

## Source-Code für C++

Der folgende Source-Code ist ein einfacher Evolutionsalgorithmus, der die Variablen einer Formel optimieren kann. Die Variablen sind Integer-Zahlen und können im Unterprogramm 'Fitness' zu einer Formel verknüpft werden.

### Beispiel für die Verknüpfungen der Variablen:

Will man die Variablen zu einer Formel:

$$(a*\cos(b))-(c/d)=0$$

verknüpfen, so muß man in der Prozedur 'Fitness' die Zeile:

```
Ergebnis [n]=.....
```

folgendermaßen abändern: Für die Variable a schreibt man V[0], für die Variable b schreibt man V[1], für c V[2] für d V[3] usw. Die Formel sieht nun so aus:

```
Ergebnis[n]=(V[0]*cos(V[1]))-(V[2]/V[3]);
```

Wobei cos() der Cosinus ist und eine Funktion von C++ ist (In der Include-Datei math.h, siehe auch 'Include-Dateien'). Auf diese Weise kann man jede beliebige Formel realisieren. Die Anzahl der Variablen die für diese Formel verwendet werden soll kann mittels der Definition Variablen\_pro\_GEN (siehe auch Kapitel Definitionen) eingestellt werden.

### Weitere Einstellungen mittels den Definitionen:

Die folgenden Einstellungen sind alle in den Definitionen zu vollziehen (siehe auch Kapitel 'Definitionen').

Anzahl\_der\_Generationen

Mit diesem Punkt kann die Anzahl der Generationen die berechnet werden sollen eingestellt werden.

Variablen\_pro\_GEN

Wie schon erwähnt, kann mit dieser Definition die Anzahl der verwendeten Variablen verändert werden.

GENE\_pro\_Generation

Diese Definition gibt die Größe der Anfangspopulation und somit auch der folgenden Populationen an.

Kreuzen\_von, Kreuzen\_bis

Mit Kreuzen\_von und Kreuzen\_bis kann man einstellen welche Variablen einer Generation verkreuzt werden sollen. (Beispiel: Die besten 16 Variablen ... Kreuzen\_von 0; Kreuzen\_bis 16).

Mutationsfaktor, Mutationswirkung

Der Mutationsfaktor gibt an wieviele der GENE einer Generation mutiert werden sollen. Die Mutationswirkung gibt an wieviele Variablen der GENE die mutieren sollen verändert werden sollen.

oben0, unten0, oben1, ...

Für jede Variable muß mithilfe dieser Variablen eine Ober- und eine Untergrenze angegeben werden. Parallel zu den Definitionen müssen die entsprechenden Zeilen im Hauptprogramm geschrieben werden (siehe Kapitel main)

### Mögliche Verbesserungen des Programmes

- Das Programm soll nur das Prinzip von Evolutionsalgorithmen darstellen und kann in vielerlei Weise verbessert werden. Ich werde hier einige Möglichkeiten anführen wie man das Programm entscheidend verbessern kann.
- Der erste Schritt kann sein, daß man die Berechnung nicht nach n Generationen, sondern wenn ein genügend guter Fitnesswert erreicht ist abbricht. (Die Fitnesswerte stehen in dem Array Ergebnis[n], wobei der beste Werte an der Stelle 0 steht (Ergebnis[0]).
- Weiters kann könnte man anstatt von Integer-Variablen auch float Variablen verwenden. Hier muß man sich aber überlegen, wie man die Erzeugung von Zufallswerten in den Prozeduren Anfangspopulation\_erstellen und Mutation programmiert.
- Sinnvoll wäre es auch, wenn man die Mutation auf einige GENE einer Generation beschränken würde.
- Die weitreichendste Verbesserung wäre aber, wenn man nicht eine gewisse Anzahl von Variablen verwendet, sondern einen Speicherbereich der n Bytes groß ist. Die Fitnessfunktion kann hier auch für wesentlich komplexere Themen verwendet werden.

## Programmaufbau

- Include-Dateien
- Definitionen
- Erstellen einer Anfangspopulation (Anfangspopulation\_erstellen)
- Fitness
- Sortieren
- Mutation
- Kreuzen
- Ausgabe
- main

### Einige Tips zum Programm

- Es kann durchaus vorkommen, daß sich das Programm beim Finden einer Lösung in einer Sackgasse (Lokales Maximum, oder Minimum) verläuft. Es empfiehlt sich in so einem Fall, einfach das Programm neu zu Starten.
- Man sollte ein wenig mit der Anzahl der GENE\_pro\_Generationen und dem Kreuzen\_von bzw. Kreuzen\_bis spielen um die optimale Zusammenstellung zu finden.
- Man sollte außerdem Divisionen durch 0 und ähnliches vermeiden, indem man die Variablen-ober- und -unter-grenzen entsprechend wählt.

