

# 高エネルギー宇宙物理学 のための ROOT 入門

– 第 3 回 –

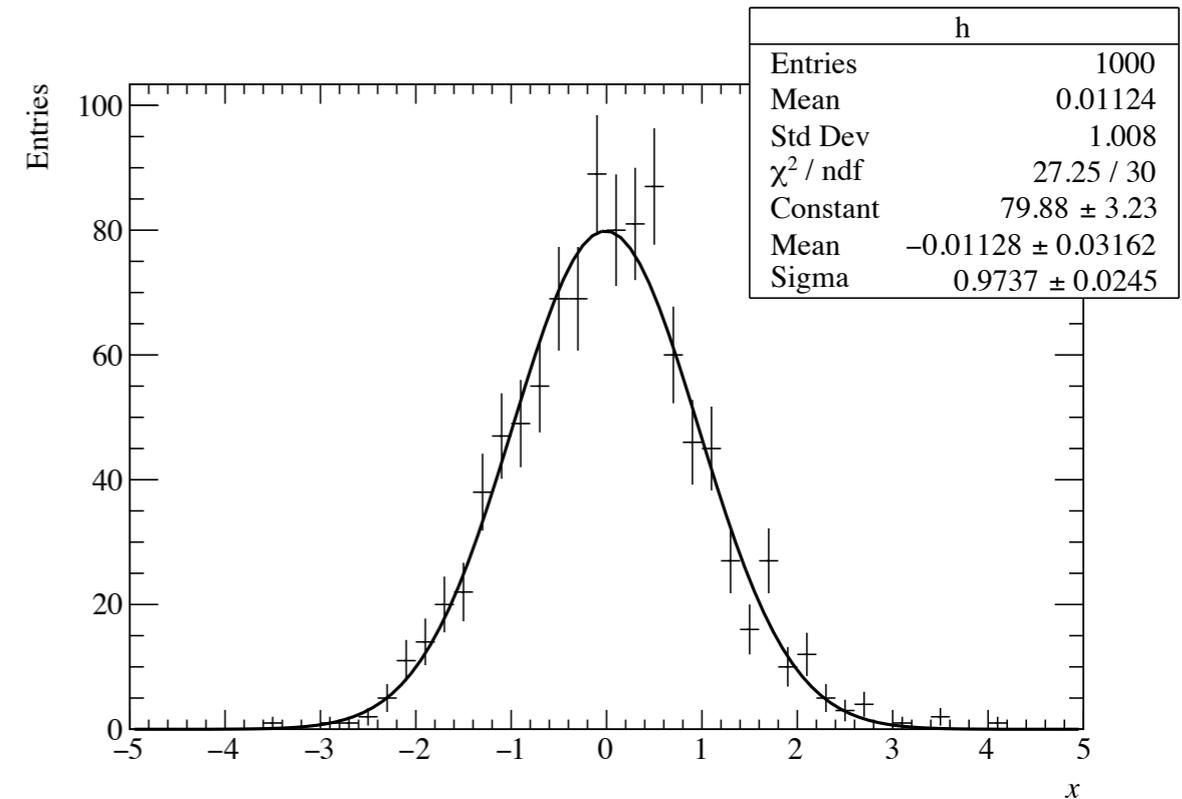
奥村 暁

名古屋大学 宇宙地球環境研究所

2018 年 5 月 24 日

# 前回のおさらい

- 各ビンには統計誤差が存在
  - ▶ そのビンに入る標本の大きさはポアソン分布に従う
  - ▶  $N > 20$  で正規分布と見なせる
  - ▶  $\delta N = \sqrt{N}$  と近似できる



- 最小二乗法を用いて、カイ二乗 ( $\chi^2$ ) を最小にするように、モデル関数の変数空間を探索する

$$\chi^2 = \sum_{i=1}^N \frac{(y_i - f(x_i))^2}{\delta y_i^2}$$

$x_i$  : ビンの中心値

$y_i$  : 各ビンの計数

$f(x_i)$  :  $x_i$  におけるモデル関数の値

$\delta y_i$  :  $y_i$  の誤差

$N$  - 変数の数 : 自由度  $\nu$

- この値はカイ二乗分布と呼ばれる確率密度関数に従う

# ポアソン分布 (Poisson Distribution) とは

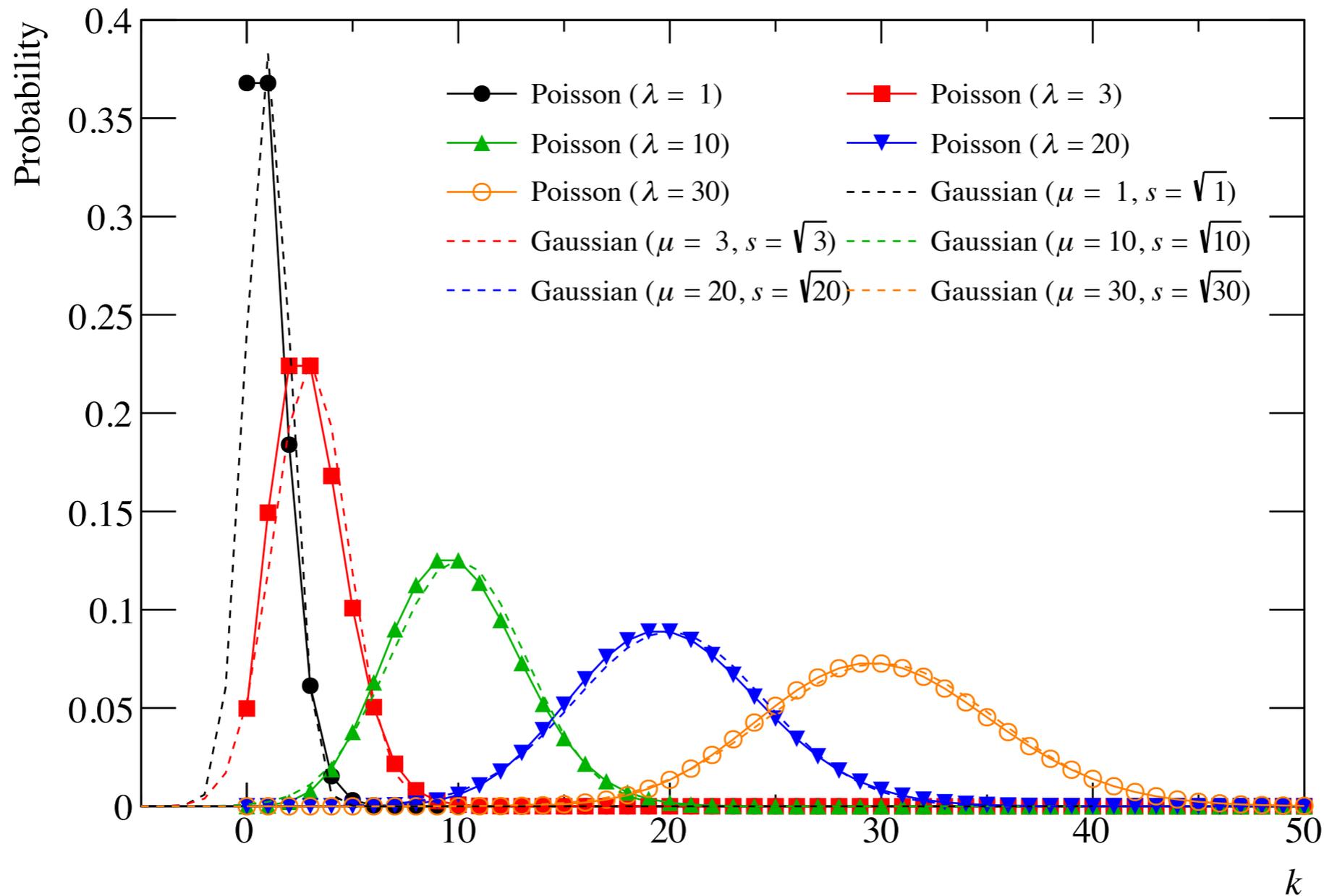
---

- ある平均値  $\lambda$  に対して、発生する自然数がどのように分布するか
  - ▶ 10 Bq の放射性物質が 1 秒間に何回崩壊するか
  - ▶ 微弱光を光検出器に当てた場合、何光電子発生するか
  - ▶ ある観測期間中に陽子が  $n$  回崩壊するとして、ゼロ回しか観測されなかった場合に崩壊事象の上限値をどう計算するか

$$P(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

- 現れる値は自然数 (0, 1, 2...) のみなので、積分できる関数分布を描くと、多数の高さの異なる  $\delta$  関数になる
- $\lambda$  が大きいと、正規分布と見なせる

