Kurze Anleitung für ROOT

Übungen zur Vorlesung Statistische Methoden der Datenanalyse

Eine Einführung in ROOT und C/C++

ROOT ist ein objektorientiertes Softwarepaket für die Datenanalyse und -visualisierung. Es basiert auf C/C++. ROOT ist frei verfügbar und kann von folgender Webseite bezogen werden

`http://root.cern.ch`

Dort gibt es viele weitere Informationen rund um ROOT, u.a. viele Beispiele und Tutorien.

Erste Schritte mit ROOT

ROOT ist im CIP-Pool installiert und wird von der Kommandozeile aus mit `'root'` gestartet. **Beendet wird es am ROOT-Prompt mit `'.q'`.**

ROOT hat einen eingebauten C/C++-Interpreter. So kann man interaktiv Code eingeben und auswerten, z.B.

```
root [0] sqrt(2.)
(double)1.41421356237309515e+00
root [1]
```

berechnet die Wurzel aus 2.

Wir werden jedoch ROOT meist nicht interaktiv benutzen, sondern den Code in eine Datei schreiben (ein "ROOT-Makro" erstellen) und anschließend diesen Code von ROOT ausführen lassen.

Makros werden in ROOT mit `.x` *`dateiname.C`* ausgeführt. Die Datei wird immer in dem Verzeichnis gesucht, in dem ROOT gestartet wurde.

Beispiel: Erstellen Sie in einem beliebigen Editor (z.B. Emacs) eine Datei mit nachfolgendem Inhalt und speicheren Sie sie als `first.C`.

```
{
  gROOT->Reset();
  cout << "Wurzel aus 2 ist gleich " << sqrt(2.) << "\n";
}
```

Starten Sie ROOT aus dem Verzeichnis, in dem sich die Datei `first.C` befindet und führen Sie das Makro mittels `.x first.C` aus.

In einer Makrodatei können allerdings auch mehrere Funktionen gleichzeitig untergebracht werden. Die Datei kann dann mit `.L` *`dateiname.C`* eingelesen werden, und danach auf die Funktionen darin zugegriffen werden.

Beispiel: Erstellen Sie in einem beliebigen Editor (z.B. Emacs) eine Datei mit nachfolgendem Inhalt und speicheren Sie sie als `second.C`.

```
double verdoppelt(double input=2.){
  return 2.*input;
}
void ausgabe(double zuverdoppeln=2.){
  cout<<verdoppelt(zuverdoppeln)<<endl;
}
```

Nach Laden mit `.L second.C` steht die Funktion `ausgabe` zur Verfügung, die bei Ausführung die als Argument übergebene Zahl verdoppelt und auf dem Bildschirm ausgibt. Wenn man sie ohne Argument aufruft, wird standardmäßig 2 angenommen.

Ein paar praktische Tipps

Bevor einige der in ROOT verwendeten Klassen beschrieben werden, erstmal ein paar Worte zur Optik und zum praktischen Arbeiten mit ROOT:

- Der beim Start von ROOT auftauchende „Splash-Screen“ ist auf Dauer ein wenig lästig. Er lässt sich dauerhaft entfernen, wenn man zu seiner `.bashrc` Datei im Homedirectory folgende Zeile hinzufügt:

  ```
  # make ROOT splash screen go away
  alias root="root -l";
  ```

- Die Standardfarben und -schriftarten von ROOT sind ein wenig merkwürdig. Um dies ebenfalls dauerhaft zu beseitigen, kann man Makrodateien erzeugen, die bei jedem Start ausgeführt werden, und so die Standards umsetzen. Sie können z.B. in Ihrer Homedirectory eine Datei `rootlogon.C` anlegen und dort bestimmte Standardwerte ändern. Um ROOT zu sagen, diese Datei zu benutzen, ist es notwenig eine Datei `.rootrc` im Homedirectory anzulegen, mit folgendem Inhalt:

  ```
  Rint.Logon:           $(HOME)/rootlogon.C
  ```

Histogramme

Ein Histogramm MyHist mit 10 Bins und einem Wertebreich von 0. bis 1. wird erzeugt durch

```
TH1F *MyHist = new TH1F("Name","Titel",10,0.,1.);
```

Ein bestehendes Histogramm kann durch seine *Fill*-Methode mit Werten gefüllt werden:

```
MyHist->Fill(x);
```

Hiermit wird ein Histogrammeintrag beim Wert x erzeugt. Ein Histogram kann mit der *Draw*-Methode angezeigt werden:

```
MyHist->Draw();
```

Ein Makro, das ein Histogramm erzeugt, einen Wert einfüllt und das Histogramm danach anzeigt, sieht so aus:

```
{
  gROOT->Reset();
  TH1F *MyHist = new TH1F("Name","Titel",10,0.,1.);
  MyHist->Fill(0.73);
  MyHist->Draw();
}
```

Punktwolken oder „Scatterplots“

Oftmals ist es hilfreich, sich die Verteilung von zweidimensionalen Größen in Form einer Punktwolke anzusehen, so dass man anhand der Dichte der Punkte in der zweidimensionalen Ebene auf die Verteilung der Größe schliessen kann. Dies kann mit der Klasse `TGraph` bewerkstelligt werden.

