

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

Константы для настройки

peakthr — порог определения пика.

pcpkthr — порог определения пика по относительной высоте над фоном.

match_l — порог определения пика в единицах σ .

fit_gap — минимальная ширина области фитирования.

max_multi_peak — максимальное число кривых Гаусса при фитировании.

median_l — ширина медианного фильтра.

maxiter — максимальное число итераций фитирования.

⟨Константы⟩≡

```
#define const static
const double peakthr      = 8;
const double pcpkthr     = .10;
const int    fit_gap      = 20;
const int    max_multi_peak = 10;
const int    median_l     = 10;
const int    match_l      = 3;
const int    maxiter      = 100;
#undef const
```

```
const double fwhm = 2.35482004503095;
```

Выполнение программы начинается с функции main. Работа программы делится на четыре этапа:

- I) загрузка входных данных из файла;
- II) выделение фона;
- III) поиск и обработка максимумов в спектре, которые в дальнейшем станут центрами для фитирования;
- IV) собственно фитирование и последующая обработка результатов
- V) запись данных

⟨Функция main⟩≡

```
int main(int argc, char *argv[])
{
    struct rlimit lim;
    getrlimit(RLIMIT_CPU, &lim);
    lim.rlim_cur = 30;
    setrlimit(RLIMIT_CPU, &lim);
    ⟨Загрузка массивов yva и xva из файла⟩
    ⟨Выделение фона⟩
    ⟨Поиск максимумов⟩
    ⟨Объединение соседних близкорасположенных максимумов⟩
    ⟨Отсеивание найденных пиков по стат. критерию⟩
    ⟨Фитируем многогауссовые пики⟩
    ⟨Обработка результатов фитирования⟩
    ⟨Записываем найденные пики в файл⟩
    cout << endl;
    return 0;
}
```

Загрузка данных

На этом этапе производится загрузка γ -спектра из файла ASCII. Имя файла передается в командной строке.

Данные загружаются в массивы `yva` и `xva`. По завершении переменная `npoint` содержит число каналов спектра, а `ch_width` — усредненную по всем точкам ширину канала.

⟨Глобальные переменные⟩≡

```
int npoint;
int nonzero_begin = 0;
valarray<double> xva;
valarray<double> yva;
double ch_width;
```

⟨Загрузка массивов yva и xva из файла⟩≡

```
if(argc != 2 && argc != 8)
{
    cout << "Usage: peakdeco [filename.dat]" << endl;
    exit(2);
}
string filename = argv[1];
ifstream file(filename.c_str());
list<double> xfile, yfile;
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);
}
npoint = xfile.size();
xva.resize(npoint);
yva.resize(npoint);

cout << "read " << setw(5) << npoint << " records from "
    << setw(10) << filename << endl;
ofstream outfile("filtered.dat");

// convert list to valarray
list<double>::iterator l;
int i = 0;
for(l = xfile.begin(); l != xfile.end(); l++)
{
    xva[i++] = *l;
    if(i>1) ch_width += xva[i-1] - xva[i-2];
}
ch_width /= npoint;
i = 0;
for(l = yfile.begin(); l != yfile.end(); l++)
{
    yva[i++] = (int) *l;
```

```

    if(*l > 0 && nonzero_begin == 0) nonzero_begin = i;
}
cout << "                energy channel width"
      << setw(10) << setprecision(4) <<  ch_width << endl;

if(argc == 8)
{
    fit_gap = atoi(argv[2]);
    cout << "fit_gap = " << fit_gap << endl;
    maxiter = atoi(argv[3]);
    cout << "maxiter = " << maxiter << endl;
    match_l = atoi(argv[4]);
    cout << "match_l = " << match_l << endl;
    peakthr = atof(argv[5]);
    cout << "peakthr = " << peakthr << endl;
    pcpkthr = atof(argv[6]);
    cout << "pcpkthr = " << pcpkthr << endl;
    median_l = atoi(argv[7]);
    cout << "median_l = " << median_l << endl;
}

```

Выделение фона

Этот этап необходим, так как спектр, снимаемый с реального детектора содержит фоновую подложку, которая затрудняет поиск и сравнение пиков по их высоте. Для выделения фона применяется медианный фильтр с шириной, примерно соответствующей полуширине одного пика.

