Computational Physics – Lecture 6

Andrei Alexandru

March 19, 2014

1 Introduction

This week we will discuss χ^2-fitting. Fitting can be used to represent the data more compactly, via a model whose parameters are extracted from a fit, or to extract physical parameters from empirical data when we have an accurate model for the process generating the data. We will focus on the second role of fitting in these lectures.

To be more precise we will take the empirical data to be a list of pairs (x_i, y_i), with $i = 1, \ldots, N$ being the sample index. Our model is represented by a function $f_\alpha(x)$, where α constitutes a collection of parameters relevant for our model. In general the function can have arbitrary form, but we will focus here on polynomial functions. For a polynomial of order M,

$$P(\alpha; x) = \sum_{k=0}^{M} \alpha_k x^k \,, \tag{1}$$

the parameters to be determined are the coefficients α. A fitting procedure varies the parameters α until the function represents a good match for the data. To make this process automatic, we need to define a function that measures the goodness of the fit. This is done using χ^2-function:

$$\chi^2(\alpha) \equiv \sum_{i=1}^{N} (f_\alpha(x_i) - y_i)^2 \,. \tag{2}$$

A graphical representation of this distance is given in Fig 1. Note that each point that the curve is missing is contributing to χ^2 function an amount proportional to the square of its distance to the curve. This will lead to incorrect results when some data points are more reliable than others. In Fig. 1 we used error bars to indicate the reliability of each point. For example the points at $x = 2$ and $x = 4$ miss the curve by roughly the same amount. However, the point at $x = 4$ has smaller error bars and the penalty associated with missing it should be greater. A better distance is then one that takes into account the "error" or uncertainty associated with each point, which we will call σ_i. We define

$$\chi^2(\alpha) \equiv \sum_{i=1}^{N} \frac{(f_\alpha(x_i) - y_i)^2}{\sigma_i^2} \,. \tag{3}$$

Using this goodness of fit function, the *residual* at every point, $f(x_i) - y_i$, is compared to the uncertainty σ_i. The points that have a residual much larger than the uncertainty contribute significantly to the χ^2 function and the fitting procedure will try to reduce the residual to values comparable to the uncertainty. This should always be possible when we have reliable estimates of the uncertainty and the model function describes the data accurately. A reasonable χ^2 in this situation is about 1 per degree of freedom. The number of degrees of freedom in a fit is the number of points minus the number of parameters in the fitting function. A more clear rationale for this rule will be provided in a later section. If your fit produces a χ^2 per degree of freedom much larges than one, this is a signal that you underestimated your uncertainty or the fitting model is not describing the data correctly. A too small χ^2 comes from overestimating your error bars.

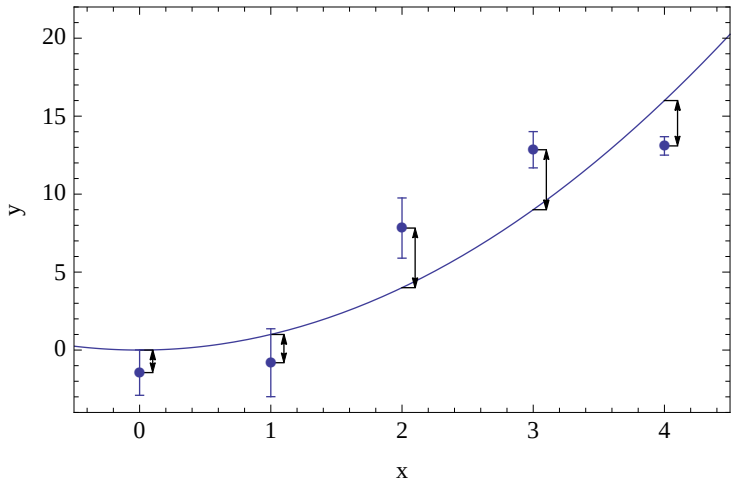

Figure 1: Fit functions try to minimize the distance between the data points and the curve.

2 Fitting a polynomial function

Fitting is then reduced to minimizing χ^2 as a function of the parameters of the model function. In general this is a numerically difficult problem, especially in the case of many parameter functions. For the polynomial case the problem can be reduced to a linear problem with a unique solution. In this section we will show how to reduce this problem to system of linear equations and how to implement such a fitter.