# ポアソン分布と正規分布の比較



- 期待値  $\lambda$  が大きくなると、段々と平均値  $\lambda$ 、標準偏差  $\sqrt{\lambda}$  の正規分布に近づく
- ROOT はヒストグラムのビンの誤差を全て正規分布と仮定する

```
$ root
root [0] .x Poisson.C

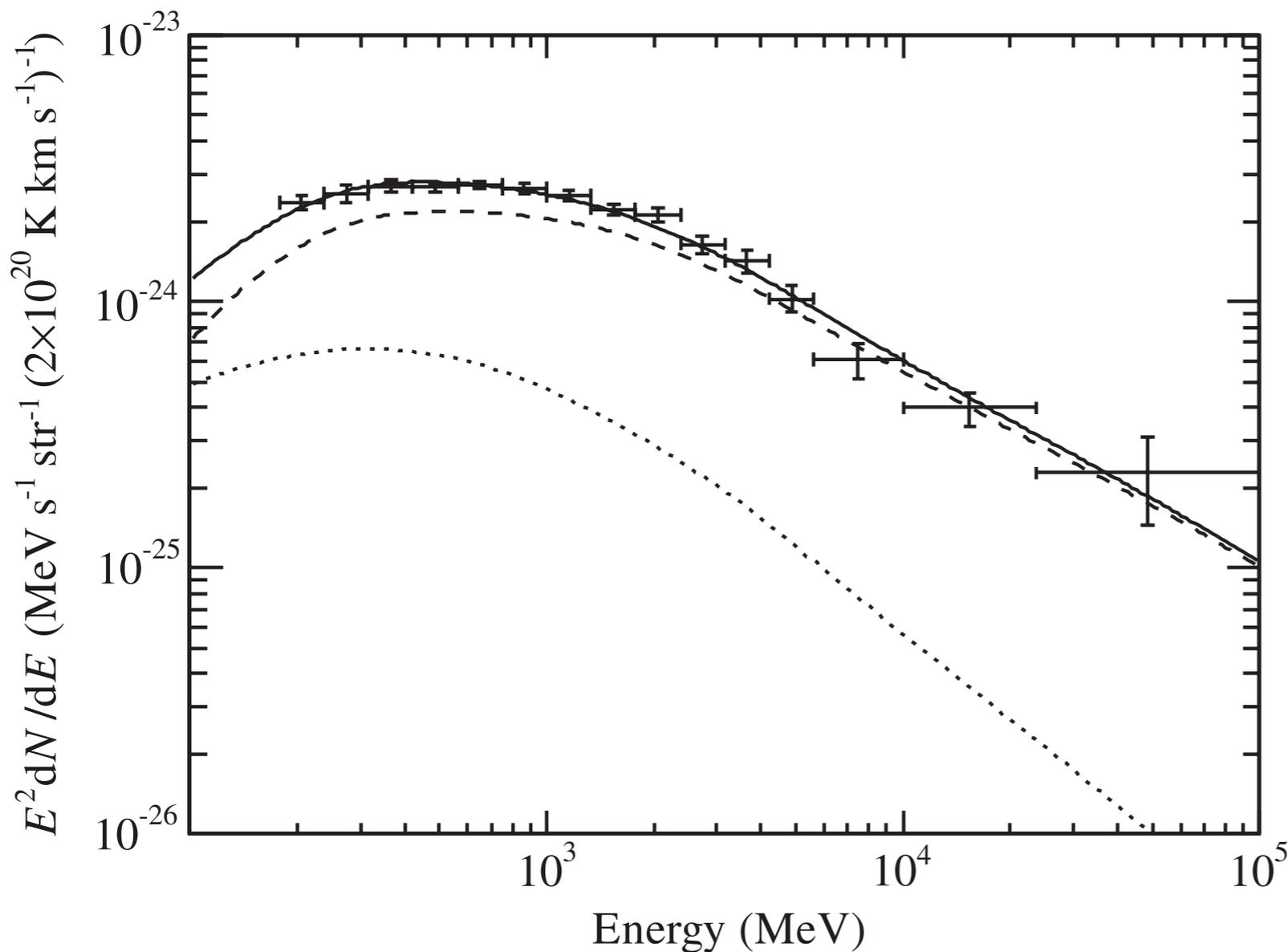
root [1] TMath::PoissonI(0, 3)    ❶ 平均 3 のポアソン分布で 0 が出る確率は 4.9%
(Double_t) 0.0497871

root [2] TMath::PoissonI(0, 2.995733) ❷ 平均 2.99... の場合は 5.0...%
(Double_t) 0.0500000                ❸ 発生回数ゼロのとき 95% の信頼度で上限は 2.99...

root [82] TMath::PoissonI(0, 4.60517) ❹ 99% の信頼度で上限は 4.60...
(Double_t) 0.0100000
```

# グラフ

# グラフ (graph) とは何か？



Ackermann et al. (2012)

- 得られたデータの変数を図表化したもの
- 狭義には2つ以上の変数の関係を示すために軸とともにデータ点を表示したもの
- 実験での使用例
  - ▶ 光検出器の印加電圧と利得の関係
  - ▶ エネルギースペクトル (energy spectrum)

# 大事なこと

---

- (2次元の場合) 独立変数  $x$  と従属変数  $y$  の違いを意識する
  - ▶ 例えば光検出器の利得 (従属変数) は、印加電圧 (独立変数) を変化させることで変化する
  - ▶ 滅多に見かけないが、これらを入れ替えて作図しない
  - ▶ 散布図の場合、どちらの変数が従属かは分からないので注意
- 無闇にデータ点を線で結ばない
  - ▶ 測定値には誤差がつきものなので、折れ線グラフはデータ解釈に先入観を持たせる
- 誤差棒の付け方 (第2回資料参照)
- エネルギースペクトルの横軸誤差棒はビン幅の場合あり

