Shell Script & gnuplot の 簡単な説明!!

日本大学文理学部情報システム解析学科
谷聖一 研究室
田中 勇歩

Shell Script 編

Shellとは？

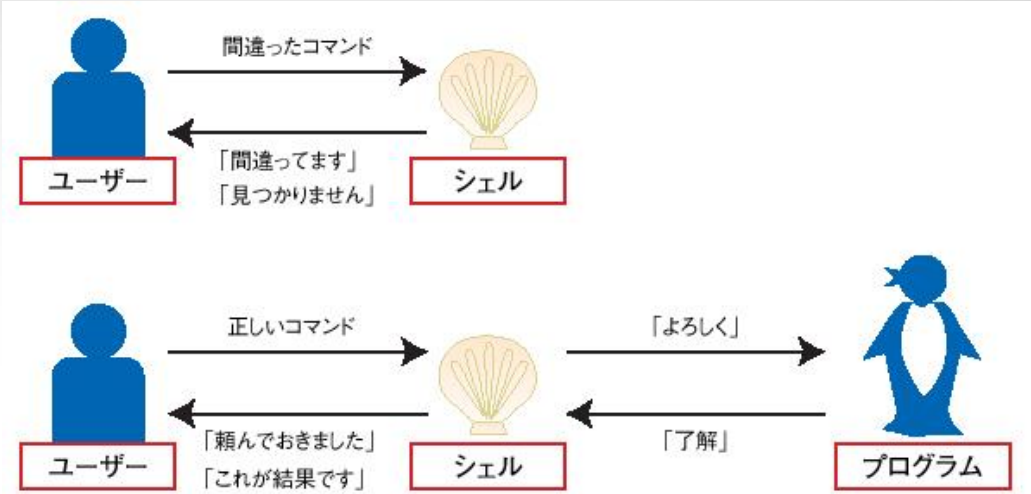
ユーザーがキーボードからコマンドラインに入力したコマンドを解釈してその実行を制御するプログラム

コマンド･インタプリタ

LinuxなどUNIX系OSで使われるコマンド・インタプリタ

Shell（シェル）

Shellの役割

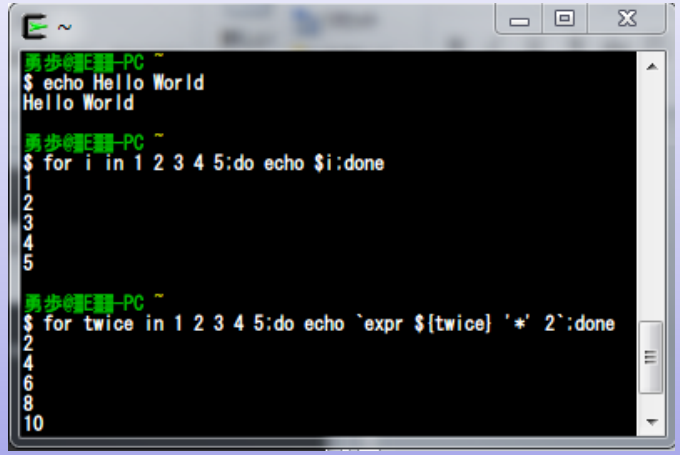


引用先(http://itpro.nikkeibp.co.jp/article/Keyword/20070907/281485/*)

Shellの役割
ユーザーが入力したコマンドを解釈

正しいコマンド⇒実行
・シェル自身が処理
・コマンドを処理するプログラムを起動

正しくないコマンド⇒エラーを通知



Shell Scriptとは？

Shell Script
- シェル・プログラム上で実行できるスクリプト言語の1つ
- コンピュータに実行させたい処理を記述
  ⇒その台本通りに動かすことが可能

※スクリプト（script）は「台本」という意味

超便利！

よってShell Scriptを上手く使えば、
複数のプログラムを起動することができる！

複数のプログラムの起動例（①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム）

```
勇歩@EE課-PC ~
$ chmod 755 plot.sh

勇歩@EE課-PC ~
$ ./plot.sh 1000 square.txt
```

←①実行権限をあげる

②プログラム実行

```
#!/bin/bash
#[グラフを作るプログラム]
#$1 指定した回数
#$2 保存するファイル名

#実行するプログラム名
square=square.cpp

#1"指定した数までの二乗を計算
echo "seuare start!!"
g++ -o a ${square}
./a ${1} ${2}
echo -e "finish¥n"

#グラフの作成
echo "make graph start!!"
gnuplot <<EOF
set terminal postscript eps
set title 'square -${1}-'
set xlabel 'x'
set ylabel 'y'
plot "${2}" using 1:2 with linespoints pt 7 ps 1
set out"image_${2%.*}.eps"
replot
EOF
echo "finish"
-u(Unix)--  plot.sh        All L1      (Shell-script[bash])
```

③C++のプログラムが実行

```
/*【1から指定の数までの二乗を計算するプログラム】
#include<iostream>
#include<fstream>//ファイルに書き出す為に必要
#include<algorithm>//atoiを使う為に必要

using namespace std;

void square(int max,string name);
/*コマンドライン引数から入力
  1:指定の数
  2:保存するファイルの名前
*/
int main(int argc,char *argv[]){
  square(atoi(argv[1]),argv[2]);//出力
  return 0;
}
void square(int max,string name){
  ofstream graph;
  graph.open(name.c_str(),ios::app);
  graph<<"#[1""<<max<<"までの二乗]"<<endl;
  graph<<"#x軸の値  y軸の値"<<endl;
  for(int i=1;i<=max;i++){
    graph<<i<<"  "<<i*i<<endl;
  }
  graph.close();
}
-u(Unix)--  square.cpp      All L1
```

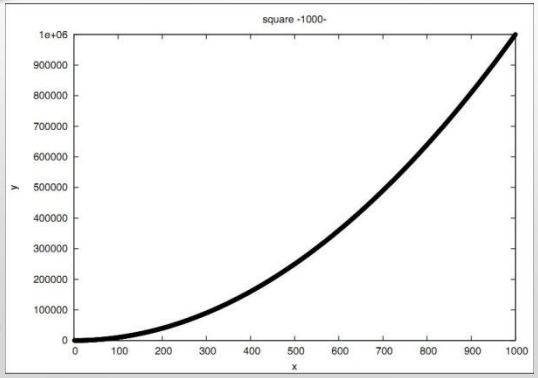
④結果がテキストに保存

```
勇歩@EE課-PC ~
$ cat square.txt | head
#[1～1000までの二乗]
#x軸の値  y軸の値
1  1
2  4
3  9
4  16
5  25
6  36
7  49
8  64

勇歩@EE課-PC ~
$ cat square.txt | tail
991  982081
992  984064
993  986049
994  988036
995  990025
996  992016
997  994009
998  996004
999  998001
1000  1000000
```