For a polynomial function the χ^2-minimization can be performed using linear algebra. This can be derived by demanding that derivatives of the χ^2-function with respect to the fit parameters α vanish:

$$\frac{\partial \chi^2}{\partial \alpha_k} = \sum_{i=1}^{N} \frac{2\left(P(\alpha; x_i) - y_i\right) x_i^k}{\sigma_i^2} = 0\,. \tag{4}$$

Above, we used

$$\frac{\partial P(\alpha; x)}{\partial \alpha_j} = \frac{\partial}{\partial \alpha_j} \sum_{k=0}^{M} \alpha_k x^k = x^j\,. \tag{5}$$

There are $M+1$ unknowns $\alpha_0, \ldots, \alpha_M$ and $M+1$ equations, linear in α. Thus this problem reduces to the matrix equation $A\alpha = b$ where

$$A_{kj} = \sum_{i=1}^{N} \frac{x_i^{j+k}}{\sigma_i^2} \qquad \text{and} \qquad b_k = \sum_{i=1}^{N} \frac{y_i x_i^k}{\sigma_i^2}\,, \tag{6}$$

where k and j indices run from 0 to M, the order of the polynomial P.

To solve this problem numerically we will use the `solve_system` routine we implement last week. All we need to do is to use the data (x_i, y_i, σ_i) to compute the matrix A and vector b. The first task is to define a structure to conveniently group each data point:

```
1 struct dpoint { double x, y, sigma; } ;
```

This allows us to pass around the list of data points to the relevant routine as a C array of `struct point`. Recall that the way we pass arrays in C is to pass a pointer to the beginning of the array and an integer to indicate its size.

We are now ready to implement the routines that compute A_{kj} and b_k. Here is the implementation

```
double computeA(int k, int j, struct dpoint* data, int N)
{
  double res = 0;
  for(int i=0; i<N; ++i) res += pow(data[i].x, j+k)/pow(data[i].sigma
      ,2);

  return res;
}

double computeb(int k, struct dpoint* data, int N)
{
  double res = 0;
  for(int i=0; i<N; ++i) res += data[i].y*pow(data[i].x, k)/pow(data[i
     ].sigma,2);

  return res;
}
```

Notice that we here follow the usual pattern when computing sums: define an accumulator and set it to zero and then loop over the elements of the sum and add each one to the accumulator.

We can now implement a routine that determines the coefficients of the polynomial that minimize the χ^2 function:

```
void fit_poly(double* coef, int M, struct dpoint* data, int N)
{
  double* A = malloc((M+1)*(M+1)*sizeof(double));
  double* b = malloc((M+1)*sizeof(double));

  for(int k=0; k<M+1; ++k)
  for(int j=0; j<M+1; ++j) A[k*(M+1)+j] = computeA(k, j, data, N);

  for(int k=0; k<M+1; ++k) b[k] = computeb(k, data, N);

  solve_system(A, M+1, b, coef);

  free(A);
  free(b);
}
```

Note that we first allocate space to hold the matrix A and vector b, then use the routines described above to compute their entries and finally call the `solve_system` routine to determine the coefficients. Before we exit the routine we have to remember to free the memory we allocated. Note that the storage space for the polynomial coefficient array `coeff` is assumed to be allocated before the function is called and we should not free it in this function. This is only logical since this array also holds the result of our calculation and presumably the caller expects to have access to it.

To test this routine, we can generate data with some noise and try to fit it using our function. We implement the following `main` routine:

```
int main()
{
  int N = 100;
  struct dpoint * data = malloc(N*sizeof(struct dpoint));

  generate_data(data, N);
  for(int i=0; i<N; ++i) printf("%e %e %e\n", data[i].x, data[i].y,
      data[i].sigma);
```

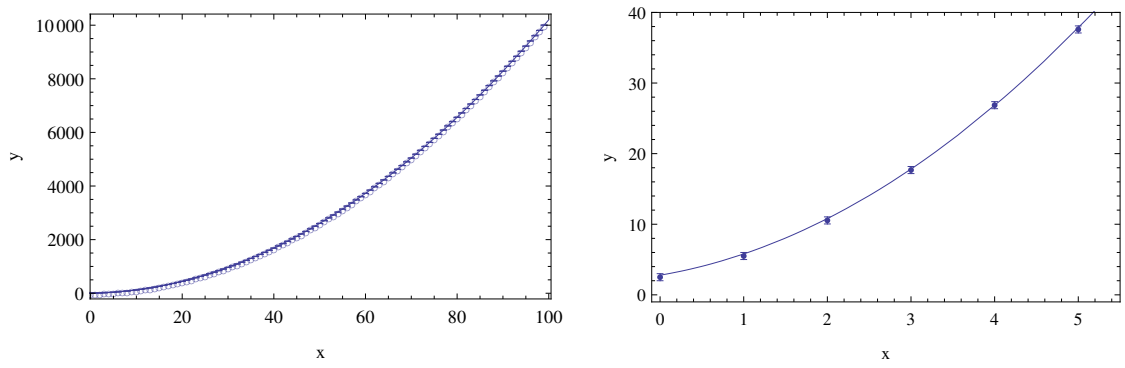

Figure 2: Fit functions vs data: full data set (left) and a detailed view of the first few points (right).

```
8
9     double coef[3];
10    fit_poly(coef, 2, data, N);
11
12    printf("%e %e %e\n", coef[0], coef[1], coef[2]);
13
14
15    free(data);
16    return 0;
17  }
```