# ROOT のクラス

---

## ■ TGraph

- ▶ 2次元のグラフ（独立変数 1 つと従属変数 1 つ）
- ▶ 誤差棒無し

## ■ TGraphErrors

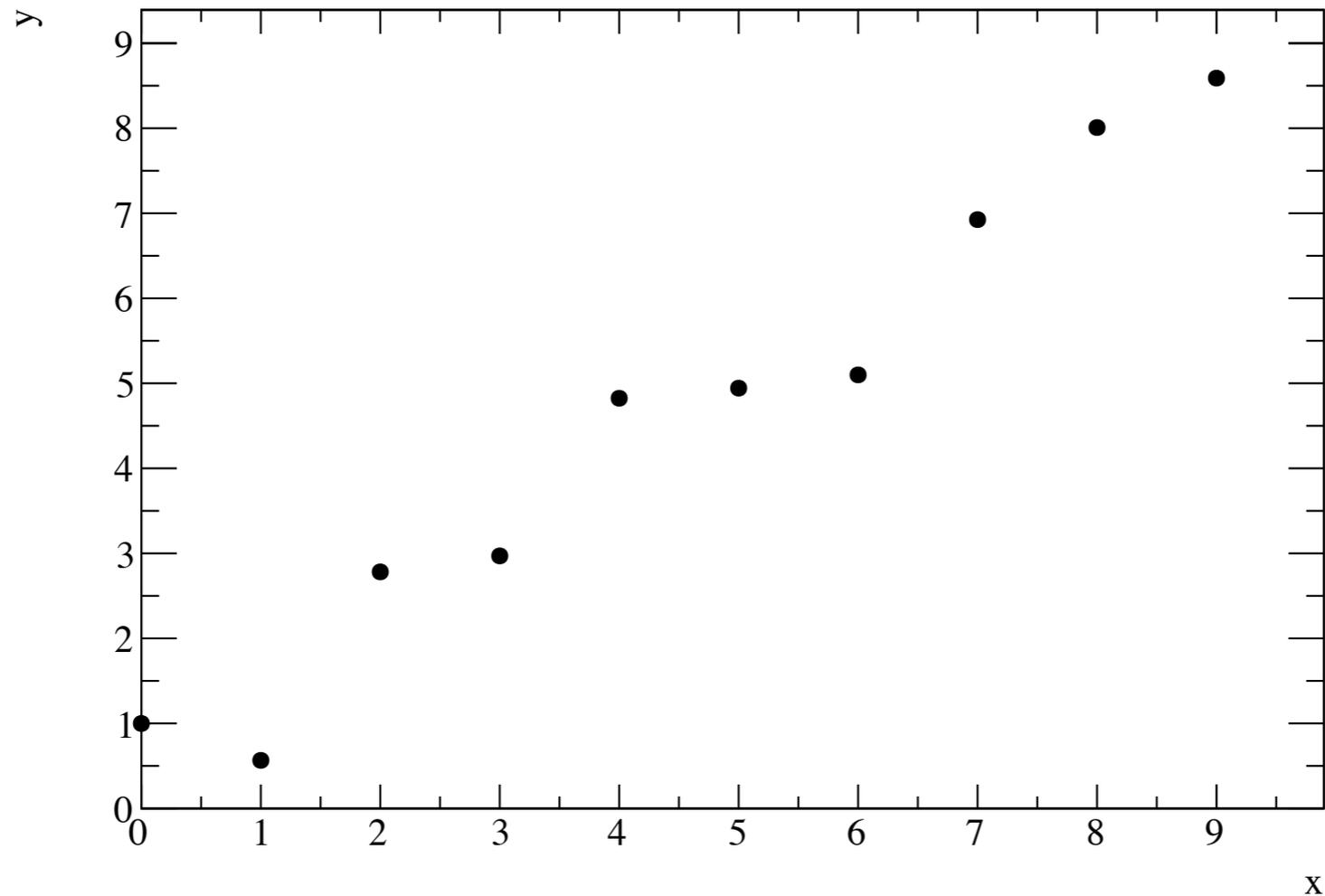
- ▶ 誤差棒あり

## ■ TGraph2D と TGaph2DErrors

- ▶ それぞれ 3次元版（独立変数 2 つと従属変数 1 つ）
- ▶ 名前が紛らわしいが、 $x/y/z$  の 3 つの値を持つ

# 2次元グラフ

# 単純な例



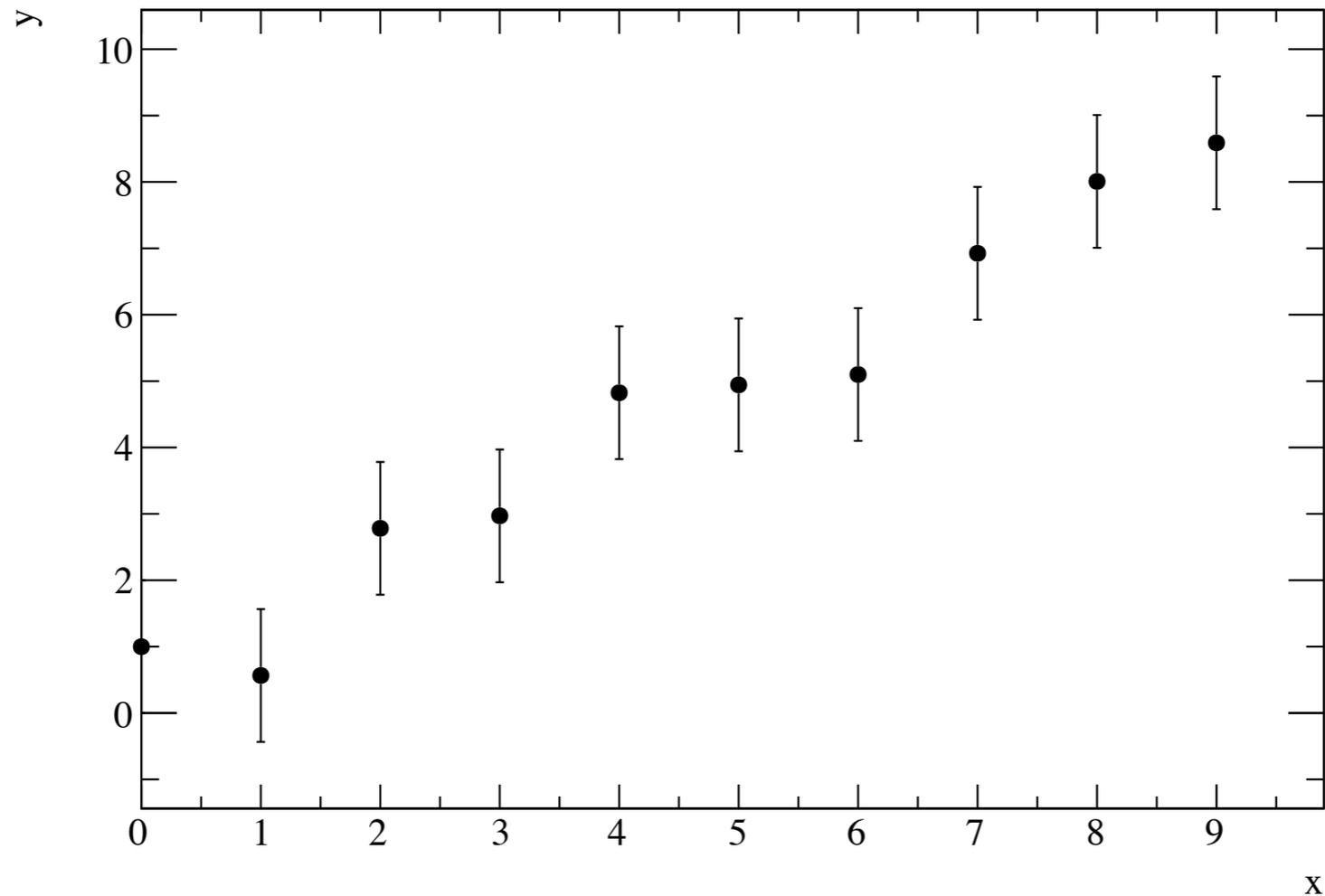
```
$ root
root [0] TGraph* graph = new TGraph;
root [1] for (int i = 0; i < 10; ++i) {
root (cont'ed, cancel with .@) [2] double x = i;
root (cont'ed, cancel with .@) [3] double y = i + gRandom->Gaus();
root (cont'ed, cancel with .@) [4] graph->SetPoint(i, x, y);
root (cont'ed, cancel with .@) [5]}
root [6] graph->SetTitle(";x;y;")
root [7] graph->SetMarkerStyle(20)
root [8] graph->Draw("ap")
```

- ① 適当に値を作り
- ② 点を追加する

- ③ タイトルはコンストラクタ外で
- ④ 初期値はドットなので変更する
- ⑤ axis と point を描く



# 誤差棒を足す



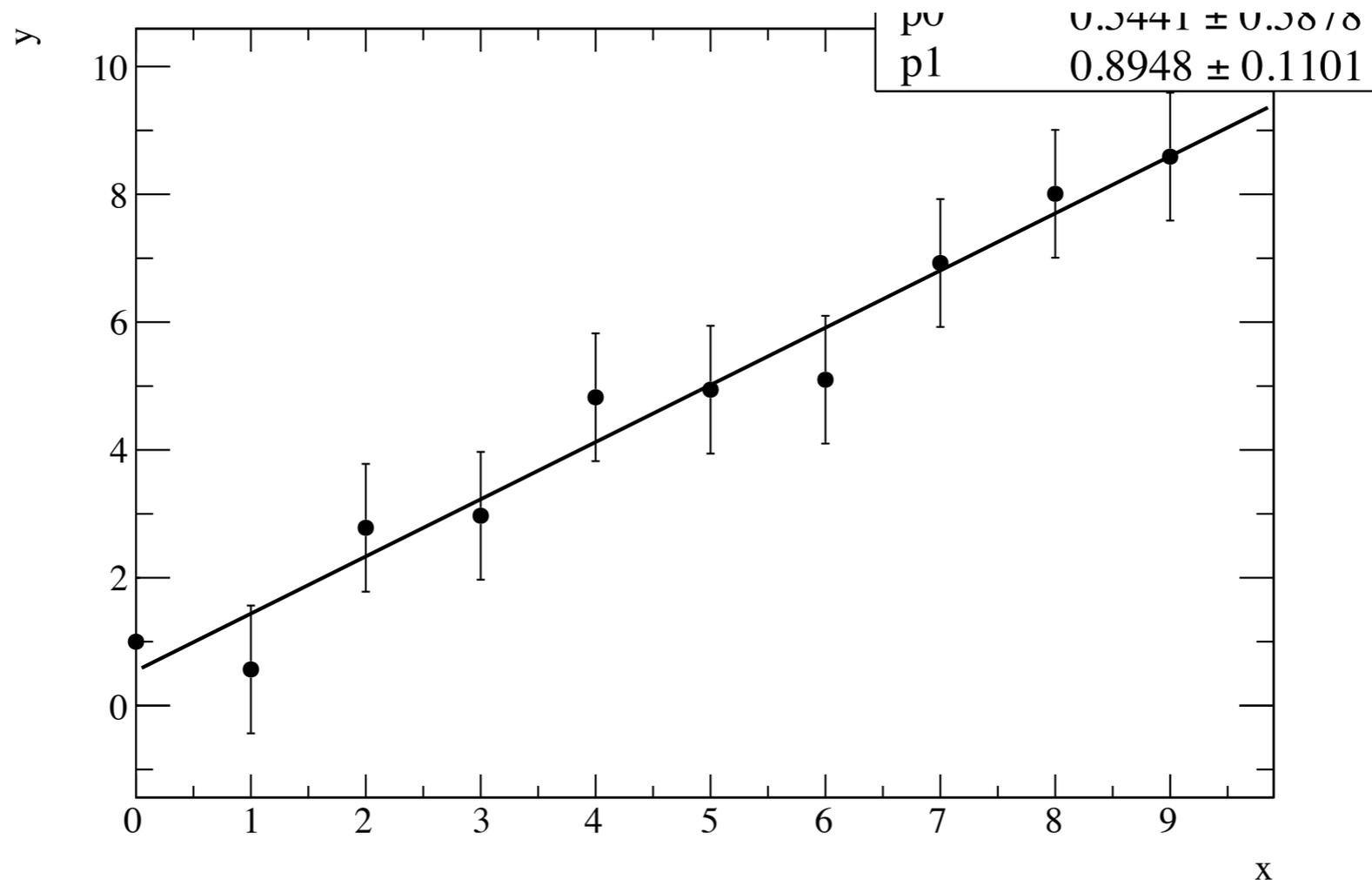
```
root [0] TGraphErrors* graph = new TGraphErrors
root [1] for (int i = 0; i < 10; ++i) {
root (cont'ed, cancel with .@) [2] double x = i;
root (cont'ed, cancel with .@) [3] double y = i + gRandom->Gaus();
root (cont'ed, cancel with .@) [4] double ex = 0;
root (cont'ed, cancel with .@) [5] double ey = 1.;
root (cont'ed, cancel with .@) [6] graph->SetPoint(i, x, y);
root (cont'ed, cancel with .@) [7] graph->SetPointError(i, ex, ey);
root (cont'ed, cancel with .@) [8]}
root [9] graph->SetTitle(";x;y;")
root [10] graph->SetMarkerStyle(20)
root [11] graph->Draw("ap")
```

