

- 本資料の著作権、文責は著者に帰属し、所属機関を代表したり、機関の意見を表明するものではありません。
- 学校、研究機関の教育、研究目的であれば自由に使用することができます。報告なく改変、再配布可能です。  
ただし、明記されている引用先の著作権に配慮してください。
- ただし使用者は誤りや誤字を報告する義務があります。



# ROOT講習会第4回資料

## TFI, TGraph, Fitting

Keita Mizukoshi (Tohoku Univ. RCNS)

# 本日のお題

- ・前回までで、ヒストグラムが描けるようになった。
- ・本日のお題
  - ・関数を表現するクラス、 TF1
  - ・データへのモデルの当てはめ、 Fitting
  - ・グラフを表現するクラス、 TGraph、 TGraphErrors
- ・実際にコードを書きながらやっていきましょう。

# TF1

- (数学的な意味での)関数を表現するクラス
- 名前、関数の定義、定義域の最小・最大をコンストラクタで指定
  - 名前はユニークであった方がいい
  - 関数の定義は文字列で与える
    - 変数の名前は  $x$
    - C++で書くように式をかける
    - 補足：本当はTFormula。
- auto 修飾されているので, `func1` の型は後ろから推測される `TF1*` になる
- ヒストグラムと同じように, `Draw()` メソッドを呼ぶとウィンドウが出てくる。

```
// ----- //
// others/math_function.C
// to explain a usage of TF1
// ----- //

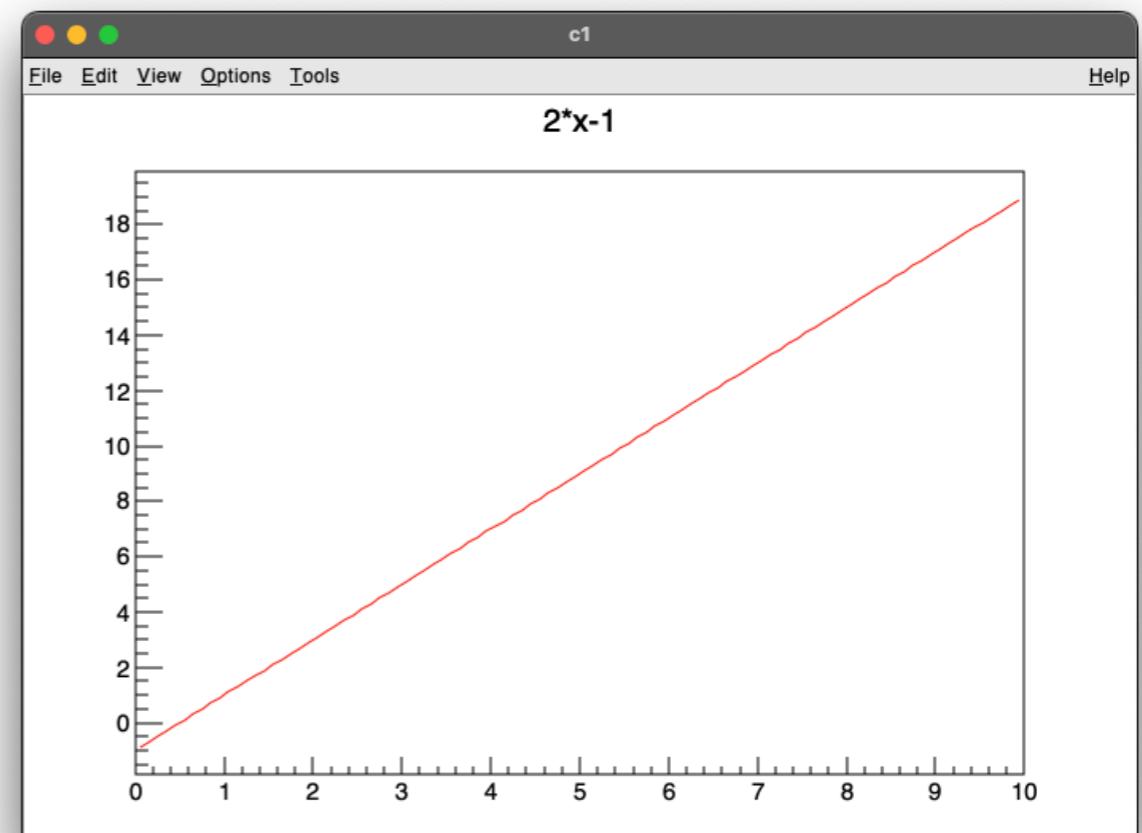
int math_function(){

  const double range_min = 0.;
  const double range_max = 10.;

  auto func1 = new TF1("func1", // function name
                      "2*x-1", // function format
                      range_min, range_max);

  func1->Draw();

  return 0;
}
```



# TF1での関数の定義

- ・ パラメータを定義する
  - ・ 大括弧で[0], [1], ...と順に定義すると、関数の中で値をいじることができるパラメータとなる
  - ・ SetParameter(<番号>, <値>) 関数で値を変えることができる
  - ・ この後のFittingで使う
- ・ 一部定義済の関数, gausやexpoなどは、関数のパラメータの最初の番号を指定
  - ・ 補足: gausは3つのパラメータのある関数なので, gaus(0) とすると, [0], [1], [2]が勝手に定義される
  - ・ SetParameters メソッドで, 一緒にパラメータの値を入れることができる
  - ・ TMathで定義済の関数も利用可能

```

// -----
// others/math_function.C
// to show function definition
// -----
// -----
// int math_functions() {

  const double range_min = 0.;
  const double range_max = 10.;

  auto func1 = new TF1("func1",
                      "[2]*x**2 + [1]*x + [0]",
                      range_min, range_max);
  func1->SetParameter(0, 2.); // for [0]
  func1->SetParameter(1, 3.); // for [1]
  func1->SetParameter(2, 1.02); // for [2]

  auto func2 = new TF1("func2",
                      "gaus(0) + expo(3)",
                      range_min, range_max);
  func2->SetParameters(50, 3, 1, 1, 1.02);

  auto func3 = new TF1("func3",
                      "20*TMath::Sin(x) + 30",
                      range_min, range_max);

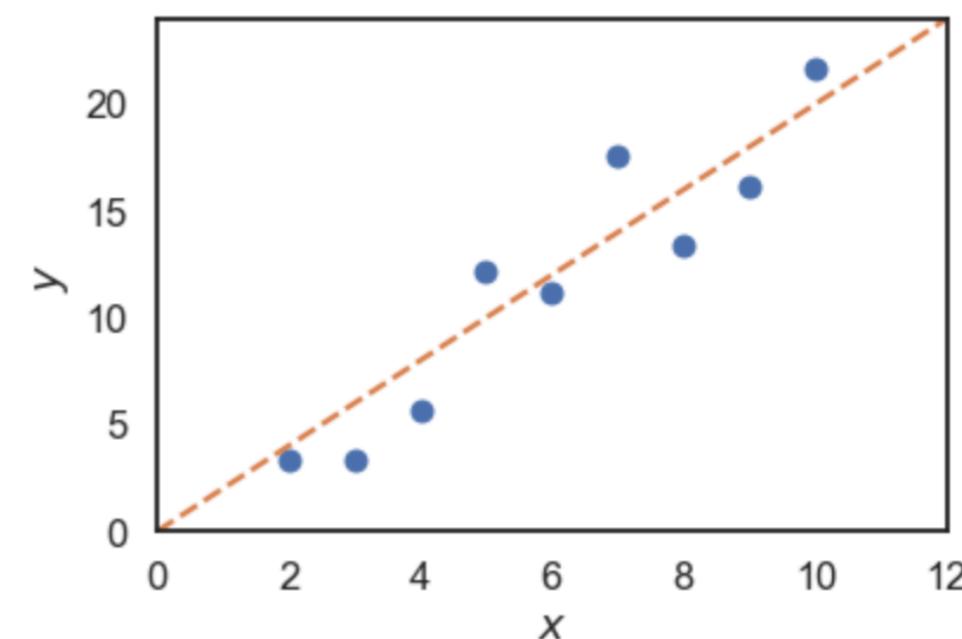
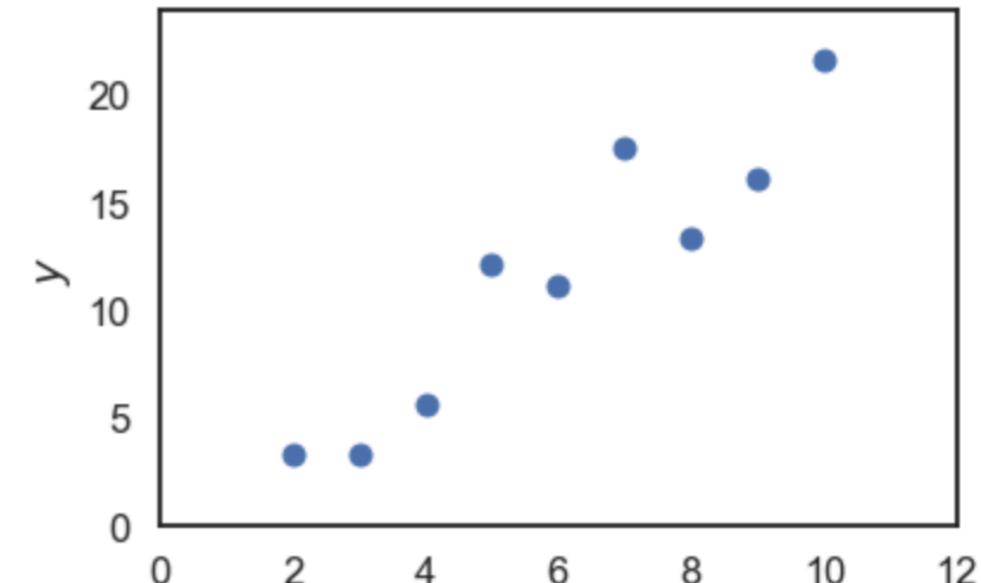
  func1->Draw();
  func2->SetLineColor(kOrange);
  func2->Draw("same");
  func3->SetLineColor(kBlue);
  func3->Draw("same");

  return 0;
}