←⑤gnuplotが実行

⑥ファイルを読み込む

⑦epsで図に保存

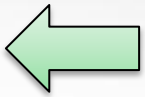


Shell Scriptの書き方＆実行

初めの行に、

```
#!/bin/bash
```

などのインタプリタを宣言



このファイルは/bin/bashをインタープリタとするよ！という意味。

実行は、

```
$ chmod 755 ファイル名
$ ./ファイル名
```

コマンドライン引数を送りたい場合は、

```
$ ./ファイル名 引数1 引数2 …
```

chmodとは？

chmod
パーミッション（所有権）の変更する為のコマンド

- Windowsではファイルの拡張子(.exeなど)によって、
  そのファイルが実行可能か不可能かを判断
- UNIXではパーミッション（あるいはファイルのモード）
  と呼ばれる特別な属性によって実行可能・不可能を判断

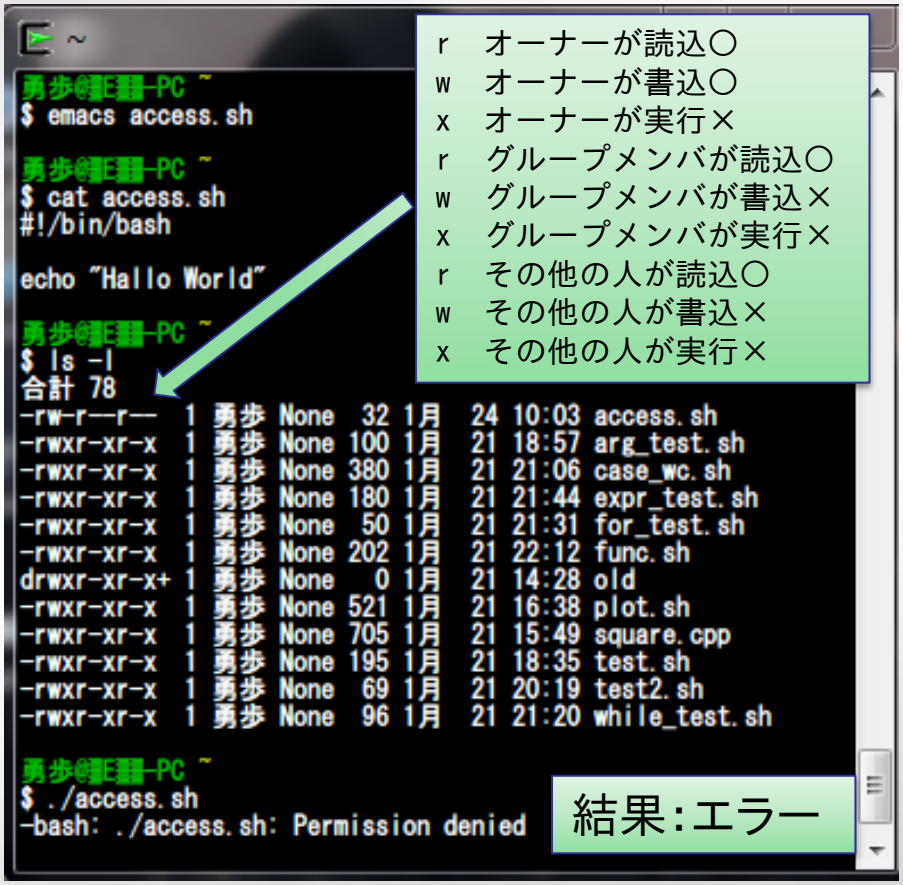
パーミッション
rwxr-xr-xなどの9文字の文字列や、755などの数値で表す
⇒パーミッションを数値で表す場合は r=4、w=2、x=1 を割り当て、3文字ずつの合計を用いる
例. rwxr-xr-x ⇒ 4+2+1, 4+1, 4+1 で、755

9文字の文字列は
r(readable), w(writable),
x(executable), ハイフン(-)
からなり、それぞれ意味をもつ

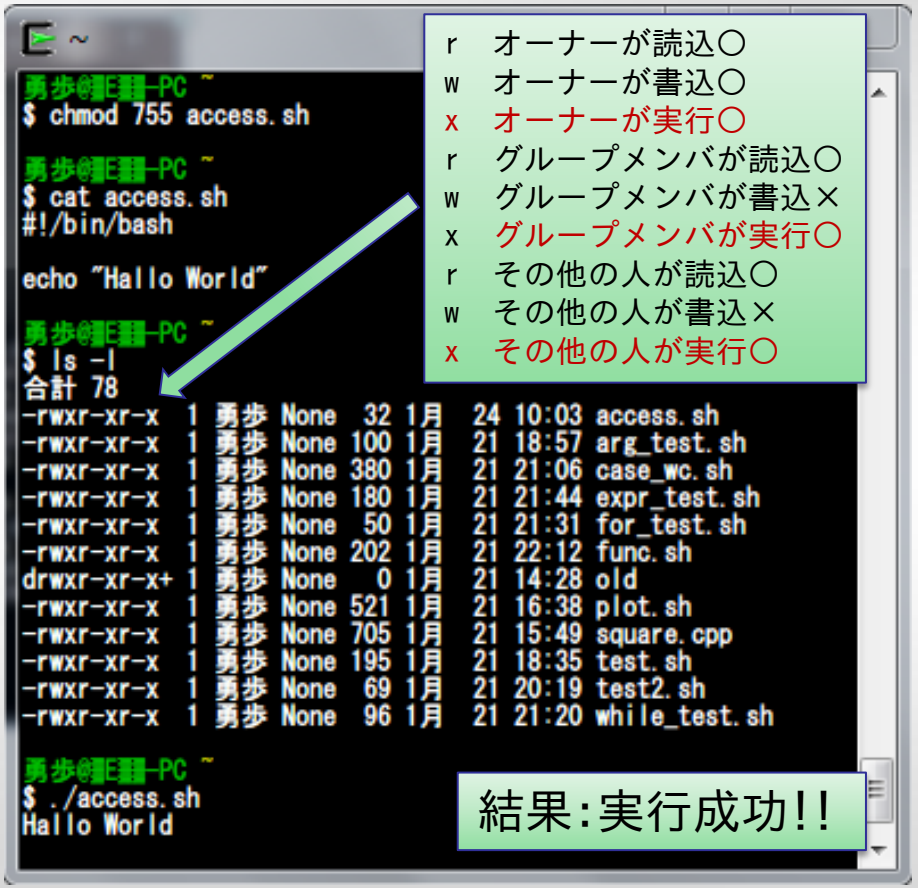
```
r  オーナーが読込可能なら r、さもなくば -
w  オーナーが書込可能なら w、さもなくば -
x  オーナーが実行可能なら x、さもなくば -
r  グループメンバが読込可能なら r、さもなくば -
w  グループメンバが書込可能なら w、さもなくば -
x  グループメンバが実行可能なら x、さもなくば -
r  その他の人が読込可能なら r、さもなくば -
w  その他の人が書込可能なら w、さもなくば -
x  その他の人が実行可能なら x、さもなくば -
```