① TGraphErrors にする

② y のばらつきと同じ量

③ 誤差を追加

# 既存の関数でのフィット

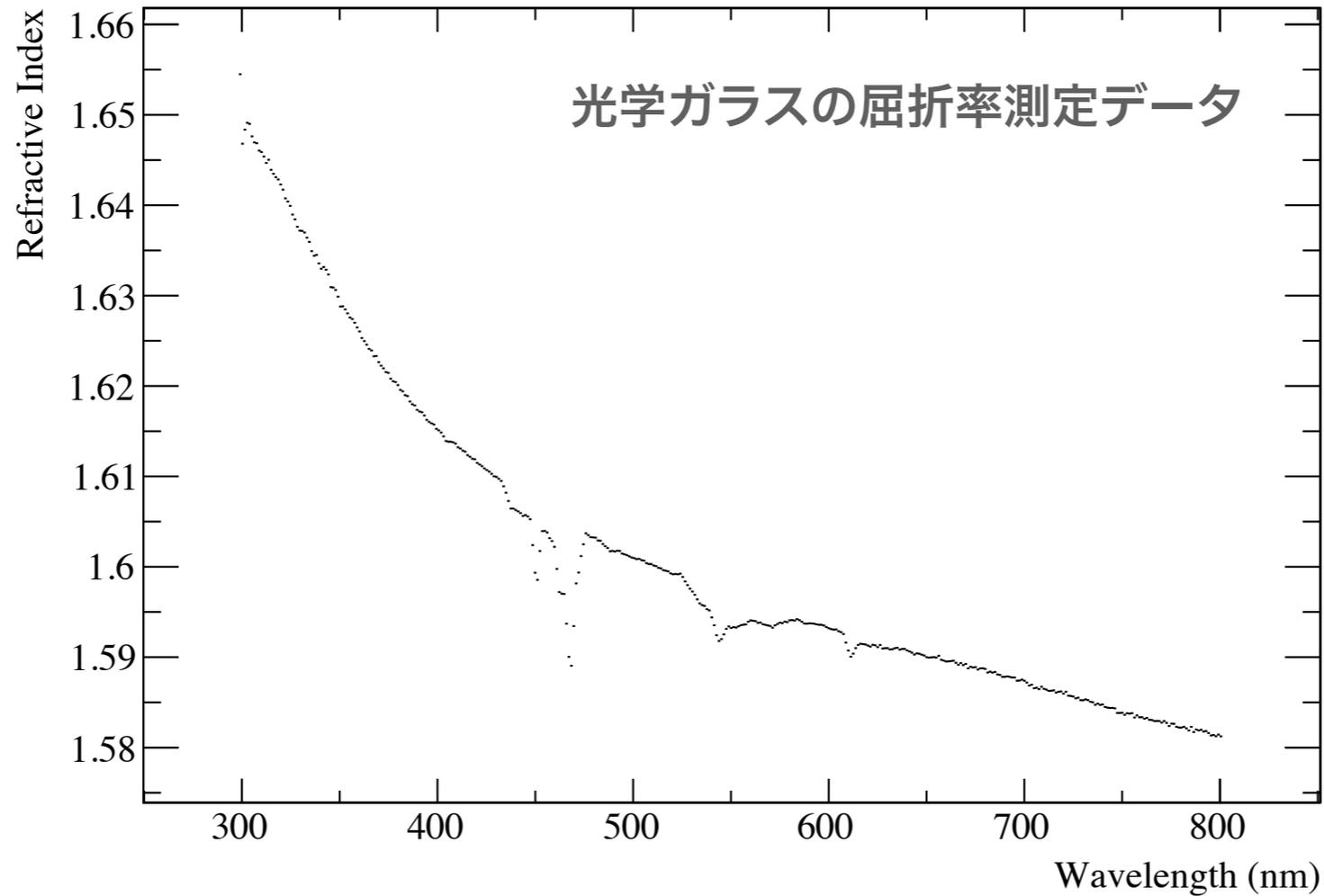


```
root [13] gStyle->SetOptFit()
root [14] graph->Fit("pol1")
*****
Minimizer is Linear
Chi2          =      2.50415
Ndf           =           8
p0            =      0.544079  +/-   0.587754
p1            =      0.894761  +/-   0.110096
(TFitResultPtr) @0x7fb24d517d50
root [15] TMath::Prob(2.504, 8)
(Double_t) 0.961544
root [22] graph->GetFunction("pol1")->GetProb()
(Double_t) 0.961537
```

① 1次関数 (pol1) でフィット  
 $f(x) = p_1 x + p_0$

②  $\chi^2$ フィットの確率を確認

# ファイルの読み込み



```
$ head -n 2 src/UVC-200B.csv
```

```
299.78,1.65449,.0363084
```

```
300.99,1.64681,.1093
```

```
$ root
```

```
root [0] TGraph* graph = new TGraph("src/UVC-200B.csv", "%lg,%lg,%*lg")
```

① ファイル名

② フォーマット指定

```
root [1] graph->SetTitle(";Wavelength (nm);Refractive Index;")
```

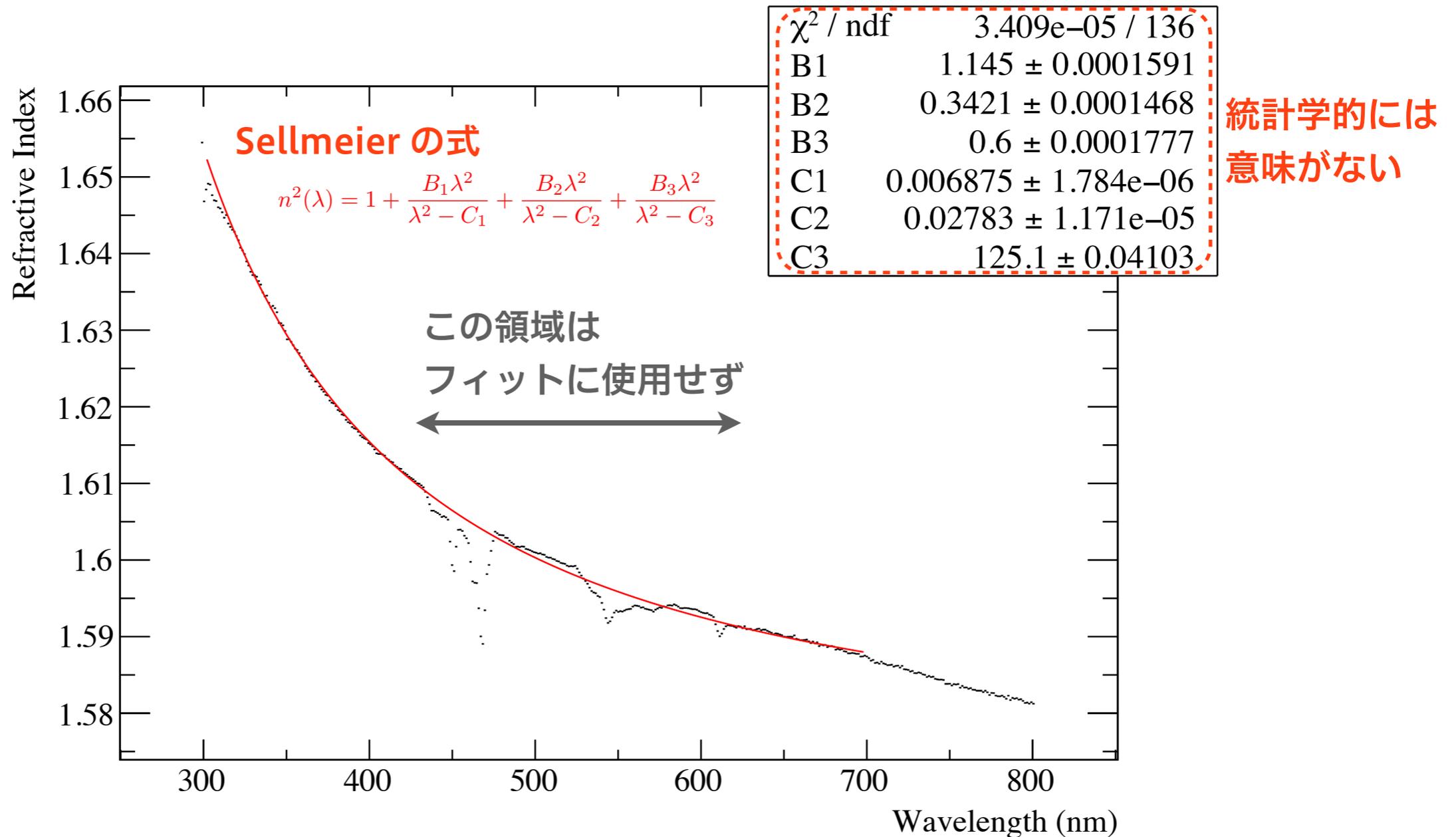
```
root [2] graph->Draw("ap")
```

# ついでに好きな関数形でフィットしてみる

```
$ cat Sellmeier.C
(略)
Double_t SellmeierFormula(Double_t* x, Double_t* par) { ❶ フィット用関数の定義
(略)
    Double_t lambda2 = TMath::Power(x[0] / 1000., 2.); ❷ 変数 x[] とパラメータ par[] から計算
    return TMath::Sqrt(1 + par[0] * lambda2 / (lambda2 - par[3]) +
        par[1] * lambda2 / (lambda2 - par[4]) +
        par[2] * lambda2 / (lambda2 - par[5]));
}

void Sellmeier() {
(略)
    TF1* sellmeier = new TF1("sellmeier", SellmeierFormula, 300, 800, 6);
    sellmeier->SetParameter(0, 1.12); ❸ 関数の初期値を与える
    sellmeier->SetParLimits(0, 0.8, 1.2);
    sellmeier->SetParName(0, "B1");
(略)
    TGraph* graph = new TGraph("UVC-200B.csv", "%lg,%lg,%*lg"); ❹ ファイルの読み込み
    graph->SetTitle(";Wavelength (nm);Refractive Index;");
    graph->Draw("ap");
    graph->Fit("sellmeier", "w m e 0", "", 300, 700); ❺ フィット
(略)
    TF1* sellmeier2 = new TF1("sellmeier2", SellmeierFormula, 300, 700, 6);
    sellmeier2->SetParameters(sellmeier->GetParameters());
    sellmeier2->SetLineWidth(1);
    sellmeier2->SetLineColor(2);
    sellmeier2->Draw("l same");
}
```

