood Fisher matrix formalism.
• Although these examples are one dimensional, the methods can be extended to two or more dimensions.
Model the Gaussian + Background

- Four parameters model
  - Level background to be fitted
  - Integral of the Gaussian
  - Centroid of the Gaussian
  - Width of the Gaussian
- Data “d” are binned into NBINS
  - The data have known measurement errors
- Find the best-fit model parameters
  - Requires the data
  - Requires the measurement errors
- Find the uncertainty in the parameters
  - Requires the model (best-fit or hypothetical)
  - Requires the errors (observed or hypothetical)
  - Hypothetical allows prediction of future experiments
  - Therefore becomes powerful tool in planning
General least squares

Model \( f(i, p) \); data bin “i”; parameter vector \( p \); \( \sigma \) = measurement error per bin

\[
\chi^2 = \sum_{i=1}^{NDAT} \frac{[f(i, p) - d_i]^2}{\sigma_i^2}
\]

Jacobian matrix \( J_{i,j} \equiv \frac{\partial f(i, p)}{\partial p_j} \)

Fisher matrix \( F_{j,k} = 1 - \frac{\partial^2 \chi^2}{2 \partial p_j \partial p_k} = \sum_{i=1}^{NDAT} \frac{1}{\sigma_i^2} \cdot \frac{\partial f}{\partial p_j} \cdot \frac{\partial f}{\partial p_k} \)

Covariance matrix \( C = F^{-1} \)

RMS error on parameter \( p_j \) is \( \sqrt{C_{jj}} \)
Make this specific

• Where does the fitting model come in?
  – It is the f(i,p) for each data point “i” and parameter set “p”
  – Create a callable function that models the expected bin content for any reasonable “p” vector

• Where do the data come in?
  – Directly into the sum-of-squares; they are the “d” values.
  – Create a function that loads these data into an array.

• And the measurement errors?
  – Those are the sigma values in the s-o-s.
  – Create a function that loads the error values into an array.

• How to navigate to the best-fit parameters?
  – Use Levenberg-Marquardt.

• Where to start the parameter adjustment sequence?
  – Somewhere close to the right answer!
Example calculation of single fit
filename GausSingle.java

public class GausSingle
{
    final static int NPARMS = 4;
    final static int NPTS = 200;
    static double parms[] = new double[NPARMS];
    static double errors[] = new double[NPTS];
    static double data[] = new double[NPTS];

    public static void main (String args[])
    {
        setInitParms(parms);
        showParms(parms);
        getData(data);
        setErrors(data, errors);
        solve(parms, data, errors);
        showParms(parms);
    }

    ///
    /// further functions follow...
    ///
The method solve() – what is that?

static void solve(double parms[], double data[], double errors[])  
// Starts with given parms, exits with best fit parms.
{
    LM myLM = new LM(NPTS, NPARMS, data, errors, parms);
    myLM.lmfit();
    myLM.getparms(parms);
}

• Class LM is a Levenberg-Marquardt nonlinear least-squares solver
• To begin, solve() calls constructor to create object “myLM” to communicate all the necessary startup information
• Then solve() calls object’s lmfit() method to begin the iterative fitting; solve() waits peacefully while the fitting iterates.
• When lmfit() is done, solve() grabs the iterated parms and returns.
What’s in the LM class?

- Constructor LM()
  - Receive startup data, constructs object with working arrays
- lmfit()
  - Performs the iterative improvement in the fit
  - Uses the s.o.s. gradient vector and curvature matrix
  - Method is combination of steepest descent and full Newton
  - Strategy is how/when to combine these for best progress
- getsos(parms[ ])
  - For any parm vector, returns sum-of-squares
- getparms(p[ ])
  - Allows client to claim iterated parm vector
- dpdf()
  - Two sided derivative of fitting function
- gaussj()
  - A reasonably efficient real matrix inverter.
Multiple fits to rerandomized data
filename GausMulti.java

