0.S1 ジュニア版 特別章(1)　　「百ます計算」と Grid Search

百ます計算

日本の小学校で行なわれている百ます計算とは、おおよそ次のようなものです。

	14	**19**	**20**	**11**	**18**	**12**	**17**	**16**	**13**	**10**
9	5	10	11	2	9	3	8	7	4	1
1	13	18	19	10	17	11	16	15	12	9
2	12	17	18	9	16	10	15	14	11	8
6	8	13	14	5	12	6	11	10	7	4
4	10	15	16	7	14	8	13	12	9	6
3	11	16	17	8	15	9	14	13	10	7
7	7	12	13	4	11	5	10	9	6	3
8	6	11	12	3	10	4	9	8	5	2
5	9	14	15	6	13	7	12	11	8	5
10	4	9	10	1	8	2	7	6	3	0

上の表は、太字の部分の縦横の引き算をしたものです。例えば、左上から 14－9＝5、その横は 19－9＝10、同じことを繰り返して、一番右端は 10－9＝1 という計算ができます。同様にして、他の部分も計算して、最終的に右下すみの 10－10＝0 を計算して終わります。これは、引き算のバージョンですが、足し算掛け算それに割り算のものもあります。実際には、太字のところが縦横それぞれ与えられていて、その下に空欄のマスがあいているわけです。そこに計算結果を埋めていき、自己タイムの更新を目指すものと言われています。

さて、これを GAUSS でやってみましょう。今回は簡単化のため、ファイルを作ることなく、直接 Command Window を使って、>>の後に電卓のように命令を打ち込んで計算することにします。以下、>>の部分は打ち込まないでください。

入力

```
>> x={14 19 20 11 18 12 17 16 13 10}
>> y={9,1,2,6,4,3,7,8,5,10}
>> print/rz  x-y
```

上のように合計３行を１行ずつ打ち込んで、それぞれの行で Enter を押します。GAUSS の Edit ファイル上での通常の操作では、各行または命令ごとの後にはセミコロン；が必要ですが、この場合は自動的に付きます。結果は、計算すべき部分だけが行列となって表示されます。小数点以下０がついてもかまわないのであれば、３行目は print x-y だけで十分です。足し算のときは x-y のところを x+y に、掛け算のときは x-y のところを x.*y とするだけです。一度入力した１行目と２行目は、以後あらためて入力する必要はありません。なお、入力を間違えたら、もう一回正しい入力をすれば前回の入力は、上書きされて無効

になります。どれも 10 秒から数十秒程度で入力から計算まで完了できると思います。
　なお、procedure を使って引き算の百ます計算を表した上で、それを呼び出して計算すれば次のようになります。見やすいように format 命令で出力数字間の幅を狭くしましょう。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
call masu100(x,y);

proc masu100(x,y);
    local z;
    z=x-y;
    format /rz 4,2;
    print miss(0,0)~x;
    print y~z;
    format /mb1 /ros 16,8;
    retp(z);
endp;
```

画面表示

