PEAKDECO. Программа для автоматического выделения пиков из γ-спектрограммы.

19 ноября 2009 г.

1. Заметки

С одной стороны эта программа нужна для проведения анализа экспериментов по определению сечений фотоядерных реакций большой множественности. Выходы каждой реакции определяются с помощью метода наведенной активности. Та часть процедуры анализа, в которой производится обработка спектров, сводится к нахождению пиков в последовательно измеренных спектрах, идентификации источника каждого пика, сопоставление наборам пиков конкретных ядер, связанное с вычислением периодов полураспада, и расчет полного числа ядер каждого типа в образце.

С другой стороны обработка серий спектров и определение ядерных параметров является специфическим требованием только этих экспериментов по гамма-активации в то время, как задача обработки отдельных спектров то и дело возникает буквально на каждом шагу. Поэтому обработка спектров должна проводиться в отдельной программе.

Основная и единственная задача этой программы состоит в нахождении пиков во входном файле и определении их параметров.

Входной спектр представляет собой текстовый файл, полученный с блока анализатора германиевого детектора (МКА). Кроме спектра в нем содержится информация о длительности и времени начала набора данных.

Введем обозначения. Входные данные можно описать суммой

$$Y_i = Y_i^{\text{пик}} + Y_i^{\text{фон}} + Y_i^{\text{шум}} = R_i + B_i + \delta_i,$$

где компонента B_i сама задается суммой соответствующих аппаратных функций.

Фоновая составляющая имеет следующие свойства.

$$\left|\frac{\partial B_i}{\partial i}\right| \ll \left|\frac{\partial R_i}{\partial i}\right|, \quad \left|\frac{\partial B_i}{\partial t}\right| \ll \left|\frac{\partial R_i}{\partial t}\right|$$

То есть фон — медленно изменяющаяся относительно сигнала компонента (второе соотношение может оказаться неверным в буквальном смысле, но в общем это свойство также можно использовать).

Как R_i, так и B_i являются суммой большого количества случайных величин, распределенных по закону Пуассона. Сумма случайных величин, подчиняющихся распределению Пуассона, также распределена по закону Пуассона с $\lambda = \lambda_1 + \lambda_2$.

Входной спектр в каждом канале распределен в соответствии с законом Пуассона, то есть вероятность того, что $y_i = k$ равна

$$p(k) = \frac{\lambda^k e^{-\lambda}}{k!}.$$

Фон находится как минимальное значение входного спектра в диапазоне n.

$$b_i = \inf_{j \in [i-n/2; i+n/2]} y_j$$

Вероятность, что в таком диапазоне попадется величина k задается следующей формулой.

$$p_n(k) = 1 - (1 - p(k))^n$$

Для того, чтобы получить вероятность того, что величина b_i будет иметь значение k, надо это выражение домножить на коэффициент, учитывающий, что во всех остальных $n - 1$ точках $y_i \geq k$.

$$p_{\geq}(k) = 1 - \sum_{i=0}^{k-1} p(i)$$

После чего получается интересующее нас выражение.

$$p_n^{\min}(k) = p_n(k)(p_{\geq}(k))^{n-1}$$

$$p_n^{\min}(k) = np(k)(p_{\geq}(k))^{n-1}$$

$$p_n^{\min}(x) = \frac{n}{\sqrt{2\pi\lambda}} e^{-\frac{(x-\lambda)^2}{2\lambda}} \left(1 - \frac{1}{\sqrt{2\pi\lambda}} \int_{-\infty}^{x-1} e^{-\frac{(x-\lambda)^2}{2\lambda}} dx\right)^{n-1}$$

Далее, можно будет найти среднее значение $b_i^{\text{теор}}$.

$$b_i^{\text{теор}} = \sum_{k=0}^{+\infty} k p_n^{\min}(k) \approx \int_0^{+\infty} x p_n^{\min}(x) dx$$

Величина λ в каждой точке даст оценку фона.

Форма пика определяется так называемой аппаратной функцией детектора. Если во время измерения не происходило перегрузки АЦП, то она описывается функцией Гаусса. Ширина пика зависит от энергии нелинейно. Считается, что эту зависимость можно представить в виде параболы.

$$h = a\sqrt{E} + b,$$

где h — FWHM, или ширина пика на полувысоте.

$$\sigma = \frac{h}{2\sqrt{2\ln 2}},$$

$$a = 0.0369074,$$

$$b = 0.553757.$$

Выбор предельного значения χ^2 при фитировании можно сделать из таких соображений:

$$\chi^2 = \sum_i (\bar{x}_i - x_i)^2 = \sum_i \sigma_i^2 = \sum_i \Delta_i = \sum_i Y_i,$$

где Δ_i — дисперсия в точке i.

Если вдуматься, то из этих же самых соображений можно и искать истинное число пиков на данном отрезке. Если фитирование fit_multi_peak() не сошлось, то рассмотреть наименьший возможный χ^2.

Оказывается, что фитирование экспоненциальными функциями неустойчиво. Это мой личный опыт. Опыт мирового сообщества говорит, что лучше фитировать логарифм.

Рассмотрим дискретное преобразование Фурье сигнала, состоящего из распределенных по закону Пуассона точек:

$$\mathrm{P}(y_i = k) = p_i = \frac{\lambda^k e^{-\lambda}}{k!}.$$

Будем считать, что истинная амплитуда сигнала λ везде одинакова. Дискретное преобразование Фурье этого сигнала, состоящего из N точек, запишется следующим образом:

$$\hat{y_j} = \sum_{i=1}^{N} y_i e^{-2\pi/N\sqrt{-1}ij} = \sum_{i=1}^{N} y_i \cos\left(-\frac{2\pi}{N}ij\right) + i\sum_{i=1}^{N} y_i \sin\left(-\frac{2\pi}{N}ij\right),$$

то есть линейной комбинацией N случайных переменных. Вопрос: каково распределение этой величины? Ответ: практически равномерное распределение.

Тот же самый подход, который я придумал для поиска фона, можно применить и для поиска пиков. Необходимо оценить не минимальное значение на данном интервале, а максимальное.

В качестве одного из критериев отбора максимумов можно использовать производную. Если рассчитать производную от спектра численно ($y'_i = y_i - y_{i-1}$), то распределение полученной величины будет по определению распределением Скеллама. В фоновой области оно хорошо аппроксимируется нормальным распределением с параметрами $M = 0$ и $\sigma = \sqrt{2\lambda}$, где λ — величина фона. Это дает возможность сделать ограничение сверху на значение модуля производной (с заданной вероятностью).

С другой стороны, в области пика производная функции Гаусса имеет максимум и минимум соответственно при $x = M \pm \sigma$. Абсолютная величина максимума и минимума равна $\frac{0}{\sqrt{2\pi}\sqrt{e}\sigma}$.

По видимому, это обстоятельство можно использовать так: если максимальное значение модуля производной в данном интервале больше, чем $3 \cdot \sqrt{2\lambda}$, то критерий дает положительный результат.

2. История изменений

Первая версия программы была написана в июне 2007 года. Она была скопирована на сервер DEPNI, и работает на нем до сих пор почти без изменений. Недостатки — жесткая и неудобная организация программы, ограничивающая внесение изменений.

Вторая версия программы, которая разрабатывается с 2008 года, должна подвести более серьезное научное обоснование к используемым методам. Кроме того программа должна позволять обрабатывать целую серию спектров для того, чтобы улучшить качество распознавания.

29 июля 2009 г. Принимаюсь за дело сразу после отпуска. Переделана обработка командной строки на getopt. Теперь опции передаются в аргументах типа –maxiter=500, все остальное трактуется как имена файлов. Функция process_options(). Загрузка файлов вынесена в load_file().

4 августа 2009 г. Работает загрузка и одновременная обработка нескольких файлов. Насколько можно судить, не имея возможности сравнить графики, результаты на выходе совпадают со старой версией. Вывод данных переписан с использованием сериализации, поэтому потребуется написать вспомогательную программу для форматирования выходных данных.

γ-спектры загружаются из файлов ASCII. Формат файлов: 1 колонка — номер канала, 2 колонка — энергия в кэВ'ах, 3 колонка — число отсчетов. В начале файла идет несколько строчек с комментариями, в которых записано время начала и конца измерения, мертвое время детектора и условный номер спектра. Имена файлов передаются в командной строке.

48 ⟨* 48⟩≡ ?? ▷

```
#include<iostream>
#include<cmath>
#include<fstream>
#include<string>
#include<sstream>
```

```
#include<cstdio>
#include<list>
#include<valarray>
#include<iomanip>
#include<set>
#include<algorithm>
#include<vector>
#include<cassert>
#include<map>
#include<cstring>
#include<complex>
#include<fenv.h>

#include<gsl/gsl_vector.h>
#include<gsl/gsl_blas.h>
#include<gsl/gsl_multifit_nlin.h>
#include<gsl/gsl_cdf.h>
#include<gsl/gsl_randist.h>
#include<fftw3.h>

#include<sys/time.h>
#include<sys/resource.h>
#include<getopt.h>

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/valarray.hpp>
#include <boost/serialization/map.hpp>

using namespace std;
```

Uses valarray.