(Функции)≡

```

double median(valarray<double> &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(valarray<double>& series, valarray<double>& output, int width)
{
    for(size_t i = 0; i < series.size(); i++)
    {
        valarray<double> 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);
    }
}

```

```

    }
}

void average_filter(valarray<double>& data, valarray<double>& 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);
        valarray<double> x = data[slice(start, len, 1)];
        output[p] = mean(x);
    }
}

double mean(valarray<double>& x)
{
    int np = x.size();
    double mean = x.sum() / np;
    return mean;
}

```

В глобальные массивы `filtered_y` и `fpeak_y` после данного этапа помещаются соответственно выделенный фон и пики, которые находятся как разность между входным спектром и фоном.

⟨Глобальные переменные⟩ +=

```
valarray<double> filtered_y, fpeak_y;
```

⟨Выделение фона⟩ ≡

```

filtered_y.resize(npoint);
median_filter(yva, filtered_y, median_l);
out_array(outfile, xva, filtered_y);
fpeak_y.resize(npoint);
smart_diff(yva, filtered_y, fpeak_y);
ofstream onlypeak("onlypeak.dat");
out_array(onlypeak, xva, fpeak_y);

```

Поиск максимумов

Для успешного фитирования методом МНК в процедуру подгонки должны передаваться некоторые начальные значения параметров, близкие к реальным. На этом этапе работы программы производится поиск в спектре максимумов, которые, вероятнее всего, являются пиками. Эта задача выполняется в три подхода. Сначала по несглаженным данным производится поиск максимумов (из-за шумов это дает несколько тысяч точек), затем

соседние объединяются для учета ширины реальных пиков (около 3 кэВ), а потом оставшиеся максимумы просеиваются по различным критериям.

⟨Глобальные переменные⟩ + ≡

```
list<int> stat_maximum;
```

⟨Поиск максимумов⟩ ≡

```
valarray<double> s_data(npoint);
average_filter(yva, s_data, 3);
list<int> maximum;
for(int i = 1; i < npoint - 1; i++)
    if(s_data[i] > s_data[i-1] && s_data[i] > s_data[i+1])
        maximum.push_back(i);
```

⟨Объединение соседних близкорасположенных максимумов⟩ ≡

```
list<int> far_maximum;
list<int>::iterator mi;
for(mi = maximum.begin(); mi != maximum.end(); )
{
    list<int> values;
    int prev, next;
    prev = next = *mi;
    while(next - prev <= 3 && ++mi != maximum.end())
    {
        values.push_back(next);
        prev = next;
        next = *mi;
    }
    list<int>::iterator vi;
    double max = 0;
    int max_pos = 0;
    for(vi = values.begin(); vi != values.end(); vi++)
        if(fpeak_y[*vi] > max)
        {
            max = fpeak_y[*vi];
            max_pos = *vi;
        }
    far_maximum.push_back(max_pos);
}
```

Отсеиваем по критерию 1

$$y_{peak_i} < l\sigma_i,$$

где $\sigma_i = y_{фон_i}$ — ст. отклонение в данной точке спектра.

Критерий 2

$$y_i > (1 + P)y_{фон_i},$$

где P — порог `psrpkthr`.

Критерий 3

$$y_i > Q,$$

где Q — порог peakthr.

```

<Отсевывание найденных пиков по стат. критерию>≡
list<int>::iterator fmi;
ofstream smfile("statmax.dat");
for(fmi = far_maximum.begin(); fmi != far_maximum.end(); fmi++)
{
    if(fpeak_y[*fmi] < sqrt(filtered_y[*fmi]) * match_l) continue;
    if(yva[*fmi] < (1 + pcpkthr) * filtered_y[*fmi]) continue;
    if(yva[*fmi] < peakthr) continue;
    stat_maximum.push_back(*fmi);
    smfile << setw(20) << *fmi << setw(20) << xva[*fmi] << endl;
}

```

Подгонка параметров пиков по МНК

Найденные на предыдущем этапе приближенные положения максимумов будут использоваться для двух вещей.

Во-первых положение максимума дает начальную оценку центра функции Гаусса, амплитуду и фон в этой точке мы также знаем.

Во-вторых, зная расстояние до соседних пиков, мы можем фитировать не одну, а несколько функций Гаусса сразу. Итак.