```
    .   14   19   20   11   18   12   17   16   13   10

    9    5   10   11    2    9    3    8    7    4    1
    1   13   18   19   10   17   11   16   15   12    9
    2   12   17   18    9   16   10   15   14   11    8
    6    8   13   14    5   12    6   11   10    7    4
    4   10   15   16    7   14    8   13   12    9    6
    3   11   16   17    8   15    9   14   13   10    7
    7    7   12   13    4   11    5   10    9    6    3
    8    6   11   12    3   10    4    9    8    5    2
    5    9   14   15    6   13    7   12   11    8    5
   10    4    9   10    1    8    2    7    6    3    0
```

プログラム各行の説明

１行目　メモリを初期化する。
２行目　スクリーンをクリアする。

３行目　変数 x に 10 個の数字からなる１行を代入する。
４行目　変数 y に 10 個の数字からなる１列を代入する。カンマまでが１行。
５行目　masu100(x,y)を呼び出して計算させる。
７～15 行目 x と y をインプットとし、z をリターンとする masu100 という関数
８行目　この関数内だけで使われるローカル変数はｃ
９行目　x－y を計算して z に代入する
10 行目 format 命令で以後 print 命令で画面表示される字間を調節する。この場合/rz は右寄せ小数点以下の０をなくすオプション。4,2 は各数値に 4 文字分が割り当てられ、小数点以下２桁という意味。ここでは小数点以下はゼロがなくされるので小数点以上の２桁または１桁が表示される。
11 行目　欠落値のドットを miss(0,0)によって１つ作成し、これと x の内容を水平方向に～でマージして、それを画面表示する。
12 行目　y と計算結果の z を水平方向に～でマージして、それを画面表示する。
13 行目　デフォールトフォーマット **format /mb1 /ros 16,8;**に戻す。
14 行目　z をリターンとして外部に返す。
15 行目　この procedure の最後を示す部分。７行目の proc から endp までが１つの関数。

なお、procedure の中で format 文を多用することは避けなければなりません。万が一使用したとしても、最後に GAUSS 本来のデフォルト設定に戻しておくことが必要です。数値のおのおのが広がった結果でもよい場合には、敢えて format 文を使用する必要はありません。上では、引き算のケースですが、足し算や掛け算の場合は x-y を x+y や x.*y に変更してください。

行列のインデックス

上の百ます計算と同じように、GAUSS などの行列言語のプログラミングにおいて最も重要な概念は行列のインデックスです。今 10 行 10 列の合計百要素からなる行列を考えます。

$$\begin{pmatrix} 5 & 10 & 11 & 2 & 9 & 3 & 8 & 7 & 4 & 1 \\ 13 & 18 & 19 & 10 & 17 & 11 & 16 & 15 & 12 & 9 \\ 12 & 17 & 18 & 9 & 16 & 10 & 15 & 14 & 11 & 8 \\ 8 & 13 & 14 & 5 & 12 & 6 & 11 & 10 & 7 & 4 \\ 10 & 15 & 16 & 7 & 14 & 8 & 13 & 12 & 9 & 6 \\ 11 & 16 & 17 & 8 & 15 & 9 & 14 & 13 & 10 & 7 \\ 7 & 12 & 13 & 4 & 11 & 5 & 10 & 9 & 6 & 3 \\ 6 & 11 & 12 & 3 & 10 & 4 & 9 & 8 & 5 & 2 \\ 9 & 14 & 15 & 6 & 13 & 7 & 12 & 11 & 8 & 5 \\ 4 & 9 & 10 & 1 & 8 & 2 & 7 & 6 & 3 & 0 \end{pmatrix}$$

これは、引き算の百ます計算の結果と同じものです。行列では横の数字の並びを「行」、そして縦の数字の並びを「列」と呼びます。通常、行列の行と列の１行１列の場所は左上と決められています。ここでは、左上すみの５が１行 1 列の場所、右下すみの０が 10 行 10 列の場所ということになります。また、例えば、３行４列の場所は、左上すみの２つ下の３行目を左に４つ行ったところ、すなわち９の場所になります。このことを GAUSS の実際の計算でどう表せるかを見てみましょう。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
print z[1,1];
print z[10,10];
print z[3,4];
```

画面表示

```
      5.0000000
     0.00000000
      9.0000000
```

プログラム各行の説明

１行目　メモリを初期化する。
２行目　スクリーンをクリアする。
３行目　変数 x に 10 個の数字からなる１行を代入する。カンマがないので１行。
４行目　変数 y に 10 個の数字からなる１列を代入する。各カンマまでが１行。
５行目　百ますの要領でｘ－ｙを計算して、変数ｚに代入する。
６行目　行列ｚの１行１列目を画面表示する。

７行目　行列ｚの 10 行 10 列目を画面表示する。
８行目　行列ｚの３行４列目を画面表示する。

上のように、１行１列目は５、10 行 10 列目は０、そして３行４列目は９で目測で計算したのと一致していることがわかります。このように、行列の行と列のインデックスを表示するには、四角括弧の中にカンマで区切って行そして列の順で記述します。

