Genetic Algorithms

2004.11.18

■ Set up the problem

In our case we have chosen the conventional binary coding. Each individual is coded as a binary string with length stringLength. The problem is limited to 1D, and for x from 0 to 1.

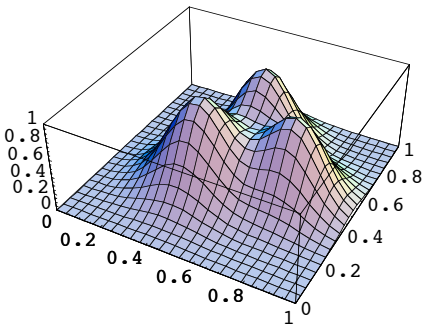
```
stringLength = 10;
mutationRate = 0.002;
popSize = 100;          (* must be a power of two *)
```

Define the fitness function. It must be defined in the interval 0 to 1. The first one is a simple one, while the second function is a little bit more difficult.

Fitness 関数は、x->{0,1}, y->{0,1} に規格化されている必要がある？　zは１以上でもよい？　ー＞　popFitness で選択しているだけなので、何でも良い。

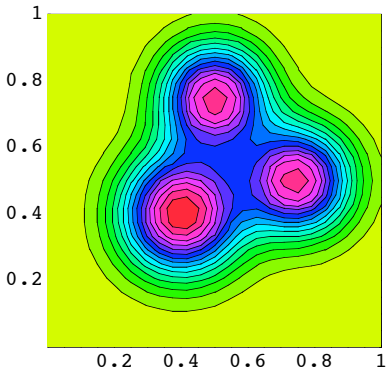
```
Clear[fitnessFunction];
fitnessFunction[x_List] :=Exp[-(8(2 x[[1]]-0.8)^2+8(2 x[[2]]-0.8)^2)]+
                        0.9Exp[-(10(2 x[[1]]-1.0)^2+10(2 x[[2]]-1.5)^2)]+
                        0.9Exp[-(10(2 x[[1]]-1.5)^2+10(2 x[[2]]-1.0)^2)]

Plot3D[fitnessFunction[{x,y}],{x,0,1},{y,0,1},PlotRange->{{0,1},{0,1},{0,1}}]
```



- SurfaceGraphics -

```
g1=Show[ContourGraphics[%],Contours->15,ColorFunction->(Hue[#/1.3+0.2]&)]
```



- ContourGraphics -

■ Miscellaneous functions

整数の二進表示型にマップする。実際のCプログラムに展開する場合には、整数化してから取り扱う方が便利と考えられる。必要な計算精度を考えてビット数を決定するべきであるが、実際には、機械のワードサイズに合わせることになる。ー＞16bit (short integer) が適切であろう。ただし、遺伝子交換を考えて、切断点の決定は、数値範囲を考慮した位置であることが必要だろう。

```
Clear[floatToBinary];
floatToBinary[x_] := IntegerDigits[Round[1000 x], 2, 10] /; (1000 x) < 2^10

Clear[binaryStringToFloat];
binaryStringToFloat[string_] :=Map[FromDigits[#1,2]&,string,{2}]/1000.
```

Scale the fitness values linearly so that max = 2 min. This is a good idea to do because otherwise the genetic material tends to be dominated by very few nonidentical members. There is always a balance between concentrating the search on the best strings, and having a search in a broad spectrum.

```
Clear[renormalizeFitness];
renormalizeFitness[fitness_List] := Module[{minF,maxF,a,b},
    minF = Min[fitness];
    maxF = Max[fitness];
    a = 0.5*maxF/(maxF+minF);
    b = (1-a)*maxF;
    Map[ a# + b &, fitness ]
]
```

■ Function definitions

■ Simple crossover

A random position is chosen along the strings. Create two new strings by swapping the parts before and after the cut respectively.

2-D crossover function definition

```
Clear[doSingleCrossover];
doSingleCrossover[ {strp_, strq_} ] := Module[{cut},
    cut = Random[Integer, {1,stringLength-1}];

{{Join[Take[strp[[1]],cut],Drop[strq[[1]],cut]],Join[Take[strq[[1]],cut],Drop[strp[[
1]],cut]]},
{Join[Take[strp[[2]],cut],Drop[strq[[2]],cut]],Join[Take[strq[[2]],cut],Drop[strp[[2
]],cut]]}}  ]
```

■ Mutations

Every gene may be excanged, from 0 to 1 and vice versa, with the probability mutationRate. The idea behind this is to introduce a mechanism for escaping from local minimas.