実行例

実行権限を付与していない場合



実行権限を付与した場合



Shell Scriptの実行例

test.sh

```
#!/bin/bash
PARA1=”string”
PARA2=string
PARA3=3
PARA4=`date ’+%Y-%m-%d’`

echo ”PARA1 = ${PARA1}”
echo ”PARA2 = ${PARA2}”
echo ”PARA3 = ${PARA3}”
echo ”PARA4 = ${PARA4}”
echo ”PARA5 = ${PARA5}”
```

実行結果

```
$ test.sh
PARA1 = string
PARA2 = string
PARA3 = 3
PARA4 = 2013-01-21
PARA5 =
```

- 変数名に使える文字　：英数字 または「_」(アンダーバー)　※1文字目に数字は使用不可
- 変数に値を代入　　：「変数名=値」　※「=」の前後にはスペースやタブの代入不可
- 変数の参照　　：変数名の前に「$」を付ける

Shell Scriptの実行例2 -特殊な変数-

変数	意味
$$	シェル自身のPID（プロセスID）
$!	シェルが最後に実行したバックグラウンドプロセスのPID
$?	最後に実行したコマンドの終了コード（戻り値）
$-	setコマンドを使って設定したフラグの一覧
$*	全引数リスト。"$*"のように「"」で囲んだ場合、"$1 $2 … $n" と全引数を一つにくっついた形で展開される。
$@	全引数リスト。"$@"のように「"」で囲んだ場合、"$1" "$2" … "$n" とそれぞれの引数を個別にダブルクォートで囲んで展開される。
$#	シェルに与えられた引数の個数
$0	シェル自身のファイル名
$1～$n	シェルに与えられた引数の値。$1は第１引数、$2は第２引数…となる。

←これらの特殊な変数は、
参照専用で値の代入不可

arg_test.sh .sh

```
#!/bin/bash
echo $$
echo $!
echo $?
echo $-
echo $*
echo $@
echo $#
echo $0
echo $1
echo $2
echo $3
```

実行結果

```
$ ./arg_test.sh a b c
7488（※毎回変化）

0
hB
a b c
a b c
3
./arg_test.sh
a
b
c
```

Shell Scriptの実行例3 –四則演算-

シェルスクリプトで数値の演算を行いたい場合は、「expr」コマンドを使用

例.変数に5と3を足した数値を格納したい場合
NG: 変数=5+3　(変数に「5+3」という文字列が格納される)
OK: 変数=`expr 5 + 3`

算術演算子	意味
a + b	aとbの和
a – b	aとbの差
a ¥* b	aとbの積
a / b	aとbの商
a % b	aとbの剰余

「expr」コマンドで
使用できる算術演算子

expr_test.sh

```
#!/bin/bash
a=`expr 5 + 3`
b=`expr 5 - 3`
ab=`expr $a + $b`
c=`expr 5 ¥* 3`
d=`expr 5 / 3`
e=`expr 5 % 3`

echo ”a=$a”
echo ”b=$b”
echo ”ab=$ab”
echo ”c=$c”
echo ”d=$d”
echo ”e=$e”
```

実行結果

```
$ ./expr_test.sh
a=8
b=2
ab=10
c=15
d=1
e=2
```

Shell Scriptの実行例4 -演算子-

if文やwhile文では使える演算子

数値評価演算子	意味
数値1 -eq 数値2	数値1と数値2が等しい場合に真
数値1 -ne 数値2	数値1と数値2が等しくない場合に真
数値1 -gt 数値2	数値1が数値2より大きい場合に真
数値1 -lt 数値2	数値1が数値2より小さい場合に真
数値1 -ge 数値2	数値1が数値2より大きいか等しい場合に真
数値1 -le 数値2	数値1が数値2より小さいか等しい場合に真

文字列評価演算子	意味
文字列	文字列の長さが0より大きければ真
-n 文字列	文字列の長さが0より大きければ真
! 文字列	文字列の長さが0であれば真
-z 文字列	文字列の長さが0であれば真
文字列1 = 文字列2	2つの文字列が等しければ真
文字列1 != 文字列2	2つの文字列が等しくなければ真

論理結合演算子	意味
! 条件	条件が偽であれば真
条件1 -a 条件2	条件1が真、かつ、条件2が真であれば真
条件1 -o 条件2	条件1が真、または、条件2が真であれば真

※数値評価演算子の由来

eq : equal
ne : not equal
gt : greater than
lt : less than
ge : greater or equal
le : less or equal

Shell Scriptの実行例5 –if 文-

```
if [ 条件1 ]
then
    処理1
elif [ 条件2 ]
then
    処理2
else
    処理3
fi
```

補足

- 条件の前後にはスペースを入れないと、エラー。
- 「if [条件] 」は「 if test 条件」と書くこともできる
- 条件が偽の場合は、「! 条件」
- 「elif」はc言語で言うところの「else if」にあたる

test2 .sh

```
#!/bin/bash
if [ $# -le 5 ]
then
    echo $@
else
    echo ”over”
fi
```

