プログラミング基礎演習レポート 2016-2

電気電子工学科 2 年　541046J　高瀬健吾

１．導入

Plus[Times[Sin[13.4],3],2]

Plus[Power[x,3],Sin[x]]

上のような表式で与えられる式もしくは一変数関数を評価するプログラム report.c を実装した。実行時引数で与えられた表式を木構造化し、木に対して微分操作を行って新たに微分後の木を作り、結果を二階微分まで同様の形式で表示する。式が与えられた場合は具体的な値を、関数が与えられた場合その局所的な最大最小を画面に出力する。認識可能な基本関数は四則演算、三角関数、指数対数であり、定義域、値域は共に実数である。

２．手法・結果

・課題１

表式を木構造化するにあたって、int operation、double number、struct node *left、struct node *right を要素に持つ構造体 node を定義した。表式と(int) operation、(double) number の対応は以下のとおりである。

式	operation	number
Plus[]	1	0
Times[]	2	0
Subtract[]	3	0
Divide[]	4	0
Sin[]	5	0
Cos[]	6	0
Exp[]	7	0
Log[]	8	0
Power[]	9	0
x(変数)	10	0
n(実数)	0	n(実数)

まず木構造と表式の変換を行う関数 struct node *create_node(int *pos,char *s)と void traverse(struct node *p)を実装した。第 11 回で実装したものとほぼ同じだが、今回は演算、数字ともに 1 文字ではないため、double read_number(int *pos,char *str)、int read_operation(int *pos,char *str)を用いて文字列の読み込みと読み込み位置 pos の移動を行っている。

さらに、int iffunc(struct node *p)で木構造化された式が x の関数であるかどうか判定する。返り値は木構造内に格納された変数 x の個数とした。ある木構造 p に対して iffunc(p)が 0 のとき、つまり p が x によらない定数であるときは double calculate_number(struct node *p)で具体的な値を返す。

実行例は以下の通り。

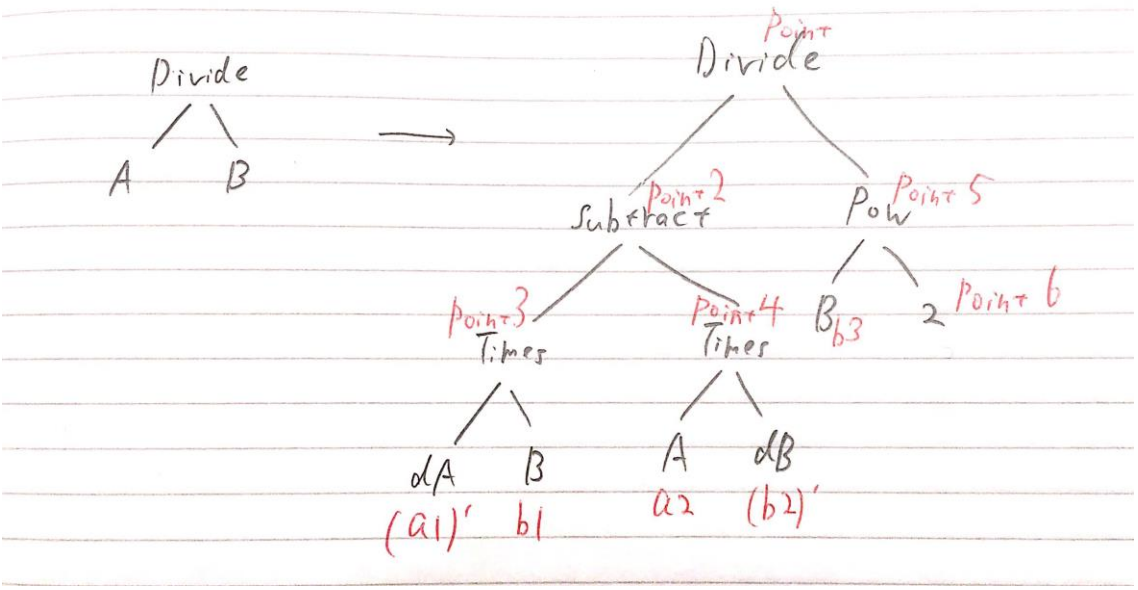
```
takase@kengo ~/C/proen/report2
$ ./a.exe Plus[Times[Sin[13.4],3.0],2.0] -2
f(x) = Plus[Times[Sin[13.4],3.0],2.0] = 4.221128
```

・課題２

木構造化した式に対して微分操作を行い、新たに作った微分後の木構造のポインタを返す関数 struct node *differentiate_node(struct node *p)を実装した。引数となる元の木構造を破壊しないために木構造を複製する struct node* copy_node(struct node *p)をあわせて実装している。ここで p->operation ごとの変換規則は以下のとおりとした。

$$(f+g)' = f' + g'$$
$$(fg)' = f'g + fg'$$
$$(f-g)' = f' - g'$$
$$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$
$$(\sin f)' = (\cos f)f'$$
$$(\cos f)' = -(\sin f)f'$$
$$(e^f) = (e^f)f'$$
$$(\log f)' = \frac{f}{f'} \qquad (f > 0, f' \neq 0)$$
$$(f^g)' = \begin{cases} 0 & (g = 0) \\ gf^{g-1}f' & (g \neq 0, g\text{は定数}) \\ f^g\left(g'\log f + g\dfrac{f'}{f}\right) & (g\text{は}x\text{の関数}) \end{cases}$$
$$x' = 1$$
$$n' = 0$$

微分して新たにできる基本関数それぞれに対して構造体を割り当てる必要があるため、コードの可読性が著しく下がってしまった。例として p->operation が 4 の時のコードの対応を挙げる。



このままでは表式が煩雑になりすぎるので、変数 x を持たない部分木を計算して新たに枝分れを減らした木構造を返す関数 struct node *optimise_node(struct node *p)もあわせて用いた。optimise_node には Times[0,f]=0 とする機能もつけた関係で、微分後の木に対して二重に optimise_node を使っている。

微分して得られた木構造は traverse で表示する。

実行例は以下の通り。

```
takase@kengo ~/C/proen/report2
$ ./a.exe Plus[Power[x,3],Sin[x]] -2
f(x) = Plus[Power[x,3.0],Sin[x]]
f'(x) = Plus[Times[Times[3.0,Power[x,2.0]],1.0],Times[Cos[x],1.0]]
f''(x) = Plus[Plus[Times[Plus[0.0,Times[3.0,Times[Times[2.0,Power[x,1.0]],1.0]]],1.0],0.0],Plus[Ti
mes[Times[Times[-1.0,Sin[x]],1.0],1.0],0.0]]
```

・課題３

与えられた関数が 0 になるような x を、ニュートン法を用いて求める関数 double search(double x0,struct node *df,struct node *dff)を実装した。ここで x0 は探索を始める初期値であり、実行時引数から与える。search 内での df(x),ddf(x)の具体的な計算は、ある木構造に対して x を代入した木構造を返す struct node *substitute(struct node *p,double x)と calculate_node(p)を用いている。

最後に ddf(x)の正負を見て f が局所最大か局所最小かを判断して画面に表示した。

実行例は以下の通り。

```
takase@kengo ~/C/proen/report2
$ ./a.exe Plus[Exp[Times[-1,x]],Power[x,2]] -2
f(x) = Plus[Exp[Times[-1.0,x]],Power[x,2.0]]
f'(x) = Plus[Times[Exp[Times[-1.0,x]],-1.0],Times[Times[2.0,Power[x,1.0]],1.0]]
f''(x) = Plus[Plus[Times[Times[Exp[Times[-1.0,x]],-1.0],-1.0],0.0],Plus[Times[Plus[0.0,Times[2.0,T
imes[Times[1.0,Power[x,0.0]],1.0]]],1.0],0.0]]
x0 = -2.000000
f|min = f(0.35172) = 0.82718

takase@kengo ~/C/proen/report2
$ ./a.exe Plus[Log[Plus[1,Power[x,2]]],Power[Plus[x,1],2]] -2
f(x) = Plus[Log[Plus[1.0,Power[x,2.0]]],Power[Plus[x,1.0],2.0]]
f'(x) = Plus[Divide[Plus[0.0,Times[Times[2.0,Power[x,1.0]],1.0]],Plus[1.0,Power[x,2.0]]],Times[Tim
es[2.0,Power[Plus[x,1.0],1.0]],1.0]]
f''(x) = Plus[Divide[Subtract[Times[Plus[0.0,Plus[Times[Plus[0.0,Times[2.0,Times[Times[1.0,Power[x
,0.0]],1.0]]],1.0],0.0]],Plus[1.0,Power[x,2.0]]],Times[Plus[0.0,Times[Times[2.0,Power[x,1.0]],1.0]
],Plus[0.0,Times[Times[2.0,Power[x,1.0]],1.0]]]],Power[Plus[1.0,Power[x,2.0]],2.0]],Plus[Times[Plu
s[0.0,Times[2.0,Times[Times[1.0,Power[Plus[x,1.0],0.0]],1.0]]],1.0],0.0]]
x0 = -2.000000
f|min = f(-0.56984) = 0.46624
```

３．考察

・微分の実装について

木構造を直接微分しようとすると微分して新たに出てきた基本関数の分だけ新たに構造体を定義する必要があり、コードが煩雑になってしまった。表式を木構造にする関数は既に作ってあったので、sprintf などを使って記号上で再帰的に微分操作を行った方が簡潔に書けたと思われる。

また、最初に定義域、値域を実数のみに制限してしまったために、$\log f$の微分が$f < 0$で定義できないなどの問題が発生した。複素数まで定義しておけば$\mathrm{Log}\, f = \log|f| + j \arg f$のように全範囲で微分を定義することができたが、正則の判定などがネックで実装できなかった。

・struct node *optimise_node(struct node *p)の処理について

実装できたのは x を含まない部分木の計算と Times[0,f]=0 のみだった。Plus[0,f]=f、Times[1,f]=1,Power[f,1]=f なども実装したかったが、単方向リストで木構造を作ってしまったためにリストを辿ることができず今回は見送った。