Mutation for 2-D

```
Clear[doMutation];
doMutation[string_] := Module[{tempstring1,tempstring2,i},
    tempstring1 = string[[1]];
    Do[ If[ Random[] < mutationRate,
        tempstring1[[i]] = 1 - tempstring1[[i]] ],
        {i,stringLength} ];
    tempstring2 = string[[2]];
    Do[ If[ Random[] < mutationRate,
        tempstring2[[i]] = 1 - tempstring2[[i]] ],
        {i,stringLength} ];
    {tempstring1,tempstring2}

]
```

■ Selection

Select two strings from the population, with a probability for each string proportional to its fitness value. Return only the two indices.

This computes the cumulative sum of the fitness values. It is necessary in the selection process.

■ Selection process の説明

```
Clear[doSingleSelection];
doSingleSelection := Module[{rfitness,ind},
    rfitness = Random[Real, {0, cumFitness[[popSize]]}];
    ind = 1;
    While[ rfitness > cumFitness[[ind]], ind++ ];
    ind--
]
```

```
General::spell1 :
 スペル間違いの可能性があります．新規シンボル "rfitness"はすでにあるシンボル "fitness"に似ています．  詳細
```

純関数とNestWhile を用いて書き換えることができるが、速度的に有利にはなっていないようで、かつC言語への変換が見通せない、という点で棚上げにした関数である。

```
Clear[doSingleSelection];
doSingleSelection := Module[{rfitness,ind},
    rfitness = Random[Real, {0, cumFitness[[popSize]]}];
    NestWhile[#1+1&,1,(rfitness>cumFitness[[#1]])&]]
```

■ 関数の書き換えーMathematica らしくする

This routine selects a pair of individuals with probability for each proportional to its fitness value.

```
Clear[selectPair];
selectPair := Module[{ind1,ind2},
    ind1 = doSingleSelection;
    While[ (ind2 = doSingleSelection) == ind1 ];
    {ind1, ind2}
]
```

■ Initialize

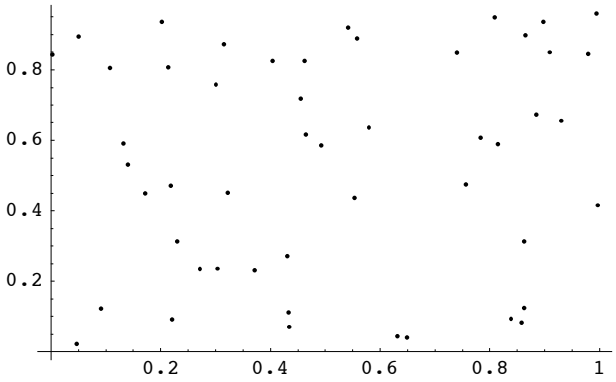
2-Dimensional Initialize function

```
Clear[doInitialize];
doInitialize := Module[{i},
    popFloats = Transpose[{Table[ Random[], {popSize} ],Table[ Random[], {popSize}
]}];
    popStrings = Map[ floatToBinary, popFloats,{2} ];
    popFitness = Map[fitnessFunction,binaryStringToFloat[ popStrings ]];

    cumFitness = Drop[FoldList[Plus, 0, popFitness],1];
    progressOfFitness = {cumFitness[[popSize]]};
    historyOfPop = { popStrings };
]

doInitialize;

ListPlot[binaryStringToFloat[ popStrings ]];
```



■ Main

■ Make a new generation; update synchronously

This represents the other way to do it. The complete parent population is chosen in a single sweep, and the children constitutes the new population.

■ This is Main

```
doInitialize;
```

2-D main calculation routine

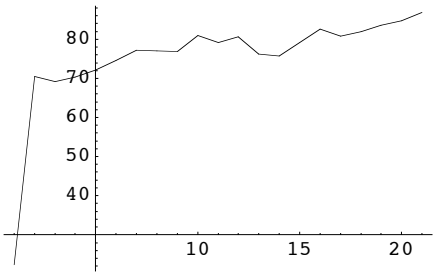
```
Do[ parentsid = Table[selectPair,{popSize/2}];
    popStrings = Flatten[Table[
            doSingleCrossover[ {popStrings[[parentsid[[ip,1]]]],
                                popStrings[[parentsid[[ip,2]]]]} ],
                                    {ip,popSize/2} ], 1];
    popStrings = Map[ doMutation, popStrings];
    popFitness = Map[fitnessFunction,binaryStringToFloat[ popStrings ]];
    popFitness = renormalizeFitness[ popFitness ];
    cumFitness = Drop[FoldList[Plus, 0, popFitness],1];

    AppendTo[ historyOfPop, popStrings ];
    AppendTo[ progressOfFitness, cumFitness[[popSize]] ],
    {20} ];
```

■ Plot results

This is the sum of fitness for the whole population plotted versus generation number.

```
ListPlot[ progressOfFitness, PlotJoined->True, PlotRange->All ];
```