<Глобальные переменные>+≡

```
list<fit_data> fitted_peaks;
```

<Фитируем многогауссовые пики>≡

```

list<int>::const_iterator si;
for(si = stat_maximum.begin(); si != stat_maximum.end(); si++)
{
    list<int>::const_iterator ti = si;
    int count = 0;
    int range1 = *si, range2 = *si;
    while(ti != stat_maximum.begin() && *si - *ti < fit_gap)
    {
        range1 = *ti;
        ti--;
        count++;
    }
    ti = si;
    while(ti != stat_maximum.end() && *ti - *si < fit_gap)
    {
        range2 = *ti;
        ti++;
        count++;
    }
    if(count > max_multi_peak)
    {
        cerr << "!!! skipping fit because Npeaks exceeds max_multi_peak" << endl;
        continue;
    }
    //if(range1 == *si)
    range1 -= fit_gap/2;
    //if(range2 == *si)

```

```

range2 += fit_gap/2;
if(range1 < nonzero_begin)
{
    range1 = nonzero_begin + 3;
    if(range1 >= *si) continue;
}
cout << " -----*-----*-----*-----*-----*-----*-----*"
    << endl << "                Doing "
    << xva[*si] << " " << xva[range1] << " " << xva[range2] << endl;
fit_multi_peak(*si, range1, range2);
cout << endl << endl;
}

```

Обработка результатов фитирования

После процедуры фитирования в списке `fitted_peaks` у нас находятся все результаты фитирования: большое число структур `fit_data`, в которых содержатся параметры каждого профитированного пика. Однако в этом списке много дубликатов одного и того же пика, которые необходимо удалить (возможно, выбрав лучший). Окончательный список найденных пиков помещается в переменную `peaks`.

(Глобальные переменные)+≡

```
list<fit_data> peaks;
```

(Обработка результатов фитирования)≡

```

vector<fit_data> fv(fitted_peaks.begin(), fitted_peaks.end());
sort(fv.begin(), fv.end(), fit_data_cmp());
int nf = fv.size();
i = 0;
while(i < nf)
{
    int k = i;
    double pM, psigma, M, sigma;
    pM = fv[i].M;
    psigma = fv[i].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 = i;
    double best_chi2 = DBL_MAX;
    for(int l = i; l < k; l++)
    {
        if(fv[l].chi2 < best_chi2)
        {

```

```

        best_chi2 = fv[l].chi2;
        best_pos = l;
    }
}
peaks.push_back(fv[best_pos]);
i = k++;
}

```

Детали фитирования

Сама процедура фитирования основана на `gsl_multifit_nlin` из библиотеки GSL. См. Gnu Scientific Library Reference Manual, стр. 391. Мой код списан оттуда.

Будем фитировать 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:

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

⟨Функции⟩ +=

```

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);
    valarray<double> 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);
    }
    valarray<double> X(x, ngauss);
    valarray<double> val(ngauss), val1(ngauss);
    val = pow(X - M, 2);
    val /= (sigma*sigma*2);
    val = exp(-val);
    val *= A;
    return val.sum() + alpha * x + b;
}

```


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

(Функции) \equiv

```
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);
    valarray<double> 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)/(sigmaj*sigmaj*sigmaj);

        ret = dfds;
        break;
    default: cerr << "oh no, what a bug!" << endl;
    }
    return ret;
}
```

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

⟨Структуры данных⟩≡

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

Подгоняем функции под интерфейсы GSL.

⟨Функции⟩+≡

```
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;
}

```

Фитирование осуществляется в функции `fit_multi_peak`. Она получает три параметра: номер канала интересующего максимума и левую и правую границу диапазона, в котором будет производится фитирование. Число гауссиан в фитировании равно числу максимумов в этом диапазоне, и их начальные параметры берутся из уже известных данных.

Эта функция возвращает структуру `fit_data`, которая содержит информацию о ближайшем к `rough_center` найденном пике.

⟨Структуры данных⟩+≡

```

struct fit_data
{
    double A, M, sigma, b, alpha, chi2, err, Merr, Aerr, Serr;
};

```

⟨Функции⟩+≡

```

void fit_multi_peak(int rough_center, int range1, int range2)
{
    // some checks
    assert(range1 >= 0);
    assert(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;

    ⟨Определить число и начальные параметры максимумов⟩
    ⟨Настройка МНК в GSL⟩
    ⟨Итерации⟩
    ⟨Проверка и обработка результатов⟩
    ⟨Исключение дубликатов⟩
}

```

Инициализация алгоритма. Первым делом мы определяем, сколько именно пиков будет фитироваться на данном интервале. Число пиков и их параметры определяются из числа максимумов, сохраненных в `stat_maximum`.