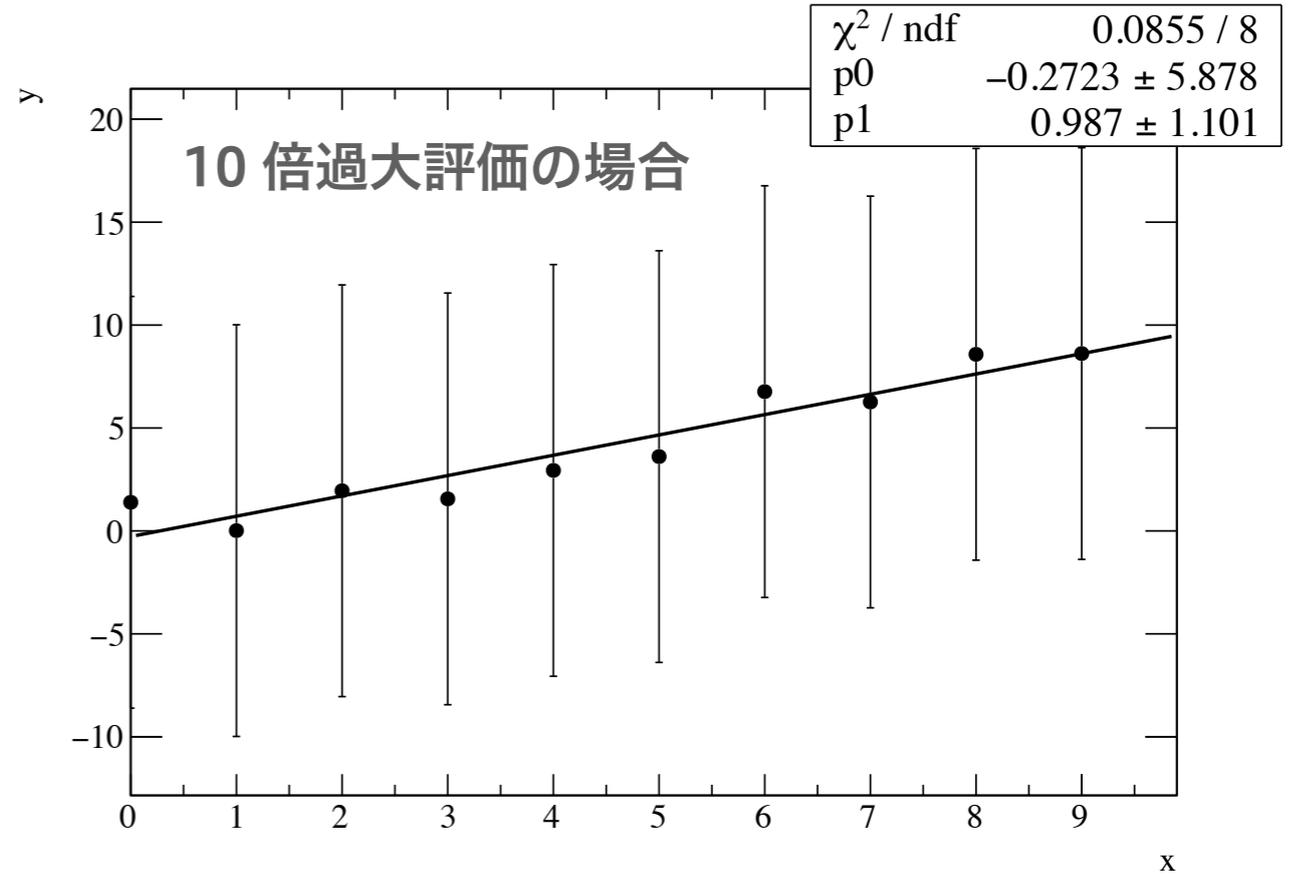
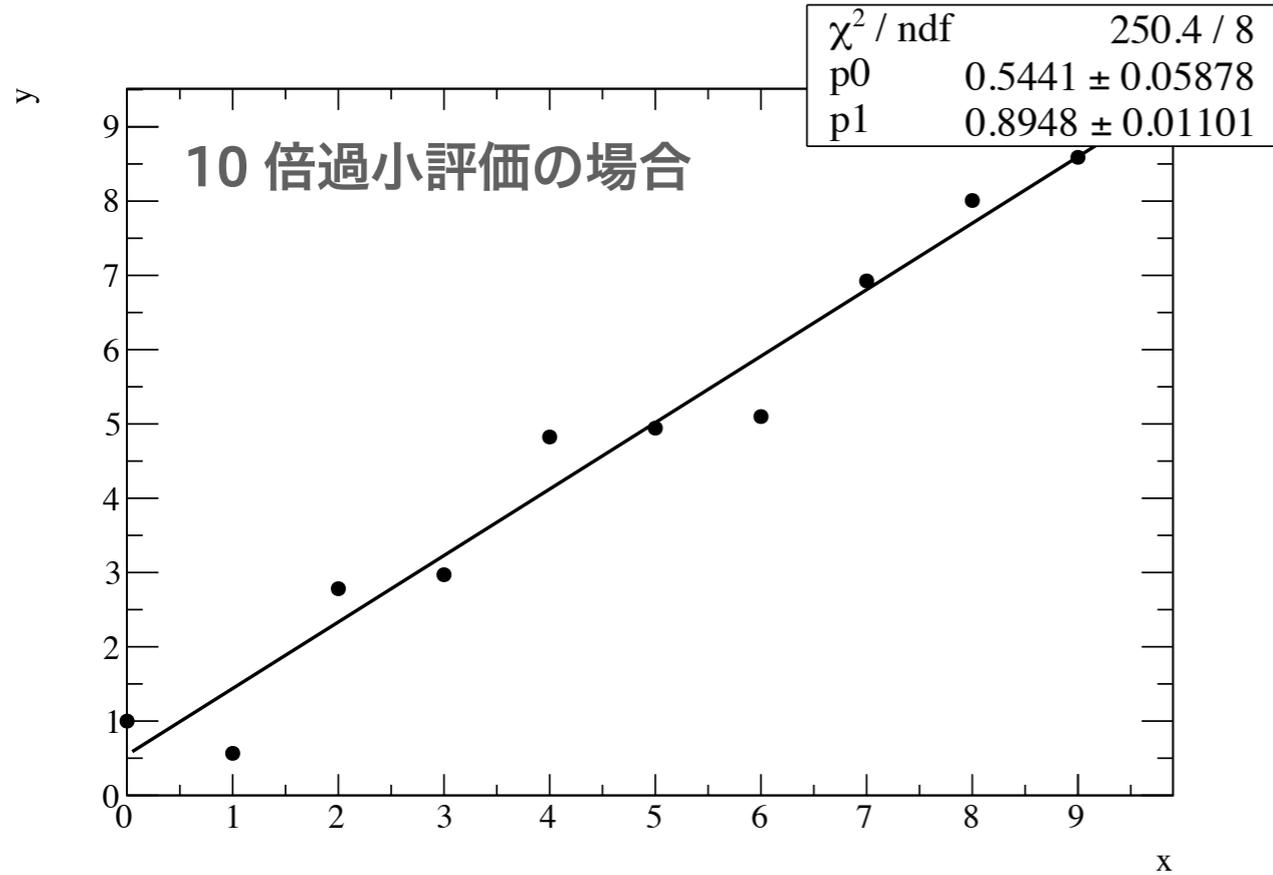
# ついでに好きな関数形でフィットしてみる



- ❖ 測定値に誤差がついていない場合、ROOT は全てのデータ点に誤差 1 をつける
- ❖ したがって、 $\chi^2/\text{ndf}$  の値は統計学的にあまり意味がない
- ❖ 得られたパラメータの誤差もあまり意味がない
- ❖ 大雑把なパラメータを知るには良いが「精度良くパラメータが求まった」とか言わない

# 誤差の過小評価、過大評価が与える影響

真値は  $p_0 = 0, p_1 = 1$



```
$ root
root [0] .x WrongErrorEstimate.C(0.1)
Probability = 1.40682e-49
root [2] .x WrongErrorEstimate.C(10)
Probability = 1
```