- Like GausSingle but gathers statistics on successive fits to data that have been rerandomized.
- Can be used to simulate experiments whose data are not yet available.

```java
public class GausMulti {
    final static int NPARMS = 4;
    final static int NPTS = 200;
    final static int NITER = 100;
    static double trueparms[] = new double[NPARMS];
    static double fittedparms[] = new double[NPARMS];
    static double cleandata[] = new double[NPTS];
    static double errors[] = new double[NPTS];
    static double noisidata[] = new double[NPTS];
    static double stats[][] = new double[2][NPARMS];
    static double parmerrors[] = new double[NPARMS];

    public static void main (String args[]) {
        setTrueParms(trueparms);
        setCleanData(trueparms, cleandata);
        setErrors(cleandata, errors);
        zeroStats(stats);
        for (int n=0; n<NITER; n++) {
            setNoisyData(cleandata, errors, noisidata);
            initParms(trueparms, fittedparms);
            solve(fittedparms, noisidata, errors);
            showParms(fittedparms);
            addStats(fittedparms, stats);
        }
        finishStats(stats);
        showParms(stats[0]); // the parameter means.
        showParms(stats[1]); // the rms parm errors.
    }
}
```
A model where the Gaussian function is just one sample at the center of the bin: OK when width >> 1 bin

```java
class F {
    static double func(int i, double p[]) {
        // samples one point at x=i
        // OK if funcx() varies only mildly over the bin.
        double x = (double) i;
        return funcx(x, p);
    }

    static double funcx(double x, double p[]) {
        // p[0] = background level
        // p[1] = integral of the Gaussian
        // p[3] = rms breadth of Gaussian distribution
        double numer = (x-p[2])*(x-p[2]);
        double denom = p[3]*p[3];
        double ratio = numer/denom;
        if (ratio < 30.0)
            return p[0] + (0.39894*p[1]/p[3])*Math.exp(-0.5*ratio);
        return p[0];
    }
}
```
A model where each bin is an integrated Gaussian:
needed when Gaussian width ~ 1 bin

class F
{
  static double func(int i, double p[])
  // samples uniformly within bin "i"
  {
    int NSAMP = 20;
    double dx = 1.0/NSAMP;
    double sum = 0.0;
    for (double x=-0.5+0.5*dx; x<0.5; x+=dx)
      sum += dx*funcx(i+x, p);
    return sum;
  }

  static double funcx(double x, double p[])
  // p[0] = background level
  // p[1] = integral of the Gaussian
  // p[3] = rms breadth of Gaussian distribution
  {
    double numer = (x-p[2])*(x-p[2]);
    double denom = p[3]*p[3];
    double ratio = numer/denom;
    if (ratio < 30.0)
      return p[0] + (0.39894*p[1]/p[3])*Math.exp(-0.5*ratio);
    return p[0];
  }
}
How to evaluate numerical derivatives?

- Double sided finite difference works well
- Step size deserves some care in choosing
- Roughly |paramSize| * cubeRoot(MACHEPS)
- Often 1E-6 works well.

```c
double dfdp(int i, int j, double parms[])
// Two sided derivative of F.func(i,p) w.r.t. any given parm[j]
// Gives one jacobian matrix element dfi/dpj at given parmvector.
{
    double pminus[] = new double[NPARMS];
    double pplus[]  = new double[NPARMS];
    for (int k=0; k<NPARMS; k++)
        pminus[k] = pplus[k] = parms[k];
    pminus[j] -= DELTAP;
    pplus[j] += DELTAP;
    return (F.func(i,pplus) - F.func(i,pminus))/(2.0*DELTAP);
}
```
But I don’t have real data yet! I just want to predict errors for a hypothetical experiment!

- Follow the formalism of chart 4.
  - The parameter errors don’t depend on any of the data!
  - They depend only on the model and the expected measurement errors “sigma”.
- So, do this…
  - Write your model $f(i,p)$ function for some nominal data concept
    - May involve observation time, nominal background, instrument
  - Compute its derivatives w.r.t each parameter at given parm set
    - This is your Jacobian matrix
  - Combine Jacobian with sigma, get Fisher information matrix.
  - Invert that, getting the covariance matrix.
  - Evaluate the square roots of its diagonal elements.
Estimating parm errors, single parm vector
filename GausErr.java

```java
public class GausErr {
    final static int NPARMS = 4;
    final static int NPTS = 200;
    static double data[] = new double[NPTS];
    static double erro[] = new double[NPTS];
    static double jacobian[][] = new double[NPTS][NPARMS];
    static double fisher[][] = new double[NPARMS][NPARMS];
    static double covar[][] = new double[NPARMS][NPARMS];
    static double parms[] = new double[NPARMS];
    static double rmsparms[] = new double[NPARMS];

    public static void main (String args[]) {
        setParms(parms);
        showParms(parms);
        setData(parms, data);
        setErro(data, erro);
        getJacobian(parms, jacobian);
        getFisher(erro, jacobian, fisher);
        getCovar(fisher, covar);
        getRMSparms(covar, rmsparms);
        showRMS(rmsparms);
    }
}
```