?? ⟨Data structures ??⟩≡ (? 0—1)

```
typedef valarray<double> varray;

struct input
{
  friend class boost::serialization::access;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int)
  {
    ar & xva; ar & yva; ar & npoint;
    ar & ch_width; ar & filename;
  }
  varray xva, yva;
  int npoint;
  double ch_width;
  string filename;
  input (): npoint (0), ch_width (-1) {}
};

struct fit_data
{
  friend class boost::serialization::access;
  double A, M, sigma, beta, b, alpha, chi2, err, Merr, Aerr, Serr;
  double N, dof, P;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int)
  {
    ar & A; ar & M; ar & sigma; ar & beta; ar & b; ar & alpha;
    ar & chi2; ar & err; ar & Merr; ar & Aerr; ar & Serr;
    ar & N; ar & dof; ar & P;
  }
  fit_data (): chi2 (-1) {}
};

struct analyze
{
  friend class boost::serialization::access;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int)
  {
    ar & index; ar & b; ar & y;
    ar & nonzero_begin; ar & maximum;
    ar & valgauss1; ar & valgauss2; ar & valgauss3;
    ar & valgauss_e;
    ar & stat_maximum; ar & fitted_peaks;
    ar & peaks;
  }

  int index;
  varray b, y;                    // b == filtered_y, y == fpeak_y
```

```
  int nonzero_begin;
  varray valgauss1, valgauss2, valgauss3, valgauss_e;
  list<int> stat_maximum;
  map<int,bool> maximum;
  varray dx, dy;
  list<fit_data> peaks, fitted_peaks;
};

struct fit_data_cmp
{
  bool operator()(const fit_data& f1, const fit_data& f2) {return f1.M < f2.M;}
};

struct data
{
  int range1, range2;
  double *y, *x;
  double *sigma;
  int ngauss;
};
```

Defines:
  valarray, used in chunks 42–44 and 48.
Uses ch_width and npoint.