Ein Makro, das einen `TGraph` für vier Punkte erstellt, diese setzt, und schliesslich einen „Scatterplot“ erstellt, sieht so aus:

```
{
  gROOT->Reset();
  TGraph *MyScatter = new TGraph(4);
  MyScatter->SetPoint(0,1.2,0.9);
  MyScatter->SetPoint(1,1.1,1.0);
  MyScatter->SetPoint(2,1.445,0.1);
  MyScatter->SetPoint(3,1.5,1.2);
  MyScatter->SetMarkerSize(0.1);// Größe der Punkte
  MyScatter->Draw("AP");
}
```

Funktionen

Eine Funktion kann folgendermaßen gezeichnet werden:

```
{
  gROOT->Reset();
  fun1 = new TF1("fun1","sin(x)/x",0.,10.);
  fun1->Draw();
}
```

Zufallszahlen

Mit der Klasse *gRandom* lassen sich Zufallszahlen erzeugen. Es gibt einige vordefinierte Verteilungen:

```
Rndm(), Uniform(xlow,xup)
Binomial(n,prob)
Poisson(mean)
Gaus(mean,sigma)
Landau(Delta,beta)
```

Das nachfolgende Makro gibt fünf Zufallszahlen im Bereich zwischen 0. und 1. aus:

```
{
  gROOT->Reset();
  int i;
  for (i=0; i<5; i++) {
    cout << gRandom->Rndm() << "\n";
  }
}
```

Mehr über Histogramme

Die *Draw*-Methode für Histogramme hat mehrere Optionen:

`"E"`	Zeichne Fehlerbalken
`"SAME"`	Zeichne Histogramm über ein existierendes
`"C"`	Zeichne glatte Kurve durch die Bins
`"L"`	Verbinde Bins mit gerader Linie

Die `"SAME"`-Methode kann auch auf Funktionen angewendet werden.

Im folgenden Beispiel wird ein Histogramm mit 1000 Zufallszahlen entsprechend einer Gaußverteilung gefüllt. Das Histogramm wird gezeichnet und darüber wird die Kurve einer Gaußfunktion gezeichnet

```
{
  gROOT->Reset();
  TH1F *MyHist = new TH1F("Name","Titel",20,-4.,4.);
  int i;
  for (i=0; i<1000; i++) {
    MyHist->Fill(gRandom->Gaus(0.,1.));
  }
  MyHist->Draw("E");

  TF1 *f1 = new TF1("f1", "gaus",-4.,4.);
  f1->SetParameters(150.,0., 1.);
  f1->Draw("SAME");
}
```

C/C++ Grundlagen

Variablen und Konstanten

Gültige Zeichen für Variablennamen sind A-Z, a-z, 0-9 und _ (maximal *ein* Unterstrich!).

Variablen und Wertebereiche

Typ	**Bit**	**Präzision**	**Wertebereich**
char	8		-128 bis 127
unsigned char	8		0 bis 255
wchar_t	16		0 bis 65 535
int	16/32		wie short/long int
unsigned int	16/32		wie unsigned short/long int
short int	16		-32 768 bis 32 767
unsigned short int	16		0 bis 65 535
long int	32		-2 147 483 648 bis 2 147 483 647
unsigned long int	32		0 bis 4 294 967 295
bool	8		*true* oder *false*
float	32	7	1.17549×10^{-38} bis $3.40282 \times 10^{+38}$
double	64	15	2.22507×10^{-308} bis $1.79769 \times 10^{+308}$
long double*	64	15	2.22507×10^{-308} bis $1.79769 \times 10^{+308}$

* ist compiler-abhängig, aber mindestens so genau wie double. *Präzision* gibt die Anzahl der Stellen (Vor- plus Nachkommastellen) an, die genau dargestellt werden.

Logische Ausdrücke

```
<    kleiner als                 NICHT   !
>    größer als                  UND     &&
<=   kleiner oder gleich         ODER    ||
>=   größer oder gleich
!=   ungleich
==   genau gleich
```

Kontrollstrukturen

Einfache Verzweigungen

```
if(BEDINGUNG) {
  ANWEISUNG;
}
else {
  ANWEISUNG;
}
```

Mehrfache Verzweigungen

```
switch(VARIABLE) {
  case 1:
```

```
    ANWEISUNG1;
    break;
  case 2:
    ANWEISUNG2;
    break;
  ...
  default:
    ANWEISUNG;
}
```

Schleifen

for-Schleife

```
int i;
for (i=1; i<=10; i++) {
  ANWEISUNG;
}
```

Bedingungsschleife

```
int i = 0;
while (i < 5) {
  ANWEISUNG;
  i++;
}
```

Literatur

Helmut Erlenkötter, *C++, Objektorientiertes Programmieren von Anfang an*, rororo, ISBN 3-499-60077-3 (€ 11,99)

C++ Reference, http://www.cplusplus.com/

The ROOT User's Guide, https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html

ROOT Tutorials, https://root.cern.ch/doc/master/group__Tutorials.html