```

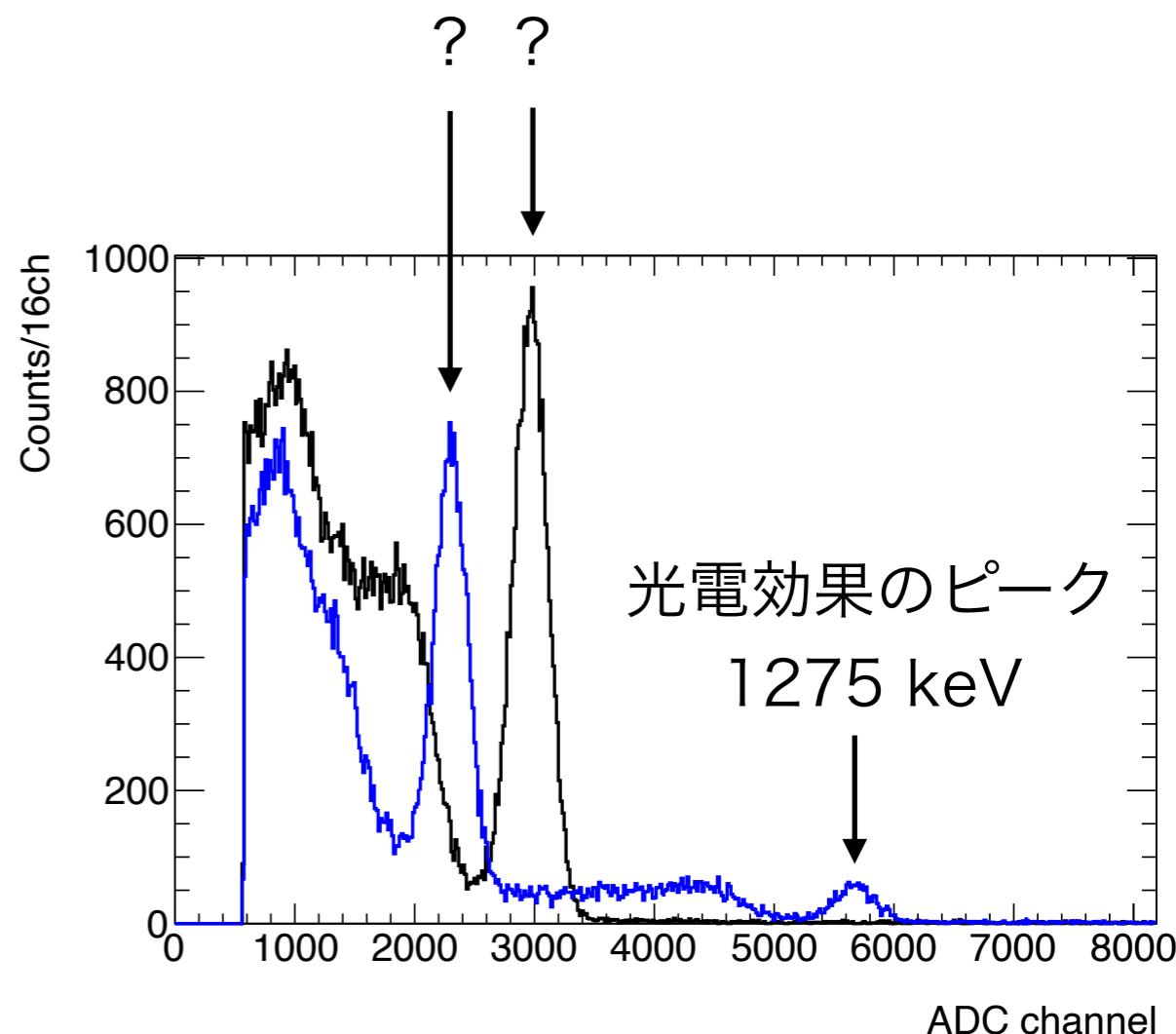
# Fittingとは

- [暫定的な定義]
- データは "真の振る舞い" の一部
- 背景の仕組みを考える
- データを一番よく説明できる線を引く
- 測定条件  $x_i$  それぞれに対して, 値  $y_i$  を得た
- 一般化した  $y = f(x)$  を求める
- なんとなく直線に見えるので, 一番あう直線  $y = f(x) = a_0 + a_1x$  を探す.
- 定規を当てる→もう少し数学的にやる



# シンチレーターの実験再訪

- ヒストグラムの重ね書きの演習で用いた図をもう一度良く見てみる。
- ADCはエネルギーのスケールに対応している
- 青のヒストグラム, Na-22線源のデータの5800付近にピークを持つ山は, 1275 keVのガンマ線の全吸収ピークであると考えられる
- 中心極限定理により, 正規分布に近づくはず



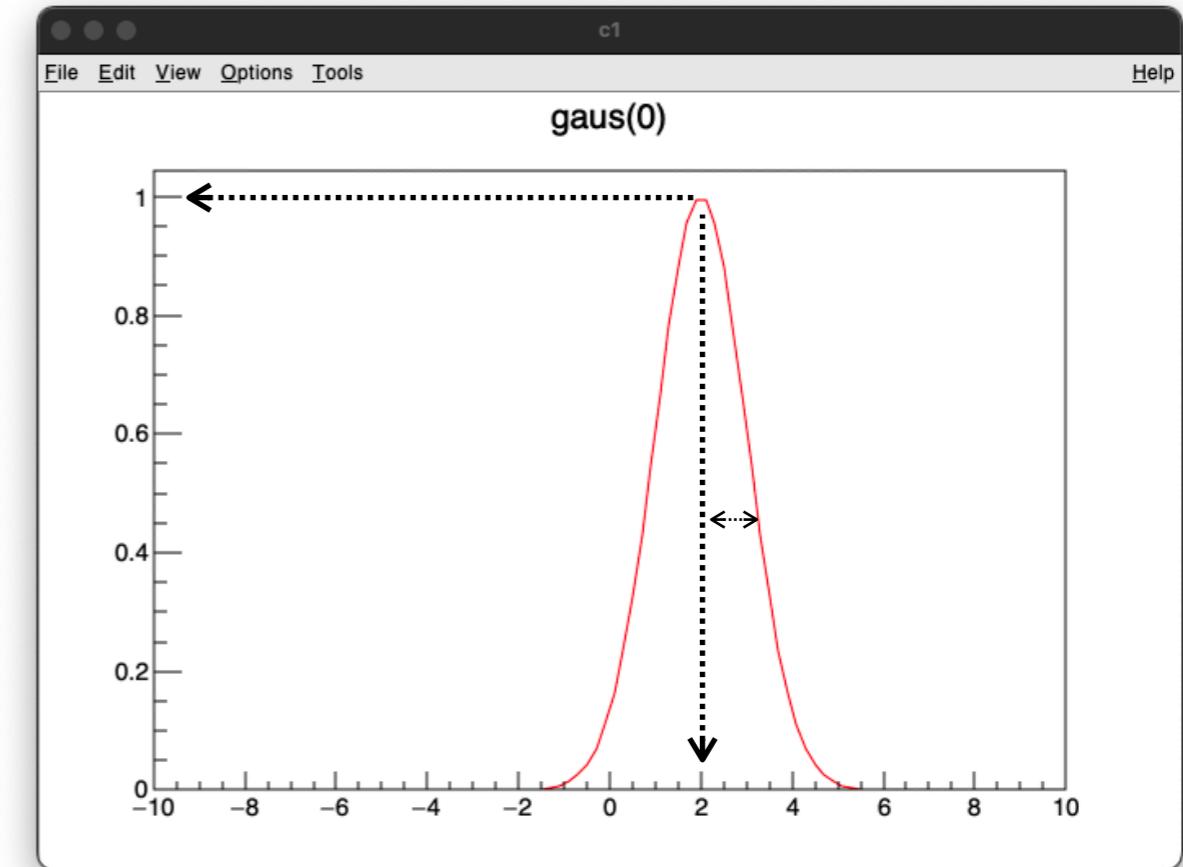
第3回で描いた図. 黒:Cs-137, 青:Na-22線源

# 正規分布

- 表式

$$N(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

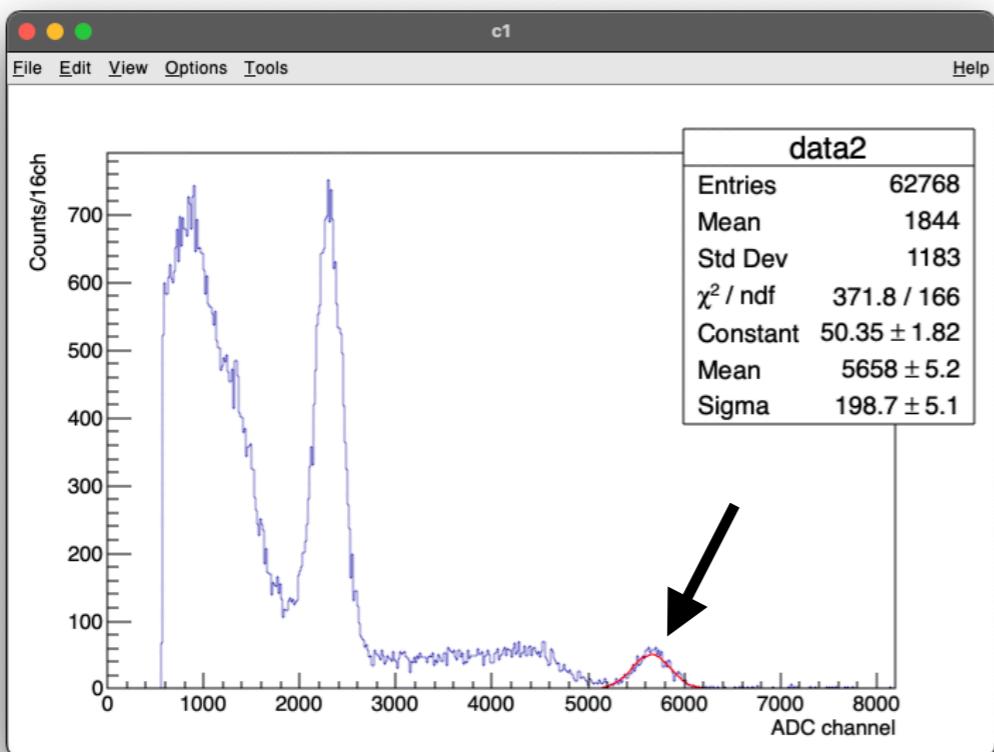
- $\mu$  平均値 → ピークの山のある  $x$  の位置
- $\sigma^2$  分散 ( $\sigma$  は測定量と同じ次元を持つ) → 山の広がりに対応
- これに加えて, ROOTではY軸方向へのスケール Const. を含めた関数として定義されている.
- ありとあらゆる所に出てくる.