?? ⟨* 48⟩+≡ ◁48 ?? ▷

```
⟨Data structures ??⟩
void median_filter (varray& series, varray& output, int w);
void average_filter (varray& series, varray& output, int w);
double median (valarray<int> &numbers);
double mean (varray &data);
void out_array (ostream& s, const varray& arrayx,
                const varray& arrayy);
void smart_diff (const varray& a1, const varray& a2,
                 varray& out);
double peak_fun (double x, double A, double M, double sigma, double b);
double peak_int (double A, double M, double sigma);
int expb_df(const gsl_vector *x, void *dat, gsl_matrix *J);
fit_data fit_multi_peak (list<int>::const_iterator rough_center, const list<int> mlist,
                         int range1, int range2, input &in, analyze &ana);
fit_data fit_multi_peak2 (list<int>::const_iterator rough_center, const list<int> mlist,
                          int range1, int range2, input &in, analyze &ana);
double mpeak_fun (double x, const gsl_vector *p, int ngauss);
double dmpeak_fun_dj (double x, const gsl_vector *p, int ngauss, int j);
double mpeak_fun2 (double x, const gsl_vector *p, int ngauss);
double dmpeak_fun2_dj (double x, const gsl_vector *p, int ngauss, int j);
int mexpb_f (const gsl_vector *x, void *dat, gsl_vector *f);
int mexpb_df (const gsl_vector *x, void *dat, gsl_matrix *J);
int mexpb_fdf (const gsl_vector *x, void *data, gsl_vector *f, gsl_matrix *J);
int mexpb_f2 (const gsl_vector *x, void *dat, gsl_vector *f);
int mexpb_df2 (const gsl_vector *x, void *dat, gsl_matrix *J);
int mexpb_fdf2 (const gsl_vector *x, void *data, gsl_vector *f, gsl_matrix *J);
double mopeak_fun (double x, double A, double M, double sigma,
  double b, double alpha);
void compute_background (varray& series, varray& output, int w);
void check_gauss (varray& xva, varray& yva, varray& valgauss1, varray& valgauss2, varray& valgauss3, v
double poisson (int a, int n);
double logpoisson (int a, int n);
double hpge_fwhm (double e);
double gauss_pdf (double x, double m, double s);
double gauss_cdf (double x, double m, double s);
double nmin (double x, double lambda, double n);
double nmin_int (double n, double lambda);
double solve (double n, double y);
void process_options (int argc, char *argv[]);
double load_file (const char *f, varray &x, varray &y);
valarray<double> fodec (valarray<double>& signal, double e);
double stddev (varray &x);
list<int> maximum_flt(input &i, analyze &a);

double peakthr = 8;
double pcpkthr = .10;
int fit_gap = 20;
int max_multi_peak = 10;
int median_l = 10;
```

```
int match_l = 3;
int maxiter = 100;
string single_peak_file;
bool chi2_dont_check_p = false;
const double fwhm = 2.35482004503095;

int npoint;
double ch_width;
vector<input> inputs;
vector<analyze> results;

int main(int argc, char *argv[])
{
  // struct rlimit lim;
  // getrlimit(RLIMIT_CPU, &lim);
  // lim.rlim_cur = 60;
  // setrlimit(RLIMIT_CPU, &lim);
  cout << "This is PEAKDECO version $Id: peakdeco.nw 53 2009-10-29 20:01:40Z hatta $" <<
  //feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
  process_options (argc, argv);
  if (optind >= argc)
    {
      cout << "Usage: peakdeco [options] <filenames...>" << endl;
      exit (2);
    }
  for (int f_index = optind; f_index < argc; f_index++)
    {
      inputs.push_back (input ());
      inputs.back().ch_width =
        load_file (argv[f_index], inputs.back().xva, inputs.back().yva);
      inputs.back().npoint = inputs.back().xva.size();
      inputs.back().filename = string (argv[f_index]);
      results.push_back (analyze ());
    }

  assert (inputs.size () > 0);
  varray input_np (inputs.size ()), input_cw (inputs.size ());
  for (int i = 0; i < inputs.size (); i++)
    {
      input_np[i] = inputs[i].npoint;
      input_cw[i] = inputs[i].ch_width;
    }
  if (inputs.size () > 1 ∧ (stddev (input_np) > 1e-6 ∨ stddev (input_cw) > 1e-6))
    {
      cout << "error: all files should have identical format" << endl;
      cout << "stddev (input_np) = " << stddev (input_np) << endl;
      cout << "stddev (input_cw) = " << stddev (input_cw) << endl;
      exit (3);
    }
  npoint = inputs[0].npoint;
```

```
ch_width = inputs[0].ch_width;
cout << "Common npoint " << npoint << " channel width " << ch_width << "." << endl;

for (int i = 0; i < inputs.size (); i++)
  {
    cout << "compute background in " << inputs[i].filename;
    results[i].b.resize (npoint);
    if (single_peak_file.empty ())
      compute_background (inputs[i].yva, results[i].b, median_l);
    results[i].y.resize (npoint);
    smart_diff (inputs[i].yva, results[i].b, results[i].y);
    cout << ": mean y " << mean (results[i].y)
         << " mean b " << mean (results[i].b) << endl;
  }
for (int i = 0; i < inputs.size (); i++)
  {
    for (int j = 0; j < inputs[i].yva.size (); j++)
      {
        if (inputs[i].yva[j] >= 0)
          {
            results[i].nonzero_begin = j;
            break;
          }
      }
  }
if (single_peak_file.empty ())
  {
    for (int i = 0; i < inputs.size (); i++)
      {
        check_gauss (inputs[i].xva, inputs[i].yva, results[i].valgauss1,
                     results[i].valgauss2, results[i].valgauss3, results[i].valgauss_e);
        for(int j = 1; j < results[i].valgauss_e.size (); j++)
          {
            const int msw = 5*5;
            int start = fmax (j - (msw - 1)÷2, 0);
            int end = fmin (j + (msw - 1)÷2, results[i].valgauss_e.size ());
            int len = end - start + 1;
            double rmin = varray (results[i].valgauss1[slice (start, len, 1)]).min ();
            double rmax = varray (results[i].valgauss1[slice (start, len, 1)]).max ();
            double v1 = results[i].valgauss1[j];
            double v2 = results[i].valgauss2[j];
            double v3 = results[i].valgauss3[j];
            if (rmin < rmax ∧ fabs (v1 - rmin) < 1e-20 ∧ v1 ≤ v2 ∧ v1 ≤ v3)
              {
                int p = 0;
                while (p < npoint - 1 ∧ inputs[i].xva[p] < results[i].valgauss_e[j]) p++;
                results[i].maximum[p] = true;
              }
          }
        cout << "Maximum search in " << inputs[i].filename << ": "
```

```
            ≪ results[i].maximum.size () ≪ " peaks." ≪ endl;
      }
    for (int i = 0; i < inputs.size (); i++)
      {
        results[i].stat_maximum = maximum_flt (inputs[i], results[i]);
        cout ≪ "In " ≪ inputs[i].filename
            ≪ " after filtering: " ≪ results[i].stat_maximum.size () ≪ " peaks."
            ≪ endl;
      }
  }
else
  {
    ifstream peak_file (single_peak_file.c_str ());
    if (¬peak_file)
      {
        cout ≪ "Can't open peaks file." ≪ endl;
        exit (2);
      }
    while (peak_file)
      {
        double e;
        peak_file ≫ e;
        if (¬peak_file) break;
        for (int i = 0; i < inputs.size (); i++)
          {
            int p = (e - inputs[i].xva[0]) ÷ ch_width;
            results[i].stat_maximum.push_back (p);
          }
      }
  }
for (int i = 0; i < results.size (); i++)
  results[i].stat_maximum.sort ();
for (int i = 0; i < inputs.size (); i++)
  {
    list<fit_data> fitted_peaks;
    // shortcut
    list<int> &s = results[i].stat_maximum;
    list<int>::const_iterator si;
    for(si = s.begin (); si ≠ s.end (); si++)
      {
        list<int>::const_iterator ti = si;
        list<int>::const_iterator r1 = si, r2 = si;
        int count = 1;
        int range1 = *si, range2 = *si;
        range1 -= fit_gap÷2;
        range2 += fit_gap÷2;
        range1 = fmax (results[i].nonzero_begin + 3, range1);
        range2 = fmin (npoint - 1, range2);
        list<int> single_peak;
        single_peak.push_back (*si);
```

```
if (*si ≤ range1 ∨ *si ≥ range2)
  {
    cout ≪ "Invalid fitting range, skipping this peak." ≪ endl;
    continue;
  }
fit_data fit = fit_multi_peak2 (si, single_peak, range1, range2, inputs[i], results[i]);
if ((fit.P ≥ 0.01 ∨ chi2_dont_check_p) ∧ fit.chi2 ≥ 0)
  {
    cout ≪ "Successful fit." ≪ endl;
    results[i].fitted_peaks.push_back (fit);
    continue;
  }
cout ≪ "Couldn't fit single peak, will try to fit multiple." ≪ endl;
while(ti ≠ s.begin () ∧ *si - *ti < fit_gap)
  {
    range1 = *ti;
    r1 = ti;
    if (ti ≠ si) count++;
    ti--;
  }
ti = si;
while (ti ≠ s.end () ∧ *ti - *si < fit_gap)
  {
    range2 = *ti;
    r2 = ti;
    if (ti ≠ si) count++;
    ti++;
  }
range1 -= fit_gap÷2;
range1 = fmax (results[i].nonzero_begin + 3, range1);
range2 += fit_gap÷2;
range2 = fmin (range2, npoint - 1);
assert (range1 < range2);
int n = range2 - range1 + 1;
// а) пробуем профитировать с найденной конфигурацией максимумов;
list<int> many_peaks (r1, ++r2);
if (*si ≤ range1 ∨ *si ≥ range2)
  {
    cout ≪ "Invalid fitting range, skipping this peak." ≪ endl;
    continue;
  }
fit = fit_multi_peak2 (si, many_peaks, range1, range2, inputs[i], results[i]);
// б) проверяем, сошлось ли фитирование, если да, то алгоритм
//    переходит к следующему блоку данных;
if ((fit.P ≥ 0.01 ∨ chi2_dont_check_p) ∧ fit.chi2 ≥ 0)
  {
    cout ≪ "Successful fit." ≪ endl;
    results[i].fitted_peaks.push_back (fit);
    continue;
  }
```

```
else
  {
    int deconv_tries = 0;
    cout << "Fit unsuccessful, trying deconvolution." << endl;
    // в) если не сошлось, то вычисляем деконволюцию;
    varray r = inputs[i].yva[slice (range1, n, 1)];
    varray deconv = fodec (r, inputs[i].xva[*si]);
    // г) находим в деконволюции максимальное значение, отсутствующее
    //    в начальном наборе максимумов, и добавляем его в набор;
  another_deconv:
    deconv_tries++;
    int max_pos = -1;
    double max_val = 0;
    for (int k = 0; k < deconv.size (); k++)
      {
        bool newp = true;
        if (deconv[k] > max_val)
          {
            for (list<int>::iterator l = many_peaks.begin (); l != many_peaks.end (); l++)
              {
                if (fabs (k - *l + range1) < 3)
                  {
                    cout << "Skipping already present " << *l
                         << " (" << inputs[i].xva[*l] << ")." << endl;
                    newp = false;
                    break;
                  }
              }
            if (newp)
              {
                max_val = deconv[k];
                max_pos = k;
              }
          }
      }
    max_pos += range1;
    cout << "Add new maximum at " << inputs[i].xva[max_pos] << "." << endl;
    range1 = fmin (range1, max_pos - fit_gap/2);
    range1 = fmax (results[i].nonzero_begin + 3, range1);
    range2 = fmax (range2, max_pos + fit_gap/2);
    range2 = fmin (npoint - 1, range2);
    many_peaks.push_back (max_pos);
    // д) снова пробуем фитирование;
    if (*si <= range1 || *si >= range2)
      {
        cout << "Invalid fitting range, skipping this peak." << endl;
        continue;
      }
    fit = fit_multi_peak2 (si, many_peaks, range1, range2, inputs[i], results[i]);
    if ((fit.P >= 0.01 || chi2_dont_check_p) && fit.chi2 >= 0)
```

```
            {
              // е) если сошлось, то сообщить об успешном завершении обработки этого блока дан
              cout ≪ "Successful fit." ≪ endl;
              results[i].fitted_peaks.push_back (fit);
            }
          else
            {
              cout ≪ "Failed to fit with " ≪ deconv_tries ≪ " additional maximums." ≪ endl
              // ж) если не сошлось и число попыток превысило все разумные
              //    пределы, то сообщить о неудаче.
              if (deconv_tries ≥ 3)
                {
                  cout ≪ "Too many tries, giving up fitting this peak." ≪ endl;
                  continue;
                }
              // з) если не сошлось, то перейти к шагу "г";
              goto another_deconv;
            }
        }
      cout ≪ endl ≪ endl;
    }
  }
for (int i = 0; i < inputs.size (); i++)
  {
    vector<fit_data> fv(results[i].fitted_peaks.begin(),
                        results[i].fitted_peaks.end());
    sort (fv.begin(), fv.end(), fit_data_cmp());
    int nf = fv.size ();
    int j = 0;
    while (j < nf)
      {
        int k = j;
        double pM, psigma, M, sigma;
        pM = fv[j].M;
        psigma = fv[j].sigma;
        const double fwhms = 2.35482004503095 ÷ 2;

        while(k < nf ∧ (
                        M = fv[k].M,
                        sigma = fv[k].sigma,
                        M - pM < psigma*fwhms + sigma*fwhms))
          {
            psigma = sigma;
            pM = M;
            k++;
          }
        int best_pos = j;
        double best_chi2 = DBL_MAX;
        for(int l = j; l < k; l++)
          {
```

```
            if(fv[l].chi2 < best_chi2)
              {
                best_chi2 = fv[l].chi2;
                best_pos = l;
              }
          }
        results[i].peaks.push_back(fv[best_pos]);
        j = k;
      }
  }
 ofstream out ("peakdeco.dat");
 boost::archive::text_oarchive oa(out);
 oa << inputs << results;
 cout << endl;
 return 0;
}

void compute_background(varray& series, varray& output, int w)
{
 assert(series.size() == output.size());
 const int np = series.size();
 varray min_array (np);
 int junk_end = 0;
 while (junk_end < np && series[junk_end] <= 0) junk_end++;
 junk_end++;
 for(int p = 0; p < np; p++)
  {
   int start = (p - w >= 0 ? p - w : 0);
   if (start < junk_end) start = junk_end;
   if (p < start)
     {
       min_array[p] = 0;
       continue;
     }
   int end = (p + w < np ? p + w : np - 1);
   int len = end - start;
   if (len <= 0)
     {
       min_array[p] = series[p];
       continue;
     }
   assert (len > 0);
   varray x = series[slice(start, len, 1)];
   double m = x.min ();
   output[p] = solve (len, m);
  }
}


double hpge_fwhm (double e)
```

```
{
  const double a = 0.0369074;
  const double b = 0.553757;
  return a * sqrt (fmax (e, 0)) + b;
}

void check_gauss (varray& xva, varray& yva, varray& valgauss1, varray& valgauss2, varray& valgauss3, v
{
  int npoint1 = npoint * 5;

  valgauss1.resize (npoint1);
  valgauss2.resize (npoint1);
  valgauss3.resize (npoint1);
  e.resize (npoint1);

  for (int p = 0; p < npoint1; p++)
    {
      double m = (xva[npoint-1] - xva[0]) ÷ npoint1 * p + xva[0];
      int p0 = 0;
      while (p0 < npoint ∧ xva[p0] < m) p0++;
      double fwhm_here = hpge_fwhm (m);
      double sigma = fwhm_here ÷ fwhm;
      int gauss_width = fwhm_here ÷ ch_width;
      gauss_width *= 2;
      int start = fmax (p0 - gauss_width, 0);
      int end = fmin (p0 + gauss_width, npoint - 1);
      int len = end - start + 1;

      varray check_x = xva[slice(start, len, 1)];
      varray check_y1 = yva[slice(start, len, 1)];
      double min = check_y1.min ();
      check_y1 -= min;
      double sum = check_y1.sum ();
      if (sum ≠ 0)
        check_y1 ÷= sum;
      else
        {
          continue;
        }
      varray check_y2 (check_y1);
      varray check_y3 (check_y1);
      varray gauss_val1 (check_y1);
      varray gauss_val2 (check_y1);
      varray gauss_val3 (check_y1);
      for (int i = 0; i < check_y1.size (); i++)
          {
            gauss_val1[i] = gsl_ran_gaussian_pdf (check_x[i] - m, sigma*1);
            gauss_val2[i] = gsl_ran_gaussian_pdf (check_x[i] - m, sigma*5);
            gauss_val3[i] = gsl_ran_gaussian_pdf (check_x[i] - m, sigma÷5);
          }
```

```
      if (gauss_val1.sum () > 0)
        gauss_val1 ÷= gauss_val1.sum ();
      if (gauss_val2.sum () > 0)
        gauss_val2 ÷= gauss_val2.sum ();
      if (gauss_val3.sum () > 0)
        gauss_val3 ÷= gauss_val3.sum ();
      check_y1 = check_y1 - gauss_val1;
      check_y2 = check_y2 - gauss_val2;
      check_y3 = check_y3 - gauss_val3;
      double r1 = pow (check_y1, 2.).sum ();
      double r2 = pow (check_y2, 2.).sum ();
      double r3 = pow (check_y3, 2.).sum ();
      valgauss1[p] = r1÷len;
      valgauss2[p] = r2÷len;
      valgauss3[p] = r3÷len;
      e[p] = m;
    }
}

list<int> maximum_flt(input &i, analyze &a)
{
  a.dx.resize(npoint - 1);
  a.dy.resize(npoint - 1);
  for (int j = 0; j < npoint - 1; j++)
    {
      a.dy[j] = i.yva[j+1] - i.yva[j];
      a.dx[j] = (i.xva[j] + i.xva[j+1])÷2;
    }
  const int diff_window_half = 4;

  list<int> r;
  map<int,bool>::iterator fmi;
  for (fmi = a.maximum.begin (); fmi ≠ a.maximum.end (); fmi++)
    {
      int f = fmi→first;
      // if (a.y[f] < sqrt(a.b[f]) * match_l) continue;
      // if (i.yva[f] < (1 + pcpkthr) * a.b[f]) continue;
      if (i.yva[f] < peakthr) continue;
      // if (a.valgauss1[f] > 0.2) continue;
      int dwa = fmax (0, f-diff_window_half);
      int dwb = fmin (npoint - 1, f+diff_window_half);
      double dmax = abs (varray(a.dy[slice (dwa, dwb-dwa, 1)])).max ();
      varray btmp = a.b[slice (dwa, dwb-dwa, 1)];
      btmp *= 2;
      double bmax = (sqrt (btmp) * 3).max ();
      if (dmax < bmax) continue;
      r.push_back (f);
    }
  return r;
}
```