注意事項

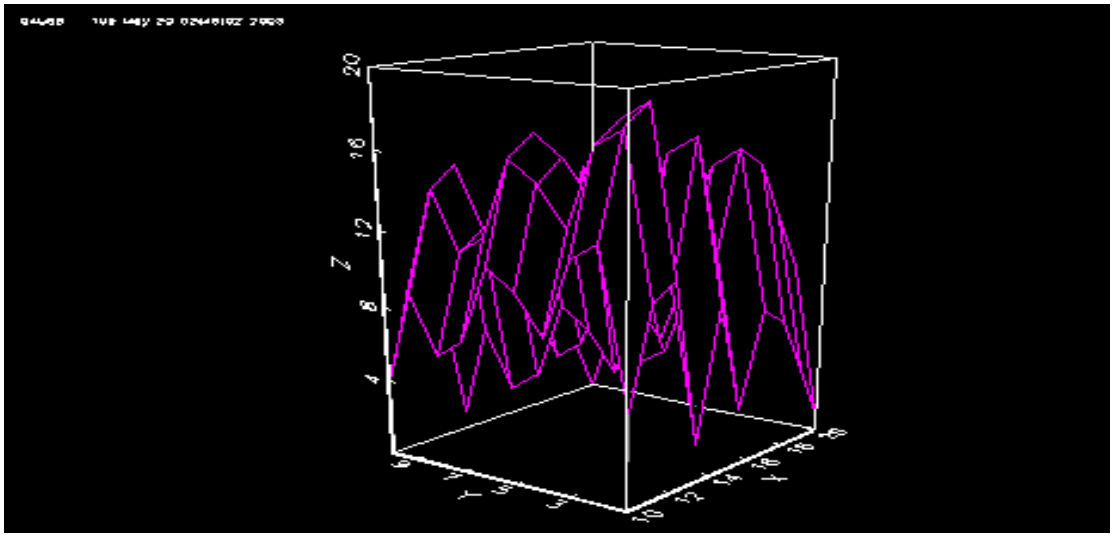
行列のインデックスは四角括弧　　例　m[3,4]
組込み関数および procedure のインプットとリターンは丸括弧　　例　kansu(x,y)
数値の代入は｛　　｝括弧　　例　{1,2,3,4,5}

Surface グラフ

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
library pgraph;
graphset;
xlabel("X");
ylabel("Y");
zlabel("Z");
surface(x,y,z);
```

グラフ表示



プログラム各行の説明

１行目　メモリを初期化する。
２行目　スクリーンをクリアする。
３行目　変数 x に 10 個の数字からなる１行を代入する。カンマがないので１行。
４行目　変数 y に 10 個の数字からなる１列を代入する。各カンマまでが１行。
５行目　百ますの要領でｘ－ｙを計算して、変数ｚに代入する。
６行目　グラフを描くライブラリ pgraph を呼び出す。
７行目　そのグローバル変数（デフォルトパラメータ設定）を初期化する。
８行目　ｘ軸のラベルとしてＸを文字列として設定。GAUSS では” “の内側は文字列。
９行目　ｙ軸のラベルとしてＹを文字列として設定。
10 行目　ｚ軸のラベルとしてＺを文字列として設定。
11 行目　surface グラフをｘ軸（１行で設定）とｙ軸（１列で設定）の値について高さをプロットしてそれを結んだ表面を作画する。

上のように、百ますでやったのと同じように、ｘ軸側には横１行（カンマなし）、ｙ軸側には縦１列（カンマまでが１行でカンマでそれぞれを区切る）をそれぞれ設定して、ｘ－ｙの計算結果の行列ｚの表面（高さに相当）をグラフ化しています。

最大値と最小値

上の結果行列ｚについて、最大値と最小値を求めてみましょう。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
print maxc(z);
print "max=" maxc(maxc(z));
```

画面表示

```
13.000000
18.000000
19.000000
10.000000
17.000000
11.000000
```

```
        16.000000
        15.000000
        12.000000
        9.0000000
max=        19.000000
```

プログラム各行の説明

１行目　メモリを初期化する。
２行目　スクリーンをクリアする。
３行目　変数 x に 10 個の数字からなる１行を代入する。カンマがないので１行。
４行目　変数 y に 10 個の数字からなる１列を代入する。各カンマまでが１行。
５行目　百ますの要領でｘ－ｙを計算して、変数ｚに代入する。
６行目　ｚの各列の最大値を組込み関数 maxc で求め画面表示する。結果は１列となる。
７行目　さらにその結果の最大値を求め画面表示する。これが行列全体の最大値となる。

なお、組込み関数 maxc のｃは column のｃで各行ごとに最大値を求めるものです。これを二重にして繰り返すことによって、行列全体の最大値が求まります。明らかに、上の行列では各列の２行目に最大値が集中していますから、それを転置して１列の形で結果は出てきます。さらにこれをその結果の 1 列に適応して、その中の３つ目の 19 が最大値となる。

同様にして、最小値を求めてみましょう。今度は minc という関数を使うことが違うだけです。この場合のｃも、それぞれの column ごとの min を求めるという意味です。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
print minc(z);
print "min=" minc(minc(z));
```

画面表示

```
        4.0000000
        9.0000000
        10.000000
        1.0000000
        8.0000000
        2.0000000
```