ありえないほど小さい確率

ありえないほど大きい確率

```
$ cat WrongErrorEstimate.C
void WrongErrorEstimate(Double_t error = 1.0) {
    TGraphErrors* graph = new TGraphErrors;
    for (int i = 0; i < 10; ++i) {
        double x = i;
        double y = i + gRandom->Gaus(); // Add fluctuation with a sigma of 1
        double ex = 0;
        double ey = error;
        graph->SetPoint(i, x, y);
        graph->SetPointError(i, ex, ey);
    }

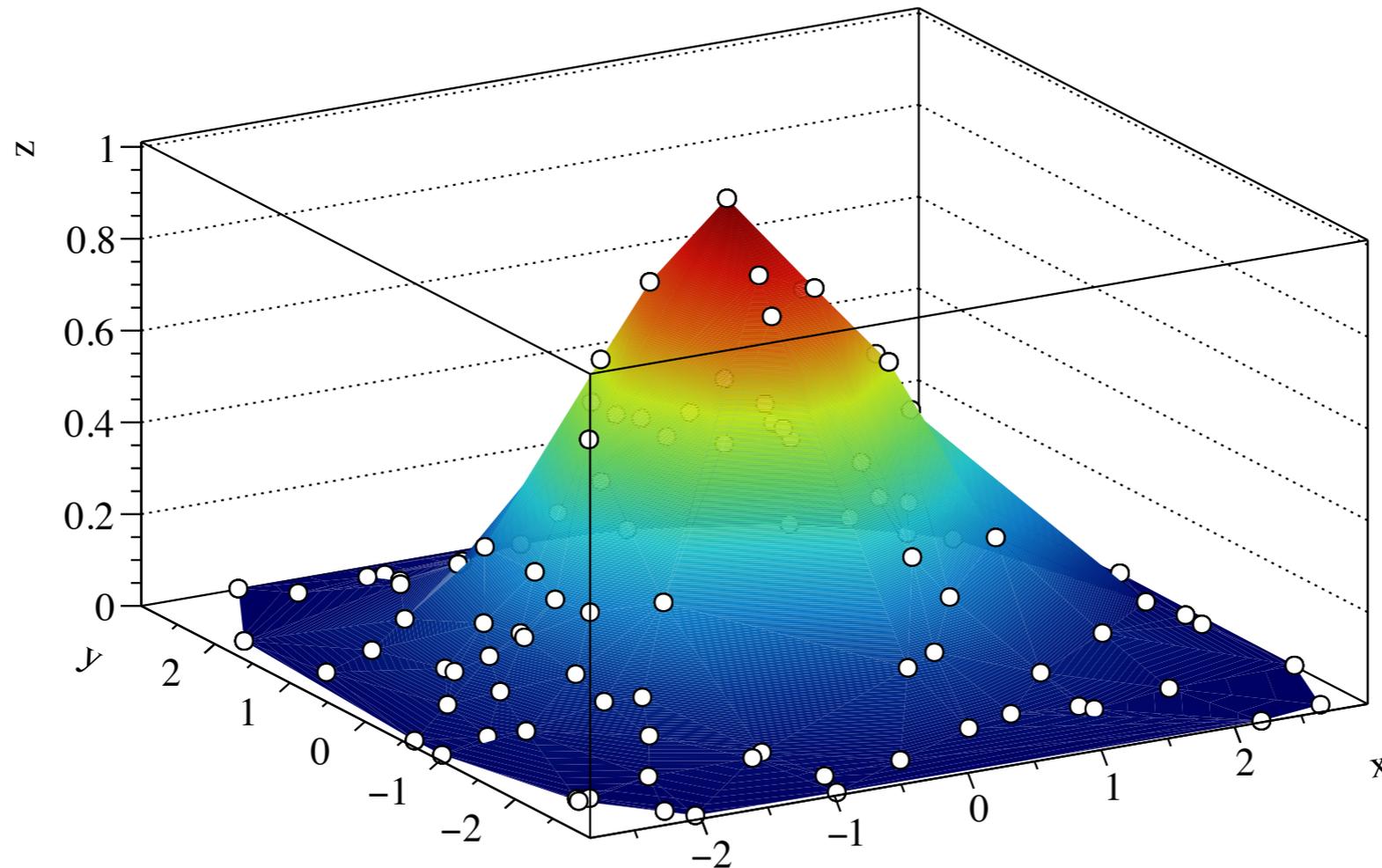
    graph->SetTitle(";x;y;");
    graph->SetMarkerStyle(20);
    graph->Draw("ap");

    gStyle->SetOptFit();
    graph->Fit("pol1");

    std::cout << "Probability = " << graph->GetFunction("pol1")->GetProb() <<
std::endl;
}
```

# 3 次元グラフ

# 単純な例



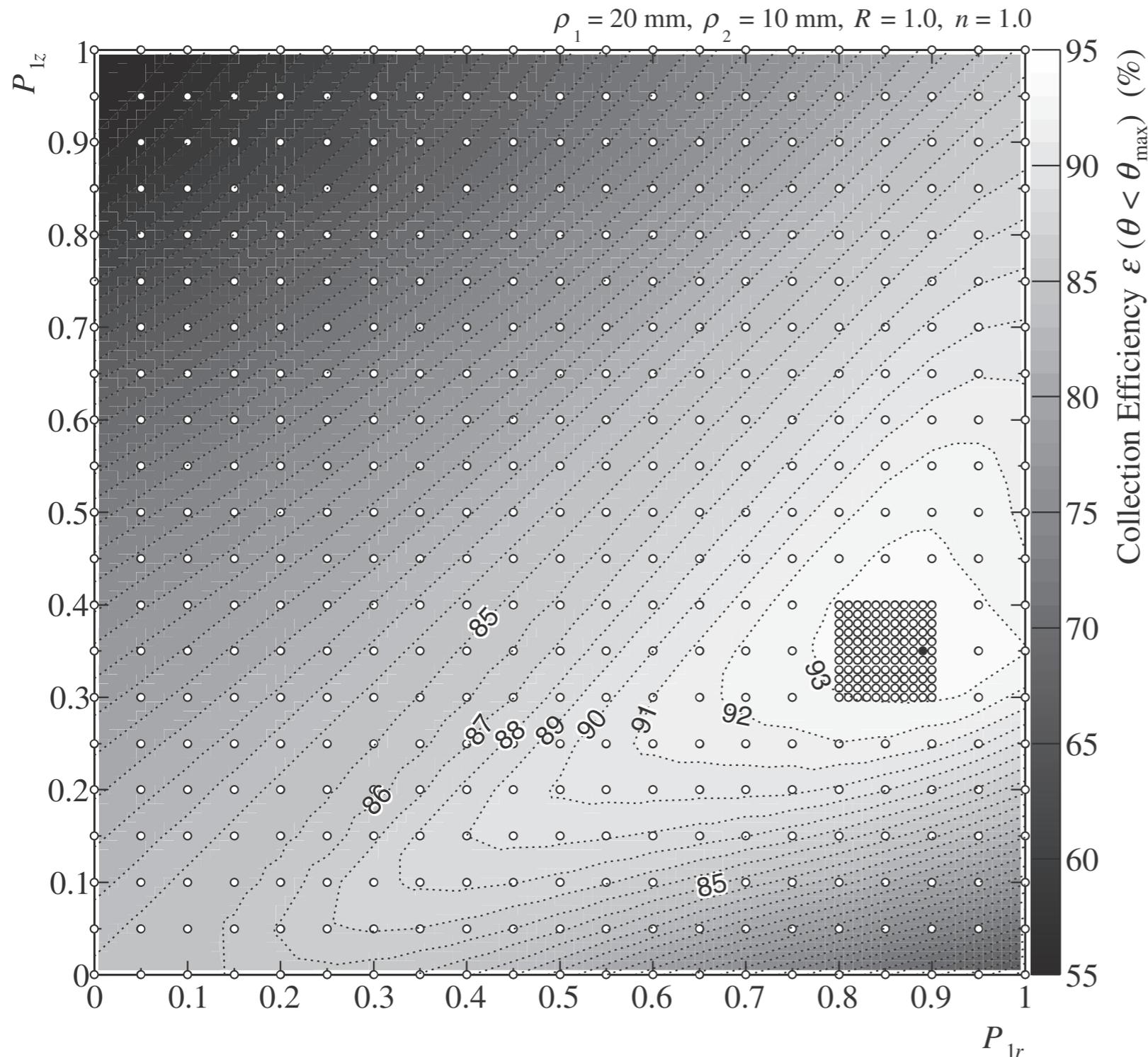
```
$ root
root [0] TGraph2D* graph = new TGraph2D
root [1] for (int i = 0; i < 100; ++i) {
root (cont'ed, cancel with .@) [2] double x = gRandom->Uniform(-3, 3);
root (cont'ed, cancel with .@) [3] double y = gRandom->Uniform(-3, 3);
root (cont'ed, cancel with .@) [4] double z = TMath::Exp(-(x*x + y*y)/2.);
root (cont'ed, cancel with .@) [5] graph->SetPoint(i, x, y, z);
root (cont'ed, cancel with .@) [6]}
root [7] graph->SetTitle(";x;y;z;")
root [8] graph->Draw("p0 tri2")
```

① TGraph2D か TGraph2DErrors を使う

② x/y/z を与える

③ 描画方法は多数あるので ROOT 公式を参照

# 実際の使用例



Okumura 2012

- 経験的には、あまり使用機会は多くない
- 2次元ヒストグラムのほうが登場頻度は高い
- XY ステージを使った測定など、離散的な測定で使用
- 限られたデータ点数から数値を補間するときにも便利
  - ▶ ドロネー図 (Delaunay diagram) を使って分割される