Defines:
ch_width, used in chunks 6, 7, 11, 44, and 48.
check_gauss, used in chunk 13a.
compute_background, used in chunks 10a and 43.
fit_gap, used in chunks 16 and 44.
fwhm, used in chunks 11, 28, 37, and 48.
hpge_fwhm, used in chunks 11, 28, 37, 43, and 48.
main, never used.
match_l, used in chunks 14a and 44.
max_multi_peak, used in chunks 28, 34, and 44.
maxiter, used in chunks 28, 36, 37, and 44.
median_l, used in chunks 10a and 44.
mopeak_fun, used in chunks 28 and 39.
npoint, used in chunks 6, 7, 10a, 11, 13a, 14a, 16, 28, 37, 44, and 48.
out_array, never used.
pcpkthr, used in chunks 14a and 44.
peakthr, used in chunks 14a, 28, 37, and 44.
smart_diff, used in chunk 10a.
Uses average_filter, dmpeak_fun2_dj, dmpeak_fun_dj, gauss_cdf, gauss_pdf, load_file, logpoisson, mean, median, median_filter, mexpb_df, mexpb_df2, mexpb_f, mexpb_f2, mexpb_fdf, mexpb_fdf2, mpeak_fun, mpeak_fun2, nmin, nmin_int, peak_int, poisson, process_options, solve, stddev, and valarray.

3. Фитирование в GSL

Процедура фитирования основана на gsl_multifit_nlin из библиотеки GSL. См. Gnu Scientific Library Reference Manual, стр. 391.

Фитируется функцию, состоящую из суммы нескольких гауссиан и фона.

$$y(x) = \alpha x + b + \sum_{k=1}^{F} A_k e^{-\frac{(x-M)^2}{2\sigma^2}},$$

где фон уже задается как наклонная линия, и общее число параметров равно $3F + 2$. Параметры передаются в следующем формате:

$$(\alpha, b, A_1, M_1, \sigma_1, \ldots A_n, M_n, \sigma_n).$$

Второй вариант для фитирования:

$$y(x) = \alpha x + b + \sum_{k=1}^{F} \left[\frac{A_k}{\sqrt{2\pi}\sigma_k} e^{-\frac{(x-M_k)^2}{2\sigma_k^2}} + \frac{\beta_k}{2} \Phi\left(\frac{x - M_k}{\sqrt{2}\sigma_k}\right) \right],$$

где добавляется функция $\Phi\left(\frac{x-M}{\sqrt{2}\sigma}\right)$ — интеграл ошибок, то есть интеграл от функции Гаусса. Формат передачи параметров:

$$(\alpha, b, A_1, M_1, \sigma_1, \beta_1, \ldots A_n, M_n, \sigma_n, \beta_n).$$

Общее число параметров во втором случае $4F + 2$.

?? ⟨* 48⟩+≡ ◁?? ??▷

```
double mpeak_fun (double x, const gsl_vector *p, int ngauss)
{
 double alpha = gsl_vector_get (p, 0);
 double b = gsl_vector_get (p, 1);
 varray A (ngauss), M (ngauss), sigma (ngauss);
 for (int i = 0; i < ngauss; i++)
 {
  A[i] = gsl_vector_get (p, i*3 + 2);
  M[i] = gsl_vector_get (p, i*3 + 3);
  sigma[i] = gsl_vector_get (p, i*3 + 4);
 }
 varray X (x, ngauss);
 varray val (ngauss), val1 (ngauss);
 val = pow (X - M, 2);
 val /= (sigma*sigma*2);
 val = exp (-val);
 val *= A;
 return val.sum () + alpha * x + b;
}

double mpeak_fun2 (double x, const gsl_vector *p, int ngauss)
{
 double alpha = gsl_vector_get (p, 0);
 double b = gsl_vector_get (p, 1);
 varray A (ngauss), M (ngauss), sigma (ngauss), beta (ngauss);
 for (int i = 0; i < ngauss; i++)
```

```
    {
      A[i] = gsl_vector_get (p, i*4 + 2);
      M[i] = gsl_vector_get (p, i*4 + 3);
      sigma[i] = gsl_vector_get (p, i*4 + 4);
      beta[i] = gsl_vector_get (p, i*4 + 5);
    }
  varray X (x, ngauss);
  varray val (ngauss), val1 (ngauss);
  val = pow(X - M, 2);
  val /= (sigma*sigma*2);
  val = exp(-val);
  val *= A;
  val /= sqrt (2*M_PI);
  val /= sigma;                    // val = A*normal_pdf[M, sigma] (x)
  for (int i = 0; i < ngauss; i++)
    val[i] += (beta[i] / 2) * erf ((x-M[i]) / (sqrt (2) * sigma[i]));
  return val.sum () + alpha * x + b;
}
```

Defines:
mpeak_fun, used in chunks 25 and 43.
mpeak_fun2, used in chunk 25.

Кроме задания функции, GSL требует якобиан этой функции. В переменной j передается номер параметра, по которому производится дифференцирование.

$$y_2 = \frac{A\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma} + \frac{\beta\,\mathrm{erf}\left(\frac{x-M}{\sqrt{2}\,\sigma}\right)}{2} + \alpha\,x + b$$

$$\frac{\partial y_2}{\partial A} = \frac{e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma}$$

$$\frac{\partial y_2}{\partial M} = \frac{A\,(x-M)\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma^3} - \frac{\beta\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma}$$

$$\frac{\partial y_2}{\partial \sigma} = \frac{A\,(x-M)^2\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma^4} - \frac{\beta\,(x-M)\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma^2} - \frac{A\,e^{-\frac{(x-M)^2}{2\,\sigma^2}}}{\sqrt{2}\,\sqrt{\pi}\,\sigma^2}$$

$$\frac{\partial y_2}{\partial \beta} = \frac{\mathrm{erf}\left(\frac{x-M}{\sqrt{2}\,\sigma}\right)}{2}$$

