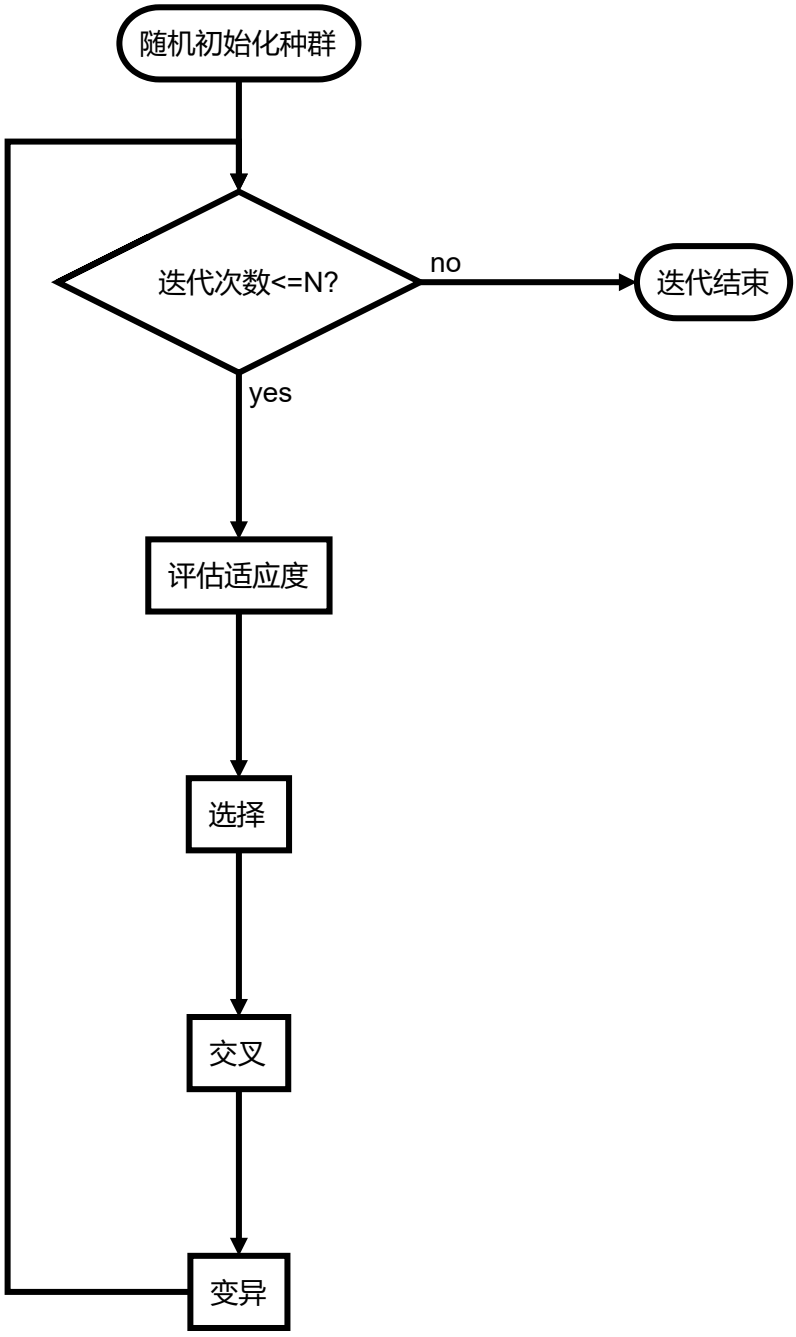
遗传算法

遗传算法（Genetic Algorithm，GA）最早是由美国的 *John holland*于20世纪70年代提出,该算法是根据大自然中生物体进化规律而设计提出的。是模拟达尔文生物进化的**自然选择**和**遗传学**机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索**最优解**的方法，它提供了一种求解复杂问题的通用框架，它不依赖于问题的具体领域，对问题的种类具有很强的**鲁棒性**。