実行結果

```
$ ./test2.sh 0 1 2 3 4
0 1 2 3 4

$ ./test2.sh 0 1 2 3 4
5
over
```

Shell Scriptの実行例6 –case 文-

```
case 変数 in
      パターン1）処理;;
      パターン2）処理;;
      パターン3 | パターン4）処理;;
      *）処理;;
esac
```

補足

- "read"は標準入力
- "*" は任意の文字列(空文字を含む)を意味
- "?" は任意の1文字を意味
- "[]" は括弧の中の文字のいずれかを意味
- "-" で範囲を指定することも可能。
- "!" が付くと括弧の中のいずれでもないという意味
- "|"はOR

case_wc.sh

```
#!/bin/bash
while :
do
  read key
  case ”$key” in
    ”q” ) echo ”終了します。”
          break ;;
    a* ) echo ”aで始まる文字列” ;;
    ?b* ) echo ”2文字目がbの文字列” ;;
    [A-Z]* ) echo ”大文字で始まる文字列” ;;
    [!xyz]* ) echo ”先頭がx、y、zではない文字列” ;;
    * ) echo ”上記のいずれでもない文字列” ;;
  esac
done
exit 0
```

実行結果

```
$ ./case_wc.sh
abc
aで始まる文字列
bbc
2文字目がbの文字列
ABC
大文字で始まる文字列
www
先頭がx、y、zではない文字列
xyz
上記のいずれでもない文字列
Abc
2文字目がbの文字列
q
終了します。
```

Shell Scriptの実行例7 –while 文-

```
while [ 条件 ]
do
    処理
done
```

補足

- 処理を途中で中断してループを抜ける
  →「break」
- ループの先頭に戻る→「continue」

while_test.sh

```
#!/bin/bash
a=1
while [ $a -le 10 ]
do
    echo ”${a} 回目の処理”
    a=`expr $a + 1`
done
```

実行結果

```
$ ./while_test.sh
1 回目の処理
2 回目の処理
3 回目の処理
4 回目の処理
5 回目の処理
6 回目の処理
7 回目の処理
8 回目の処理
9 回目の処理
10 回目の処理
```

Shell Scriptの実行例8 –for 文-

```
for 変数 in 引数1 引数2 …
do
    処理
done
```

補足

- 処理を途中で中断してループを抜ける
  →「break」
- ループの先頭に戻る→「continue」

for_test.sh

```
#!/bin/bash
for arg in $@
do
    echo ${arg}
done
```

実行結果

```
$ ./for_test.sh aa bb cc
dd
aa
bb
cc
dd
```

Shell Scriptの実行例9 –関数-

```
関数名()
{
    処理・・・
}
```

補足

- C言語のように括弧の中に引数を記述する必要はない
- 関数に引数を渡す場合、シェルと同じくスペース区切りで渡す
- 関数に渡された引数を参照する場合、シェルと同じくに「$1, $2, ...」と参照
- 呼び出し元に終了コードを返す場合「return コード」と記述

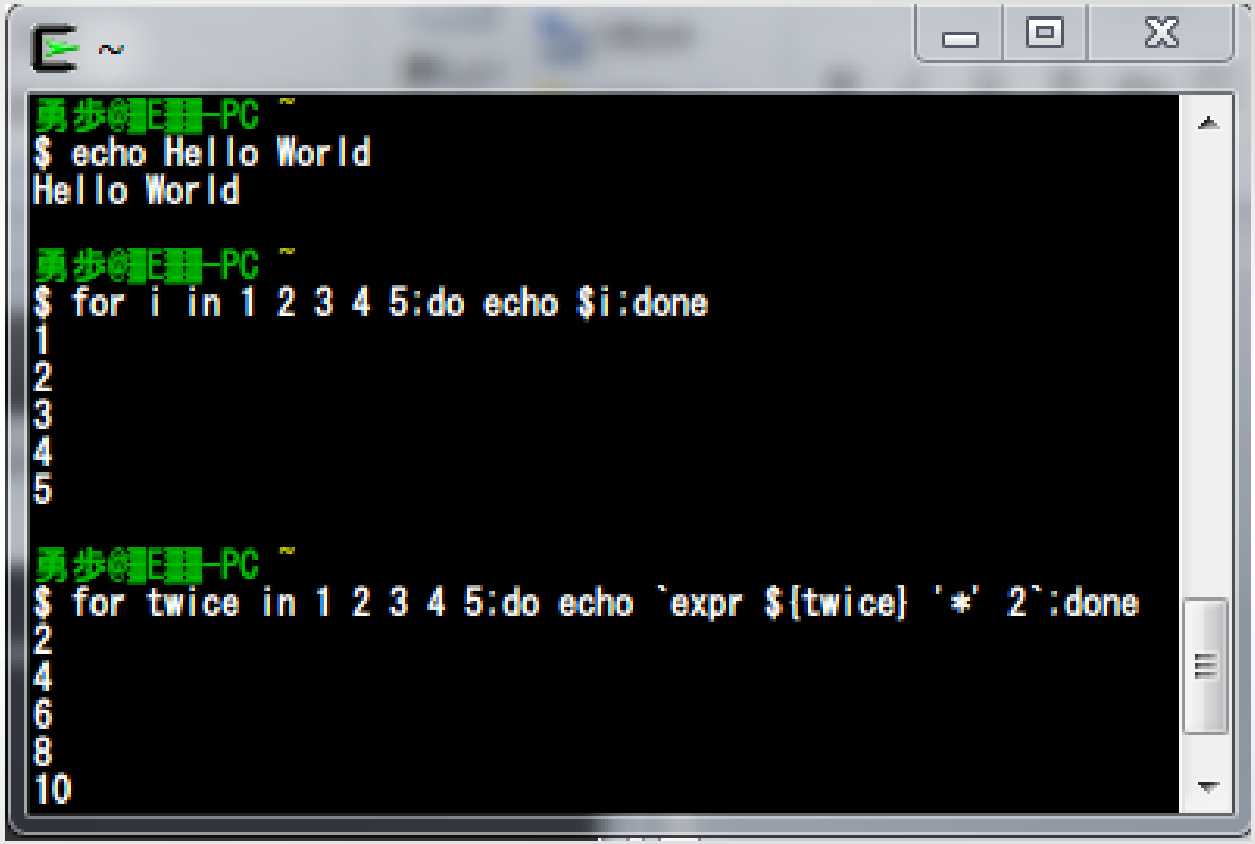
for_test.sh

```
#!/bin/bash
arg_chk()
{
# 引数の数が2の場合
if test $# -eq 2
then
    return 0
# 引数の数が2以外の場合
else
    return 1
fi
}

arg_chk $@
ret=$?
echo "引数チェック結果=${ret}"
```