Here we generate 100 data points and use our fitter to fit a second order polynomial. We print to the console the generated list of data points and the results of the fit. The data is generated using the following routine:

```
1  void generate_data(struct dpoint* data, int N)
2  {
3    for(int i=0; i<N; ++i)
4    {
5      data[i].x = i;
6      data[i].y = 2*i + i*i + 3 + (drand48()-0.5);
7      data[i].sigma = 0.5;
8    }
9  }
```

This basically generates a set of (x, y) pairs, with $x = 0, \ldots, N-1$ and $y(x) = x^2 + 2x + 3$. We use `drand48` function that generates a random number uniformly distributed between 0 and 1 to add some noise to our data. Note that we offset the number generated by `drand48` by 0.5 to make sure that our noise averages to 0. This way the noise does not shift the curve up or down. Finally, we associate with each point an uncertainty of 0.5 since every point can shift up or down by that amount, because of our noise.

Without the noise we would expect the fitter to return exactly the polynomial we used in generating the data, that is $\alpha_0 = 3$, $\alpha_1 = 2$, and $\alpha_2 = 1$. In the presence of noise, the results will deviate from these exact values. In my testing I got the following values, $\alpha_0 = 2.78$, $\alpha_1 = 2.01$, and $\alpha_3 = 1.00$, which seem to agree with our expectations. To confirm that the fit is good, we can use `ErrorListPlot` in Mathematica to show the data points with their error bars and compare it with the fit. The plot is presented in Fig. 2.

3 χ^2-distribution

The quality of the fit can be gauged using the value of χ^2. This value fluctuates depending on the fit function and the data points. Earlier we said that a successful fit should have a value of χ^2 per degree of freedom of

around 1. However, this value will sometime be larger or smaller, even for good fits. In order to understand whether our value indicates that we have a good fit, we need to understand the distribution of the χ^2 values as the data points fluctuate around their mean values. We will generate this distribution empirically and then compare it with a model that is quite successful. We will then use this model to define the *confidence level* for our fit.

We will then generate data points as before with a mean given by a model, for example $y(x) = x^2+2x+3$. We will add noise to these points by generating random numbers. The random numbers we will generate will average to zero, so that the model is not systematically shifted. The standard deviation at each data point will be σ_i. We can use any distribution for this noise, but the agreement with the model we will discuss is exact only for *normally distributed* fluctuations, that is, noise drawn from a gaussian distribution. Note that in C code we can easily generate random numbers uniformly distributed in the interval $[0, 1]$ using `drand48`. To generate gaussian distributed noise with standard deviation of 1 we use Box-Muller transformation [1]: if we have two independent random numbers $u_{1,2}$ drawn from an uniform distribution in the interval $(0, 1]$ then the numbers

$$\begin{aligned} z_1 &= -\sqrt{-2\log u_1}\cos 2\pi u_2\,, \\ z_2 &= -\sqrt{-2\log u_1}\sin 2\pi u_2\,, \end{aligned} \tag{7}$$

are normally distributed. A simple, and not very efficient, implementations is given below

```
1  double gausrnd()
2  {
3    double u1 = drand48();
4    double u2 = drand48();
5    return sqrt(-2*log(u1))*cos(2*M_PI*u2);
6  }
```

Using this routine we can now modify our `generate_data` routine to produce data using gaussian noise. Here is the modified implementation

```
1  void generate_data(struct dpoint* list, int N)
2  {
3    for(int i=0; i<N; ++i)
4    {
5      list[i].sigma = 1;
6      list[i].x = i;
7      list[i].y = i*i + 2*i + 3 + list[i].sigma*gausrnd();
8    }
9  }
```

This routine generates synthetic data with a gaussian noise of width of 1 for each point. We can easily modify the width by changing line 5 in the listing above. We will now create a `main` routine that generates N_s sets of data of length N and fits a second order polynomial. For each fit we print the value of the fitted coefficients and the value of χ^2, which is important in order to determine its distribution. The value of N and N_s are read from the command line.

```
1  int main(int argc, char** argv)
2  {
3    int N = atoi(argv[1]);
4    int Ns = atoi(argv[2]);
5
6    struct dpoint* list = malloc(N*sizeof(struct dpoint));
7
8    for(int i=0; i<Ns; ++i)
9    {
10     generate_data(list, N);
11     double coef[3];
```