⟨Определить число и начальные параметры максимумов⟩+≡

```

for(mi = stat_maximum.begin(); mi != stat_maximum.end(); mi++)
{
    if(*mi >= range1 && *mi <= range2)
        what_to_fit.push_back(*mi);
}

```

```

const int N = range2 - range1;
const int ngauss = what_to_fit.size();
const int p = 3*ngauss + 2;

```

```

assert(ngauss > 0);
if(p >= N)
{
    cerr << "skipping fit because p >= N" << endl;
    return;
}
double x_init[p];
x_init[0] = 0; // alpha
x_init[1] = filtered_y[rough_center]; // b
for(int i = 0; i < ngauss; i++)
{
    x_init[i*3 + 2] = fpeak_y[what_to_fit[i]]; // A
    x_init[i*3 + 3] = xva[what_to_fit[i]]; // M
    x_init[i*3 + 4] = 0.4; // sigma
}

⟨Настройка MHK в GSL⟩≡
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] = yva[i + range1];
    e[i] = 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);

```

Теперь собственно цикл фитирования. Кроме максимального числа итераций здесь есть еще и минимальное — 5. Зачем это сделано — секрет. Выход из итераций по критерию `gsl_multifit_test_delta`.

⟨Итерации⟩≡

```
do
{
    iter++;
    status = gsl_multifit_fdfsolver_iterate(s);
    if(status) break;
    // epsabs == 0, esprel == 1e-4
    status = gsl_multifit_test_delta(s->dx, s->x, 0, 1e-4);
} while (iter < 5 || (status == GSL_CONTINUE && iter < maxiter));
```

Так. Итерации завершены и теперь необходимо обработать результат фитирования. Если число итераций, которые потребовались для работы алгоритма, равно максимальному, то этот результат следует отбросить, потому что фит не сошелся. Кроме того здесь есть еще и проверки на правдоподобность параметров пика.

⟨Проверка и обработка результатов⟩≡

```
if(iter == maxiter) return;
double alpha = gsl_vector_get(s->x, 0);
double b = gsl_vector_get(s->x, 1);
valarray<double> 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 result alpha =" << setw(15) << alpha;
cout << " b =" << setw(15) << b << " iter" << setw(5) << 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;

for(int i = 0; i < ngauss; i++)
{
    if(fpeak_y[what_to_fit[i]] > maxval)
        maxval = fpeak_y[what_to_fit[i]];
}

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 < 0) continue;
    if(ret.sigma == 0) continue;
    if(ret.A < peakthr) continue;
    if(ret.sigma > 2 || ret.sigma < 0.1) continue;
    if(ret.A > 100*maxval) continue;
    double input_diff = DBL_MAX;
    for(int l = 0; l < ngauss; l++)
    {
        double d = fabs(ret.M - xva[what_to_fit[l]]);
        if(d < input_diff)
            input_diff = d;
    }
    if(input_diff > ch_width * 5) continue;
    if(ret.b + ret.alpha * xva[rough_center] >
        filtered_y[rough_center] * 10) continue;
    if(iter == 1) continue;
    gsl_multifit_covar(s->J, 0, covar);
    double chi = gsl_blas_dnorm2(s->f);
    double dof = N - p;
    double chisq = chi*chi/dof;
    ret.chi2 = chisq;
    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_fdsolver_free(s);

<Структуры данных>+=
struct fit_data_cmp
{
    bool operator()(const fit_data& f1, const fit_data& f2) {return f1.M < f2.M;}
};

<Исключение дубликатов>=
sort(fit_results.begin(), fit_results.end(), fit_data_cmp());
int nf = fit_results.size();
int i = 0;
while(i < nf)
{
    int k = i;
    double pM, psigma, M, sigma;
    pM = fit_results[i].M;
    psigma = fit_results[i].sigma;
    const double fwhms = 2.35482004503095 / 2;

    while(k < nf && (