?? ⟨* 48⟩+≡ ◁?? ??▷

```
double dmpeak_fun_dj(double x, const gsl_vector *p, int ngauss, int j)
{
  if(j ≡ 0) return x;
  if(j ≡ 1) return 1;
  double alpha = gsl_vector_get(p, 0);
  double b = gsl_vector_get(p, 1);
  varray A(ngauss), M(ngauss), sigma(ngauss);
  for(int i = 0; i < ngauss; i++)
  {
    A[i] = gsl_vector_get(p, i*3 + 2);
    M[i] = gsl_vector_get(p, i*3 + 3);
    sigma[i] = gsl_vector_get(p, i*3 + 4);
  }
  int g = (j - 2) ÷ 3; // generation
  int i = (j - 2) % 3; // serial number
  double ret = 0;
  double Mj = M[g];
  double Aj = A[g];
  double sigmaj = sigma[g];
  double dfdA, dfdM, dfds;
  switch(i)
  {
  case 0: // A
    dfdA = exp(-(x-Mj)*(x-Mj)÷(2*sigmaj*sigmaj));
    ret = dfdA;
    break;
  case 1: // M
    dfdM = Aj*exp(-(x-Mj)*(x-Mj)÷(2*sigmaj*sigmaj))
      * (x-Mj)÷(sigmaj*sigmaj);
    ret = dfdM;
    break;
```

```
  case 2: // sigma
    dfds = Aj*exp(-(x-Mj)*(x-Mj)÷(2*sigmaj*sigmaj))
      * (x-Mj)*(x-Mj);
    dfds ÷= sigmaj;
    dfds ÷= sigmaj;
    dfds ÷= sigmaj;
    ret = dfds;
    break;
  default: cout ≪ "Bug!" ≪ endl;
  }
  return ret;
}

double dmpeak_fun2_dj (double x, const gsl_vector *p, int ngauss, int j)
{
  if (j ≡ 0) return x;
  if (j ≡ 1) return 1;
  double alpha = gsl_vector_get (p, 0);
  double b = gsl_vector_get (p, 1);
  varray A (ngauss), M (ngauss), sigma (ngauss), beta (ngauss);
  for (int i = 0; i < ngauss; i++)
    {
      A[i] = gsl_vector_get (p, i*4 + 2);
      M[i] = gsl_vector_get (p, i*4 + 3);
      sigma[i] = gsl_vector_get (p, i*4 + 4);
      beta[i] = gsl_vector_get (p, i*4 + 5);
    }
  int g = (j - 2) ÷ 4; // generation
  int i = (j - 2) % 4; // serial number
  double ret = 0;
  double Mj = M[g];
  double Aj = A[g];
  double sigmaj = sigma[g];
  double betaj = beta[g];
  double dfdA, dfdM, dfds, dfdb;
  double isigma3 = 1;
  double isigma2 = 1;
  isigma2 ÷= sigmaj;
  isigma2 ÷= sigmaj;
  isigma3 = isigma2 ÷ sigmaj;
  double isigma4 = isigma3 ÷ sigmaj;
#define EM2(x, M, sigma) (exp (-((x)-(M))*((x)-(M))/(2*(sigma)*(sigma))))
  switch(i)
    {
    case 0: // A
      dfdA = (1÷(sqrt (2*M_PI) * sigmaj)) * EM2 (x, Mj, sigmaj);
      ret = dfdA;
      break;
    case 1: // M
      dfdM = (Aj ÷ (sqrt (2*M_PI))) * isigma3 * (x - Mj) * EM2 (x, Mj, sigmaj)
```

```
      - betaj * (1 ÷ (sqrt (2*M_PI) * sigmaj)) * EM2 (x, Mj, sigmaj);
    ret = dfdM;
    break;
  case 2: // sigma
    dfds = (-Aj ÷ sqrt (2*M_PI)) * isigma2 * EM2 (x, Mj, sigmaj)
      + (Aj ÷ sqrt (2 * M_PI)) * isigma4 * (x-Mj) * (x-Mj) * EM2 (x, Mj, sigmaj)
      - (betaj * (x-Mj) ÷ (sqrt (2*M_PI))) * isigma2 * EM2 (x, Mj, sigmaj);
    ret = dfds;
    break;
  case 3: // beta
    dfdb = (1.0 ÷ 2.0) * erf ((x-Mj) ÷ (sigmaj * sqrt (2)));
    ret = dfdb;
    break;
  default: cout ≪ "Bug2!" ≪ endl;
  }
#undef EM2
  return ret;
}
```

Defines:
dmpeak_fun2_dj, never used.
dmpeak_fun_dj, used in chunks 25 and 43.
EM2, never used.

Диапазоны будем передавать через параметр *dat GSL'a, который будет структурой типа data.