TF1で "gaus(0)" で描いた正規分布.  
 パラメータは,  
 [0] Const. → 1  
 [1] Mean → 2  
 [2] Sigma → 1  
 をとりあえず入れて描いてみた.

# Fit() 関数

- 前に作成したmake\_hist() 関数を流用し、ヒストグラムを作成するところから始める。
- モデル当てはめを行うための関数、TF1を準備する
- hist->Fit(<TF1の名前>, "", "", 最小値, 最大値); とやるとFitができる。



赤線でFitしたモデルが描かれている。

Fitの情報のボックスはメニューバーから Options→Fit Parametersと押すと出てくる。

```

// -----
// fitting_gaus.C -- macro
// Author: K. Mizukoshi
// Date : Jun. 18 2025
// ----- //

TH1D* make_hist(TString filename = "data1") {

    //gROOT->SetStyle("ATLAS");
    const int MCACh = 8192;
    const double HistMin = 0.;
    const double HistMax = 8192.;

    TH1D* hist = new TH1D(filename, filename,
                         MCACh, HistMin-0.5, HistMax-0.5);

    const TString data_dir = "../data/";
    double BufferValue;
    ifstream ifs(data_dir + filename + ".txt");
    while(ifs >> BufferValue) {
        hist->Fill(BufferValue);
    }

    hist->Rebin(16);
    hist->SetTitle(";ADC channel;Counts/16ch");
    //hist->Draw();
    return hist;
}

int fitting_gaus(){

    TH1D* hist2 = make_hist("data2");
    hist2->Draw();

    const double range_min = 5000.;
    const double range_max = 8000.;
    auto func = new TF1("func", "gaus(0)", range_min, range_max);
    // Set initial parameter values
    func->SetParameter(0, 100);
    func->SetParameter(1, 6000);
    func->SetParameter(2, 500);

    hist2->Fit("func", "", "", range_min, range_max);

    return 0;
}

```

# Fittingの部分の詳説

- ・ パラメータのあるTF1を作成し、Fit()を呼ぶとデータに合うようにパラメータを調整してくれる
- ・ Fit()の第1引数はFitに使うTF1の名前 (C++の名前ではなく、中に文字列で与えた方)かTF1のポインタ
- ・ Fit()の第2、第3引数はFitting、Drawのオプション
- ・ Fit()の第4、第5引数はFitの範囲
- ・ Fit()を呼ぶ前にSetParameter()しているが、これはある程度正しい値を事前に入れておくため。あまりにも外れているとFitに失敗する
- ・ func->SetParLimits(<id>, 最小, 最大); でパラメータの取りうる範囲を制限することもできる
- ・ Fit結果がターミナルに出力されるはず。

```

int fitting_gaus() {
    TH1D* hist2 = make_hist("data2");
    hist2->Draw();

    const double range_min = 5000.;
    const double range_max = 8000.;
    auto func = new TF1("func", "gaus(0)",
                        range_min, range_max);
    // Set initial parameter values
    func->SetParameter(0, 100);
    func->SetParameter(1, 6000);
    func->SetParameter(2, 500);

    hist2->Fit("func", "", "",
               range_min, range_max);
    return 0;
}
  
```

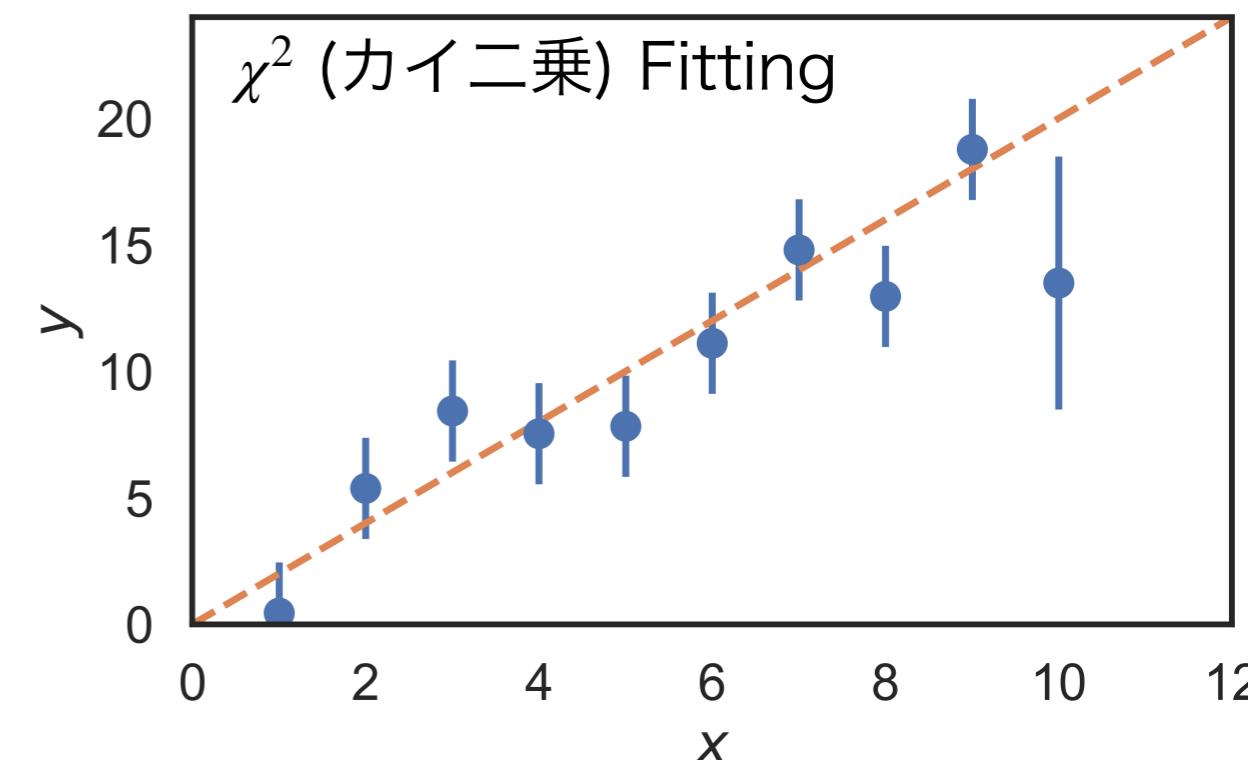
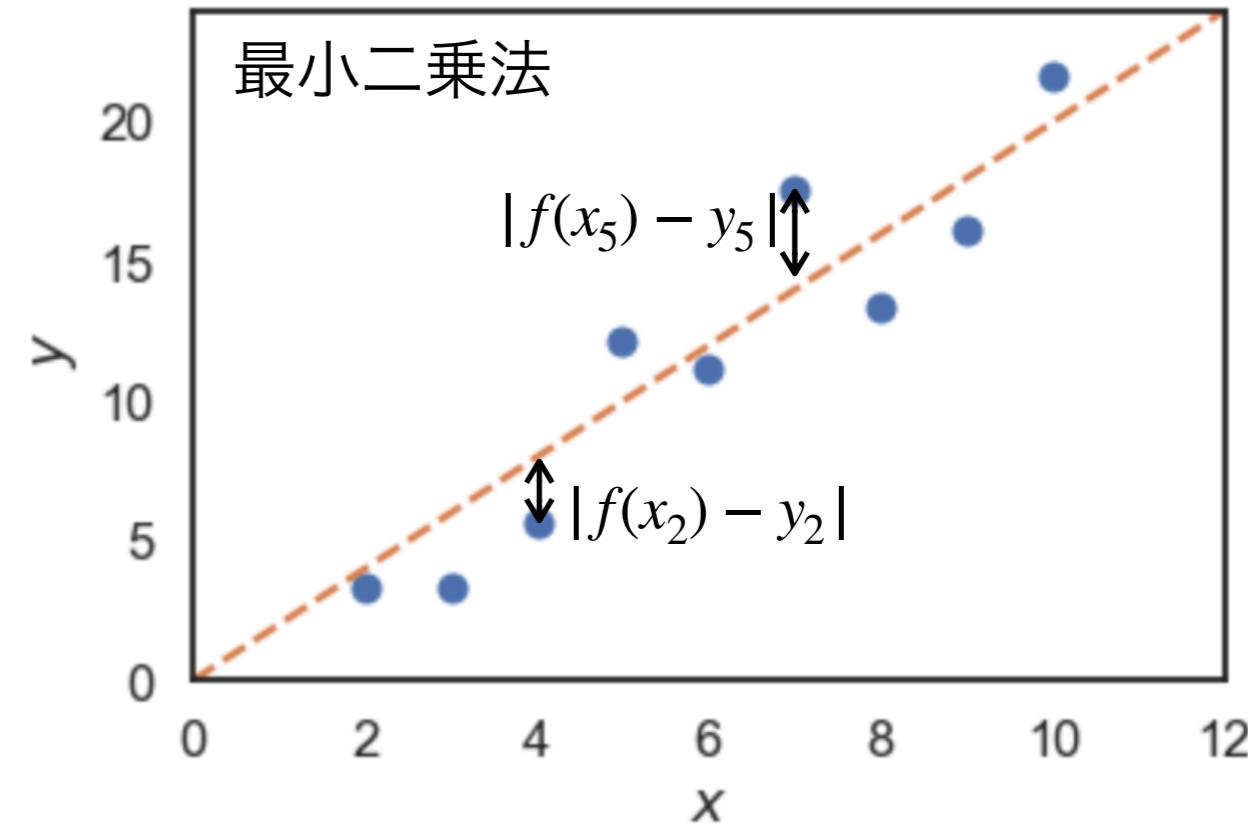
```

Minimizer is Minuit2 / Migrad
Chi2 = 371.806
NDf = 166
Edm = 9.22382e-07
NCalls = 118
Constant = 50.3495 +/- 1.81536
Mean = 5657.53 +/- 5.19466
Sigma = 198.667 +/- 5.14164
(limited)
  