To explore a grid of parm values, put this sequence into a loop that steps through the grid wanted.
Implementation details

• Where to find a Java Levenberg-Marquardt nonlinear least squares routine to use in getting best-fit parameter set?
  – “Levenberg-Marquardt Demo” at MikeLampton.com

• Where to find a Java matrix inverter, to convert my Fisher matrix to a covariance matrix?
  – Use gaussj() from “Levenberg-Marquardt Demo” at MikeLampton.com

• How to best illustrate my Gaussian fitting results?
  – It’s all about how errors propagate from the data world into the parameters. For that, a useful approach is to make plots of how the parameter errors depend on the broad model characteristics. Expect that a big Gaussian on top of a feeble background will have photometric error ~ sqrt(Area), so that relative photometric error will vary as 1/sqrt(Area). Expect error in breadth to be proportional to breadth/sqrt(Area). Expect rising background will increase the fitting errors especially with fainter targets.
Other code snippets too trivial to publish

static void setParms(double p[]) {
    p[0] = 1;  // background level
    p[1] = 1E4; // integral of Gaussian
    p[2] = 100.5; // centroid of Gaussian
    p[3] = 1.0;  // RMS width of Gaussian
}

static void setData(double p[], double d[]) {
    for (int i=0; i<NPTS; i++)
        d[i] = F.func(i, p);
}

static void setErro(double d[], double e[]) // Installs Poisson errors based on model data {
    for (int i=0; i<NPTS; i++)
        e[i] = Math.sqrt(d[i]);
}

static void showParms(double p[]) {
    for (int i=0; i<NPARMS; i++)
        System.out.print(U.fwd(p[i], 16,4));
    System.out.println();
}

static void showRMS(double r[]) {
    for (int i=0; i<NPARMS; i++)
        System.out.print(U.fwd(r[i], 16,4));
    System.out.println();
}

static void getJacobian(double p[], double jac[][]) {
    for (int i=0; i<NPTS; i++)
        for (int j=0; j<NPARMS; j++)
            jac[i][j] = dfdp(i, j, p);
}

static void getFish(double e[], double jac[][], double f[][]) {
    for (int j=0; j<NPARMS; j++)
        for (int k=0; k<NPARMS; k++)
            f[j][k] = 0.0;
    for (int i=0; i<NPTS; i++)
        for (int j=0; j<NPARMS; j++)
            for (int k=0; k<NPARMS; k++)
                f[j][k] += jac[i][j]*jac[i][k]/(e[i]*e[i]);
}

static double getCovar(double fish[][], double cov[][]) {
    for (int j=0; j<NPARMS; j++)
        for (int k=0; k<NPARMS; k++)
            cov[j][k] = fish[j][k];
    double det = gaussj(cov, NPARMS);
    return det;
}

static void getRMSparms(double cov[][], double rms[]) {
    for (int j=0; j<NPARMS; j++)
        rms[j] = Math.sqrt(cov[j][j]);
}
Alternative method: Monte Carlo

- Start with a hypothetical “true” model
- Use it to evaluate a model data pattern
- Repeat 1000 times:
  - Add Gaussian random errors to the model data pattern
  - Fit this data vector, getting parameter vector
  - Accumulate mean & variance statistics on each parameter
- Are the means close to the initial true values?
- Are the variances in agreement with the diagonal values of the covariance matrix obtained via Fisher formalism?
Simple estimate of Gaussian moments

- Zero background
- Step through a grid of Totals and Widths
- At each point, run 1000 iterations
  - Produce random data
  - Calculate zeroth moment \( \text{tot} = \text{sum}(\text{di}) \)
    - Not optimum if background is present!
  - Calculate first moment \( \text{xbar} = \text{sum}(x \cdot \text{di}) \)
    - Not optimum especially if background is present!
  - Build statistics on these
- Make a plot of rms errors in \( \text{tot} \) and \( \text{xbar} \)
- Notice that there are two extreme cases for \( \text{xbar} \) estimate
  - True centroid at center of bin
  - True centroid at a bin edge
Simple Check: results
filename Pachinko.java

Note bifurcation into two extreme cases: centroid at mid-bin and centroid on edge.
References: