Lab 5

Ai Nguyen

December 4, 2015

Lab to explore higher-order functions and more practice using floating-point instructions. The diagrams to illustrate the idea of the bisection algorithm are done using gnuplot.

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

$$bisection : ((\mathbb{R} \times \mathbb{R}) \times \mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$$

$$bisection(f, l, h) = \begin{cases} m, & \text{if} |f(m)| < \epsilon \\ m, & \text{if} |h - l| < \delta \\ bisection(f, l, h), & \text{if} f(m) < 0 \\ bisection(f, l, m), & \text{if} f(m) \geq 0 \\ & \text{where m } = \frac{l+h}{2} \end{cases}$$

This is the commands for plotting Fig.1

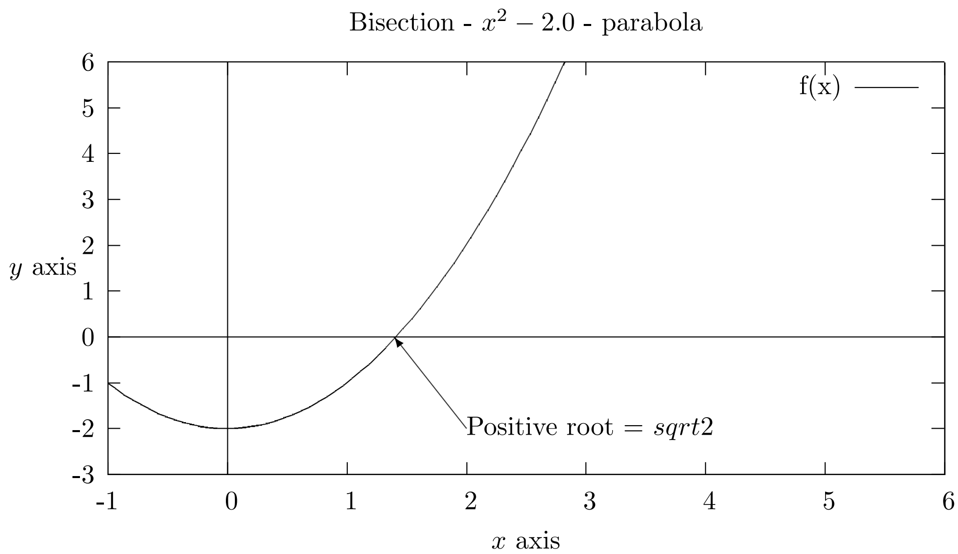
Text written to file bisectionfig1.gp

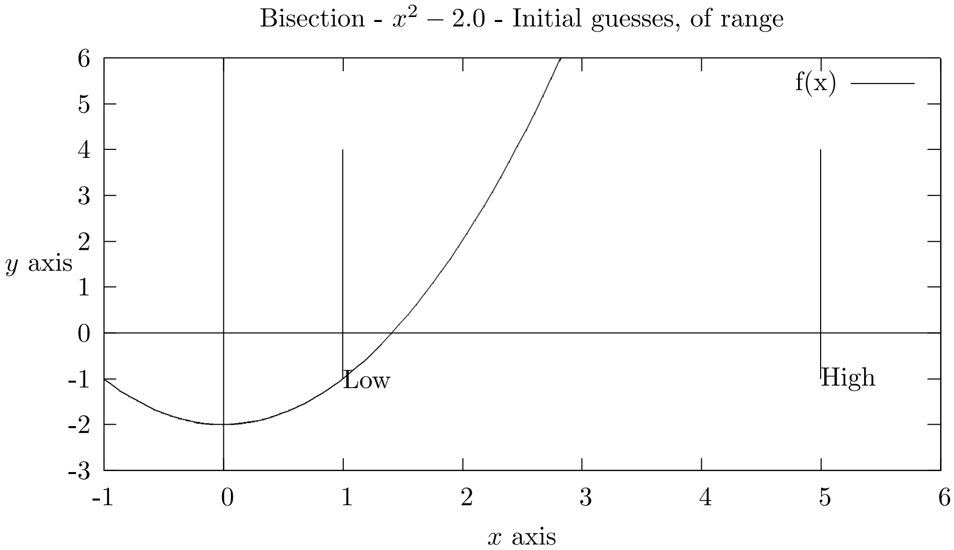
```
set terminal latex size 5.0, 3.0
set title "Bisection - $x^2 - 2.0$ - parabola"
set xlabel "$x$ axis"
set ylabel "$y$ axis"
set output "bisectionfig1.tex"
set xzeroaxis
set yzeroaxis
set label "Positive root = $\sqrt{2}$" at 2,-2
set arrow from 2,-2 to 1.4,0
f(x) = x*x -2
plot [-1:6][-3:6] f(x)
```

This is the commands for plotting Fig.2

Text written to file bisectionfig2.gp

```
set terminal latex size 5.0, 3.0
set title "Bisection - $x^2 - 2.0$ - Initial guesses, of range"
set xlabel "$x$ axis"
set ylabel "$y$ axis"
set output "bisectionfig2.tex"
set xzeroaxis
set yzeroaxis
set label "High" at 5,-1
set label "Low" at 1, -1
set arrow from 5,-1 to 5,4 nohead
set arrow from 1,-1 to 1,4 nohead
f(x) = x*x - 2.0
plot [-1:6][-3:6] f(x)
```


Bisection - $x^2 - 2.0$ - parabola
f(x)
y axis
6
5
4
3
2
1
0
-1
-2
-3
Positive root = $sqrt2$
-1
0
1
2
3
4
5
6
x axis


Bisection - $x^2 - 2.0$ - Initial guesses, of range
f(x)
Low
High
y axis
x axis
6
5
4
3
2
1
0
-1
-2
-3
-1
0
1
2
3
4
5
6

1 SML

Here is an implementation of the bisection algorithm in SML. This is a higher-order function since one of its parameters is another function. The algorithm works the same no matter what function we are trying to find the root for (as long as it is increasing).

SML

```
val epsilon = 0.00000001;
val delta = 0.000000001;
fun bisection f low high =
        let
                val mid = (low + high) / 2.0;
                val image = f mid
        in
                if abs image < epsilon then mid
                else if abs (high - low) < delta then mid
                else if image < 0.0 then bisection f mid high
                                        else bisection f low mid
        end;
```

Test the SML bisection function

SML

```
fun parabola x = x*x -2.0;
bisection parabola ~1.0 5.0;
```

This calculates the positive root $\sqrt{2}$

```
debian@debian:~/labs/lab5$ poly < lab5.sml
Poly/ML 5.5.2 Release
val epsilon = 1E~8: real
val delta = 1E~9: real
val bisection = fn: (real -> real) -> real -> real -> real
val parabola = fn: real -> real
val it = 1.414213564: real
```

2 C Code

We can implement this in C

C source code written to file lab5.c

```c
#include <stdio.h>
#include <math.h>
const double epsilon = 0.00000001, delta = 0.00000001;
double parabola(double x) {return x*x -2.0;}
double bisection(double f(double),double low, double high)
{
        double r;
        double mid = (low+high)/2.0;
        double image = f(mid);
        if(fabsl(image) < epsilon) return mid;
        else if( fabsl(high - low) < delta) return mid;
        else if (image < 0.0) r = bisection(f,mid,high);
        else r = bisection(f,low,mid);
        return r;
}
int main()
{
        printf("%f\n", bisection(parabola,-1.0,15.0));
}
```

```
debian@debian:~/labs/lab5$ ./lab5c
1.414214
```

3 ASM Code

3.1 Parabola Function

asm source code written to file lab5.s

```
.data
two: .float 2.0
.text
parabola:
push %ebp
mov %esp, %ebp
fld 8(%ebp)             # st0 = x
fmul 8(%ebp)            # st0 = x^2
fsub two                # st0 = x^2 - 2.0
mov %ebp, %esp
pop %ebp
ret
```

8(%ebp) is your parameter so *fld 8(%ebp)* will push your parameter to FPU stack. At this point, $st0 = parameter$.