# ROOT オブジェクトの名前

# ROOT オブジェクトの名前

```
$ root
root [0] TH1D* hist = new TH1D("h", ";#it{x};Entries", 5, -5, 5)
(TH1D *) 0x7fdc3c64c040
root [1] TH1D* hist2 = hist
(TH1D *) 0x7fdc3c64c040
root [2] h
(TH1D *) 0x7fdc3c64c040
root [3] hist->Draw()
root [4] hist2->Draw()
root [5] h->Draw()
root [6] gDirectory->ls()
OBJ: TH1D      h      : 0 at: 0x7fdc3c64c040
root [7] TGraph* graph = new TGraph
root [8] graph->SetName("g")
root [9] gDirectory->GetList()->Add(graph)
root [10] gDirectory->ls()
OBJ: TH1D      h      : 0 at: 0x7fdc3c64c040
OBJ: TGraph    g      : 0 at: 0x7fdc3c0be610
```

① hist: C++ 上の変数名      ② “h”: ROOT の管理する名前  
③ オブジェクトの実体はメモリ上にある  
④ C++ 上で新たに hist2 という変数名を使って  
同じものを指すことができる  
⑤ ROOT のインタプリタ上では特別に  
ROOT の管理する名前でもオブジェクトに触れる  
⑥ 実体はどれも同じなので、結果は同じ  
⑦ “h” というオブジェクトは、gDirectory に登録されている  
⑧ TGraph はコンストラクタで命名の必要がない  
⑨ 後から名前を付けられる  
⑩ gDirectory に追加すると “g” でもアクセス可

# なぜ名前が必要？

---

- C++ や Python 内での変数名はいつでも変更できてしまう
- 「どのオブジェクトがどれ」と区別をつけるには、ROOT 側で名前をつけておくと便利なことがある
- ROOT オブジェクトを ROOT ファイルに保存するとき、名前がついていないとオブジェクト同士の区別がつかない
- ROOT はヒストグラムと TTree のみに、名前の付与と gDirectory への登録を自動で行う（理由は知らない）

# ROOT ファイル

---

- ROOT のクラスから作られたオブジェクトは、ほとんど全てが ROOT ファイルに保存できる
- 拡張子 .root
- データ収集の際に直接 ROOT ファイルとして保存してしまえば、解析時にいちいち ROOT オブジェクトとして作成し直さなくて良い
  - ▶ 例：オシロの波形を TGraph や TH1 として保存する
- 解析結果も ROOT ファイルにしてしまえば、可搬性が高くなる
- 描画した図も TCanvas のまま保存可能

# ROOT ファイルに保存する例

```
$ root
root [0] TH1D* hist = new TH1D("h", ";#it{x};Entries", 5, -5, 5)
root [1] hist->Draw()
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [2] TGraph* graph = new TGraph
root [3] graph->SetName("g")
root [4] gDirectory->GetList()->Add(graph)

root [5] TFile f("mydata.root", "recreate") ① ROOT ファイルを新規もしくは作成する
root [6] c1->Write() ② ROOT ファイルにオブジェクトを書き込む
root [7] graph->Write()
root [8] f.Close()
```

# ROOT ファイルを開く例

```
$ root
root [0] TFile f("mydata.root")
root [1] f.ls()
TFile**      mydata.root
TFile*       mydata.root
KEY: TCanvas  c1;1c1
KEY: TGraph   g;1
root [2] TGraph* graph = (TGraph*)f.Get("g")
root [3] TCanvas* can = (TCanvas*)f.Get("c1")

root [4] can->Draw()

root [5] TH1* h = (TH1*)can->GetPrimitive("h")
```

- ① ROOT ファイルを開く
- ② 中身を確認すると、“c1”という TCanvas と “g”という TGraph が保存されている
- ③ オブジェクトを取得し、キャストする  
Python の場合はキャスト不要  
名前がないと、取り出すのが面倒
- ④ TCanvas は保存時の状態で再度開ける
- ⑤ TCanvas 内に描画されたオブジェクトも取り出すことができる

# C++ と ROOT と Python

- コンパイルという作業が必要→コンパイラ型言語
- Python に比べると色々面倒くさい
  - ▶ 使う側も面倒くさい（機能が少ない、書く量が多いなど）
  - ▶ 教える側も面倒くさい（メモリの処理、ポインタなど）
- 「簡単なデータ解析しかしません」「修士で就職します」の場合、C++ を学ぶ必要性は近年は低い
- C/C++ を学んだほうがよい学生
  - ▶ ハードウェア制御を実行速度重視で行う
  - ▶ ROOT や Geant4 をガリガリ使う
  - ▶ ソフトウェアの開発側に回る（ユーザに終わらない）

# Python

---

- スクリプト型言語、コンパイルの必要がない
- テキスト処理などを初め、C++ より豊富な機能を標準で備える
  - ▶ 自分で色々と機能を実装する必要がない
  - ▶ 間違いが混入しにくく、ソフト開発も素早くできる
- C/C++ より実行速度が遅い場合が多い
  - ▶ ボトルネックの箇所だけ C/C++ で書いたりすることもある
  - ▶ Python の標準ライブラリなどに含まれる機能のほうが自作 C/C++ プログラムより最適化されており早い場合もある
- 理解が簡単、教えるのも簡単
- 修士で卒業する、データ解析しかしないなら Python だけでも生きていける

# 基本的な流れ

```
$ cd RHEA/src
```

```
$ g++ hello_world.cxx
```

```
$ ./a.out
```

```
Hello World!
```

① コンパイラでコンパイルし、実行ファイルを生成する

② 実行ファイルを実行する

```
$ g++ hello_world.cxx -O2
```

```
$ ./a.out
```

```
Hello World!
```

③ 最適化オプションをつける

※単純なプログラムだと変化ないが、一般的には速度が向上

```
$ g++ hello_world.cxx -O2 -o hello_world
```

```
$ ./hello_world
```

```
Hello World!
```

④ a.out はダサいので、実行ファイル名を変更

```
$ clang++ hello_world.cxx -O2 -o hello_world
```

```
$ ./hello_world
```

```
Hello World!
```

⑤ OS X だと Clang を使用する

※g++ と打っても同じコンパイラが走る

# コンパイラとは

---

- ❖ 人間の読めるコードを計算機が読める形式（機械語）に変換する
- ❖ コンパイルしないと動かない
  - ▶ ただし ROOT は特殊で、コンパイルしていない C++ を実行することができる（後述）
- ❖ Linux では GNU Compiler Collection (GCC)、OS X では Clang を使用するのが一般的
- ❖ 実際の大規模なソフトウェアでは多数のオプション指定が必要
- ❖ CMake や autotools で自動化が可能

# C/C++ の基本

```
$ cat hello_world.cxx
#include <cstdio>

int main() {
    printf("Hello World!\n");

    return 0;
}
```

- ① 非常に初歩的なことをする場合以外は、高度な機能を使うためにヘッダーファイルを #include する
- ② 必ず main 関数が実行される。他の関数は全て main 関数から呼び出される。
- ③ main 関数は int の返り値が必要。ここでエラーコードを返して main を抜ける。0 は正常終了の意味。

- ❖ ROOT の場合は特殊で、main 関数は ROOT 自体が既に実行している
- ❖ ROOT5 の場合は CINT が、ROOT6 は Cling がスクリプト内の関数を呼び出すため、スクリプト内に main 関数は不要
- ❖ ROOT を使わない純粋な C++ の場合、コンパイルしないと実行できない

# ROOT スクリプトの場合

```
$ cat hello_world.C
void hello_world() {
    printf("Hello World!\n");
}
```

- ❖ main 関数は定義する必要なし
- ❖ 多くの標準的なヘッダーファイルも #include する必要なし

# Python の場合

```
$ cat hello_world.py
#!/usr/bin/env python

def hello_world():
    print("Hello World!")

if __name__ == "__main__":
    hello_world()

$ python hello_world.py
Hello World!
$ ./hello_world.py
Hello World!
$ python
>>> import hello_world
>>> hello_world.hello_world()
Hello World!
```

① スクリプト内部で Python を呼び出すときに必要

② 関数の定義の仕方、def を使う

③ 1つ目、2つ目のやり方で必要

④ python コマンドにスクリプトを食わせる

⑤ 実行ファイルとして使う

⑥ module として使う

## ❖ 書き方と実行方法は何通りがある

- ▶ スクリプトを python コマンドに実行させる
- ▶ スクリプト自体を実行し内部で python コマンドを走らせる
- ▶ module として使う方法 (import する)

# ROOT スクリプトで main を再定義すると

```
$ cat main.C
int main() {
    return 0;
}
$ root
root [0] .x main.C
Error in <TApplication::TApplication>: only one instance of TApplication allowed
-----
| Welcome to ROOT 6.06/04                http://root.cern.ch |
|                                     (c) 1995-2016, The ROOT Team |
| Built for macosx64                   |
| From tag v6-06-04, 3 May 2016         |
| Try '.help', '.demo', '.license', '.credits', '.quit'/''.q' |
|-----|
/Users/oxon/.rootlogon.C:38:7: error: redefinition of 'fontid'
Int_t fontid=132;
    ^
/Users/oxon/.rootlogon.C:38:7: note: previous definition is here
Int_t fontid=132;
    ^
```

- ❖ 新しく作られた main ではなく、ROOT が新たに走り出す
- ❖ main は特殊な関数なので、ROOT スクリプト内では使わないこと
- ❖ ~/.rootlogon.C が 2 回呼び出されてエラーを吐いている