```

ターミナルに出てくるFit結果の表示例

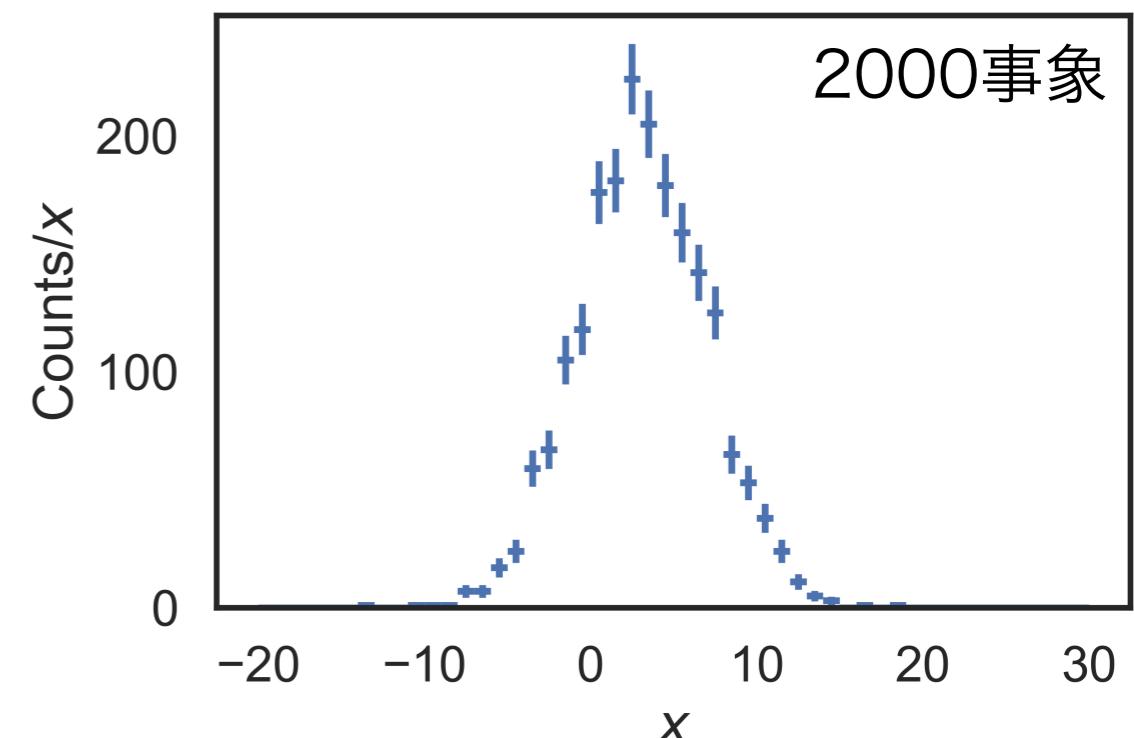
# Fittingが行っていること

- 「データへの関数の当てはめ」は具体的に何をやっているのか?
- 最小二乗法
  - モデルとデータ点の差の二乗の和が最小になるようにパラメータを決める
  - 特定のケースでは解析的に解ける
- $\chi^2$ (カイ二乗) Fitting
  - 最小二乗法ではデータのエラーを無視している
  - モデルとデータの差を、データのエラーの単位で測れば良い
- $$\chi^2(\mathbf{a}) = \sum_i \left( \frac{f(x_i) - y_i}{\sigma_i} \right)^2$$
- この値(:=コスト関数)を最小化するパラメータの組を見つける→Fitting
  - 関数の最小値を得るパラメータを得る問題→最適化問題(数理計画問題)



# ヒストグラムの $\chi^2$ Fitting

- ヒストグラムでは、各Binが各点に対応する。
- 各Binに対して、以下のように取り扱う
  - $x_i$  : Bin中心 ※
  - $y_i$  : Binに入る計数
  - $\sigma_i$  :  $\sqrt{y_i}$  二項分布の極限から、計数 $N$ に対し  
 $\sqrt{N}$ のエラー ( $N$ が大きい時)



- 例えば、Fit関数（モデル関数）として

$$N'(x) = \frac{C}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

などを用いる。

- $C$ は高さ方向のパラメータ。 $\propto$ 事象数

- 最小化を目指す関数は、 $\chi^2(\mu, \sigma, C) = \sum_i \left( \frac{N'(x_i) - y_i}{\sigma_i} \right)^2$

ヒストグラムの例

y方向には $\sqrt{N}$ のエラーがつく。  
 x方向の棒は単にBinの範囲を示す

※ ここではわかりやすくBin中心  
 としているが、一般には積分で定義し、  
 Binのとりかたによらないようにする

# 結果の取得

- Fit()の第2引数に"S"を指定して, Fit()の返値をfit\_resultに入れた
- この型はTFitResultPtr\*だけど, 面倒なのでautoで受けておく.
- パラメータの値取得
  - fit\_result->Parameter(0);
- パラメータのエラー取得
  - fit\_result->ParError(0);
- $\chi^2$ や自由度
  - fit\_result->Chi2();
  - fit\_result->Ndf();
- TF1もFit結果の一部情報を持っている.
  - func->GetParameter(0);
  - func->GetParError(0);
  - func->GetChisquare();
  - func->GetNDF();

```
int fitting_gaus(){

    TH1D* hist2 = make_hist("data2");
    hist2->Draw();

    const double range_min = 5000.;
    const double range_max = 8000.;
    auto func = new TF1("func", "gaus(0)", range_min, range_max);
    // Set initial parameter values
    func->SetParameter(0, 100);
    func->SetParameter(1, 6000);
    func->SetParameter(2, 500);

    auto fit_result = hist2->Fit("func", "S", "", range_min, range_max);

    return 0;
}
```

```
TF1* fitting_gaus(){

    TH1D* hist2 = make_hist("data2");
    hist2->Draw();

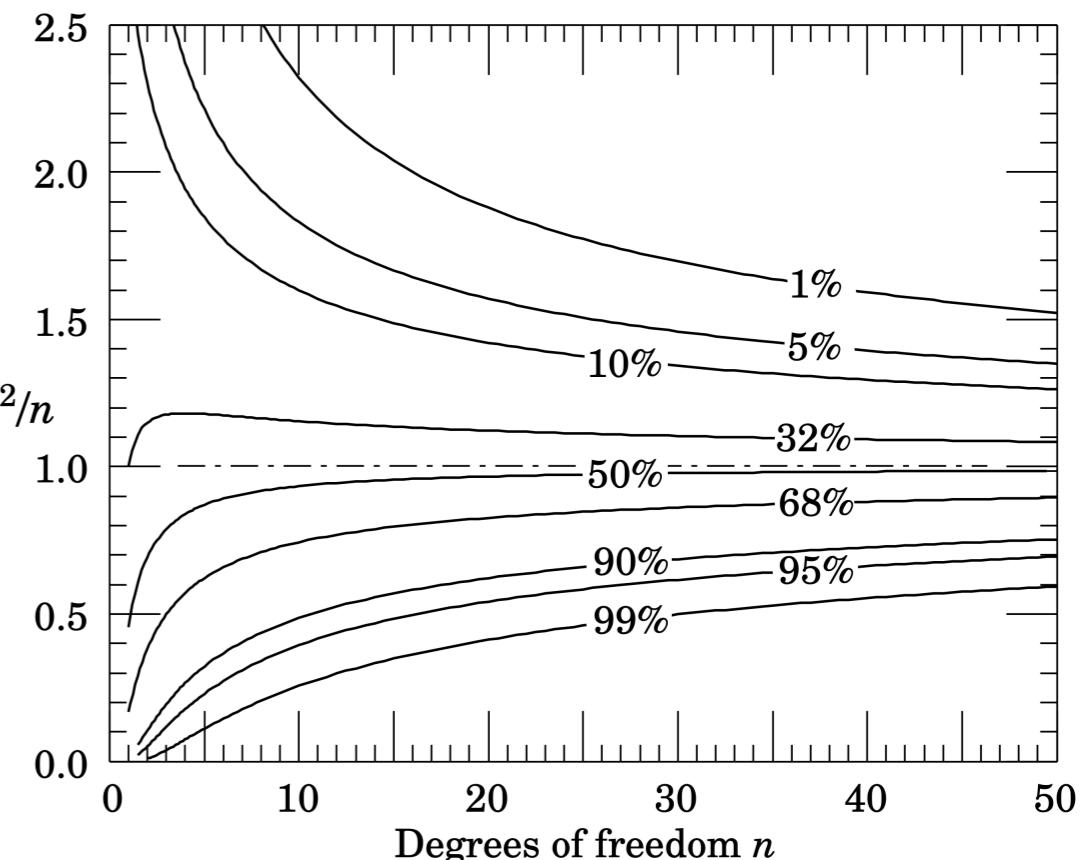
    const double range_min = 5000.;
    const double range_max = 8000.;
    auto func = new TF1("func", "gaus(0)", range_min, range_max);
    // Set initial parameter values
    func->SetParameter(0, 100);
    func->SetParameter(1, 6000);
    func->SetParameter(2, 200);

    hist2->Fit("func", "", "", range_min, range_max);

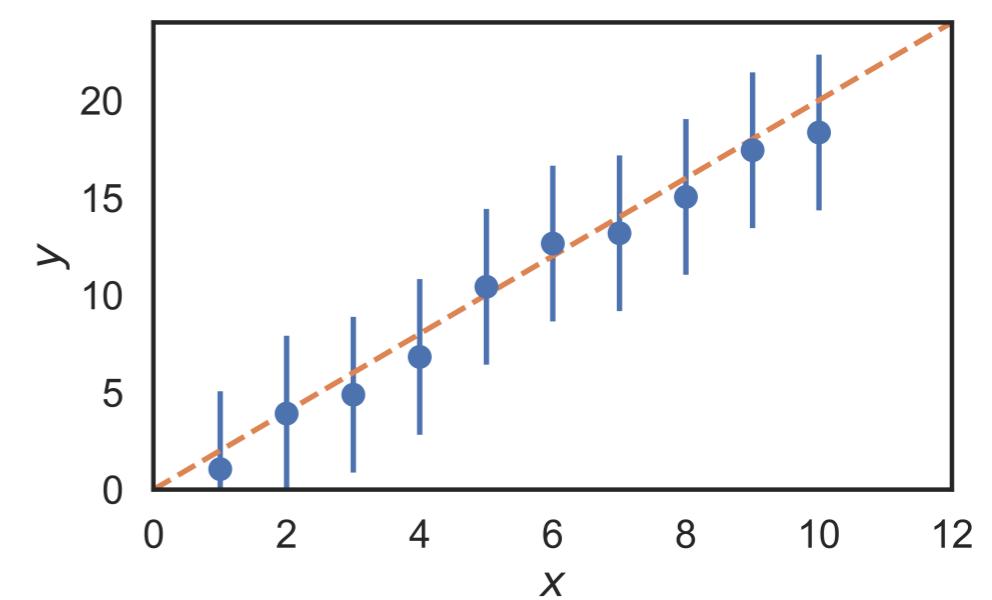
    return func;
}
```