```
        7.0000000
        6.0000000
        3.0000000
      0.00000000
min=      0.00000000
```

一重の minc は各列の最小値を列として表示するものです。それをもう一回繰り返すように二重にすると、この最小値の列の結果の min ということで全体の最小値が得られます。

最小値を返すインデックス

上では、最大値と最小値をそれぞれ求めましたが、ここでは最大値および最小値を与える場所、つまり何行何列目であるのかをプログラムで計算してみましょう。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
print maxindc(maxc(z')) maxindc(maxc(z));
```

画面表示

```
        2.0000000        3.0000000
```

上のプログラムの最終行では、行列ｚを転置したもの（すなわち行と列をすべて逆転させたもの）の列ごとの最大値を求め 1 列で出てきたもの中で最大値を与えるインデックス番号を画面表示させます。すなわち、「行ごと」の最大値を求め、その最大値を与えるインデックス番号を画面表示させてます。これは最大値が存在する行番号です。後半は、そのままのｚを用いで、その最大値を求め、その最大値を与えるインデックス番号を合わせてその横に画面表示させています。関数を２重にするところは行列の最大値を求めたのと同じです。ただし、ここではインデックス番号が必要です。どの行から最大値が出てくるのか、そしてどの行から最大値が出てくるのかを求め、合わせて画面表示させているのです。

同様にして、行列ｚの最小値を与えるインデックス番号も次のようにしてできます。

プログラム

```
new;
cls;
x={14 19 20 11 18 12 17 16 13 10};
y={9,1,2,6,4,3,7,8,5,10};
z=x-y;
print minindc(minc(z')) minindc(minc(z));
```

画面表示

```
        10.000000        10.000000
```

内側の maxc が minc にかわり、外側の maxindc が minindc にかわっただけです。前半では転置させることによって、行と列を逆転させたことになり、行ごとの最小値も求めたものをさらに minindc でその中の最小値を与えるインデックス番号を求めています。後半では、ぞのままのｚを用いていますから、列ごとの最小値を求め、さらにその中で最小値を与えるインデックス番号、すなわち、どこの列から最小値は出てくるのかを求めて画面表示させています。これらは、まさに Grid Search の原型となる計算です。

固定ステップの２次元 Grid Search

プログラム

```
new;
cls;
x=seqa(0,0.01,100)';
y=seqa(0,0.01,100);
z=x-y;
print maxindc(maxc(z')) maxindc(maxc(z));
print "max:" x[maxindc(maxc(z'))] y[maxindc(maxc(z))];
print minindc(minc(z')) minindc(minc(z));
print "min:" x[minindc(minc(z'))] y[minindc(minc(z))];
```

画面表示

```
        1.0000000        100.00000
max:       0.00000000       0.99000000
        100.00000        1.0000000
min:       0.99000000       0.00000000
```

プログラム各行の説明

１行目　メモリを初期化する。

２行目　スクリーンをクリアする。

３行目　０から 0.01 ステップで 100 個の数列、すなわち{0,0.01,0.02,…0.99}を転置したもの、つまり１行に変換したものを変数ｘに代入する。