?? ⟨* 48⟩+≡ ◁??

```
int mexpb_f (const gsl_vector *x, void *dat, gsl_vector *f)
{
  double *y = ((data *)dat)->y;
  double *e = ((data *)dat)->x;
  double *sigma = ((data *)dat)->sigma;
  int range1 = ((data *)dat)->range1;
  int range2 = ((data *)dat)->range2;
  int ngauss = ((data *)dat)->ngauss;
  int n = range2 - range1;

  for(int i = 0; i < n; i++)
  {
    double Xi = e[i];
    double Yi = mpeak_fun (Xi, x, ngauss);
    gsl_vector_set (f, i, (Yi - y[i])/sigma[i]);
  }
  return GSL_SUCCESS;
}

int mexpb_df (const gsl_vector *x, void *dat, gsl_matrix *J)
{
  double *e = ((data *)dat)->x;
  double *sigma = ((data *)dat)->sigma;
  int range1 = ((data *)dat)->range1;
  int range2 = ((data *)dat)->range2;
  int n = range2 - range1;
  int ngauss = ((data *)dat)->ngauss;

  for(int i = 0; i < n; i++)
  {
    double Xi = e[i];
    for(int k = 0; k < ngauss*3 + 2; k++)
      gsl_matrix_set (J, i, k, dmpeak_fun_dj (Xi, x, ngauss, k) / sigma[i]);
  }
  return GSL_SUCCESS;
}

int mexpb_fdf (const gsl_vector *x, void *data, gsl_vector *f, gsl_matrix *J)
{
  mexpb_f (x, data, f);
  mexpb_df (x, data, J);
  return GSL_SUCCESS;
}

int mexpb_f2 (const gsl_vector *x, void *dat, gsl_vector *f)
{
  double *y = ((data *)dat)->y;
```

```
  double *e = ((data *)dat)->x;
  double *sigma = ((data *)dat)->sigma;
  int range1 = ((data *)dat)->range1;
  int range2 = ((data *)dat)->range2;
  int ngauss = ((data *)dat)->ngauss;
  int n = range2 - range1;

  for(int i = 0; i < n; i++)
  {
    double Xi = e[i];
    double Yi = mpeak_fun2 (Xi, x, ngauss);
    gsl_vector_set (f, i, (Yi - y[i])/sigma[i]);
  }
  return GSL_SUCCESS;
}

int mexpb_df2 (const gsl_vector *x, void *dat, gsl_matrix *J)
{
  double *e = ((data *)dat)->x;
  double *sigma = ((data *)dat)->sigma;
  int range1 = ((data *)dat)->range1;
  int range2 = ((data *)dat)->range2;
  int n = range2 - range1;
  int ngauss = ((data *)dat)->ngauss;

  for(int i = 0; i < n; i++)
  {
    double Xi = e[i];
    for(int k = 0; k < ngauss*4 + 2; k++)
      gsl_matrix_set (J, i, k, dmpeak_fun2_dj (Xi, x, ngauss, k) / sigma[i]);
  }
  return GSL_SUCCESS;
}

int mexpb_fdf2 (const gsl_vector *x, void *data, gsl_vector *f, gsl_matrix *J)
{
  mexpb_f2 (x, data, f);
  mexpb_df2 (x, data, J);
  return GSL_SUCCESS;
}

fit_data fit_multi_peak(list<int>::const_iterator rough_center, const list<int> mlist,
                        int range1, int range2, input &in, analyze &ana)
{
  // some checks
  assert (range1 >= 0 && range2 > 0);
  assert (range1 < range2 && *rough_center > range1 && *rough_center < range2);
  vector<int> what_to_fit;
  vector<fit_data> fit_results;
  list<int>::iterator mi;
```

```
double maxval = 0;
list<int>::const_iterator rr;
int ngauss = mlist.size ();
what_to_fit.resize (ngauss);
copy (mlist.begin (), mlist.end (), what_to_fit.begin ());
const int N = range2 - range1 + 1;
const int p = 3*ngauss + 2;
const double dof = N - p;
cout << "Trying to fit " << ngauss << " peaks with center at "
    << in.xva[*rough_center] << ", fitting range from "
    << in.xva[range1] << " to "
    << in.xva[range2]
    << " (" << N << " points)." << endl;
cout << "List of initial estimates: ";
for (vector<int>::iterator li = what_to_fit.begin (); li != what_to_fit.end (); li++)
  cout << in.xva[*li] << "; ";
cout << endl;

assert (ngauss > 0);

if(p >= N)
  {
    cout << "Skipping fit because p >= N." << endl;
    return fit_data ();
  }
if (ngauss > max_multi_peak)
  {
    cout << "Skipping fit because Npeaks exceeds max_multi_peak ("
        << ngauss << ")." << endl;
    return fit_data ();
  }

double x_init[p];
x_init[0] = 0;                    // alpha
x_init[1] = ana.b[*rough_center]; // b
for(int i = 0; i < ngauss; i++)
  {
    x_init[i*3 + 2] = ana.y[what_to_fit[i]];  // A
    x_init[i*3 + 3] = in.xva[what_to_fit[i]]; // M
    x_init[i*3 + 4] = 0.4;                    // sigma
  }

const gsl_multifit_fdfsolver_type *T;
gsl_multifit_fdfsolver *s;
int status;
int iter = 0;
gsl_matrix *covar = gsl_matrix_alloc(p,p);
double y[N], e[N], sigma[N];
struct data d;
gsl_multifit_function_fdf f;
```

```
gsl_vector_view x = gsl_vector_view_array(x_init, p);
f.f = &mexpb_f;
f.df = &mexpb_df;
f.fdf = &mexpb_fdf;
f.n = N;
f.p = p;
f.params = &d;
d.range1 = 0;
d.range2 = N;
d.sigma = sigma;
d.y = y;
d.x = e;
d.ngauss = ngauss;

for(int i = 0; i < N; i++)
  {
    y[i] = in.yva[i + range1];
    e[i] = in.xva[i + range1];
    sigma[i] = sqrt(y[i]);
    if(sigma[i] == 0) sigma[i] = 1; // correction for sigma_i = 0
  }

T = gsl_multifit_fdfsolver_lmsder;
s = gsl_multifit_fdfsolver_alloc(T, N, p);
gsl_multifit_fdfsolver_set(s, &f, &x.vector);
do
  {
    iter++;
    status = gsl_multifit_fdfsolver_iterate(s);
    if(status)
      {
        cout << gsl_strerror (status) << endl;
        //break;
      }
    // epsabs == 0, esprel == 1e-4
    varray A(ngauss), M(ngauss), var(ngauss);
    for(int i = 0; i < ngauss; i++)
      {
        A[i] = gsl_vector_get (s->x, i*3 + 2);
        M[i] = gsl_vector_get (s->x, i*3 + 3);
        var[i] = fabs(gsl_vector_get (s->x, i*3 + 4));
        if (var[i] > 10000)
          {
            cout << "Fit failed, because sigma to big." << endl;
            return fit_data ();
          }
      }
    status = gsl_multifit_test_delta(s->dx, s->x, 0, 1e-4);
  } while (status == GSL_CONTINUE && iter < maxiter);
```

```
if(iter ≡ maxiter)
  {
    cout ≪ "This is a failed fit, because maximum steps reached." ≪ endl;
    return fit_ data ();
  }
double alpha = gsl_ vector_ get(s→x, 0);
double b = gsl_ vector_ get(s→x, 1);
varray A(ngauss), M(ngauss), var(ngauss);
for(int i = 0; i < ngauss; i++)
  {
    A[i] = gsl_ vector_ get(s→x, i*3 + 2);
    M[i] = gsl_ vector_ get(s→x, i*3 + 3);
    var[i] = fabs(gsl_ vector_ get(s→x, i*3 + 4));
  }

cout ≪ "Fitting complete: alpha=" ≪ alpha;
cout ≪ " b=" ≪ b ≪ " iter=" ≪ iter ≪ endl;
cout ≪ setw(15) ≪ "A" ≪ setw(15) ≪ "M" ≪ setw(15)
     ≪ "sigma" ≪ endl;
cout ≪ "===========================================" ≪ endl;
for (int i = 0; i < ngauss; i++)
  cout ≪ setw(15) ≪ A[i] ≪ setw(15) ≪ M[i]
       ≪ setw(15) ≪ var[i] ≪ endl;
gsl_ vector *test = gsl_ vector_ alloc(N);
mexpb_ f (s→x, &d, test);
double chi2 = 0;
for (int ii = 0; ii < N; ii++)
  chi2 += pow (gsl_ vector_ get (test, ii), 2);
double chi2_ prob = 1 - gsl_ cdf_ chisq_ P (chi2, dof);
cout ≪ "Chisq=" ≪ chi2 ≪ ", P=" ≪ chi2_ prob ≪ "." ≪ endl;
maxval = varray (ana.y[slice (range1, N, 1)]).max ();

for (int i = 0; i < ngauss; i++)
  {
    fit_ data ret;
    ret.A = A[i];
    ret.M = M[i];
    ret.sigma = var[i];
    ret.b = b;
    ret.alpha = alpha;
    // Фильтрация
    if(ret.M < in.xva[0] - ret.sigma*10)
      {
        cout ≪ "This is a failed fit, because peak center is outside of the data." ≪ end
        return fit_ data ();
      }
    if(ret.M > in.xva[in.npoint - 1] + ret.sigma*10)
      {
        cout ≪ "This is a failed fit, because peak center is outside of the data." ≪ end
```

```
      return fit_data ();
    }
  if(ret.sigma ≤ 0)
    {
      cout ≪ "This is a failed fit, because sigma < 0." ≪ endl;
      return fit_data ();
    }
  if(ret.A < peakthr)
    {
      cout ≪ "This is a failed fit, because the peak is too small." ≪ endl;
      return fit_data ();
    }
  const double hpge_sigma = hpge_fwhm (ret.M) ÷ fwhm;
  if(fabs(ret.M - 511) > 2
     ∧ (ret.sigma > hpge_sigma∗1.5
        ∨ ret.sigma < hpge_sigma÷1.5))
    {
      cout ≪ "This is a failed fit, because the peak is too wide or too thin." ≪ endl;
      return fit_data ();
    }
  gsl_multifit_covar(s→J, 0, covar);
  ret.chi2 = chi2;
  ret.dof = dof;
  ret.P = chi2_prob;
  ret.Aerr = sqrt(gsl_matrix_get(covar, i ∗ 3 + 2, i ∗ 3 + 2));
  ret.Merr = sqrt(gsl_matrix_get(covar, i ∗ 3 + 3, i ∗ 3 + 3));
  ret.Serr = sqrt(gsl_matrix_get(covar, i ∗ 3 + 4, i ∗ 3 + 4));
  fit_results.push_back(ret);
 }
gsl_matrix_free(covar);
gsl_multifit_fdfsolver_free(s);
sort(fit_results.begin(), fit_results.end(), fit_data_cmp());
int nf = fit_results.size();
int i = 0;
int best = 0;
while(i < nf)
{
 if (fabs (fit_results[i].M - in.xva[∗rough_center]) < fabs (fit_results[best].M - in.xva[∗rough_center]))
  best = i;
 i++;
}
cout ≪ endl;
return fit_results[best];
}

fit_data fit_multi_peak2(list<int>::const_iterator rough_center, const list<int> mlist,
                         int range1, int range2, input &in, analyze &ana)
{
// some checks
assert (range1 ≥ 0 ∧ range2 > 0);
```

```
assert (range1 < range2 ∧ *rough_center > range1 ∧ *rough_center < range2);
vector<int> what_to_fit;
vector<fit_data> fit_results;
list<int>::iterator mi;
double maxval = 0;
list<int>::const_iterator rr;
int ngauss = mlist.size ();
what_to_fit.resize (ngauss);
copy (mlist.begin (), mlist.end (), what_to_fit.begin ());
const int N = range2 - range1 + 1;
const int p = 4*ngauss + 2;
const double dof = N - p;
cout ≪ "Trying to fit better " ≪ ngauss ≪ " peaks with center at "
≪ in.xva[*rough_center] ≪ ", fitting range from "
≪ in.xva[range1] ≪ " to "
≪ in.xva[range2]
≪ " (" ≪ N ≪ " points)." ≪ endl;
cout ≪ "List of initial estimates: ";
for (vector<int>::iterator li = what_to_fit.begin (); li ≠ what_to_fit.end (); li++)
cout ≪ in.xva[*li] ≪ "; ";
cout ≪ endl;

assert (ngauss > 0);

if(p ≥ N)
{
  cout ≪ "Skipping fit because p >= N." ≪ endl;
  return fit_data ();
}
if (ngauss > max_multi_peak)
{
  cout ≪ "Skipping fit because Npeaks exceeds max_multi_peak ("
  ≪ ngauss ≪ ")." ≪ endl;
  return fit_data ();
}

double x_init[p];
x_init[0] = 0;                    // alpha
x_init[1] = ana.b[*rough_center]; // b
for(int i = 0; i < ngauss; i++)
{
  x_init[i*4 + 2] = ana.y[what_to_fit[i]];  // A
  x_init[i*4 + 3] = in.xva[what_to_fit[i]]; // M
  x_init[i*4 + 4] = 0.4;                    // sigma
  x_init[i*4 + 5] = ana.b[what_to_fit[i]];  // beta
}

const gsl_multifit_fdfsolver_type *T;
gsl_multifit_fdfsolver *s;
int status;
```

```
int iter = 0;
gsl_matrix *covar = gsl_matrix_alloc(p,p);
double y[N], e[N], sigma[N];
struct data d;
gsl_multifit_function_fdf f;

gsl_vector_view x = gsl_vector_view_array(x_init, p);
f.f = &mexpb_f2;
f.df = &mexpb_df2;
f.fdf = &mexpb_fdf2;
f.n = N;
f.p = p;
f.params = &d;
d.range1 = 0;
d.range2 = N;
d.sigma = sigma;
d.y = y;
d.x = e;
d.ngauss = ngauss;

for(int i = 0; i < N; i++)
{
  y[i] = in.yva[i + range1];
  e[i] = in.xva[i + range1];
  sigma[i] = sqrt(y[i]);
  if(sigma[i] == 0) sigma[i] = 1; // correction for sigma_i = 0
}

T = gsl_multifit_fdfsolver_lmsder;
s = gsl_multifit_fdfsolver_alloc(T, N, p);
gsl_multifit_fdfsolver_set(s, &f, &x.vector);
do
{
  iter++;
  status = gsl_multifit_fdfsolver_iterate(s);
  if(status)
  {
    cout << gsl_strerror (status) << endl;
    //break;
  }
  // epsabs == 0, esprel == 1e-4
  varray A (ngauss), M (ngauss), var (ngauss);
  for (int i = 0; i < ngauss; i++)
  {
    A[i] = gsl_vector_get (s->x, i*4 + 2);
    M[i] = gsl_vector_get (s->x, i*4 + 3);
    var[i] = fabs(gsl_vector_get (s->x, i*4 + 4));
    if (var[i] > 10000)
    {
      cout << "Fit failed, because sigma to big." << endl;
```

```
      return fit_data ();
    }
  }
  status = gsl_multifit_test_delta (s→dx, s→x, 0, 1e-4);
} while (status ≡ GSL_CONTINUE ∧ iter < maxiter);

if (iter ≡ maxiter)
{
  cout ≪ "This is a failed fit, because maximum steps reached." ≪ endl;
  return fit_data ();
}
double alpha = gsl_vector_get (s→x, 0);
double b = gsl_vector_get (s→x, 1);
varray A (ngauss), M (ngauss), var (ngauss), beta (ngauss);
for (int i = 0; i < ngauss; i++)
{
  A[i] = gsl_vector_get (s→x, i*4 + 2);
  M[i] = gsl_vector_get (s→x, i*4 + 3);
  var[i] = gsl_vector_get (s→x, i*4 + 4);