# Fit結果の評価

- $\chi^2$ を使ってFitの適切さを確認する.
- 自由度 $n$  (number of degrees of freedom) :  
[データ点の数] - [Fitパラメータの数]
- p値 : モデル(Fit)が正しいと仮定し, 観測値が  $\chi^2/n$  それ以上に外れる余事象の確率
- $\chi^2/n$  : reduced chi-square と呼ぶ
- nが大きい時,  $\chi^2/n$ が1に近ければ,  
**妥当なFittingだと言える**
- $\chi^2/n$ が大きい → 全然あってない
  - 誤差が正規分布だというのは仮定.
  - $\chi^2/n \sim 5$ とかでもFit 자체が即座に誤りとい  
うわけではないが, 理由は考えたい
- $\chi^2/n$ が小さい → Fitがあいすぎ



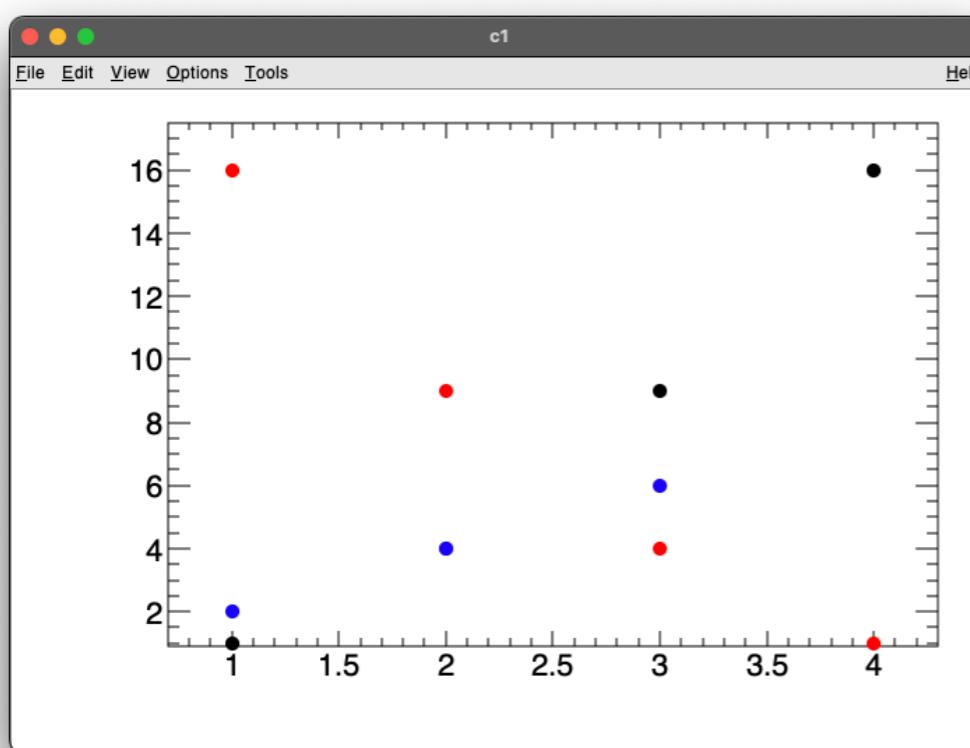
$\chi^2$ とp値の関連 (PDG2022より)



Fitがあいすぎている例

# TGraph

- グラフを表現するクラス
  - (x, y)の組みなどの散布データ
- 作成方法
  - Case 1 → 配列で定義
  - Case 2 → std::vectorで定義
  - Case 3 → とりあえず作ってから、1点ずつ入れていく
- 描画機能は貧弱。最初のグラフは"A"をオプションに入れて、軸も書くようにする必要がある。



```

// -----
// draw_graph.C
// initial example to draw
// TGraph
// -----
int draw_graph() {
  gROOT->SetStyle("ATLAS");

  // Case 1
  int n1 = 4;
  double x1[] = {1, 2, 3, 4};
  double y1[] = {1, 4, 9, 16};
  TGraph* graph1 = new TGraph(n1, x1, y1);

  // Case 2
  std::vector<double> x2 = {1, 2, 3, 4};
  std::vector<double> y2 = {16, 9, 4, 1};
  TGraph* graph2 = new TGraph(x2.size(),
                               x2.data(),
                               y2.data());

  // Case 3
  TGraph* graph3 = new TGraph();
  for(int i=0; i<4; ++i){
    graph3->AddPoint(i, i*2);
  }

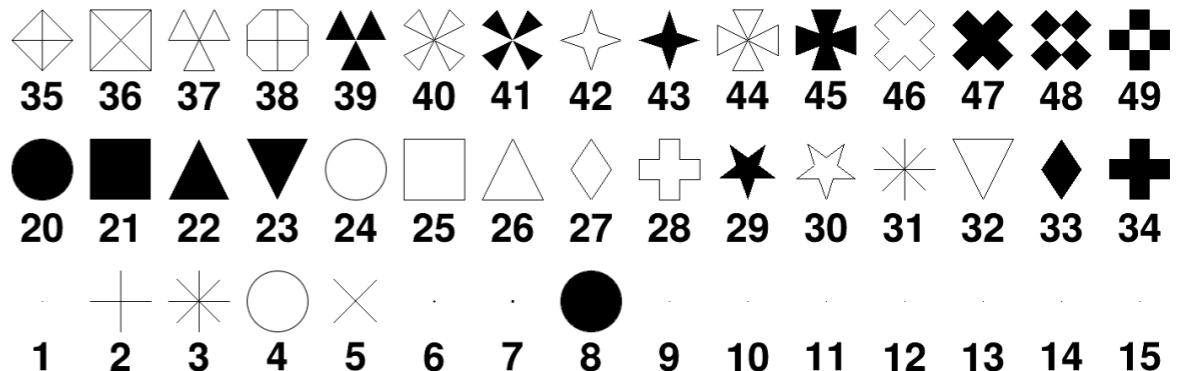
  graph1->Draw("AP");
  graph2->SetMarkerColor(kRed);
  graph2->Draw("P");
  graph3->SetMarkerColor(kBlue);
  graph3->Draw("P");

  return 0;
}

```

# Graphの見た目の微調整

- graph->SetMarkerStyle()
  - ・マーカーのスタイルを変える
  - ・適当に使って見やすくする
  - ・十字はエラーバーと誤解されかねない  
のでやめておく
- graph->SetMarkerSize()
  - ・サイズの変更
- graph->SetMarkerColor()
  - ・マーカーの色を変える.  
SetMarkerColor(  
TColor::GetColor("#ff6600"));とかで,  
好き放題できるはず
- graphもDraw("same")としても, 全てが収  
まるような自動調節がされないので, 最初  
にダミーのgraphを書いて, Marker style  
1, size 0, color kWhiteなどと指定して,  
見えないようにしておくと良い.



+	*	×	○	□	△	◊	+	☆	▽	◊	□	▽	◊	⊕	*	◊	✖	✖
104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	
+	*	×	○	□	△	◊	+	☆	▽	◊	□	▽	◊	⊕	*	◊	✖	✖
86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	
+	*	×	○	□	△	◊	+	☆	▽	◊	□	▽	◊	⊕	*	◊	✖	✖
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	
+	*	×	○	□	△	◊	+	☆	▽	◊	□	▽	◊	⊕	*	◊	✖	✖
50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	



# 正規分布とエラー

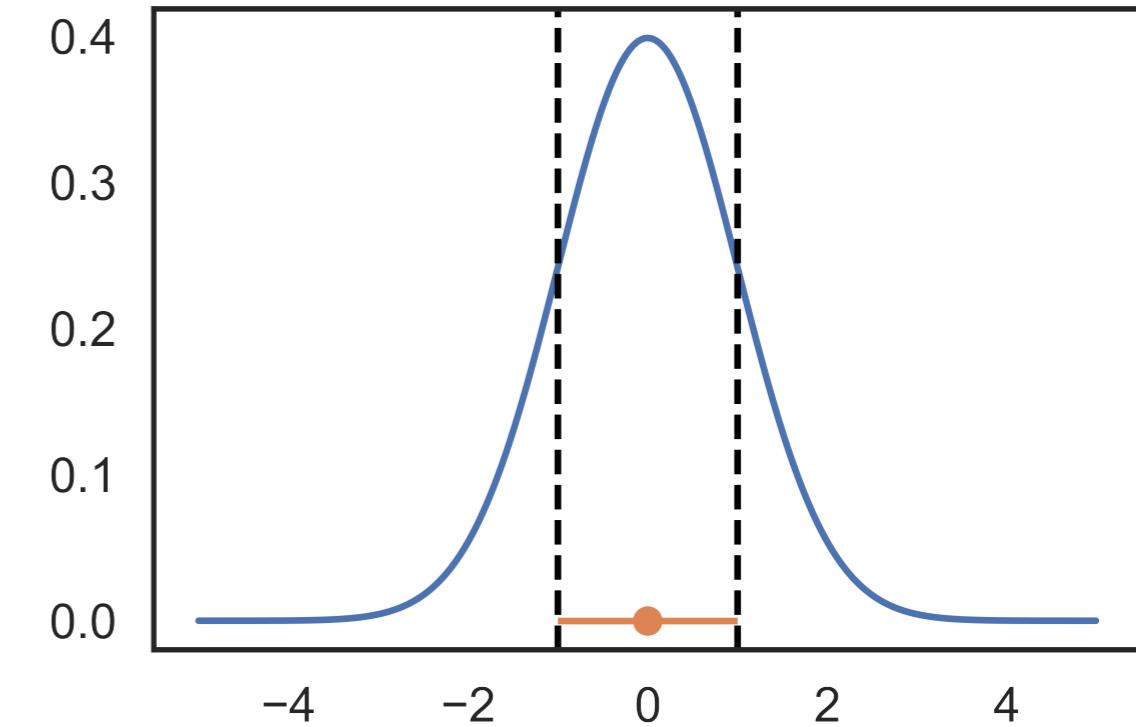
- 一般に測定値は誤差 (error)を含み, 広がりを持つ
- 統計誤差 (statistical error)
  - 背後に正規分布 を仮定する

$$N(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

- $\mu$  平均値
- $\sigma^2$  分散 ( $\sigma$ は測定量と同じ次元を持つ)

- 測定の結果, 青色の正規分布が得られた時,  
点とエラーバーで  $\mu$ ,  $\sigma$  を代表として表示する.

