- # 数値計算演習
  2.11.05
  --Make--
- makeやlibraryなどは，compileの手間・時間を省く知識の集大成．一見ややこしいことも，なぜするのかを考えて理解すると筋が通るはず．基本は「あるものは利用せよ」あるいは「二度手間をかけるな」．
- ▼ 課題
  - 巨大なプログラムも簡単なコードの寄せ集め．先ずは簡単なコードでheader，include，library，makeなどの基本を理解せよ．覚える必要はない．．．
  - ▼ 【課題１】
    - 以下のコマンドを適当に打ち込んで理解せよ．
  - ▼ 【課題２】
    - ▼ 以下の関数の解を求める2分法のプログラムを作成し，実働を確かめよ．
      - x^2-4X+1=0
  - ▼ 【課題３】
    - 関数(func)を分割し，archiveにしてlibraryをincludeしてcompileせよ．またそれらの作業を自動化するMakefileを作れ．
  - ▼ 【課題４】
    - Newton法も同様の作業をおこなえ．2分法と結果を比較せよ．
  - なお，2分法，Newton法の簡単なプログラムを後ろに示した．また，2分法，Newton法の原理の理解には数値計算のテキスト
    http://ist.ksc.kwansei.ac.jp/~nishitani/Lectures/NumRecipe/fsolve.pdf
    を参照せよ．
- ▼ header fileのinclude
  - 関数が増えてくると，その宣言をいちいちするのが面倒になる．header fileは関数宣言を一括しておこなうことができる．header fileにちゃんと宣言しておけば，それをincludeするだけで宣言部を省略することができる．<stdio.h>などもheader file.
  - ▼ 先ずは普通のプログラム．
    - [BobsNewPBG4:~/Desktop/test2] bob% gcc prog.c

```
[BobsNewPBG4:~/Desktop/test2] bob% ./a.out
Hello world!!
[BobsNewPBG4:~/Desktop/test2] bob% cat
prog.c
#include <stdio.h>

int main(void){
  printf("Hello world!!\n");
  exit(0);
}
```

- 関数を沢山持った場合.
  - [BobsNewPBG4:~/Desktop/test2] bob% cat

```
prog.c
#include <stdio.h>
void f1(void);
void f2(void);

int main(void){
  f1();
  f2();
  exit(0);
}

void f1(void){
  printf("Hello world from f1!!\n");
  return;
}

void f2(void){
  printf("Hello world from f2!!\n");
  return;
}
[BobsNewPBG4:~/Desktop/test2] bob% gcc
prog.c
[BobsNewPBG4:~/Desktop/test2] bob% ./a.out
Hello world from f1!!
Hello world from f2!!
```

- header fileを使った場合.
  - [BobsNewPBG4:~/Desktop/test2] bob% cat

```
prog.c
#include "prog.h"

int main(void){
  f1();
  f2();
  exit(0);
}

void f1(void){
  printf("Hello world from f1!!\n");
  return;
}

void f2(void){
  printf("Hello world from f2!!\n");
  return;
}
[BobsNewPBG4:~/Desktop/test2] bob% gcc
prog.c
[BobsNewPBG4:~/Desktop/test2] bob% ./a.out
Hello world from f1!!
Hello world from f2!!
```

- [BobsNewPBG4:~/NumRecipeEx05/Make] bob% cat prog.h

```
#include <stdio.h>
void f1(void);
void f2(void);
```

▼ 分割コンパイル

- コードを２つにわけた場合．prog.hは前と同じ．

▼ [BobsNewPBG4:~/NumRecipeEx05/Make] bob% cat prog.c

- #include "prog.h"

```
int main(void){
  f1();
  f2();
  exit(0);
}
```

```
void f1(void){
  printf("Hello world from f1!!\n");
  return;
}
```

- [BobsNewPBG4:~/NumRecipeEx05/Make] bob% cat prog2.c
  - ```
    #include "prog.h"

    void f2(void){
      printf("Hello world from f2!!\n");
      return;
    }
    ```
- コンパイルと実行.
  - ```
    [BobsNewPBG4:~/NumRecipeEx05/Make] bob% gcc prog.c prog2.c
    [BobsNewPBG4:~/NumRecipeEx05/Make] bob% ./a.out
    Hello world from f1!!
    Hello world from f2!!
    ```