```

```

        M = fit_results[k].M,
        sigma = fit_results[k].sigma,
        M - pM < psigma*fwhms + sigma*fwhms))
    {
        psigma = sigma;
        pM = M;
        k++;
    }

    double final_sigma = 0;
    double final_M = 0;
    double max_A = 0;
    double sum = 0, wsum = 0;
    double Merr = 0, Aerr = 0, Serr = 0;
    double final_A = 0;
    for (int j = i; j < k; j++)
    {
        double tmp_sigma = fit_results[j].sigma;
        // if(tmp_sigma
        Merr += fit_results[j].Merr;
        Aerr += fit_results[j].Aerr;
        Serr += fit_results[j].Serr;
        if(max_A < fit_results[j].A)
        {
            max_A = fit_results[j].A;
            final_sigma = tmp_sigma;
        }
    }
    for (int j = i; j < k; j++)
    {
        sum += fit_results[j].A/max_A * fit_results[j].M;
        wsum += fit_results[j].A/max_A;
    }

    if(i != k - 1)
    {
        cout << " Combining peaks: ";
        for (int j = i; j < k; j++)
            cout << setw(7) << setprecision(2) << fixed << fit_results[j].M
                << " ";
        cout << endl;
    }
    final_M = sum/wsum;
    for (int j = i; j < k; j++)
    {
        final_A += mopeak_fun(final_M, fit_results[j].A, fit_results[j].M,
                               fit_results[j].sigma, 0, 0);
    }

    fit_data final;
    final.M = final_M;
    final.sigma = final_sigma;
    final.A = final_A;

```

```

    final.Merr = Merr;
    final.Aerr = Aerr;
    final.Serr = Serr;
    final.chi2 = fit_results[i].chi2;

    fitted_peaks.push_back(final);
    i = k++;
}
cout << endl;

```

Вывод данных

На выходе мы должны представить следующую основную информацию: центр пика и его площадь.

Фитируемая функция это практически функция Гаусса, но с заменой множителя перед экспонентой, который зависит от σ на масштабную константу A . После фитирования пиков производится вычисление площади, для чего необходим интеграл от этой функции. Он выглядит следующим образом:

$$I = \int_{-inf}^{+inf} A e^{-\frac{(x-M)^2}{2\sigma^2}} dx = A\sigma\sqrt{2\pi}$$

(Функции) +=

```

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;
}

```

(Записываем найденные пики в файл) +=

```

{
    ofstream peakfile("peaks.dat");
    list<fit_data>::iterator si;

    peakfile << " "
              << setw(8) << "M" << " "
              << setw(9) << "A" << " "
              << setw(8) << "sigma" << " "
              << setw(8) << "M err" << " "
              << setw(7) << "Square" << " "
              << setw(10) << "chi2" << endl;

    peakfile << " =====<< endl;

```



```

for(si = peaks.begin(); si != peaks.end(); si++)
{
    fit_data f = *si;
    // peak sum
    double sum = peak_int(f.A, f.M, f.sigma);
    peakfile << fixed << " "
        << setw(8) << setprecision(3) << f.M << " "
        << setw(9) << setprecision(2) << f.A << " "
        << setw(8) << setprecision(5) << f.sigma << " "
        << setw(8) << setprecision(4) << f.Merr << " "
        << setw(7) << setprecision(2) << sum << " "
        << setw(10) << setprecision(2) << scientific << f.chi2 << endl;

}
peakfile << endl << endl << fixed;
for(si = peaks.begin(); si != peaks.end(); si++)
{
    fit_data f = *si;
    for(double d = f.M - 4*f.sigma; d < f.M + 4*f.sigma; d += .1)
        peakfile << setw(20) << d << setw(20)
            << mopeak_fun(d, f.A, f.M, f.sigma, f.b, f.alpha) << endl;
    peakfile << endl << endl;
}
<Выдача в HTML-таблицу>
}