- 両側  $1\sigma$  範囲は, 正規分布**全体の68%**
- 点とエラーバーが表示されている時,  
**真の値が必ずその中に入っているという意味では  
ない**
- 新たに測定すると, 68%でエラーバーの範囲に,  
32%でその外に値があることが期待される.



$\sigma$ (両側)	割合
1	<b>0.6827</b>
2	0.9545
3	0.9973
1.64	0.9
2.58	0.99
3.29	0.999

正規分布の両側  $\sigma$  の範囲の割合

# TGraphErrors

- 誤差付きのTGraph→TGraphErrorsを使う
- 配列やVectorで定義する方法だと、TGraphのコンストラクタにErrorを追加するだけ
- TGraphErrorsは残念ながらAddPoint()で一氣に入れることができない。
- SetPoint(<index>, <x>, <y>);とSetPointError(<index>, <x>, <y>); を併用する。
- indexは飛ばすと怒られるので、graph->GetN(); などで調べる。
- 対称ではないエラーを表現するTGraphAsymmErrorsもある。

```

// -----
// draw_graph_errors.C
// initial example to draw
// TGraphErrors
// -----
//



int draw_graph_errors(){
  gROOT->SetStyle("ATLAS");

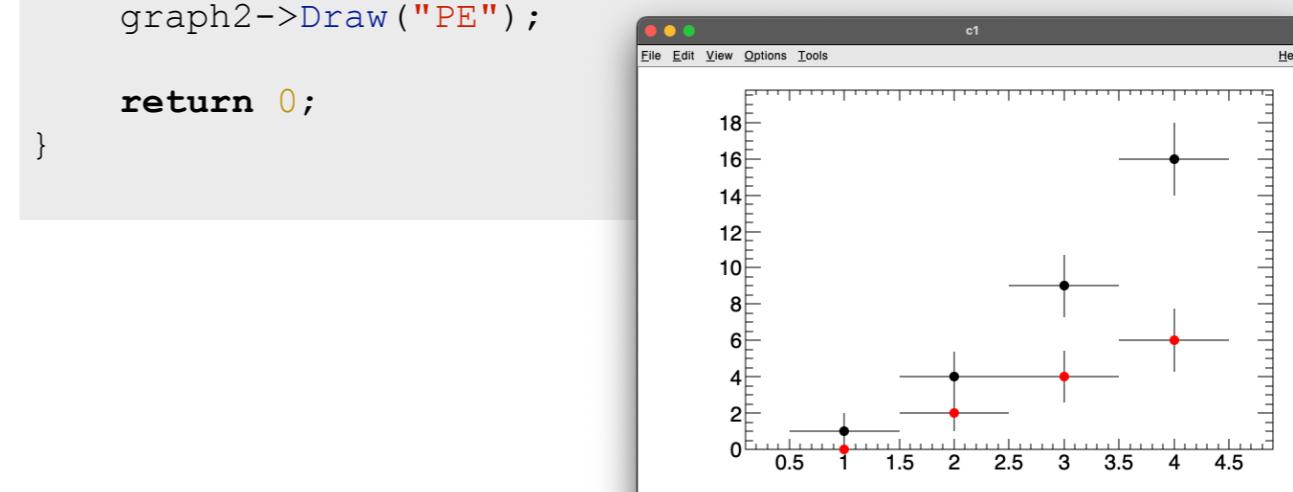
  std::vector<double> x = {1, 2, 3, 4};
  std::vector<double> y = {1, 4, 9, 16};
  std::vector<double> x_err = {0.5, 0.5, 0.5, 0.5};
  std::vector<double> y_err = {1, 1.4, 1.7, 2};
  auto graph1 = new TGraphErrors(x.size(),
                                 x.data(),
                                 y.data(),
                                 x_err.data(),
                                 y_err.data());

  auto graph2 = new TGraphErrors();
  for(int i=0; i<4; ++i){
    graph2->SetPoint(i, i+1, i*2);
    graph2->SetPointError(i, 0.5, TMath::Sqrt(i));
  }

  graph1->Draw("APE");
  graph2->SetMarkerColor(kRed);
  graph2->Draw("PE");

  return 0;
}

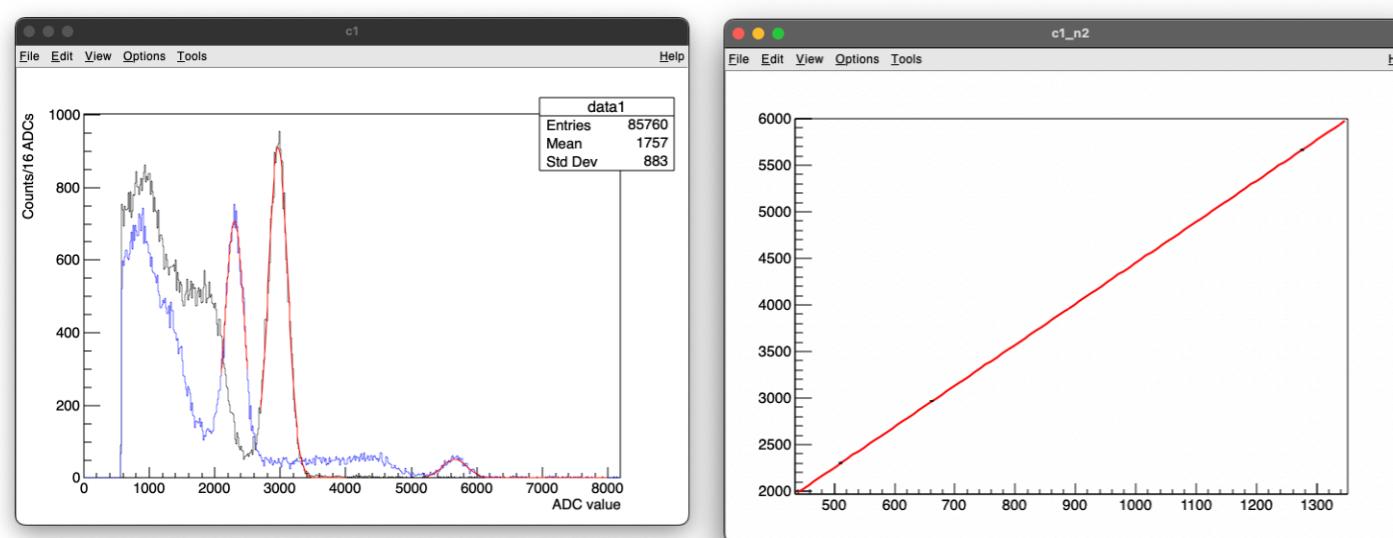
```



# 演習

- Scintillatorの演習で重ね書きした, Cs-137線源 (data1)とNa-22線源 (data2)のスペクトルに見られる, 3つの光電ピーク (511, 662, 1275 keV)を, 範囲を適当に制限して正規分布でFitする. ピークの中心値とエラーを求める.
- x軸はエネルギーの文献値、y軸は得られたADC値とエラーを示す、TGraphErrorsを作成し、プロットする.
- 実は、TGraphErrorに対してもFit()ができる.
- ADC値とエネルギーの関係式を求める.
- つまり、実に一般的なエネルギーキャリブレーションをしよう

# 演習の解答例



- Graphの点が小さくて見えづらいけど、できている。
- たまたま初期値を入れなくてもなんとかなった。
- Fit()をする部分だけ関数に切り出すのも良いが、初期値や範囲などの微調整をするために、あえて1つの関数内でやっている
- Fitのオプションをいくつか指定した(下を参照)。
- できたらFitがあつてあるか評価してみる

## 主要なFit Option

- L : カイ2乗の代わりにLog likelihoodを使う
- I : Bin中心の値を使う代わりに関数の積分値を使う
- E : Minosのエラー推定を使う(大抵よくなる)
- R : TF1関数の値域を使用する
- W : 空でないBinの重みを1にする(エラーバー無視)
- O : 描画しない