- make
  - 実際のexec file(実行ファイル)の作成は,
    source(*.c) -> objects(*.o) -> exec(a.out)
    という手順になっている.
    gcc -c prog.cでprog.oが
    gcc -o prog prog.o prog2.oでprogと名付けたexec fileができる.
  - これを自動的にやってくれるのが, make. Makefileにその手順を記述する. タブとspaceが違うので注意.
    - ```
      [BobsNewPBG4:~/Desktop/test2] bob% cat Makefile
      prog: prog.o prog2.o
              gcc -o prog prog.o prog2.o

      prog.o: prog.c prog.h
              gcc -c prog.c

      prog2.o: prog2.c prog.h
      ```
      ```

```
gcc -c prog2.c
```

- makeはMakefileを参照しながら，必要最低限のcompileを実行する．全部のファイルを再compileするには，touch *.cでディレクトリー内のcファイルの最終変更時間を最新にする．
- [BobsNewPBG4:~/NumRecipeEx05/Make] bob% touch *.c
  [BobsNewPBG4:~/NumRecipeEx05/Make] bob% make
  gcc -c prog.c
  gcc -c prog2.c
  gcc -o prog prog.o prog2.o
  [BobsNewPBG4:~/NumRecipeEx05/Make] bob% ./a.out
  Hello world from f1!!
  Hello world from f2!!

- .cや.hの依存関係はgcc -MMで作成可能．
  - [BobsNewPBG4:~/NumRecipeEx05/Make] bob% gcc -MM prog.c prog2.c
    prog.o: prog.c prog.h
    prog2.o: prog2.c prog.h

- autoconf，automakeで，より複雑なMakefileを作成可能．

- archive
  - よく使う関数をまとめておくのにライブラリが使われる．
  - ar
    - -- create and maintain library archives
    - ライブラリアーカイブの作成と維持．
    - ar -r libfile.a prog2.o
    - r(置換)，d(削除)，q(追加)，x(取出)，t(リスト表示)
  - ranlib(必要がない場合もある)
    - 高速リンクのためのtable __.SYMDEF SORTED を作成．
    - ranlib libfile.a
    - ar -sでも同じ
    - nm libfile.a

でテーブルの中身を見ることができる.

- libを使ったコンパイル
  - gcc -o prog prog.o libfile.a
- makeによる自動化
  - [BobsNewPBG4:~/NumRecipeEx05/Make] bob% cat Makefile
    - ```
      CC = gcc
      LIB=libfile.a

      prog: prog.o $(LIB)
              $(CC) -o $@ prog.o -L. -lfile

      prog.o: prog.c prog.h
              $(CC) -c $(CFLAGS) $*.c

      prog2.o: prog2.c prog.h
              $(CC) -c $(CFLAGS) $*.c

      $(LIB): prog2.o
              ar -r $(LIB) prog2.o
              ranlib $(LIB)
      ```
    - ```
      CC = gcc
      OBJS = prog.o prog2.o
      LIB=libfile.a
      LIBO = prog2.o
      CFLAGS = -Wall -g
      #CFLAGS = -O3

      prog: $(OBJS) $(LIB)
              $(CC) -o $@ $(CFLAGS) $(OBJS) $
      (LIB)

      prog.o: prog.c prog.h
              $(CC) -c $(CFLAGS) $*.c

      prog2.o: prog2.c prog.h
              $(CC) -c $(CFLAGS) $*.c

      $(LIB): $(LIBO)
              rm -f $(LIB)
      ```

```
ar -r $(LIB) $(LIBO)
ranlib $(LIB)
```

- [BobsNewPBG4:~/NumRecipeEx05/Make] bob% make
  gcc -c -Wall -g prog.c
  prog.c: In function `main':
  prog.c:6: warning: implicit declaration of function `exit'
  gcc -c -Wall -g prog2.c
  rm -f libfile.a
  ar -r libfile.a prog2.o
  ar: creating archive libfile.a
  ranlib libfile.a
  gcc -o prog -Wall -g libfile.a prog.o prog2.o

▼ Path(パス，経路)

- コマンドやライブラリ，manなど特殊なファイルが置かれているディレクトリを探しやすいように，あらかじめ明示的に宣言することに相当します．代表的なのがコマンドサーチパスで，環境変数PATHで設定します．

▼ 設定されている環境変数の表示

- setenv
- echo $PATH

▼ パスの設定

▼ 使っているシェルによって設定の仕方が違います．何を使っているかは以下のコマンド．

- [BobsNewPBG4:~/XML] bob% echo $SHELL
  /bin/tcsh

▼ csh, tcsh

- setenv PATH ~/bin:$PATH

▼ bash

- PATH=~/bin:$PATH
  export PATH

▼ コマンド，ライブラリの確認

- which, whereis

▼ ld

- オブジェクトとライブラリのリンク(結合).
  gcc -lm Coin.c
  とかの-lmというのは何というのをよく聞かれる

が，それは以下の通り．

- -l(library_name)
- lib(library_name).aとなっている
- ▼ ライブラリのサーチパス(捜すdirectory)は-Lで明示．ここの例では，
  - gcc -o prog prog.o -L. -lfile
  - -L. -lfileなどはprog.oより後にないとだめ．
- -lmとはどこかにあるlibm.aというライブラリをリンクしなさいという命令．
- ▼ .aと.so
  - .aは静的なライブラリで，exec fileの中に取り込まれる．一方，.soは動的なライブラリで，実行時にシステムによって自動的に取り込まれる．実行時に*.soがありませんというエラーが出たときには，*.soに明示的にpathを通す必要がある．