  beta[i] = gsl_vector_get (s→x, i*4 + 5);
}

cout ≪ "Fitting complete: alpha=" ≪ alpha;
cout ≪ " b=" ≪ b ≪ " iter=" ≪ iter ≪ endl;
cout ≪ setw (15) ≪ "A" ≪ setw (15) ≪ "M" ≪ setw (15)
≪ "sigma" ≪ setw (15) ≪ "beta" ≪ endl;
cout ≪ "============================================================" ≪ endl;
for (int i = 0; i < ngauss; i++)
cout ≪ setw (15) ≪ A[i] ≪ setw (15) ≪ M[i]
≪ setw (15) ≪ var[i] ≪ setw (15) ≪ beta[i] ≪ endl;
gsl_vector *test = gsl_vector_alloc(N);
mexpb_f2 (s→x, &d, test);
double chi2 = 0;
for (int ii = 0; ii < N; ii++)
chi2 += pow (gsl_vector_get (test, ii), 2);
double chi2_prob = 1 - gsl_cdf_chisq_P (chi2, dof);
cout ≪ "Chisq=" ≪ chi2 ≪ ", P=" ≪ scientific ≪ chi2_prob ≪ fixed ≪ "." ≪ endl;
maxval = varray (ana.y[slice (range1, N, 1)]).max ();

for (int i = 0; i < ngauss; i++)
{
  fit_data ret;
  ret.A = A[i];
  ret.M = M[i];
  ret.sigma = var[i];
  ret.b = b;
  ret.alpha = alpha;
  // Фильтрация
  if (ret.M < in.xva[0] - ret.sigma*10)
  {
```

```
    cout ≪ "This is a failed fit, because peak center is outside of the data." ≪ endl;
    return fit_data ();
  }
  if (ret.M > in.xva[in.npoint - 1] + ret.sigma∗10)
  {
    cout ≪ "This is a failed fit, because peak center is outside of the data." ≪ endl;
    return fit_data ();
  }
  if (ret.sigma ≤ 0)
  {
    cout ≪ "This is a failed fit, because sigma < 0." ≪ endl;
    return fit_data ();
  }
  if (ret.A < peakthr)
  {
    cout ≪ "This is a failed fit, because the peak is too small." ≪ endl;
    return fit_data ();
  }
  const double hpge_sigma = hpge_fwhm (ret.M) ÷ fwhm;
  if (fabs (ret.M - 511) > 2
  ∧ (ret.sigma > hpge_sigma∗1.5
  ∨ ret.sigma < hpge_sigma÷1.5))
  {
    cout ≪ "This is a failed fit, because the peak is too wide or too thin." ≪ endl;
    return fit_data ();
  }
  gsl_multifit_covar (s→J, 0, covar);
  ret.chi2 = chi2;
  ret.dof = dof;
  ret.P = chi2_prob;
  ret.Aerr = sqrt (gsl_matrix_get (covar, i ∗ 3 + 2, i ∗ 3 + 2));
  ret.Merr = sqrt (gsl_matrix_get (covar, i ∗ 3 + 3, i ∗ 3 + 3));
  ret.Serr = sqrt (gsl_matrix_get (covar, i ∗ 3 + 4, i ∗ 3 + 4));
  fit_results.push_back (ret);
}
gsl_matrix_free (covar);
gsl_multifit_fdfsolver_free (s);

sort (fit_results.begin (), fit_results.end (), fit_data_cmp ());
int nf = fit_results.size ();
int i = 0;
int best = 0;
while (i < nf)
{
  if (fabs (fit_results[i].M - in.xva[∗rough_center]) < fabs (fit_results[best].M - in.xva[∗rough_center]))
    best = i;
  i++;
}
cout ≪ endl;
return fit_results[best];
```

```
}


double mopeak_fun(double x, double A, double M, double sigma,
                  double b, double alpha)
{
  double Yi = A*exp(-(x-M)*(x-M)÷(2*sigma*sigma)) + alpha*x + b;
  return Yi;
}

double peak_int(double A, double M, double sigma)
{
  double sum = A*sigma*sqrt(2*M_PI);
  return sum;
}

void process_options (int argc, char *argv[])
{
  int i;
  char ch;
  struct option opts[] = {
    {"peakthr", required_argument, NULL, 0},
    {"pcpkthr", required_argument, NULL, 0},
    {"fit_gap", required_argument, NULL, 0},
    {"max-multi-peak", required_argument, NULL, 0},
    {"median-l", required_argument, NULL, 0},
    {"match-l", required_argument, NULL, 0},
    {"maxiter", required_argument, NULL, 0},
    {"peaks-file", required_argument, NULL, 0},
    {"no-chi2", no_argument, NULL, 'n'},
    {0, 0, 0, 0}};
  while ((ch = getopt_long(argc, argv, "", opts, &i)) ≠ -1)
    {
      if (ch ≡ '?' ∨ ch ≡ ':') exit (4);
      if (ch ≡ 'n') chi2_dont_check_p = true;
      if (ch ≡ 0)
        {
          if (strcmp (opts[i].name, "pcpkthr") ≡ 0) pcpkthr = atof (optarg);
          if (strcmp (opts[i].name, "peakthr") ≡ 0) peakthr = atof (optarg);
          if (strcmp (opts[i].name, "fit-gap") ≡ 0) fit_gap = atoi (optarg);
          if (strcmp (opts[i].name, "max-multi-peak") ≡ 0) max_multi_peak = atoi (optarg);
          if (strcmp (opts[i].name, "median-l") ≡ 0) median_l = atoi (optarg);
          if (strcmp (opts[i].name, "match-l") ≡ 0) match_l = atoi (optarg);
          if (strcmp (opts[i].name, "maxiter") ≡ 0) maxiter = atoi (optarg);
          if (strcmp (opts[i].name, "peaks-file") ≡ 0) single_peak_file = optarg;
        }
    }
  cout ≪ setw(15) ≪ left ≪ "Option" ≪ setw(8) ≪ right ≪ "Value"
       ≪ endl;
  cout ≪ "======================" ≪ endl;
```

```
  cout << setfill ('.');
  cout << left << setw (15) << "peakthr" << right << setw (8) << peakthr << endl;
  cout << left << setw (15) << "pcpkthr" <<  right << setw (8) << pcpkthr << endl;
  cout << left << setw (15) << "fit_gap" <<  right << setw (8) << fit_gap << endl;
  cout << left << setw (15) << "max_multi_peak" << right << setw (8) << max_multi_peak<< endl;
  cout << left << setw (15) << "median_l" << right << setw (8) << median_l << endl;
  cout << left << setw (15) << "match_l" << right << setw (8) << match_l << endl;
  cout << left << setw (15) << "maxiter" << right << setw (8) << maxiter << endl;
  cout << left << setw (15) << "peaks_file" << right << setw (8) << single_peak_file << endl;
  cout << left << setw (15) << "chi2_dont_check_p" << right << setw (8) << chi2_dont_check_p << end
  cout << setfill (' ') << endl;
}

double load_file (const char *f, varray &x, varray &y)
{
  varray out;
  ifstream file(f);
  list<double> xfile, yfile;
  double ch_width;
  while(¬file.eof())
    {
      char buf[256];
      file.getline(buf, 255);
      double x, y, dummy;
      if (sscanf(buf, "%lf %lf %lf", &dummy, &x, &y) ≠ 3) continue;
      xfile.push_back(x);
      yfile.push_back(y);
    }
  int npoint = xfile.size();
  x.resize(npoint);
  y.resize(npoint);
  cout << "Read " << npoint << " lines from " << f << ", ";
  // convert list to valarray
  list<double>::iterator l;
  int i = 0;
  for(l = xfile.begin(); l ≠ xfile.end(); l++)
    {
      x[i++] = *l;
      if(i>1) ch_width += x[i-1] - x[i-2];
    }
  ch_width ÷= npoint;
  i = 0;
  for(l = yfile.begin(); l ≠ yfile.end(); l++)
    {
      y[i++] = (int) *l;
//       if(*l > 0 && nonzero_begin == 0) nonzero_begin = i;
    }
  cout << "energy channel width " << setprecision(4) << ch_width << endl;
  return ch_width;
}
```

```
double median(varray &numbers)
{
  set<int> uniq;
  for(unsigned int i = 0; i < numbers.size(); i++)
    uniq.insert((int) numbers[i]);
  int pos = uniq.size() ÷ 2;
  double ret = -1;
  set<int>::iterator i2 = uniq.begin();
  for(int j = 0; j < pos; j++)
    ret = *(i2++);
  return ret;
}

void median_filter(varray& series, varray& output,int width)
{
  for(size_t i = 0; i < series.size(); i++)
  {
    varray slice(width*2 + 1);
    for(size_t j = 0; j < slice.size(); j++)
    {
      int orig_pos = i + j - width;
      if(orig_pos < 0 ∨ orig_pos ≥ series.size()) continue;
      slice[j] = series[orig_pos];
    }
    output[i] = median(slice);
  }
}

double logpoisson (int a, int n)
{
  double ret = log((double) a) * n - a - lgamma (n+1);
  return ret;
}

double poisson (int a, int n)
{
  if (n < 100)
    {
      double f = tgamma (n+1);
      double ret = exp (- (double) a) * pow ((double) a, (double) n) ÷ f;
      return ret;
    }
  else
    {
      double sigma = sqrt ((double) n);
      double ret = (1÷(sigma*sqrt(2*M_PI)))
                * exp (-pow(((double) a - (double) n), 2)÷(2*sigma*sigma));
      return ret;
    }
```

```
}

void average_filter (varray& data, varray& output, int w)
{
  assert (w % 2 ≡ 1);          // w is even
  assert (data.size() ≡ output.size());
  const int np = data.size ();
  const int gap = (w - 1) ÷ 2;
  for (int p = 0; p < np; p++)
  {
    int start = (p ≥ gap? p - gap : 0);
    int len   = (p + gap < np? w : gap + np - p);
    varray x = data[slice(start, len, 1)];
    output[p] = mean (x);
  }
}

double mean (varray &x)
{
  int np = x.size ();
  double mean = x.sum() ÷ np;
  return mean;
}

double stddev (varray &x)
{
  assert (x.size () > 1);
  double m = mean (x);
  return sqrt (pow (x - m, 2).sum() ÷ (x.size() - 1));
}

void out_ array(ostream& s, const varray& arrayx,
                            const varray& arrayy)
{
  int npoint = arrayx.size();
  for(int i = 0; i < npoint; i++)
  {
    s ≪ setw(20) ≪ arrayx[i] ≪ "  "
      ≪ setw(20) ≪ arrayy[i]
      ≪ endl;
  }
}

void smart_ diff(const varray& a1, const varray& a2,
                 varray& out)
{
  int npoint = a1.size();
  for(int i = 0; i < npoint; i++)
  {
    double aa1 = a1[i];
```

```
    if(aa1 < 0) aa1 = 0;
    double aa2 = a2[i];
    if(aa2 < 0) aa2 = 0;
    double res = aa1 - aa2;
    if(res < 0) res = 0;
    out[i] = res;
  }
}

double gauss_pdf (double x, double m, double s)
{
  double a = s * sqrt (2*M_PI);
  return exp (-pow (x-m, 2) ÷ (2*s*s)) ÷ a;
}

double gauss_cdf (double x, double m, double s)
{
  return (1 + erf ( (x-m) ÷ (s*sqrt (2.0)))) ÷ 2;
}

double nmin (double x, double lambda, double n)
{
  double pm = 1 - pow(1 - gauss_pdf (x, lambda, sqrt (lambda)), n);
  double pg = 1 - gauss_cdf (x - 1, lambda, sqrt (lambda));
  return pm * pow (pg, n - 1);
}

double nmin_int (double n, double lambda)
{
  double x;
  double sum = 0;
  for (x = lambda - 10*sqrt(lambda); x < lambda + 10*sqrt (lambda); x+=1)
    {
      sum += x * nmin (x, lambda, n);
    }
  return sum;
}

double solve (double n, double y)
{
  static map<int,double> cache;
  int code = int (n) * 10000000 + int (y);
  if (cache.count (code) ≥ 1) return cache[code];
  double a = y, b = y*20;
  double r;
  double yres;
  do
    {
      r = (a+b)÷2;
      yres = nmin_int (n, r);
```

```
    if (yres > y)
      b = r;
    else
      a = r;
    /*       printf ("%lf\n", r); */
  }
  while (fabs(y - yres) ≥ 1);
  /*  printf ("n = %lf y = %lf r = %lf\n", n, y, r); */
  cache[code] = r;
  return r;
}

valarray<double> fodec (valarray<double>& signal, double e)
{
  fftw_complex *s_in, *s_out;
  fftw_complex *r_in, *r_out;
  fftw_complex *i_in, *i_out;
  fftw_plan p_s, p_r, p_i;

  double fwhm = hpge_fwhm (e);
  double sigma = fwhm ÷ 2.35482004503095;
  double sigma1 = sigma ÷ ch_width;
  int l = round (sigma1 * 6);
  double h = round (sigma1 * 3);
  int npoint = signal.size () + l;

  s_in = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
  s_out = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
  for (int i = 0; i < npoint; i++)
    {
      s_in[i][0] = 0;
      s_in[i][1] = 0;
    }
  for (int i = 0; i < signal.size (); i++)
    s_in[i+l][0] = signal[i];
  p_s = fftw_plan_dft_1d (npoint, s_in, s_out, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute (p_s);

  r_in = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
  r_out = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
  for (int i = 0; i < npoint; i++)
    {
      r_in[i][0] = gauss_pdf ((double) i, sigma1÷2, sigma1);
      r_in[i][1] = 0;
    }
  p_r = fftw_plan_dft_1d (npoint, r_in, r_out, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute (p_r);

  i_in = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
  i_out = (fftw_complex*) fftw_malloc (sizeof(fftw_complex) * npoint);
```

```
  for (int i = 0; i < npoint; i++)
    {
      complex<double> s = complex<double> (s_out[i][0], s_out[i][1]);
      complex<double> r = complex<double> (r_out[i][0], r_out[i][1]);
      complex<double> d = s ÷ r;
      i_in[i][0] = real (d);
      i_in[i][1] = imag (d);
    }
  p_i = fftw_plan_dft_1d (npoint, i_in, i_out, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_execute (p_i);
  valarray<double> ret (signal.size ());
  for (int i = 0; i < ret.size (); i++)
    {
      complex<double> c = complex<double> (i_out[i+1][0], i_out[i+1][1]);
      ret[i] = real (c);
    }

//  fftw_destroy_plan (p);
//  fftw_free (in);
//  fftw_free (out);
  return ret;
}
```