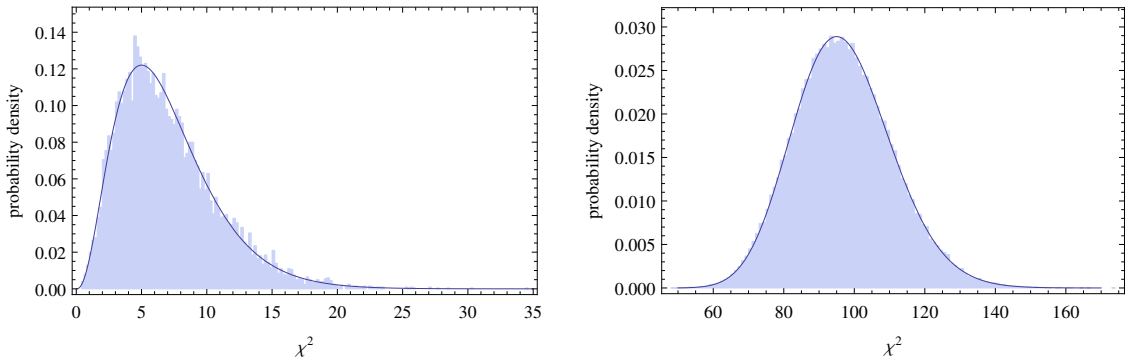

Figure 3: χ^2-distribution for two cases: $N = 10$ and $N_s = 10,000$ compared with the theoretical expectation $P_7(\chi^2)$ (left) and $N = 100$ and $N_s = 100,000$ compared with the theoretical expectation $P_{97}(\chi^2)$ (right).

```
12      fit_poly(coef, 2, list, N);
13      double chi = chisq(coef, 2, list, N);
14
15      printf("%e %e %e %e\n", coef[0], coef[1], coef[2], chi);
16    }
17
18    free(list);
19
20    return 0;
21  }
```

Note that to compute the value of χ^2 we use the following routines

```
1   double poly(double* coef, int M, double x)
2   {
3     double res = 0;
4     for(int i=0; i<M+1; ++i) res += coef[i]*pow(x,i);
5
6     return res;
7   }
8
9   double chisq(double* coef, int M, struct dpoint* data, int N)
10  {
11    double res = 0;
12    for(int i=0; i<N; ++i)
13    {
14      double tmp = poly(coef, M, data[i].x) - data[i].y;
15      res += pow(tmp/data[i].sigma, 2);
16    }
17
18    return res;
19  }
```

We implement the routine `poly` separately to compute the value $P(\alpha; x)$ where the coefficients α are stored in the array `double* coef` of length $M + 1$. The value of χ^2 is computing using the formula in Eq. 3.

We can then use this code to generate a large number of fits, for similar data that differs only in the random noise. We generate a large table in Mathematica and used `Histogram` function to generate the empirical probability distribution function. In Fig. 3 we show the histogram for two cases, $N = 10$ and

$N = 100$. The number of degrees of freedom in the two cases is 7 and 97 since our model has three parameters. The rule of thumb is indeed observed in that the distributions are indeed peaked around the values of 7 and 97. In fact we can be more precise: the distribution follows the so-called χ^2-distribution

$$P_n(\chi^2) = \frac{1}{2^{n/2}\Gamma\left(\frac{n}{2}\right)} e^{-\chi^2/2} \chi^{n-2} \,. \tag{8}$$

This is the distribution followed by the sum of squares of n gaussian random vectors drawn from distribution with unit variance. The number n here corresponds to the number of degrees of freedom in the fit. This means that every degree of freedom generates a gaussian contribution to χ^2 of unit variance. In Fig. 3 we indicate P_7 and P_{97} and show that the agreement is excellent. Thus the χ^2-distribution is a very good model for the distribution of χ^2 as it results from a fit.

Armed with a good model for the distribution of the fit, we can now define a *confidence level* associated with a value of χ^2 for a fit with n degrees of freedom. The confidence level measures the probability that the χ^2 as drawn from $P_n(\chi^2)$ is greater than the one determined in our fit, that is

$$\text{confidence level}\,(n;\chi^2) \equiv \int_{\chi^2}^{\infty} P_n(x)dx \,. \tag{9}$$

In Mathematica this function can be determined using `1-CDF[ChiSquareDistribution[n],`χ^2`]`. Here we compute some values and present them in Table 1. Note that for χ^2 per degree of freedom equal to one, the confidence level approaches 0.5 as the number of degrees of freedom increase. For values slightly smaller than one the limit goes to 1 indicating a very good fit, and for values slightly larger than one the confidence level goes to zero. This indicates that the fitting function is not appropriate or that the estimates for the error bars are wrong.

χ^2/dof	degrees of freedom			
	5	10	100	1000
0.9	0.47988	0.53210	0.75320	0.98928
1.0	0.41588	0.44049	0.48119	0.49405
1.1	0.35795	0.35752	0.23221	0.01461

Table 1: Confidence level for different values of χ^2 per degree of freedom and number of degrees of freedom.

References

[1] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 06 1958.