# もう少し ROOT っぽい例 (C++)

```
$ cat first_script2.C
void first_script2(int nbins, int nevents) {
  TH1D* hist =
    new TH1D("myhist", "Gaussian Histogram (#sigma = 1)", nbins, -5, 5);
  hist->FillRandom("gaus", nevents);
  hist->Draw();
}
$ root
root [0] .x first_script2.C(500, 100000)
root [1] myhist->GetName()
(const char *) "myhist"
root [2] gROOT->Get("myhist") ① 普通の C++ の教科書的には、TH1D hist("myhist" ...) とする
(TObject *) 0x7fe79d0572d0
root [3] myhist ② ROOT が名前でオブジェクトの管理をしている
(TH1D *) 0x7fe79d0572d0
root [4] delete gROOT->Get("myhist") ③ ROOT では名前を使ってオブジェクトのアドレスを取り出せる
root [5] gROOT->Get("myhist") ④ delete でオブジェクトをメモリ上から消すと、
(TObject *) nullptr ROOT の管理からも外れる
```

- C++ にはスコープ (scope) という概念が存在する
- {} や関数を抜けると、その変数は消えてしまう
- new してオブジェクトのアドレスをポインタ変数として扱うと、delete が呼ばれるまでオブジェクトがメモリ上から消えない (変数 TH1D\* hist は消える)
- ROOT が "myhist" という名前のオブジェクトを記憶しているので、後から参照できる

# new を使わないと

```
$ cat first_script2_wo_new.C
void first_script2_wo_new(int nbins, int nevents) { ① ポインタでない変数にする
    TH1D hist("myhist", "Gaussian Histogram (#sigma = 1)", nbins, -5, 5);
    hist.FillRandom("gaus", nevents);
    hist.Draw(); ② メンバ関数の呼び出しは -> ではなく . を使う
}
$ root
root [0] .x first_script2_wo_new.C(500, 100000) ③ TCanvas に何も表示されない
root [1] myhist->GetName()
input_line_79:2:3: error: use of undeclared identifier 'myhist'
(myhist->GetName()) ④ オブジェクトが消えているので、ROOT も既に管理していない
^
root [2] gROOT->Get("myhist")
(TObject *) nullptr
root [3] gROOT->ls()
```

- ❖ この書きかたは教科書的な C++ では普通
- ❖ ROOT の場合、生成したオブジェクトをスクリプト終了後にも引き続き描画させ解析したい
- ❖ ポインタを使わないとこれができない

# Python の場合

```
$ cat first_script2.py
import ROOT

def first_script2(nbins, nevents):
    global hist
    hist = ROOT.TH1D('myhist', 'Gaussian Histogram (#sigma = 1)', nbins, -5, 5)
    hist.FillRandom('gaus', nevents)
    hist.Draw()

$ python
>>> import first_script2
>>> first_script2.first_script2(500, 100000)
>>> first_script2.hist.GetName()
'myhist'
>>> import ROOT
>>> ROOT.myhist.GetName()
'myhist'
```

- ❖ Python も同様に、関数を抜けるとその変数は消えてしまう
- ❖ C++ の delete の相当する機能も働くため、オブジェクト自体も消える
- ❖ これを防ぐには global 変数を使う

# なぜ ROOT はスクリプト型言語のように動くのか

---

- ROOT 5 では CINT (シーイント) という C/C++ のインタプリタ (コンパイルしないで機械語に逐次変換する) が使われており、C/C++ を (ほぼ) 実行できる
- ROOT 6 では Clang を使用した Cling というインタプリタが使われるようになった
  - ▶ より C/C++ の文法に則っている
  - ▶ 実行速度の向上
  - ▶ エラーが分かりやすい、読みやすい

# 型 (Type)

---

- C/C++ には型がある
  - ▶ 符号あり整数型 : char (8 bit)、short (16)、int (32 or 64)
  - ▶ 符号なし整数型 : unsigned char など
  - ▶ 浮動小数点型 : float (32 bit)、double (64)
  - ▶ 32 bit OS か 64 bit かで int の大きさが違う
- ROOT では環境依存をなくすため、Short\_t や Long\_t などが定義されている (C の教科書で見たことのない型が ROOT の例で出てくるのはこのため)
- C++11 (新しい規格の C++) では、このような混乱をなくすために int8\_t (8 bit) などが追加された

# クラス (Class)

---

- 色々な変数や機能をひとまとまりにした、型の「ような」もの
- 好きなものを自分で追加できる。型は追加できない。
- TGraph や TH1D は ROOT が持つクラス
  - ▶ 内部にデータ点やビン幅などの数値情報
  - ▶ 名前、タイトルなどの文字情報
  - ▶ Draw() や GetStdDev() などのメンバ関数

# C++ のクラスの例

```
double x1 = 1.5, y1 = 2.3, z1 = -0.4;  
double x2 = -3.1, y2 = 5.6, z2 = 1.9;  
double x3 = x1 + x2, y3 = y1 + y2, z3 = z1 + z2
```

① 型だけでやると見づらく煩雑

```
Vector3D v1(1.5, 2.3, -0.4);  
Vector3D v2(-3.1, 5.6, 1.9);  
Vector3D v3 = v1 + v2;
```

② クラスにすることでより直感的に

- ❖ 情報をクラスにまとめることで扱いやすくなる
- ❖ 数値データに限らず、なんでもクラスにできる

# 簡単なクラスの例 (Vector3D.h)

```
#ifndef VECTOR_3D
#define VECTOR_3D

class Vector3D {
private:
    double fX;
    double fY;
    double fZ;

public:
    Vector3D();
    Vector3D(double x, double y, double z);
    Vector3D(const Vector3D& other);
    virtual ~Vector3D();

    virtual double X() const { return fX; }
    virtual double Y() const { return fY; }
    inline virtual double Z() const;
    virtual void Print() const;
};
```

- ❖ 「宣言」はヘッダーファイルに、定義はソースファイルに書くのが一般的
- ❖ 拡張子はそれぞれ .h/.hpp/.hxx/.hh などか、.cc/.cpp/.cxx など

# 使用例 (Vector3D\_main.cxx)

```
#include <cstdio>
#include "Vector3D.h"

int main() {
    Vector3D v0;           // default constructor
    Vector3D v1(1.5, 2.3, -0.4); // constructor with arguments
    Vector3D v2 = Vector3D(-3.1, 5.6, 1.9); // operator=, constructor
    Vector3D v3 = v1 + v2; // operator=, operator+
    Vector3D v4(v1 - v2); // copy constructor, operator-
    double product = v1 * v2; // operator*

    v0.Print();
    v1.Print();
    v2.Print();
    v3.Print();
    v4.Print();
    printf("v1*v2 = %f\n", product);

    return 0;
}
```

- ❖ 自分で作ったヘッダーファイルを #include することで、新たな機能として使えるようになる

# 実行例

```
$ g++ -c Vector3D.cxx
$ g++ -c Vector3D_main.cxx
$ g++ Vector3D.o Vector3D_main.o -o Vector3D
$ ./Vector3D
(x, y, z) = (0.000000, 0.000000, 0.000000)
(x, y, z) = (1.500000, 2.300000, -0.400000)
(x, y, z) = (-3.100000, 5.600000, 1.900000)
(x, y, z) = (-1.600000, 7.900000, 1.500000)
(x, y, z) = (4.600000, -3.300000, -2.300000)
v1*v2 = 7.470000
```

- ❖ 各ファイルを順次コンパイルし、オブジェクトファイル (.o) を生成する
- ❖ 最後にオブジェクトファイルを結合し、実行ファイルを作る

# Python のクラスの例 (vector.py)

```
class Vector3D(object):
    def __init__(self, x = 0., y = 0., z = 0.):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        return "(x, y, z) = (%f, %f, %f)" % (self.x, self.y, self.z)

    def __add__(self, other):
        return Vector3D(self.x + other.x, self.y + other.y, self.z + other.z)

    def __sub__(self, other):
        return Vector3D(self.x - other.x, self.y - other.y, self.z - other.z)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y + self.z*other.z
```

- C++ とは書き方がかなり違うので、よく見比べてください
- あくまで例なので、実際には既存のライブラリを使うことの方が多い
  - ▶ 開発速度が速い
  - ▶ 自作することによる間違いの混入を防ぐ

# 実行例

```
$ python vector.py
(x, y, z) = (0.000000, 0.000000, 0.000000)
(x, y, z) = (1.500000, 2.300000, -0.400000)
(x, y, z) = (-3.100000, 5.600000, 1.900000)
(x, y, z) = (-1.600000, 7.900000, 1.500000)
(x, y, z) = (4.600000, -3.300000, -2.300000)
v1*v2 = 7.470000
```

# C++11 について

---

- C++ の拡張として C++11 が策定された
  - ▶ スマートポインタ
  - ▶ 正規表現
  - ▶ マルチスレッド
  - ▶ 型の増加
  - ▶ 型推定と auto
  - ▶ 書ききれない、理解しきれていない
- ROOT を使うだけであれば初学者は気にしなくて良い
- C++ は勉強したことがあるのに理解できない構文で書かれた C++ (のような) コードに遭遇したら C++11 かも
- C++0x は以前の名前、C++14 はさらに後継
- CTA のソフトウェア開発では C++11 と Python 3 が推奨されている

# 第3回のまとめ

---

- ポアソン分布
- TGraph による簡単なグラフの作成とフィット例
- C++/Python の非常に簡単な説明
  
- 分からなかった箇所は、各自おさらいしてください