４行目　同様に数列{0,0.01,0.02,…0.99}自身を変数ｙに代入する。

５行目　百ますの要領でｘ－ｙを計算して、変数ｚに代入する。

６行目　行列ｚの最大値を与える行インデックスと列インデックス番号を画面表示する。

７行目　ｘの行インデックス番号目とｙの列インデックス番号目の実際の値を、引用符のなかの max:というコメント部分とともに画面表示する。

８行目　行列ｚの最小値を与える行インデックスと列インデックス番号を画面表示する。

９行目　ｘの行インデックス番号目とｙの列インデックス番号目の実際の値を、引用符の

なかの min:というコメント部分とともに画面表示する。
ｘとｙのところがきちんと小さいものから大きなものに同じ間隔で並んだ数の行または列にかわり、あとはこれまでと同じように最大値および最小値を与えるインデックス番号を求め、まずそれらを表示しています。行数と列数です。次に、ｘの行数番目の実際の値とｙの列数番目の実際の値とを表示させているのです。なお、行または列どちらか一方が１列または１行しかないもののインデックスの与え方は、変数の後の四角括弧の中に、ただ１つの数字を与える式を書けばよいだけです。もう一方の行または列のインデックスは考える必要はありません。上の結果の意味する所は、ｘおよびｙが０から 0.99 の間にある時に最大値は行列ｚの１行 100 列目、すなわち x=0 ,y=0.99 が最大値を与えるｘとｙです。また、最小値は同じく行列ｚの 100 行 1 列目、すねわち x=0.99 ,y=0 が最小値を与えるｘとｙの組となります。この場合、0.01 刻みにしていて、またこれ以上計算はしませんから０から 0.99 の間の小数第２位までの値の組を正確に求めていることになります。この場合は「１万ます計算」をしていることになるのですが、さらに細かく目盛りをすればその小数位までの値の組が正確に求められますが、これ以上は Light 版のメモリの制約の範囲を超えてしまいます。また、通常版で展開できる行列のディメンションもある程度決まっていて、そう小数の桁数を上げることは物理的にできません。その場合には、本編の最適値計算の章で扱うように、その回の最小値（または最大値）のさらにその周辺を徐々に絞りこむことによって計算を進めることになります。なお、上では０から 0.99 の範囲ですが、seqa で数列を作成する初期値の０のところを適当に取替え、その他の部分も調整すればその他の範囲の計算も簡単にできます。上は関数 f(x,y)=x-y の範囲内での最大値と最小値を与えるｘとｙの値を求めていると考えることもできます。

固定ステップの１次元 Grid Search

１変数の関数のある範囲の最小値（または最大値）を求めるのは、下のように、これまでの２次元のケースに比べて格段に易しく求められます。

プログラム

```
new;
cls;
x=seqa(0,0.0001,10000);
fn f(x)=x^2-x+1;
print x[minindc(f(x))];
library pgraph;
graphset;
xy(x,f(x));
```

画面表示

```
      0.50000000
```

プログラム各行の説明

１行目　メモリを初期化する。

２行目　スクリーンをクリアする。

３行目　０から 0.0001 ステップの１万個の数列（すなわち{0,0.0001,0.0002,…,0.9999}）を変数ｘに代入する。この場合は１変数なので列のままでよい。

４行目　関数 f(x)=x^2-x+1 を fn 命令で定義する。

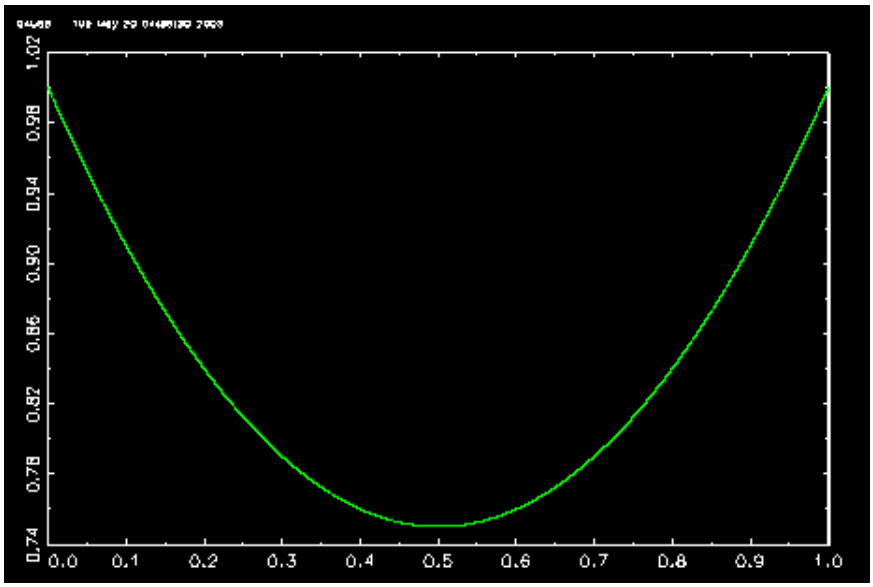
５行目　f(x)に数列ｘを入れた結果について minindc を適用し、その最小値を与えるインデックス番号を計算し、それを変数ｘのあとの四角括弧のなかに入れることによって、実際のｘの値を求めている。

６行目　グラフを描くライブラリーpgraph を呼び出す。

７行目　そのグローバル変数を初期化する。

８行目　数列ｘに対して、f(x)のなかに代入した結果をｘｙグラフでプロットする。

グラフ表示



上では、ちょうど 0.5 あたりで関数 f(x)は実際に最小になっていることがわかります。実際に結果も、0.5 で一致しています。１変数の場合は、２変数と違ってたくさんの刻みをメモリ的制約を考えずに使えますから探す範囲を０からではなくて、適当に変更してやればだいたいの最小値または最大値を与える値を計算することができます。この方法は、単に高さの高低を見ているだけで、微分による数学的な計算をしているわけではありません。上で言えば、小数第４位まで正確に求められます。ただし、さらなる精度を求める場合や、探す範囲がこれよりも広い場合には、繰り返しをともなう計算をしなくてはなりません。これについては本論で扱っています。