Next *fmul 8(%ebp)* will multiply st0 by your parameter, which means $st0 = parameter^2$.

Then *fsub two* will subtract 2.0 from st0, and $st0 = parameter^2 - 2.0$. We can't do *fsub 2* because the operand for *fsub* just is memory address or register only, that's why we have to create varible *two* to store float value 2.0.

3.2 Bisection Function

This is the data used in bisection function. The same reason with variable *two* why we create variable *zero* (FPU instruction operand is memory address and register only). We create variable ST0 for storing value from *%st(0)*, and also for popping unnesscary values in FPU stack.

asm source code appended to file lab5.s

```
.data
epsilon: .float 0.000001
delta: .float 0.000001
zero: .float 0.0
ST0: .float 0.0
```

Start of bisection function.It will take 3 parameters:

parabola 6(%ebp)

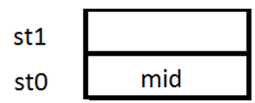
low 12(%ebp)

high 8(%ebp)

asm source code appended to file lab5.s

```
.text
bisection:
push %ebp
mov %esp, %ebp
fld 8(%ebp)          #st0 = high
fadd 12(%ebp)        #st0 = high + low
fdiv two             #st0 = (high+low)/2
```

First, we create variable $mid = \frac{high+low}{2}$. We push *high* onto FPU stack, so $st0 = high$ Then add *low* to st0, and divide it by 2. You stack at this time :



asm source code appended to file lab5.s

```
fst ST0
push ST0
mov 16(%ebp), %eax
call *%eax
add $4,%esp
```

The code above creates varible $image = f(mid)$. We store value from *%st0 = mid* to variable ST0 by using *fst ST0*, then push ST0 onto stack for calling function.
We passed memory adress of function parabola to parameter for bisection function: $\$parabola = 16(\%ebp)$. That's why we copy *16(%ebp)* into EAX for using *call *%eax*, which means calling function with address stored in EAX.

After calling function parabola, FPU stack will look like this:

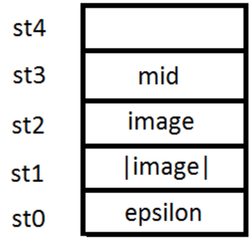
Then we will move to first condition:

if(fabsl(image) <epsilon) return mid;

asm source code appended to file lab5.s

```
fld %st(0)                # st0 = image
fabs                      # st0 = |image|
fld epsilon               # st0 = epsilon
fcomip                    # compare st0 and st1
#pop |image| out, st0 =image
fstp ST0                  #st0 = image, st1= mid
ja _end                   # if st0 > st1
```

Since we need *image* for later use, we push another *image* onto FPU stack. Then *fabs* will take absolute of st0.
To compare floating point number, we use *fcomi* to compare st0 and st1 (st0 will be in left side of expression).
That's why we push *epsilon* to FPU stack.



ja _end means that if st0 >st1, we jump to *_end* label (end of function).

But after comparing, we don't need *epsilon* on FPU stack anymore; hence, we use *fcomip* instead of *fcomi* to pop st0 after comparing $\Rightarrow$ $st0 = |image|$.

We also don't need $|iamge|$ on stack, so we pop it out (store to ST0 for temporary). FPU stack after cleaning:

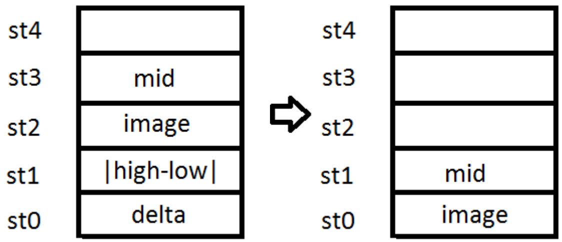
Move to second condition:

if(fabsl(high-low) <delta) return mid;

asm source code appended to file lab5.s

```
fld 8(%ebp)            #st0 = high
fsub 12(%ebp)          #st0 = high − low
fabs                   #st0 = |high − low|
fld delta              #st0 = delta
fcomip
#pop |high −low|, st0 =image
fstp ST0               #st0 = image, st1= mid
ja _end                # if st0 > st1
```