Defines:
  average_filter, used in chunk 43.
  gauss_cdf, used in chunk 43.
  gauss_pdf, used in chunk 43.
  load_file, used in chunks 7a and 43.
  logpoisson, used in chunk 43.
  mean, used in chunks 10a and 43.
  median, used in chunk 43.
  median_filter, used in chunk 43.
  mexpb_df, used in chunks 28, 35, and 43.
  mexpb_df2, never used.
  mexpb_f, used in chunks 28, 35, 37, and 43.
  mexpb_f2, used in chunk 28.
  mexpb_fdf, used in chunks 28, 35, and 43.
  mexpb_fdf2, never used.
  mopeak_fun, used in chunks 28 and 39.
  nmin, used in chunk 43.
  nmin_int, used in chunk 43.
  out_array, never used.
  peak_int, used in chunk 43.
  poisson, used in chunk 43.
  process_options, used in chunks 7a and 43.
  smart_diff, used in chunk 10a.
  solve, used in chunks 8 and 43.
  stddev, used in chunk 7b.
Uses ch_width, dmpeak_fun2_dj, dmpeak_fun_dj, fit_gap, fwhm, hpge_fwhm, match_l, max_multi_peak, maxiter, median_l, mpeak_fun, mpeak_fun2, npoint, pcpkthr, peakthr, and valarray.

⟨* 48⟩
⟨Другие члены структуры analyze 12⟩
⟨Фитируем многогауссовые пики 16⟩
⟨Функции 8⟩
⟨Функция main 5b⟩
⟨Глобальные переменные 6a⟩
⟨Исключение дубликатов 39⟩
⟨Итерации 36⟩
⟨Константы 4⟩
⟨Настройка МНК в GSL 35⟩
⟨Обработка результатов фитирования 19⟩
⟨Определить число и начальные параметры максимумов 34⟩
⟨Отсеивание найденных пиков по стат. критерию 14b⟩
⟨Подключение заголовочных файлов 42⟩
⟨Поиск максимумов 13a⟩
⟨Прототипы функций 43⟩
⟨Проверка и обработка результатов 37⟩
⟨Структуры данных 5a⟩
⟨Выделение фона 10a⟩
⟨Загрузка массивов yva и xva из файла 7a⟩
⟨Записываем найденные пики в файл 41b⟩