実行結果

```
$ ./func.sh a
引数チェック結果=1

$ ./func.sh a b
引数チェック結果=0

$ ./func.sh a b c
引数チェック結果=1
```

コマンドラインからShell Script



ちょっとしたものなら、コマンドライン上から実行できる！

コマンドラインからShell Script②

このようなこともできる

```
$ for M in A B C;do for S in {1..5};do echo ${M}_${S};done;done
A_1
A_2
A_3
A_4
A_5
B_1
B_2
B_3
B_4
B_5
C_1
C_2
C_3
C_4
C_5
```

同じ

```
#!/bin/bash
for M in A B C
do
 for S in {1..5}
 do
  echo ${M}_${s}
 done
done
```

gnuplot 編

gnuplotとは

gnuplot
2次元および3次元のグラフを描画するためのフリーウェア

ユーザインターフェース
- CUI（利用者がコマンドを打ち込んでゆく形態）
  ⇒最低限のコマンドをいくつか覚えておくことが必要
- コマンド help で閲覧することができるオンラインマニュアルが充実
  ⇒コマンドさえ覚えてしまえば，高級な使い方をするときに困らない

特徴
- 2次元グラフ描画機能が極めて強力（各種の関数やデータのグラフが自由自在に作成可能）
- 多様な画像の形式をサポート（PostScript,EPS,tgif,PNG,PBMなど）
- 3次元グラフ描画機能は、2次元ほど強力ではない

gnuplotの基本的な使い方

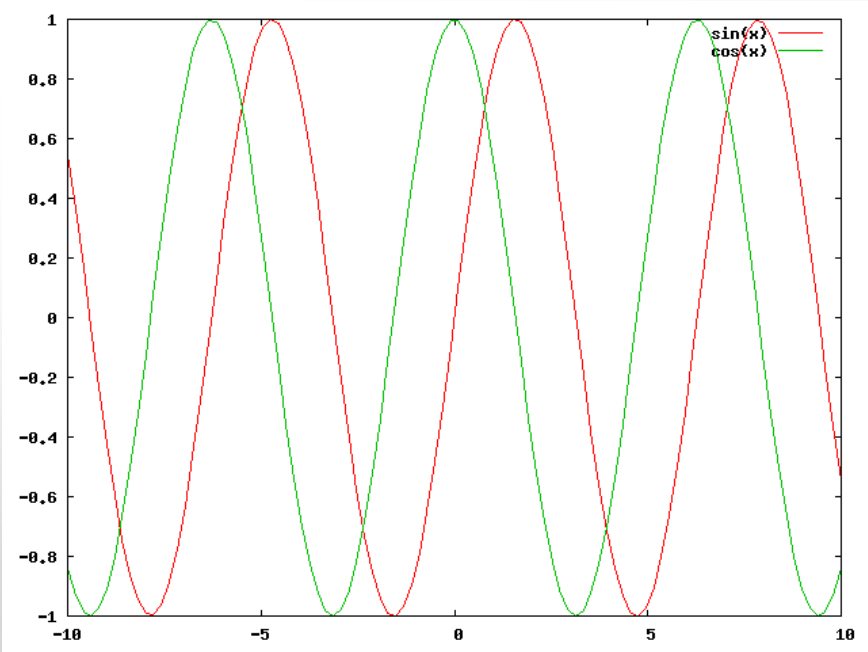
gnuplotを起動し、下記のコマンドを入力

```
$gnuplot
 G N U P L O T
       Version 4.6 patchlevel 0     last modified 2012-03-04
       Build System: CYGWIN_NT-6.1-WOW64 i686

       Copyright (C) 1986-1993, 1998, 2004, 2007-2012
       Thomas Williams, Colin Kelley and many others

       gnuplot home:     http://www.gnuplot.info
       faq, bugs, etc:   type "help FAQ"
       immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'x11'
gnuplot> plot sin(x)
gnuplot> replot cos(x)
gnuplot> replot
```

終了コマンド

```
gnuplot> exit
```

x11で表示される
（入ってない場合はエラー）

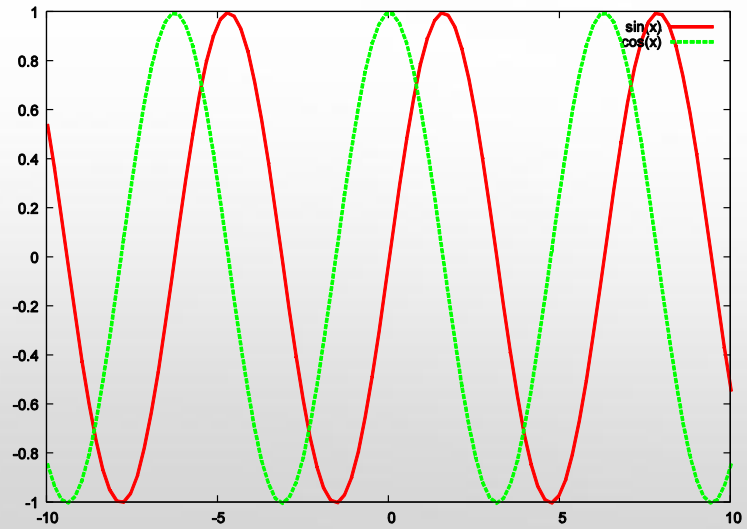


gnuplotの基本的な使い方

```
!#/bin/bash

gnuplot<<EOF
set term postscript eps enhanced color
plot sin(x) lw ${1}
replot cos(x) lw ${2}
set out"${3}"
replot
exit
EOF

echo "作成完了"
echo -n "保存先："
pwd

-u(Unix)--  cloer.sh        All L13     (S
C-; is undefined
```