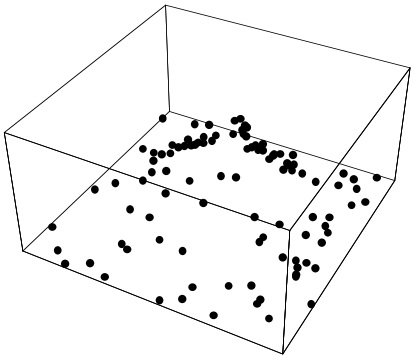


2-D GA by 3-D plot routines

```
fitnessplot = Plot[fitnessFunction[x], {x, 0, 1},
    Axes->True,
    DisplayFunction->Identity];

graphicsSingleGeneration[igen_] := Module[{graphelement,xy},
    graphelement = {};
    Do[ xy = binaryStringToFloat[ historyOfPop[[igen]] ][[i]];
        AppendTo[ graphelement,
            Point[{xy[[1]],xy[[2]],fitnessFunction[xy]}]  ],
        {i, popSize} ];
    graphelement
]
```

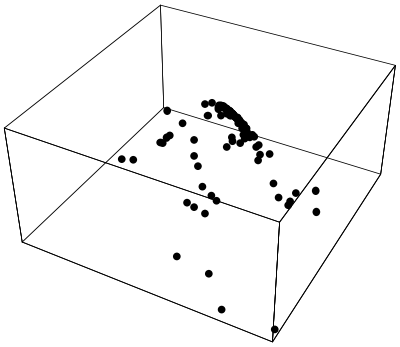
```
Show[Graphics3D[{PointSize[0.02],graphicsSingleGeneration[1]}],
    PlotRange->{{0,1},{0,1},{0,1.1}},BoxRatios->{1,1,0.5}];
```



```
Length[ historyOfPop ]
```

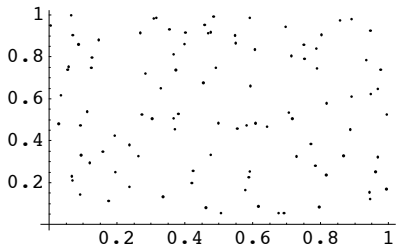
```
21
```

```
Show[Graphics3D[{PointSize[0.02],graphicsSingleGeneration[21]}],
    PlotRange->{{0,1},{0,1},{0,1.1}},BoxRatios->{1,1,0.5}];
```



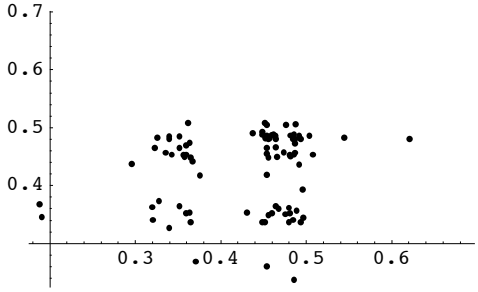
```
xy = binaryStringToFloat[ historyOfPop[[1]] ];
```

```
g2=ListPlot[xy]
```

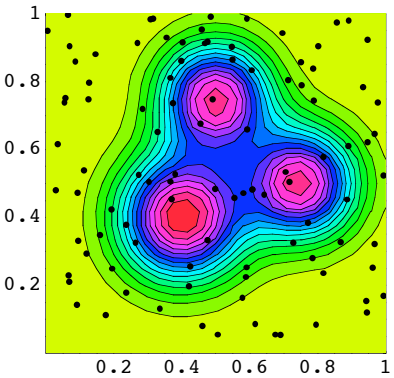



```
- Graphics -
```

```
g3=ListPlot[binaryStringToFloat[ historyOfPop[[21]] ]];
```

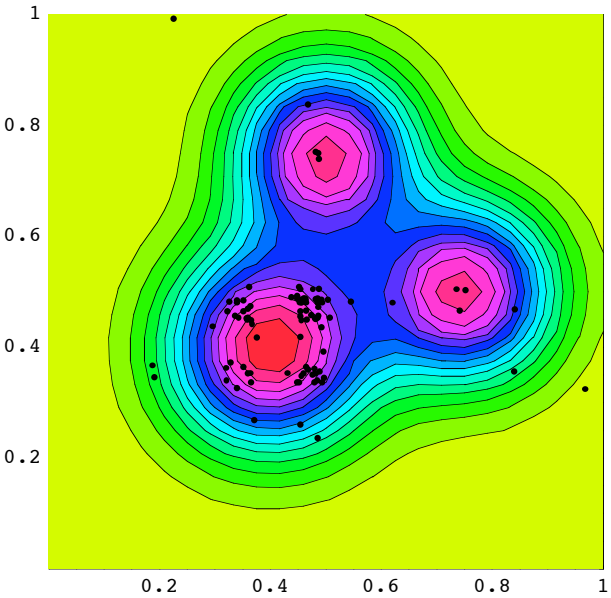



```
Show[g1,g2,Prolog->AbsolutePointSize[3]]
```



- Graphics -

```
Show[g1,g3,Prolog->AbsolutePointSize[3]]
```



- Graphics -