```

```

<Выдача в HTML-таблицу>≡
peakfile.close();
peakfile.open("webout.html");
peakfile << endl << endl
    << "<table><thead><tr>"
    << "<th>E, keV</th>"
    << "<th>&Delta;E, keV</th>"
    << "<th>A</th>"
    << "<th>&Delta;A</th>"
    << "<th>&Gamma;, keV</th>"
    << "<th>&Delta;&Gamma;, keV</th>"
    << "<th>&alpha;</th>"
    << "<th>b</th>"
    << "<th>Integral</th>"
    << "<th>&Delta;I</th>"
    << "<th>&chi;2</th></tr></thead>" << endl;

peakfile << endl << endl
    << "<tfoot><tr>"
    << "<th>E, keV</th>"
    << "<th>&Delta;E, keV</th>"
    << "<th>A</th>"
    << "<th>&Delta;A</th>"
    << "<th>&Gamma;, keV</th>"
    << "<th>&Delta;&Gamma;, keV</th>"

```

```

        << "<th>&alpha;</th>"
        << "<th>b</th>"
        << "<th>Integral</th>"
        << "<th>&Delta;I</th>"
        << "<th>&chi;2</th></tr></tfoot><tbody>" << endl;

int count = 0;
for(si = peaks.begin(); si != peaks.end(); si++)
{
    fit_data f = *si;
    // peak sum
    double sum = peak_int(f.A, f.M, f.sigma);
    double sum_err = sqrt(pow(sqrt(2*M_PI)*f.A*f.Serr, 2)
        + pow(sqrt(2*M_PI)*f.Aerr*f.sigma, 2));
    peakfile << fixed
        << "<tr class=\"\" << (count++ % 2 == 0? "odd" : "even") << "\">"
        << "<td>" << setw(8) << setprecision(3) << f.M << "</td>"
        << "<td>" << setw(8) << setprecision(3) << f.Merr << "</td>"
        << "<td>" << setw(9) << setprecision(2) << f.A << "</td>"
        << "<td>" << setw(9) << setprecision(2) << f.Aerr << "</td>"
        << "<td>" << setw(8) << setprecision(5) << f.sigma*fwhm << "</td>"
        << "<td>" << setw(8) << setprecision(5) << f.Serr*fwhm << "</td>"
        << "<td>" << setw(7) << setprecision(2) << f.alpha << "</td>"
        << "<td>" << setw(7) << setprecision(2) << f.b << "</td>"
        << "<td>" << setw(7) << setprecision(2) << sum << "</td>"
        << "<td>" << setw(7) << setprecision(2) << sum_err << "</td>"
        << "<td>" << setw(10) << setprecision(2) << scientific << f.chi2
        << "</td></tr>" << endl;
}

// peakfile << endl << endl
// << "</tbody><tr>"
// << "<th>E, keV</th>"
// << "<th>&Delta;E, keV</th>"
// << "<th>A</th>"
// << "<th>&Delta;A</th>"
// << "<th>&Gamma;, keV</th>"
// << "<th>&Delta;&Gamma;, keV</th>"
// << "<th>&alpha;</th>"
// << "<th>b</th>"
// << "<th>Integral</th>"
// << "<th>&Delta;I</th>"
// << "<th>&chi;2</th></tr>" << endl;
peakfile << "</tbody></table>" << endl;

```

Дальше идет всякая сервисная ерунда.

⟨Подключение заголовочных файлов⟩≡

```
#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<gsl/gsl_vector.h>
#include<gsl/gsl_blas.h>
#include<gsl/gsl_multifit_nlin.h>
```

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

```
using namespace std;
```

⟨Прототипы функций⟩≡

```
struct fit_data;
void median_filter(valarray<double>& series, valarray<double>& output, int w);
void average_filter(valarray<double>& series, valarray<double>& output, int w);
double median(valarray<int> &numbers);
double mean(valarray<double> &data);
void out_array(ostream& s, const valarray<double>& arrayx,
               const valarray<double>& arrayy);
void smart_diff(const valarray<double>& a1, const valarray<double>& a2,
               valarray<double>& 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);

void fit_multi_peak(int rough_center, int range1, int range2);
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);
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);
double mopeak_fun(double x, double A, double M, double sigma,
                  double b, double alpha);
```

⟨⟩*≡

⟨Подключение заголовочных файлов⟩

⟨Константы⟩

⟨Прототипы функций⟩
⟨Глобальные переменные⟩
⟨Структуры данных⟩
⟨Функции⟩
⟨Функция main⟩

```
void out_array(ostream& s, const valarray<double>& arrayx,
               const valarray<double>& 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 valarray<double>& a1, const valarray<double>& a2,
                valarray<double>& 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;
    }
}
```