## Include-Dateien

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

## Definitionen

```
#define word unsigned int
#define byte unsigned char
#define V (float)Variablen[n]
#define Anzahl_der_Generationen 100
#define Variablen_pro_GEN 4
#define GENE_pro_Generation 150
#define Kreuzen_von 0
#define Kreuzen_bis 25
#define Mutationsfaktor 5
#define Mutationswirkung 2
#define oben0 500
#define unten0 1
#define oben1 300
#define unten1 -300
#define oben2 10
#define unten2 -500
#define oben3 30000
#define unten3 1
```

## Erstellen einer Anfangspopulation

```
void Anfangspopulation_erstellen
(int Variablen [GENE_pro_Generation][Variablen_pro_GEN],
 int Grenzen [Variablen_pro_GEN][2])
{
    int m,n;
    gotoxy (2,2); printf ("Anfangspopulation erstellen ");
    for (n=0; n<GENE_pro_Generation; n++)
        for (m=0; m<Variablen_pro_GEN; m++)
            {
                Variablen [n][m]=(random (Grenzen [m][1]-Grenzen [m][0]))+
                    Grenzen [m][0];
            }
}
```

## Fitness

```
void Fitness (int Variablen [GENE_pro_Generation][Variablen_pro_GEN],
 float Ergebnisse [GENE_pro_Generation])
{
    int n;
    gotoxy (2,2); printf ("Fitness berechnen ");
    for (n=0; n<GENE_pro_Generation; n++)
        {
            Ergebnisse [n]=( V[0]*V[0] )*( (V[1]/V[3]) - (V[2]/V[0]) );
        }
}
```

## Kurzbeschreibung

Wie im Kapitel 'Beispiel für die Verknüpfung der Variablen' beschrieben, können man durch Verändern der Zeile Ergebnisse[n]=.... beliebige Formeln erstellt werden.

## Sortieren

```
void Sortieren (int Variablen [GENE_pro_Generation][Variablen_pro_GEN],
               float Ergebnisse [GENE_pro_Generation])
{
    int i, j, k, Var [Variablen_pro_GEN];
    float v;

    gotoxy (2, 2); printf ("Sortieren");
    for (i=1; i<GENE_pro_Generation; i++)
    {
        v=Ergebnisse [i]; j=i;
        for (k=0; k<Variablen_pro_GEN; k++)
        {
            Var [k]= Variablen [i][k];
        }
        while (j>=1 && (fabs(Ergebnisse [j-1])>fabs(v)) // Sortierrichtung
        {
            Ergebnisse [j]=Ergebnisse [j-1];
            for (k=0; k<Variablen_pro_GEN; k++)
            {
                Variablen [j][k]=Variablen [j-1][k];
            }
            j--;
        }
        Ergebnisse [j]=v;
        for (k=0; k<Variablen_pro_GEN; k++)
        {
            Variablen [j][k]= Var [k];
        }
    }
}
```

## Kurzbeschreibung

Hier werden die Fitness-Werte (Ergebnisse[n]) geordnet. Es wird Insertion-Sort verwendet, es kann aber im Prinzip jeder Sortieralgorithmus verwendet werden. Die Zeile die mit Sortierrichtung gekennzeichnet ist ist ausschlaggebend dafür, ob die Ergebnisse der Fitnessfunktion auf- oder absteigend sortiert werden. So wie es bereits angeführt ist, werden die Ergebnisse so sortiert, daß der Wert Null das beste Ergebnis ist und alle anderen Ergebnisse dementsprechend schlechter sind (Je kleiner die absoluten Werte der Ergebnisse sind, desto besser... fabs() ist eine Funktion von C++). Um die Variablen auf das höchstmögliche Ergebnis zu optimieren läßt man die fabs() weg und ändert das '>' in ein '<'-Zeichen. Will man die Variablen auf das kleinste Ergebnis optimieren so läßt man nur die beiden fabs() weg. Mit dem fabs() erhält man immer den Absolutbetrag der Ergebnisse, verwendet man fabs() und das '>'-Zeichen, dann werden die Variablen, wie auch in der Prozedur angeführt, so optimiert, daß das das angestrebte Ergebnis 0 ist.

## Mutation

```
void Mutation (int Variablen [GENE_pro_Generation][Variablen_pro_GEN],
              int Grenzen [Variablen_pro_GEN][2])
{
    int n, m, variable, welche;
    gotoxy (2, 2); printf ("Mutation");
    for (n=0; n<Mutationsfaktor; n++)
    {
        welche=random (GENE_pro_Generation);
        for (m=0; m<Mutationswirkung; m++)
        {
            variable=random (Variablen_pro_GEN);
            Variablen [welche][variable]=
                (random (Grenzen [variable][1]-Grenzen [variable][0]))+
                Grenzen [variable][0];
        }
    }
}
```

## Beschreibung

In diesem Teil des Programmes werden einige GENE mutiert.