一般流程

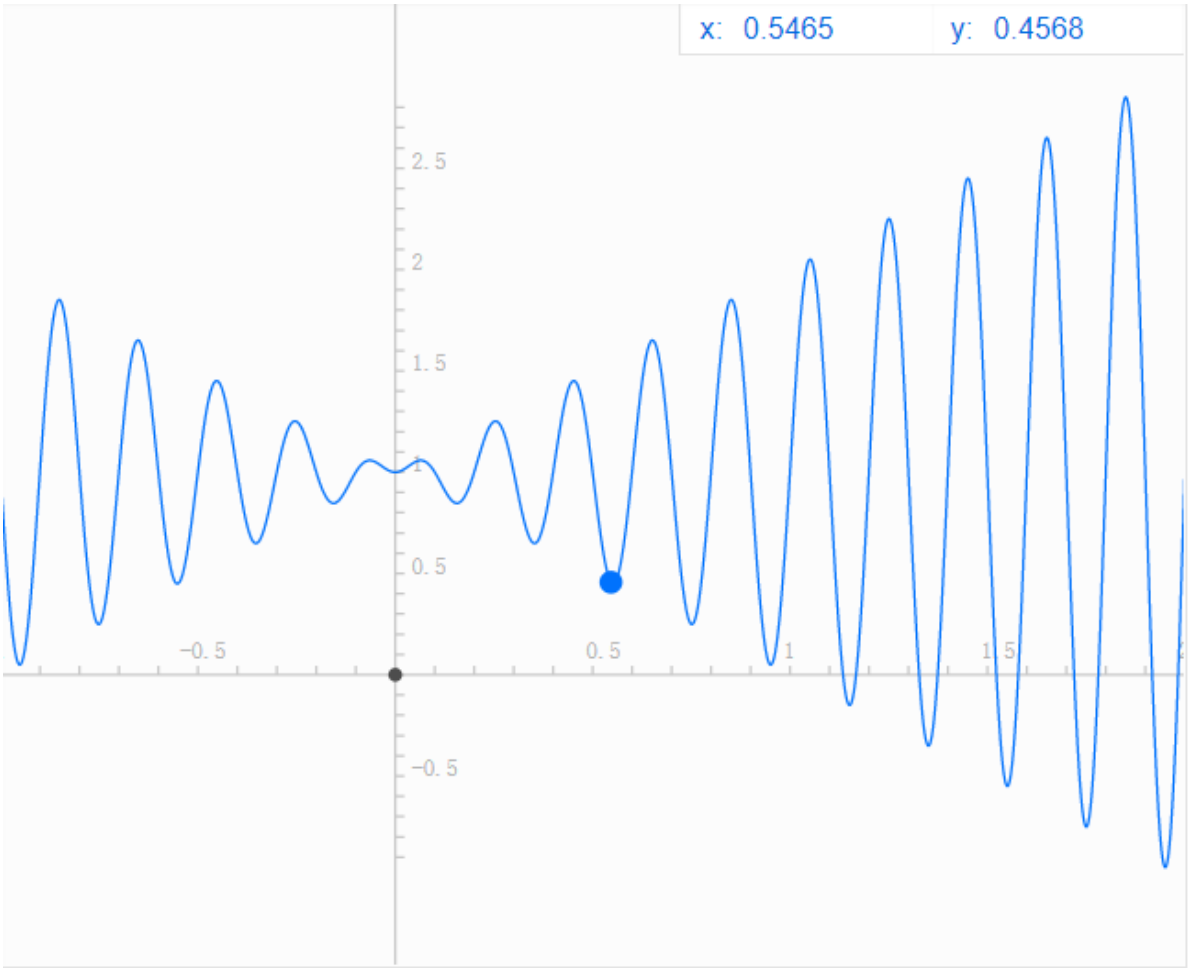


问题引出

求解$y = xsin(10\pi x) + 1,\ x \in [-1, 2]$的**最大值。**

用遗传算法求解

首先我们可以先看一下他的图像长什么样：



看图我们可以知道，这个函数有**许多极大极小值**，像**一座座山**一样。

随机初始化种群

其实这一步就相当于在这个函数上**随机取点**，问题主要是点的**精度**问题，这里我们取小数点后六位，但是貌似没有什么随机函数能够控制生成的随机数的精度，所以解决方法就是对点进行**编码**,编码还能方便交叉和变异的操作。

编码

这里采用**二进制编码**，由于是精确到小数点后六位，区间长度是2-（-1）=3，所以至少要把区间分为$3 * 10^6$等份，又因为$2^{21} < 3 * 10^6 < 2^{22}$,所以编码的长度至少要是**22**位。

解码

由于后面需要计算适应度，需要将编码转为实际的点的坐标，于是我们需要解码操作，分为两步。

1. 将编码转化为十进制数。

$(a_0a_1a_2\dots a_{21})_2 = (\sum_{i=0}^{21} a_i * 2^i)_{10} = x_t$

2. 将十进制数转为区间内的实数。

$x = -1 + \frac{(2-(-1))}{2^{22}-1}x_t$

随机初始化编码实现如下：

```
int** initPopulation(){
    int** pop = new int*[num];
    for(int i=0;i < num;i++){
        pop[i] = new int[lenOfDNA];
        for(int j=0;j < lenOfDNA;j++){
            pop[i][j] = rand()%2;
        }
    }
    return pop;
}
```