We do the same thing for first condition, push $|high - low|$ instead of $|image|$. Remind you that *8(%ebp) = high, 12(%ebp) = low*. Process of stack changing:



Move to last two conditions:

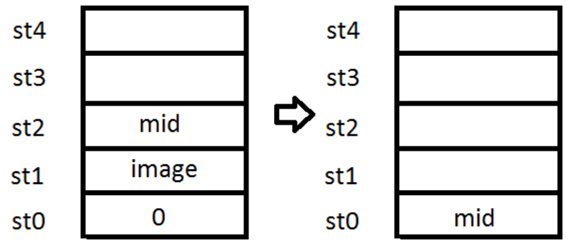
if (image <0.0) r = bisection(f,mid,high);

else r = bisection(f,low,mid);

asm source code appended to file lab5.s

```
fld zero          # st0 = 0
fcomip
fstp ST0                   # pop image, st0 = mid
ja _if                     # if image < 0
```

The same thing above, to compare with 0, we push 0 onto stack for using *fcomip*. After compare, 0 is popped out. At this point, we need to clear *image* as well, since we don't need it for later ⇒ Doing *fstp ST0* again.



This is *else* part for calling

bisection(f,low,mid)

asm source code appended to file lab5.s

```
push 16(%ebp)        # parabola
push 12(%ebp)        # low
fstp ST0
push ST0             # mid
call bisection
add $12, %esp
jmp _end
```

This is *if* part for calling

bisection(f,mid,high)

asm source code appended to file lab5.s

```
_if:
push 16(%ebp)        #parabola
fstp ST0             #mid
push ST0
push 8(%ebp)         #high
call bisection
add $12, %esp
```

Since our FPU stack for last time has $st0 = mid$, when we do *fstp ST0*, $ST0$ will be equal to mid, then we can go ahead push it on stack for using as parameter. *fstp* is used instead of *fst* because we don't need *mid* after calling bisection. At this moment, our FPU stack is clear totally.

This is the *_end* label to end function calling.

asm source code appended to file lab5.s

```
_end:
mov %ebp, %esp
pop %ebp
ret
```

3.3 Main function

Datas used for main only

asm source code appended to file lab5.s

```
.data
high: .float 15.0
low: .float -1.0
fmt: .string "The root for function f is : %f\n"
```

Start of program, call bisection with 3 parameters: address of parabola, low, high.

asm source code appended to file lab5.s

```
.text
.globl _start
_start:
push $parabola
push low
push high
call bisection
add $12, %esp
fstp ST0
```

Bisection is a recursive function, and remember that the last time of bisection calling inside bisection function will fall into situation either $|image| < epsilon$ or $|high - low| < delta$, which means that after call bisection, our stack:

So we need to pop *image* out because *mid* is the final answer we need.

Since variabel ST0 is just 32-bit and printf need 64-bit floating point, we can't pop value in $st0$ to ST0. Instead, we *add $-8, %esp* to create room for 64-bit floating point number that we pop from $st0$ by using $fstpl$ (that's why we *add $12, %esp* in the end.)

asm source code appended to file lab5.s

```
add $-8, %esp
fstpl (%esp)
push $fmt
call printf
add $12, %esp
```

End main function.

asm source code appended to file lab5.s

```
mov $1, %eax
mov $0, %ebx
int $0x80
```

Testing result for ASM code

```
debian@debian:~/labs/lab5$ ./lab5asm
The root for function f is : 1.414214
```

4 Create build.sh file

This file is used for building gnuplot and making lab5 pdf. Run *./build.sh* will do all these code in terminal

Text written to file build.sh

```
docsml lab5.doc
gnuplot bisectionfig1.gp
gnuplot bisectionfig2.gp
doctex lab5.doc
pptexenv pdflatex lab5.tex
```

This command make *buid.sh* executable.

Bourne Shell

```
chmod 777 build.sh
```