$1:sin xの線の太さ
$2:cos xの線の太さ
$3:保存するファイル名

```
$./cloer.sh 5 5 image.eps
```



gnuplotの基本的な使い方2 -画像の保存-

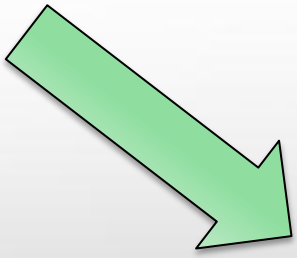
gnuplotを起動し、描写ソフトを指定
（このほかにも色々ある）

```
gnuplot> set terminal eps
```

```
gnuplot> set terminal png
```

初期設定はx11(x11に戻したい場合)

```
gnuplot> set terminal x11
```



```
gnuplot> set out  “名前.拡張子”
gnuplot> replot
```

gnuplotの基本的な使い方3

中でもおすすめはこれ！

```
gnuplot> set terminal dumb
```

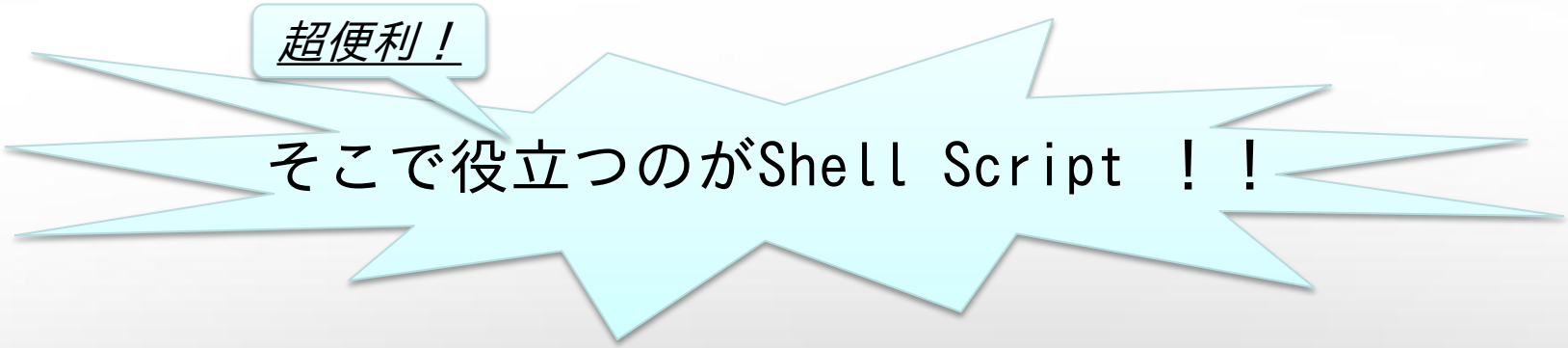
Gnuplotを起動して、`gnuplot> plot sin(x)`

と是非やってみよう！

Shell Script & gnuplot 編

Shell Script と gnuplot

gnuplotはとても便利なソフトだが、毎回毎回コマンドを打ち込んでグラフを作るのは苦痛！！



Shell Script と gnuplot

例えば、$\sin x, \cos x, \tan x$を作りたい場合、あらかじめ Shell Script を使い、作成しておく

```
#!/bin/bash

echo "sin(x)"
gnuplot <<EOF
set terminal postscript eps
plot sin(x)
set out "image_sin${i}.eps"
replot
EOF

echo "cos(x)"
gnuplot <<EOF
set terminal postscript eps
plot cos(x)
set out "image_cos${i}.eps"
replot
EOF

echo "tan(x)"
gnuplot <<EOF
set terminal postscript eps
plot tan(x)
set out "image_tan${i}.eps"
replot
EOF

--(Unix)--  new2.sh        All L1      (Shell-script[bash])-
```

超便利！

これなら毎回コマンド打ち込まずに済み、かつ、
エラーを起こした個所もすぐ修正できる。
また、コマンドを保存しておける。

Shell Script と gnuplotを使ったループ処理

new.sh

new.sh

new.sh 〉 No Selection

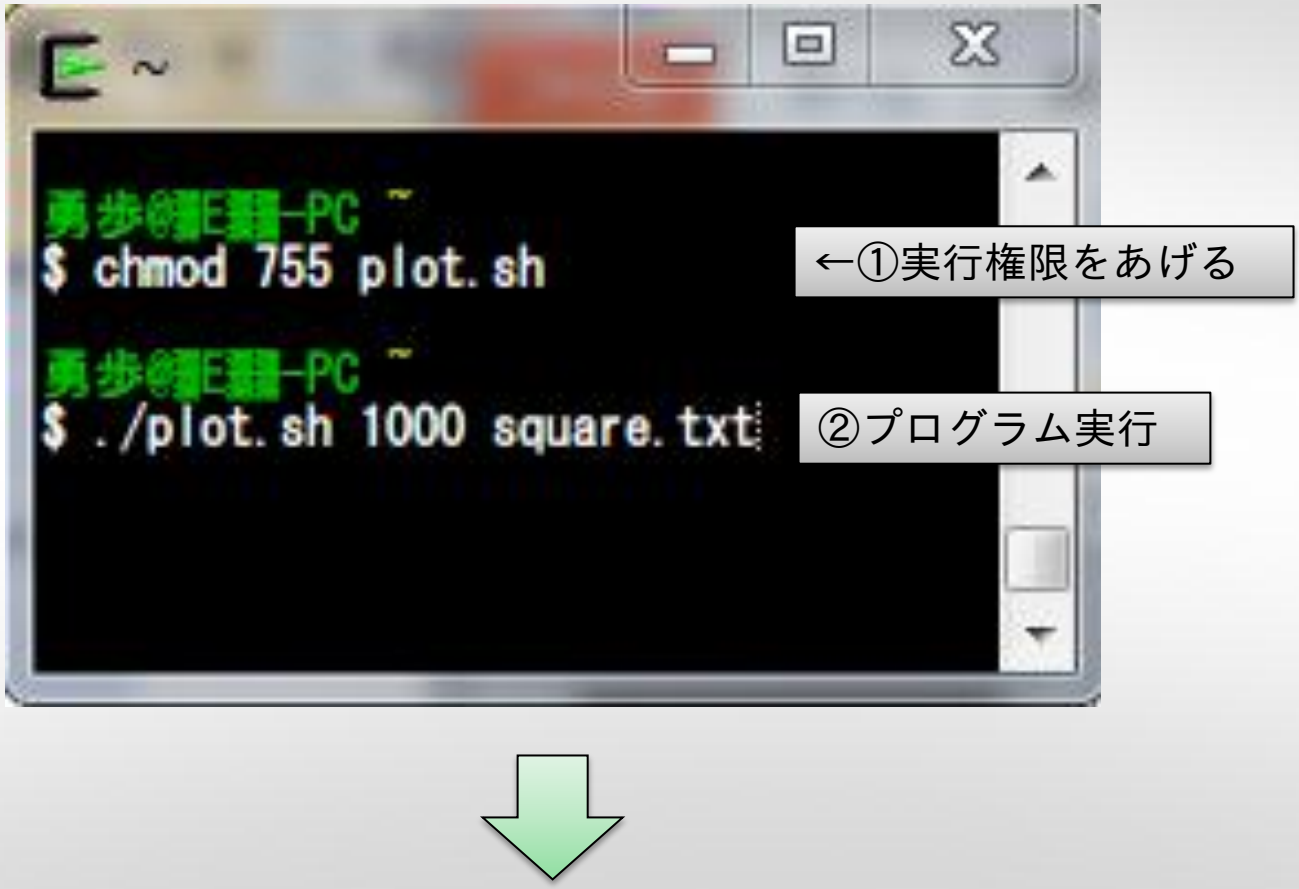
```
#!/bin/bash
#エラーの原因の調査用プログラム

echo "sin(x)"
i=1
while [ $i -le 10 ];
do
    gnuplot <<EOF
    set terminal postscript eps
    plot sin(x)
    set out "image_sin${i}.eps"
    replot
EOF#←これはok
    echo "finish${i}"
    i=`expr ${i} + 1`
done

echo "cos(x)"
i=1
while [ $i -le 10 ];
do
    gnuplot <<EOF
    set terminal postscript eps
    plot cos(x)
    set out "image_cos${i}.eps"
    replot
    EOF#←これはアウト（エラーの原因）
    i=`expr ${i} + 1`
done

echo "tan(x)"
i=1#←ここでエラー
while [ $i -le 10 ]
do
    gnuplot <<EOF
    set terminal postscript eps
    plot tan(x)
    set out "image_tan${i}.eps"
    replot
    EOF
    i=`expr ${i} + 1`
done
```