解码代码：

```
double* decode(int** pop){
    int m = _msize(pop)/sizeof(*pop);              //计算种群的个体数量，注意_msize是widows独有
    double* X = new double[m];                     //申请数组，代表每个个体的坐标
    for(int i=0;i < m;i++){
        int dna = 0;
        for(int j=0;j < lenOfDNA;j++){
            dna += pop[i][j]*pow(2,j)*1.0;         //对二进制编码加权求和转为十进制
        }
        double x = X1 + dna*(X2-X1)/(pow(2,22)-1.); //转为对应区间的实数坐标
        X[i] = x;
    }
    return X;
}
```

适应度函数

由于我们要求这个函数的最大值，所以y越大的点越好，更能**适应**，被选中的概率就更高，然而个体的y值**有正有负**，而**概率**却是**非负**的，所以我们可以将当前个体的y值减去当前种群中最小的y值，这样就可以保证适应度非负了。

但是我们不能只去选择y大的而完全不选择y小的，因为有可能y大的个体**都在一个山上**，也就是**一个极大值附近**，这样就会陷入**局部最优解**，所以最后要在适应度最后加上一个很小的**正数**，保证有最小y的个体也有被选择的概率。

代码如下：

```
double* fitness(int** pop){
    double* X = decode(pop);                    //先解码获得坐标X的矩阵
    double* Y = F(X);                           //解坐标对应的Y值
    int m = _msize(Y)/sizeof(double);
    double mini = min(Y);
    double* fit = new double[m];
    for(int i = 0;i < m;i++){
        fit[i] = Y[i] - mini + eplison;
    }
    delete [] X;          //释放空间
    X = NULL;
    return fit;
}
```

累加概率

累加概率区间为[0,1]，**适应度越高**的个体占领的**区间也就越大**，那么被选中的概率也就越大，我们只需从第一个个体开始，分别计算他们的概率并加上前一次的**累加概率**，这样就能获得一个累加概率**数组**，并且累加概率数组中的元素值肯定是**递增**的。

代码如下：

```
double* prop(double* fit){
    int m = _msize(fit)/sizeof(double);
    double sum = 0;
    for(int i = 0;i < m;i++){
        sum += fit[i];
    }
    double* fitProb = new double[m];
    fitProb[0] = fit[0]/sum;
    for(int i = 1;i < m;i++){
        fitProb[i] = fitProb[i-1] + fit[i]/sum;      //概率累加
    }
    delete [] fit;               //释放空间
    fit = NULL;
    return fitProb;
}
```

选择

有了累加概率数组，我们使用**轮盘赌算法**选择个体了，只需产生一个[0,1]之间的随机数，它落在数组的**第几个区间**内，就选择**第几个个体**，只需要将这个操作**重复n次**(n为种群中个体的数量)，也就是对每一个个体进行选择，然后产生一个**新的种群**，得到了一个新的编码矩阵。

代码如下：

```
int** select(int** pop,double* fitProb){
    int m = _msize(pop)/sizeof(*pop);
    int* sIndex = new int[m];
    for(int i = 0;i < m;i++){
        double sr = getRand();           //产生随机因子
        int index = binarySearch(sr,fitProb);  //选择个体
        sIndex[i] = index;
```

```
    }
    int* index = getIndex(sIndex);        //去除重复下标
    int n = _msize(index)/sizeof(int);
    int** newPop = new int*[n];
    for(int i = 0;i < n;i++){
        newPop[i] = new int[lenOfDNA];
        for(int j = 0;j < lenOfDNA;j++){
            newPop[i][j] = pop[index[i]][j];      //深拷贝
        }
    }
     for(int i = 0;i < m;i++){
        delete [] pop[i];
        pop[i] = NULL;
    }
    delete [] pop;            //释放空间
    pop = NULL;
    delete [] fitProb;        //释放空间
    fitProb = NULL;
    return newPop;
}
```

交叉

我们通过**选择**获得一个**新的种群**，这些个体的适应度还不错，但为了产生**更高适应度**的个体，就要进行交叉操作，所谓的交叉也就是生物上所说的繁殖，来获得更好的个体，这里我选择了**单点交叉**，也就是在父亲个体的编码上**随机产生一个点**，并将母亲个体的编码的相应位置往后的编码全部给父亲个体的编码产生的那个点之后的编码，这样就产生了一个新的个体，但是交叉是**有概率**的，并不是所有个体都要进行交叉操作。