１変数の解を求める

上の１変数の単純 Grid Search の考え方を、関数の絶対値に対してあてはめてやれば、方程式の解（その関数が０になるところの値）が求まります。次のようにします。

プログラム

```
new;
cls;
x=seqa(0,0.0001,10000);
fn g(x)=x^3+2*x-1;
print x[minindc(abs(g(x)))];
library pgraph;
graphset;
xy(x,g(x));
```

画面表示

```
    0.45340000
```

プログラム各行の説明

１行目　メモリを初期化する。

２行目　スクリーンをクリアする。

３行目　０から 0.0001 ステップの１万個の数列（すなわち{0,0.0001,0.0002,…,0.9999}）を変数ｘに代入する。この場合は１変数なので列のままでよい。

４行目　関数 g(x)=x^3+2*x-1 を fn 命令で定義する。

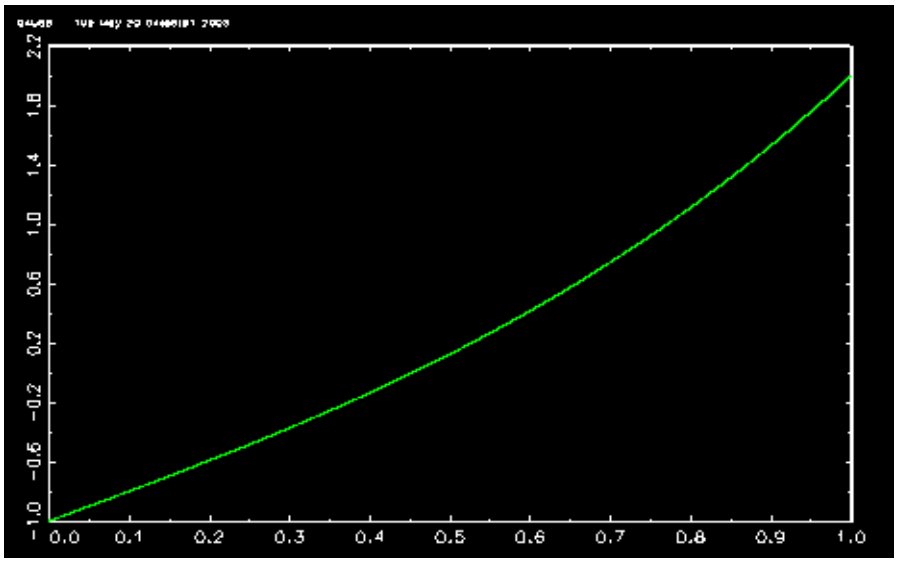
５行目　g(x)に数列ｘを入れた結果をさらに abs 関数で絶対値変換し、その結果について同様に minindc を適用し、その最小値を与えるインデックス番号を計算し、それを変数ｘのあとの四角括弧のなかに入れることで、実際のｘの値を求める。

６行目　グラフを描くライブラリーpgraph を呼び出す。

７行目　そのグローバル変数を初期化する。

８行目　数列ｘに対して、f(x)のなかに代入した結果をｘｙグラフでプロットする。

グラフ表示



違う点は、g(x)の関数が上の数列の範囲で０を一回だけ横切るということと、f(x)ではなくて、abs(g(x))の計算結果の minindc を求めていることです。この考えている数列の範囲でg(x)が０を一回だけ横切るのならば、その絶対値の最小値はそこで０になり、それが最小値です。この解と絶対値の関係を用いているわけです。上のグラフでは縦軸の 0.2 と-0.2 の中間点が０でそれを水平に横に見ていきグラフの軌跡と交わるあたりの値 0.4 すぎのどこかが求める解になります。

練習問題

【問１】横縦ともに１から 10 までの数のシークエンスとして、足し算の「百ます計算」を GAUSS でやってみよう。

【問２】横縦ともに１から 10 までの数のシークエンスとして、掛け算の「百ます計算」を GAUSS でやってみよう。

【問３】f(x)=x^3-2*x+1 の０から１の間の最小値を固定ステップの１次元 Grid Search で求めてみよう。

発展

⇒　本編　3.8

⇒　Luenberger　　１変数のケース
　　２変数のケース　16.6b