Shell Script と gnuplotを使えばループ処理も可能

だが、このように書くとエラーを起こすので注意！

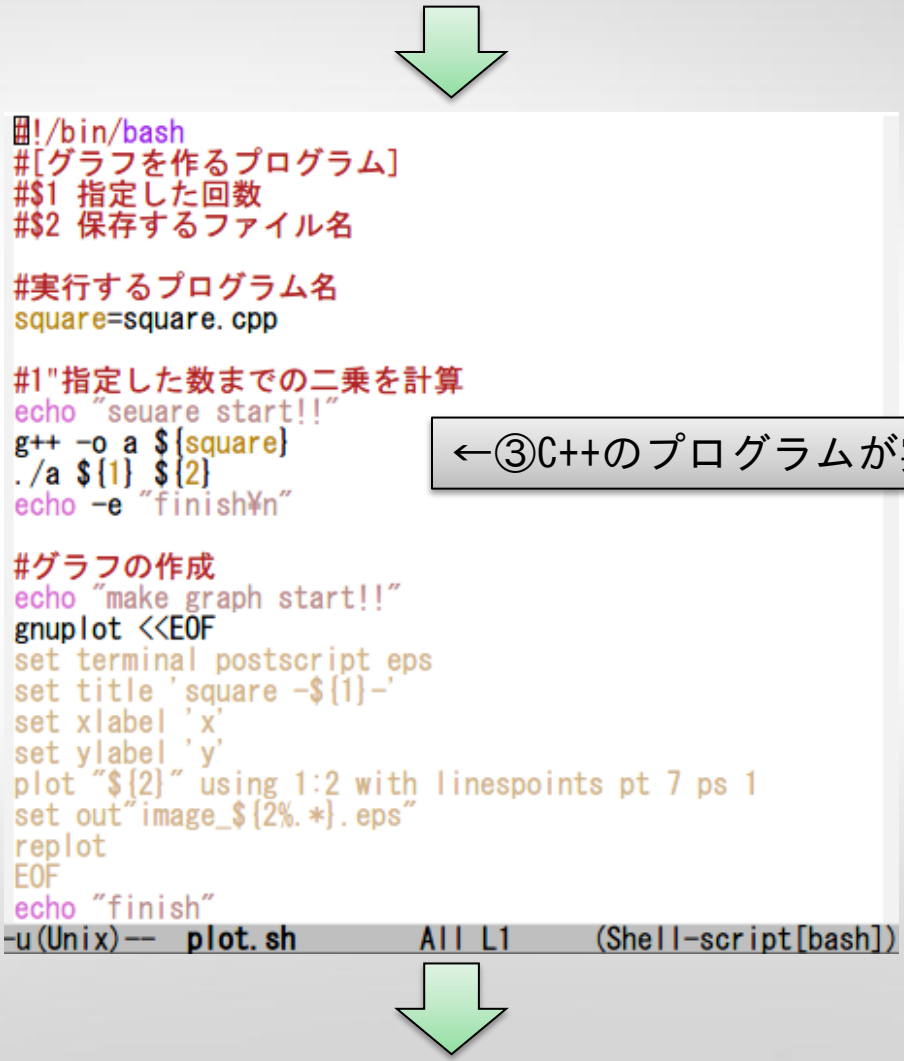
複数のプログラムの起動例(①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム)

Step1



複数のプログラムの起動例(①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム)

Step2



```
#!/bin/bash
#[グラフを作るプログラム]
#$1 指定した回数
#$2 保存するファイル名

#実行するプログラム名
square=square.cpp

#1"指定した数までの二乗を計算
echo "seuare start!!"
g++ -o a ${square}
./a ${1} ${2}
echo -e "finish¥n"

#グラフの作成
echo "make graph start!!"
gnuplot <<EOF
set terminal postscript eps
set title 'square -${1}-'
set xlabel 'x'
set ylabel 'y'
plot "${2}" using 1:2 with linespoints pt 7 ps 1
set out"image_${2%.*}.eps"
replot
EOF
echo "finish"
-u(Unix)--  plot.sh        All L1      (Shell-script[bash])
```