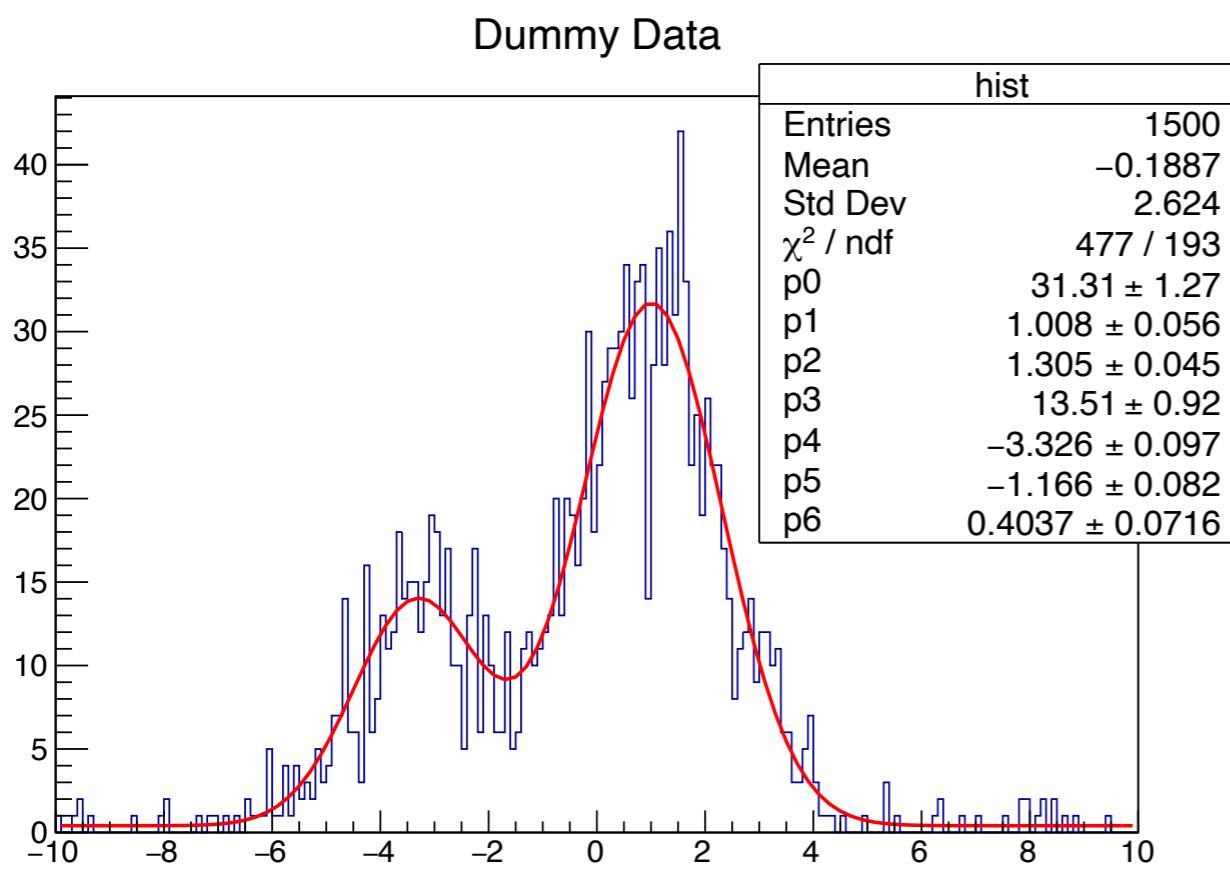
```
TH1D* make_hist(TString filename = "data1") {  
  
    const int nbins = 8192;  
    const double bin_min = 0., bin_max = 8192.;  
    TH1D* hist = new TH1D(filename, filename,  
                         nbins, bin_min-0.5, bin_max-0.5);  
  
    const TString data_dir = "../data/";  
    double BufferValue;  
    ifstream ifs(data_dir + filename + ".txt");  
    while(ifs >> BufferValue){  
        hist->Fill(BufferValue);  
    }  
  
    hist->Rebin(16);  
    hist->SetTitle(";ADC value;Counts/16 ADCs");  
    return hist;  
}  
  
void fit_all(){  
  
    auto C1 = new TCanvas();  
    auto hist1 = make_hist("data1"); // Cs-137  
    auto hist2 = make_hist("data2"); // Na-22  
    hist1->SetLineColor(kBlack);  
    hist1->Draw();  
    hist2->SetLineColor(kBlue);  
    hist2->Draw("same");  
  
    auto f1 = new TF1("f_cs137", "gaus(0)", 2700, 4000);  
    auto f2 = new TF1("f_na22_1", "gaus(0)", 2100, 2500);  
    auto f3 = new TF1("f_na22_h", "gaus(0)", 5200, 8000);  
    hist1->Fit(f1, "R0", "");  
    hist2->Fit(f2, "R0", "");  
    hist2->Fit(f3, "R0", "");  
    f1->Draw("same");  
    f2->Draw("same");  
    f3->Draw("same");  
  
    auto C2 = new TCanvas();  
    auto graph = new TGraphErrors();  
    graph->SetPoint(0, 662, f1->GetParameter(1));  
    graph->SetPointError(0, 0., f1->GetParError(1));  
    graph->SetPoint(1, 511, f2->GetParameter(1));  
    graph->SetPointError(1, 0., f2->GetParError(1));  
    graph->SetPoint(2, 1275, f3->GetParameter(1));  
    graph->SetPointError(2, 0., f3->GetParError(1));  
    graph->Draw("APE");  
  
    auto f_lin = new TF1("f_lin", "pol1");  
    graph->Fit(f_lin);  
}
```

# Likelihood Fit

- $\chi^2$ (カイ二乗) Fittingは、値の入っていないBinを無視する。
- でも、本来、Binの値がゼロ、という情報があるはず。
- この場合はLikelihood Fit (最尤推定法) を用いる
- 数学的な詳細はBackupに載せるが、以下の際にLikelihood Fitを行う。
  - 各Binのカウント数が少ない (およそ <10 events / binくらい)
    - 例えば、例で、Rebin(16)しないならlikelihoodの方が良い
  - ゼロbinを含む
  - 統計の精度について議論したい
  - 指導教員やレフェリーが要求してきた
  - こちらを使いたい時は、オプションに "L" を指定するだけ。
  - $\chi^2$  fitの方が計算は安定している傾向がある。適宜使い分ける。

# Likelihood fittingの例

- Fit optionに "L" を追加すると Likelihood を使う
- 2つの正規分布は関数形からは区別できない ← 初期値で区別



```

1 // ----- //
2 // fit_example1.C
3 // Example of fitting with ROOT
4 // Usage: root -l fit_example1.C
5 //
6 // Keita Mizukoshi (JAXA)
7 // 2023 Sep. 28
8 // -----
9 int fit_example1(){
10
11    // Generate dummy data histogram
12    auto hist = new TH1D("hist", "Dummy Data", 200, -10, 10);
13    TRandom r;
14    const int N_GAUS1 = 1000;
15    const int N_GAUS2 = 400;
16    const int N_UNIFORM = 100;
17    for(int i=0; i<N_GAUS1; ++i){
18        hist->Fill(r.Gaus(1.1, 1.3));
19    }
20    for(int i=0; i<N_GAUS2; ++i){
21        hist->Fill(r.Gaus(-3.3, 1.2));
22    }
23    for(int i=0; i<N_UNIFORM; ++i){
24        hist->Fill(r.Uniform(-10, 10));
25    }
26    hist->Draw();
27
28
29    // Prepare Fit function
30    auto fitf = new TF1("fitf", "gaus(0)+gaus(3)+pol0(6)");
31    fitf->SetParameter(0, 100); // gaus0 - Const
32    fitf->SetParameter(1, 1); // gaus0 - mu
33    fitf->SetParameter(2, 1); // gaus0 - sigma
34    fitf->SetParameter(3, 80); // gaus1 - Const
35    fitf->SetParameter(4, -3); // gaus1 - mu
36    fitf->SetParameter(5, 2); // gaus1 - sigma
37    fitf->SetParameter(6, 10); // pol0
38
39    hist->Fit("fitf", "LIE" );
40    gStyle->SetOptFit(1);
41
42
43    return 0;
44 }

```

# パフォーマンスの向上

- Fitなどは比較的計算量が多めのタスク.
- コンパイルして高速に動かすいくつかのテクニックがある.
- コマンドラインで, root fit\_all.C+ と, +をつけるとコンパイルされて, 実行時間の高速化が期待できる.
  - ただし, 色々とincludeする必要があることに注意.
- あるいは, 本物のC++のコードとしてコンパイルする.
  - 最初に呼ばれる関数名はint main() になる
  - g++ fit\_all.C \$(root-config --cflags --glbs)
  - ./a.out

# 系統誤差

- 実験の誤り (機器の較正不足, 理論の誤り, 個人の癖など)
- 統計誤差は手続き的に求められるが, 系統誤差は人間が定める値
- おおよそ統計誤差で $1\sigma$ の領域とできるだけ対応するようにつける
- つまり, **物理屋のセンスであり, 良心であり, 自信である.**
- 具体的な例 (紹介する方法が使えるかは状況による)
  - 機器の最小目盛を誤差にしてしまう
  - 2つの理論値の幅を誤差とする
  - 複数人で目盛を読んで, 標準偏差を誤差にする
  - 決定論的でないシミュレーションを10回走らせて, その標準偏差を誤差にする
- または, 系統誤差がキャンセルするようなうまい実験を考える
  - 多くの場合, 全ての系統誤差がキャンセルされるわけではない
- 系統誤差源がわかっていて独立であれば, 誤差の二乗和の平方根を系統誤差として示すこともある.
  - それぞれバイアスのある誤差でも, いっぱい足せば中心極限定理で正規分布になるよね、みたいなイメージ. 数学的に厳密ではない.
- 系統誤差もパラメータとして表すことができる時 (nuisance parameterと呼ぶ) は likelihoodに一緒に入れて最小化しちゃう (Profile likelihood) 方法もある

# 第4回のまとめ

- (数学的な意味での) 関数を表現するクラス → TF1
- グラフ (scatter plot) を表現するクラス → TGraph, TGraphErrors, TGraphAsymmErrors
- ヒストグラム、グラフに対しては, Fittingが簡単にできる
- データをモデルに当てはめて, 物理解析に繋げる
- 今回はテクニカルに重たい回だったので, 色々と弄って, 適宜 Discordで質問してください.
- Fitについては根本の概念以外ほとんど伝えられていません. まずは使い始めて, 理解が進んできたら教科書などを参照してください.
- 次回 TTreeで最終回の予定です.

# Backups

初学者向けの範囲に  
収まらなかつた進んだ話題

Keita Mizukoshi (Tohoku Univ. RCNS)

# TF1の作り方 1

- 定義済みの関数の組み合わせ

  - `auto fitf = new TF1("fitf", "gaus(0)+expo(3)");`

  - 使える関数 gaus, gausn, expo, pol0, pol1, pol2, sin, cos, 等.

  - ()の中に数字を書くと、その番号からパラメータが使われる

- 自分で関数形をかく

  - `auto fitf = new TF1("fitf", "[0]*x*sin([1]*x)+TMath::Exp(-0.5*x/[2]);`

  - [0], [1], [2], ... がパラメータ.