▼ 2分法，Newton法の簡単なプログラム例

- ▼ 2分法(Bisection method)
  -
```
#include <stdio.h>
double func(double x);

int main(void){
  double x1,x2,f1,f2,f,x;
  int i,i_max=20;
  x1=0.0;
  x2=0.8;
  f1=func(x1);
  f2=func(x2);
  for (i=0;i<i_max;i++){
    x=(x1+x2)/2.0;
    f=func(x);
    if (f*f1>=0.0){
      x1=x;
      f1=f;
    } else {
      x2=x;
      f2=f;
    }
    printf("%4d %20.15f %20.15f\n",i,x,func
(x));
  }
```

```
  exit(0);
}

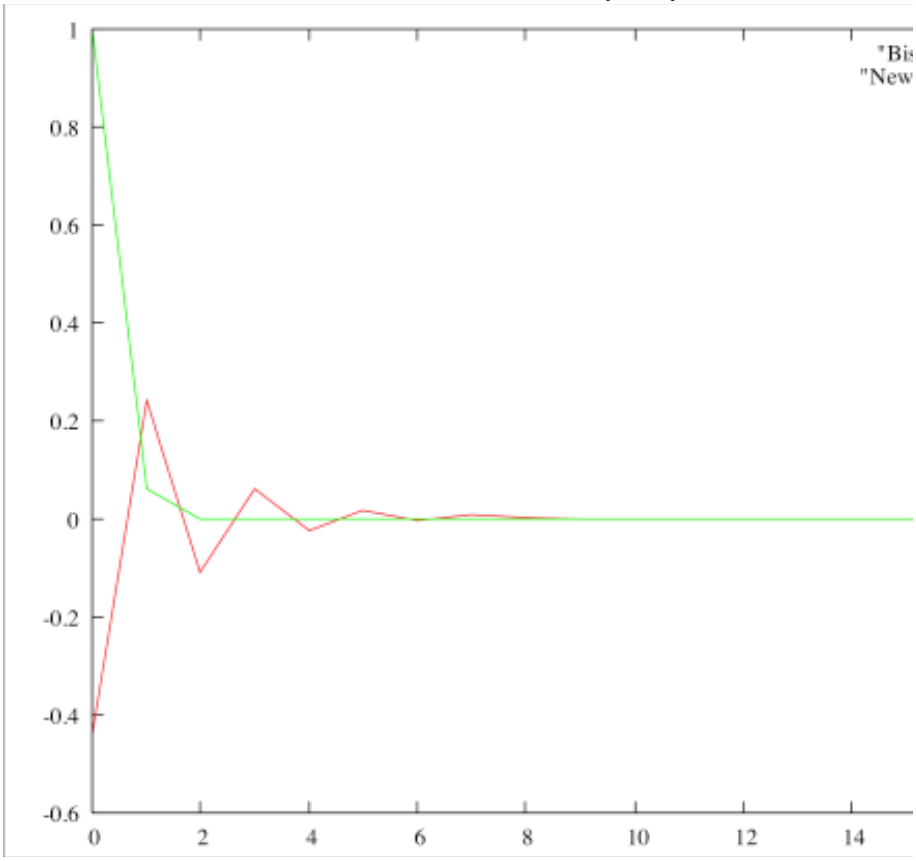
double func(double x){
  return x*x-4.0*x+1;
}
```

- Newton法
  -
```
#include <stdio.h>
double func(double x);
double dfunc(double x);

int main(void){
  double f,x;
  int i,i_max=20;
  x=1.0;
  f=func(x);
  for (i=0;i<i_max;i++){
    x=x-f/dfunc(x);
    f=func(x);
    printf("%4d %20.15f %20.15f\n",i,x,func
(x));
  }
  exit(0);
}

double func(double x){
  return x*x-4.0*x+1;
}
double dfunc(double x){
  return 2.0*x-4.0;
}
```

- gnuplot
  -
```
gnuplot> plot "Bisect.res" using 1:3 with
lines,\
> "Newton.res" using 1:3 with lines
```


"Bis
"New
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
2
4
6
8
10
12
14