## Kreuzen

```
void Kreuzen (int Variablen [GENE_pro_Generation][Variablen_pro_GEN])
{
    int n, k, Teilung;
    gotoxy (2, 2); printf ("Kreuzen");
    for (n=Kreuzen_von; n<Kreuzen_bis; n+=2)
    {
        Teilung=random (Variablen_pro_GEN);
        for (k=0; k<Teilung; k++)
        {
            Variablen [GENE_pro_Generation-n-1][k]=Variablen [n][k];
        }
        for (k=Teilung; k<=Variablen_pro_GEN; k++)
        {
            Variablen [GENE_pro_Generation-n-1][k]=Variablen [n+1][k];
        }
        Teilung=random (Variablen_pro_GEN);
        for (k=0; k<Teilung; k++)
        {
            Variablen [GENE_pro_Generation-n-2][k]=Variablen [n][k];
        }
        for (k=Teilung; k<=Variablen_pro_GEN; k++)
```

```
{
    Variablen [GENE_pro_Generation-n-2][k]=Variablen [n+1][k];
}
}
```

## Beschreibung

In diesem Kode-Teil werden die besten GENE miteinander gekreuzt.

## Ausgabe

```
void Ausgabe (int Variablen [GENE_pro_Generation][Variablen_pro_GEN],
             float Ergebnisse [GENE_pro_Generation])
{
    int n;
    gotoxy (2, 2); printf ("Ausgabe");
    gotoxy (2, 6); printf ("Variablen");
    for (n=0; n<Variablen_pro_GEN; n++)
    {
        gotoxy (2, 7+n); printf (" Variable %i: %i",
                                n, Variablen [0][n]);
    }
    gotoxy (2, 22); printf ("bestes Ergebnis: %F",
                           Ergebnisse [0]);
}
```

## main

```
void main ()
{
    int n, Variablen [GENE_pro_Generation][Variablen_pro_GEN];
    int Grenzen [Variablen_pro_GEN][2];
    float Ergebnisse [GENE_pro_Generation];

    Grenzen [0][0]=unten0; Grenzen [0][1]=oben0; Grenzen [1][0]=unten1;
    Grenzen [1][1]=oben1; Grenzen [2][0]=unten2; Grenzen [2][1]=oben2;
    Grenzen [3][0]=unten3; Grenzen [3][1]=oben3;

    clrscr ();
    randomize ();
    Anfangspopulation_erstellen (Variablen, Grenzen); // Punkt 1
    Fitness (Variablen, Ergebnisse);
    Sortieren (Variablen, Ergebnisse);
    for (n=0; n<Anzahl_der_Generationen; n++)
    {
        Kreuzen (Variablen);
        Mutation (Variablen, Grenzen);
        Fitness (Variablen, Ergebnisse);
        Sortieren (Variablen, Ergebnisse);
        Ausgabe (Variablen, Ergebnisse); // Punkt 2
    }
    gotoxy (36, 23); printf ("!TASTE!");
    getch ();
}
```

## Kurzbeschreibung

Der Programmcode den Sie hier sehen, ist ein einfaches Gerüst eines Evolutionsalgorithmus (von Punkt 1 bis Punkt 2). Man versucht mit Evolutionsalgorithmen die Vererbungsstrategien in der Natur nachzubilden. Wir erstellen zuerst eine sogenannte 'Anfangspopulation' (siehe auch Kapitel: Anfangspopulation), diese enthält für jede Variable, die durch das Programm verändert werden kann, einen zufälligen Wert. Wieviele Variablen verwendet werden sollen, kann man mit der Definition Variablen\_pro\_GEN (siehe auch Kapitel: Definition) festlegen. Man muß entsprechend der Anzahl von Variablen durch die Definitionen für deren obere und untere Grenzen ergänzen. Außerdem müssen diese Definitionen in die Variablen 'grenzen' geschrieben werden. Wieviele Wertegruppen (Variablengruppen, GENE) die Anfangspopulation und somit jede folgende Generation enthalten soll, wird in der Definition GENE\_pro\_Generation bestimmt (siehe Kapitel: Definitionen). Die Variablen sind im Feld Variablen [GENE\_pro\_Generation][Variablen\_pro\_GEN] gespeichert.

## Literatur

Thomas Otto; Auf zufälligen Wegen zum Ziel, Random-Walk-Algorithmus in der Evaluationstechnik; c't, 1994, H5, S.258.

Christiak Rieck; Modell Natur, Naturanaloge Verfahren in der Computer-Simulation; c't, 1993, H.11, S 201. □