  - TMathの関数が使える。大抵どんなものでも揃うはず。

  - 汚くても良ければ正規分布もこれでかける。

- C++の関数をTF1にする

  - `double user_func(double x, double p0, double p1) {  
 return p1*x+p0;  
}`

  - `auto fitf = new TF1("fitf", "user_func(x, [0], [1])");`
  - 文字列でC++の関数の名前を渡す必要があるのが若干気持ち悪い

# TF1の作り方 2

- やりたい放題する方法は、クラスを作り、  
`double operator()  
(double* x, double* p)`  
を定義すること。
  - $x[0] \rightarrow x$ ,  $x[1] \rightarrow y$  と次元に対応
  - $p[0], \dots$  がパラメータ
- クラスの中でヒストグラムや外部の値を好きに使えるよう
- 右はSimulationで作成したヒストグラムをpdfとして、高さのFittingを行う例

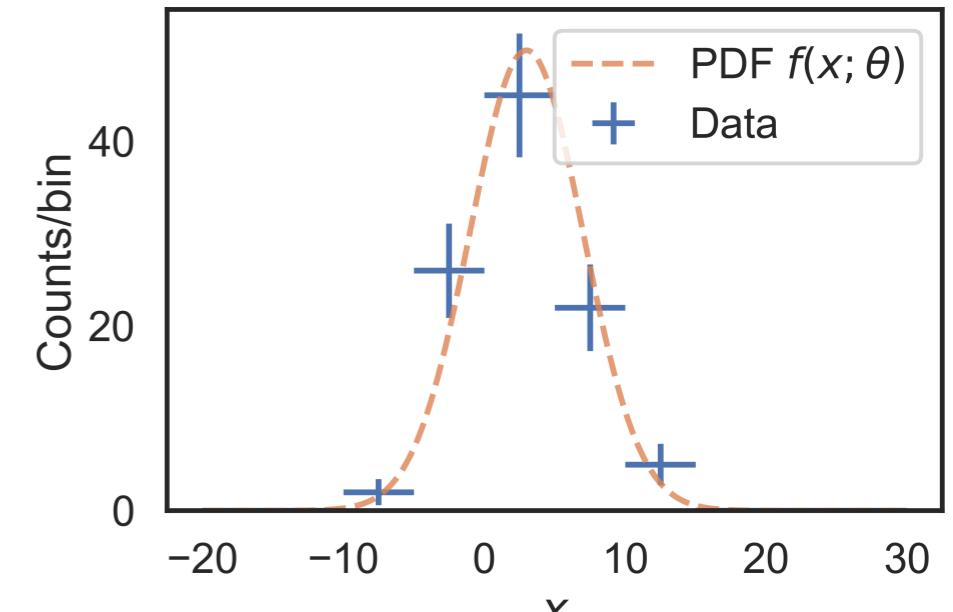
```

1 class TF1_generator {
2     public:
3         TH1D* hist;
4         TF1_generator(TH1D* hist){
5             this->hist = hist;
6         }
7         double operator() (double *x, double *p) {
8             return p[0] * hist->GetBinContent(hist->GetXaxis()
9                                         ->FindBin(x[0]));
10        }
11    };
12    int fit_example2(){
13        // Distribution generated by simulations
14        const int n_simulation_samples = 100000;
15        const double range_min = -100.;
16        const double range_max = 100.;
17        auto hist_sim = new TH1D("hist_sim", "hist_sim", 2000,
18                               range_min, range_max);
19        TRandom r;
20        for(int i=0;i<n_simulation_samples;++i){
21            hist_sim->Fill(r.Gaus(0, 5));
22        }
23        hist_sim->Scale(1./hist_sim->GetEntries());
24
25        auto fgen = TF1_generator(hist_sim);
26        auto f = new TF1("f",fgen, range_min, range_max, 1);
27        // 1 is a number of parameters
28        f->SetParameter(0,1200);
29
30        // Dummy data histogram
31        auto hist_data = new TH1D("hist_data",
32                               "hist_data", 100,
33                               range_min, range_max);
34        for(int i=0;i<520;++i){
35            hist_data->Fill(r.Gaus(0, 5));
36        }
37        hist_data->Draw();
38
39        hist_data->Fit(f, "LE", "", range_min, range_max);
40    }

```

# Likelihood fitting

- Unbinned Likelihood
  - 未知のパラメーター $\theta$ をもつPDFを仮定する
    - 例えば正規分布,  $\theta = (\mu, \sigma)$
  - このPDFである値 $x$ をとる確率  $f(x; \theta)$
  - データ  $\mathbf{x} = (x_0, x_1, \dots, x_n)$ 
    - このデータは各イベントで得られるデータ. 例 2.1, 5.2, 3.3, ...
  - $L(\theta) = \prod_{x_i \in \mathbf{x}} f(x_i; \theta)$  の最大化
  - つまり  $-\ln L(\theta) = -\sum_{x_i \in \mathbf{x}} \ln f(x_i; \theta)$  の最小化
- Binned likelihood
  - 未知のパラメーター $\theta$ をもつモデルを仮定する
    - 例えば正規分布,  $\theta = (\mu, \sigma, C)$
  - このモデルで, ある値 $x$ の時の計数  $y = f(x; \theta)$
  - データはBinningされており,  $i$  binの値  $\mathbf{x} = (x_0, x_1, \dots, x_n)$ ,  $\mathbf{y} = (y_0, y_1, \dots, y_n)$ . ここで,  $\mathbf{y}$  は計数
  - 各Binに対して, あるモデルを仮定した時に, そのBinの計数が得られる確率を求める
  - 計数に対しては, Poisson分布が使えるので, 各Binの値が得られる確率  $\text{Poi}(\lambda = f(x_i; \theta), k = y_i)$
- $L(\theta) = \prod_{i \in \text{bins}} \text{Poi}(\lambda = f(x_i; \theta), k = y_i)$  の最大化
- つまり  $-\ln L(\theta) = -\sum_{i \in \text{bins}} \ln \text{Poi}(\lambda = f(x_i; \theta), k = y_i)$  の最小化



# Extended binned Likelihood

- $\chi^2$  Fittingでは空ビンを扱えない (除外すると, "計数がない"という情報を捨てている)

- 確率密度関数  $f(x; \theta)$  (probability density function, p.d.f) に従ってデータが得られるとする. (前項だと正規分布)

- ただし, パラメーター  $\theta$  はわからないので推定する (前項だと  $\theta = (\mu, \sigma)$ )
- データを, 計  $N$  個のビンに詰めたヒストグラムがあり,  $i$  bin の値は  $n_i$

- $i$  bin の値として, モデルから推定される計数  $\nu_i$  は  $\nu_i = n_{\text{tot}} \int_{x_i \text{min}}^{x_i \text{max}} f(x; \theta) dx$

$$\bullet x_i \text{min}, x_i \text{max} \text{ は各Binの最小値, 最大値, } n_{\text{tot}} = \sum_i n_i$$

- このモデルで,  $i$  bin に係数  $n_i$  が観測される確率は Poisson 分布で  $\text{Poi}(\lambda = \nu_i, k = n_i) = \frac{\nu_i^{n_i}}{n_i!} e^{-\nu_i}$

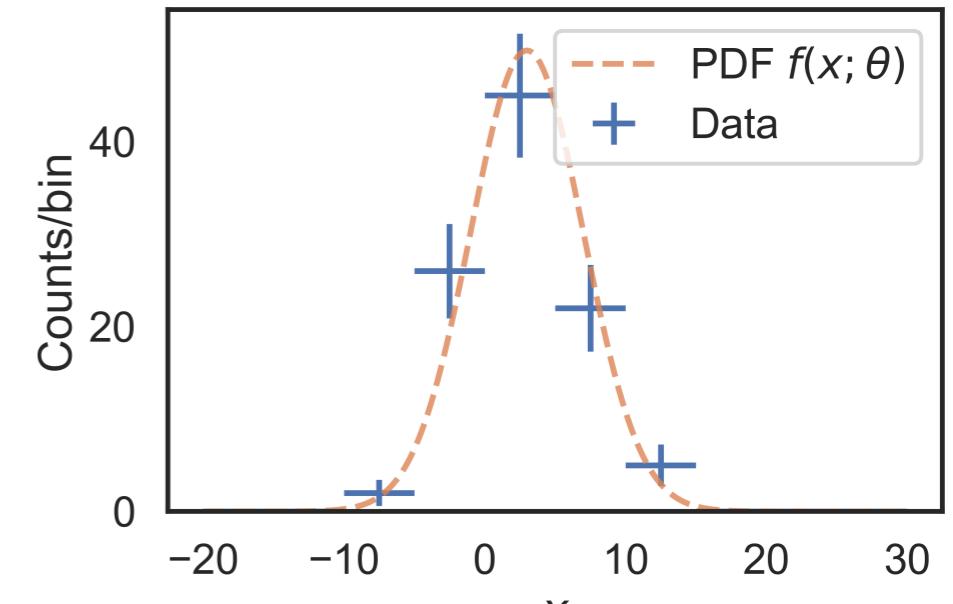
- 全ての bin に対して積をとると,  $\theta$  で決まるモデルのもとで,  $(\nu_1, \dots, \nu_N)$  を観測する確率になる

- ただし, 各々の確率は非常に小さいので, 掛け算で計算機の精度を超えててしまう → 対数をとって足し算にする
- 最大化より最小化の方が計算機向き → -2をかける

- $\chi^2$  の代わりに, NLL (Negative Log Likelihood) :  $\text{NLL} = -2 \ln L(\theta) = -2 \ln \prod_{i \in \text{bins}} \text{Poi}(\lambda = \nu_i, k = n_i)$  これを最小化する  $\theta$  を求める

- $\nu_i$  は  $\theta$  に依存するが,  $n_i$  は  $\theta$  に依存しない.  $\theta$  に依存する項のみ残し,  $\nu_{\text{tot}} = \sum_{i \in \text{bins}} \nu_i$  と定義すれば,  $-2 \ln L(\nu_{\text{tot}}, \theta) = 2\nu_{\text{tot}} - 2 \sum_{i=1}^N n_i \ln \nu_i$

- NLL 最小値から 1 大きくなる領域が, パラメータの誤差  $1\sigma$  (になるように 2 をかけた)



# Unbinned Likelihood

- $\chi^2$  Fittingでは空ビンを扱えない
  - 除外すると, "計数がない"という情報を捨てている
- 確率密度関数  $f(\mathbf{x}; \theta)$  (probability density function, p.d.f) に従ってデータが得られるとする. (前項だと正規分布)
- ただし, パラメーター  $\theta$  はわからないので推定する (前項だと  $\theta = (\mu, \sigma)$ )
- あるパラメータの組  $\theta$  で, 値  $\mathbf{x}$  を観測する確率は,  $f(\mathbf{x}; \theta)$  なので,  $\mathbf{x} = (x_0, x_1, \dots, x_n)$  の全データを観測する確率は  $\prod_{x_i \in \mathbf{x}} f(x_i; \theta)$ , これが大きくなるような  $\theta$  の組みを探す.
- ただし, 各々の確率は非常に小さいので, 掛け算で計算機の精度を超えてしまう
  - → 対数をとって足し算にする
- 最大化より最小化の方が計算機向き → マイナスをつける
- NLL (Negative Log Likelihood) :  $NLL = -2 \sum_{x_i \in \mathbf{x}} \ln f(x_i; \theta)$
- 解析的に解くのはほとんどの場合無理なので, 数値計算で最小値を求める
- NLL最小値から1大きくなる領域が, Parameterの誤差  $1\sigma$  (になるように2をかけた)