←③C++のプログラムが実行

複数のプログラムの起動例(①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム)

Step3

```
//【1から指定の数までの二乗を計算するプログラム】
#include<iostream>
#include<fstream>//ファイルに書き出す為に必要
#include<algorithm>//atoiを使う為に必要

using namespace std;

void square(int max,string name);
/*コマンドライン引数から入力
  1:指定の数
  2:保存するファイルの名前
*/
int main(int argc,char *argv[]){
  square(atoi(argv[1]),argv[2]);//出力
  return 0;
}
void square(int max,string name){
  ofstream graph;
  graph.open(name.c_str(),ios::app);
  graph<<"#[1""<<max<<"までの二乗]"<<endl;
  graph<<"#x軸の値　y軸の値"<<endl;
  for(int i=1;i<=max;i++){
    graph<<i<<"　"<<i*i<<endl;
  }
  graph.close();
}
```

-u(Unix)-- square.cpp All L1 (C++/l Abbrev)--

④結果がテキストに保存

```
勇歩@[redacted]-PC ~
$ cat square.txt | head
#[1～1000までの二乗]
#x軸の値　y軸の値
1　1
2　4
3　9
4　16
5　25
6　36
7　49
8　64

勇歩@[redacted]-PC ~
$ cat square.txt | tail
991　982081
992　984064
993　986049
994　988036
995　990025
996　992016
997　994009
998　996004
999　998001
1000　1000000
```

複数のプログラムの起動例(①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム)

Step4

```
#!/bin/bash
#[グラフを作るプログラム]
#$1 指定した回数
#$2 保存するファイル名

#実行するプログラム名
square=square.cpp

#1"指定した数までの二乗を計算
echo "seuare start!!"
g++ -o a ${square}
./a ${1} ${2}
echo -e "finish¥n"

#グラフの作成
echo "make graph start!!"
gnuplot <<EOF
set terminal postscript eps
set title 'square -${1}-'
set xlabel 'x'
set ylabel 'y'
plot "${2}" using 1:2 with linespoints pt 7 ps 1
set out"image_${2%.*}.eps"
replot
EOF
echo "finish"
-u(Unix)--  plot.sh        All L1      (Shell-script[bash])
```

←⑤gnuplotが実行

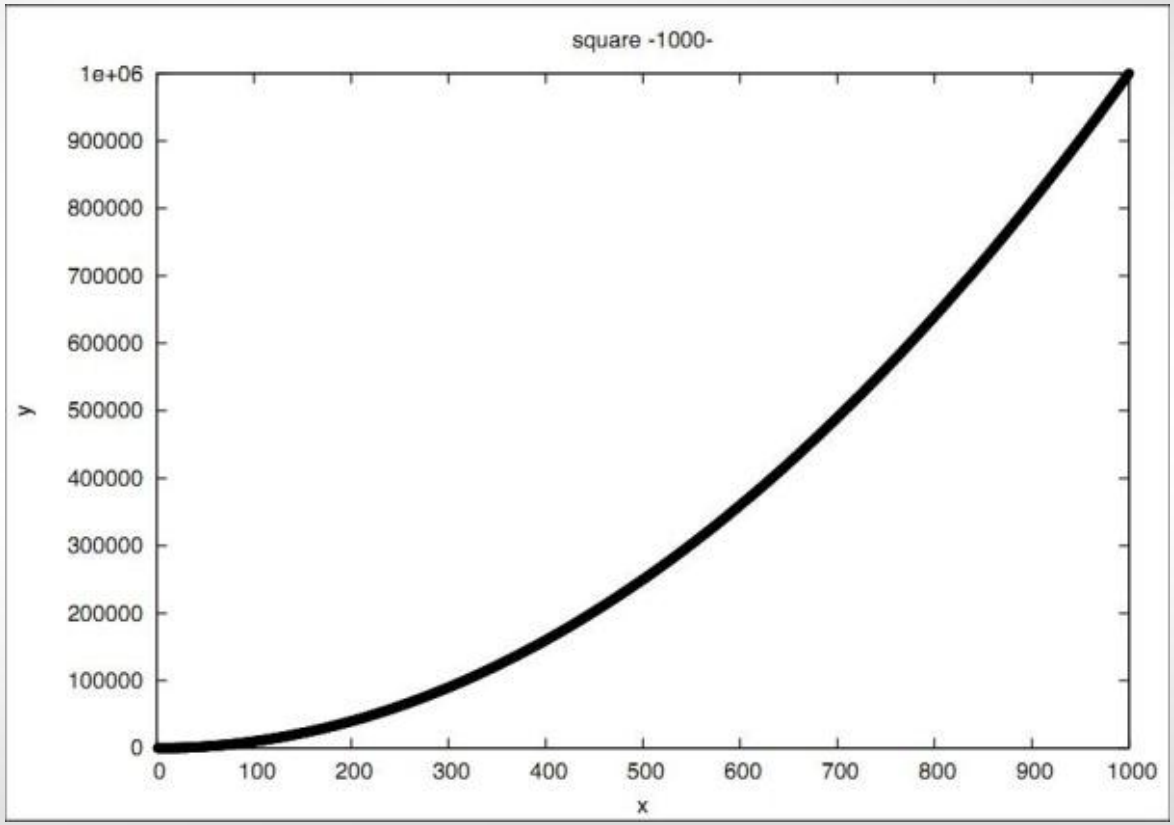
⑥ファイルを読み込む

```
~
勇歩@[redacted]-PC ~
$ cat square.txt | head
#[1～1000までの二乗]
#x軸の値   y軸の値
1   1
2   4
3   9
4   16
5   25
6   36
7   49
8   64

勇歩@[redacted]-PC ~
$ cat square.txt | tail
991   982081
992   984064
993   986049
994   988036
995   990025
996   992016
997   994009
998   996004
999   998001
1000   1000000
```

複数のプログラムの起動例(①1から指定の数までの二乗を計算するプログラム ②ファイルを読み取り,図に保存(eps)するプログラム)

Step5

⑦epsで図に保存



参考にしたWEBサイト（2013.2/10 UTC 9:02）
http://www.k4.dion.ne.jp/~mms/unix/shellscript/index.html