代码如下：

```
int** reproduce(int** pop){
    int m = _msize(pop)/sizeof(*pop);
    int** nextPop = new int*[m];
    for(int i = 0;i < m;i++){
        nextPop[i] = new int[lenOfDNA];
    }
    int* child = new int[lenOfDNA];
    int* mother = new int[lenOfDNA];
    for(int i = 0;i < m;i++){
        for(int j = 0;j < lenOfDNA;j++){
            child[j] = pop[i][j];
        }
        if(getRand()<reproduce_rate){
            int other = rand()%m;
            for(int j = 0;j < lenOfDNA;j++){
                mother[j] = pop[other][j];
            }
            int cross_point = rand()%lenOfDNA;
            for(int j = cross_point;j < lenOfDNA;j++){
                child[j] = mother[j];
            }
        }
        mutation(child);
        for(int j = 0;j < lenOfDNA;j++){
```

```
            nextPop[i][j] = child[j];
        }
    }
    delete [] child;
    child = NULL;
    delete [] mother;
    mother = NULL;
    for(int i = 0;i < m;i++){
        delete [] pop[i];
        pop[i] = NULL;
    }
    delete [] pop;
    pop = NULL;
    return nextPop;
}
```

变异

变异是**有概率**发生的，也就是产生新的个体后，他有可能发生变异，也就是编码上的**某个点**发生了突变，从1变成0或者从0变成1，变异的目的主要是使遗传算法具有局部的**随机搜索**能力，以及让个体**跳出局部最优解**，防止**早熟**，寻找一个全局最优解。

代码如下：

```
void mutation(int* child){
    if(getRand()<mutation_rate){
        int mutation_point = rand()%lenOfDNA;        //变异点
        child[mutation_point] ^= 1;
    }
}
```

最后只需将这些模块**整合**在一起就能求解这个问题了。

完整代码

```
#include<iostream>
#include<stdlib.h>
#include<math.h>
#include<time.h>
using namespace std;

//定义圆周率
#define PI 3.141592657

//定义常数项
#define eplison 1e-3

//定义域
#define X1 -1.000000
#define X2 2.000000

//定义初始种群中有830个个体
#define num 830
//精度需要小数点后六位，编码长度至少需要22位
```

```
#define lenOfDNA 22

//繁殖率
#define reproduce_rate 0.7

//变异率
#define mutation_rate 0.25

//定义函数
double* F(double* X){
    int m = _msize(X)/sizeof(double);          //计算坐标个数
    double* Y = new double[m];
    for(int i = 0;i < m;i++){
        Y[i] = X[i]*sin(10*PI*X[i]) +1.0;
    }
    return Y;
}

//随机初始化种群并对其编码
int** initPopulation(){
    int** pop = new int*[num];
    for(int i=0;i < num;i++){
        pop[i] = new int[lenOfDNA];
        for(int j=0;j < lenOfDNA;j++){
            pop[i][j] = rand()%2;            //初始化DNA，注意，每次初始化都是相同的结果，因为用的
rand()
        }
    }
    return pop;
}

//解码操作，将DNA转为坐标x
double* decode(int** pop){
    int m = _msize(pop)/sizeof(*pop);              //计算种群的个体数量，注意_msize是widows独有
    double* X = new double[m];                     //申请数组，代表每个个体的坐标
    for(int i=0;i < m;i++){
        int dna = 0;
        for(int j=0;j < lenOfDNA;j++){
            dna += pop[i][j]*pow(2,j)*1.0;         //对二进制编码加权求和转为十进制
        }
        double x = X1 + dna*(X2-X1)/(pow(2,22)-1.); //转为对应区间的实数坐标
        X[i] = x;
    }
    return X;
}

//找出最小值,双指针法
double min(double* Y){
    int m = _msize(Y)/sizeof(double);
    double min = Y[0];
    for(int i = 1;i < m;i++){
        min = min<Y[i]?min:Y[i];
    }
    return min;
}

//定义适应度函数,群体中的最大值减去最小值加上一个非常小的数来评估
double* fitness(int** pop){
```

```
    double* X = decode(pop);                    //先解码获得坐标X的矩阵
    double* Y = F(X);                           //解坐标对应的Y值
    int m = _msize(Y)/sizeof(double);
    double mini = min(Y);
    double* fit = new double[m];
    for(int i = 0;i < m;i++){
        fit[i] = Y[i] - mini + eplison;
    }
    delete [] X;            //释放空间
    X = NULL;
    return fit;
}

//计算累计概率
double* prop(double* fit){
    int m = _msize(fit)/sizeof(double);
    double sum = 0;
    for(int i = 0;i < m;i++){
        sum += fit[i];
    }
    double* fitProb = new double[m];
    fitProb[0] = fit[0]/sum;
    for(int i = 1;i < m;i++){
        fitProb[i] = fitProb[i-1] + fit[i]/sum;      //概率累加
    }
    delete [] fit;                  //释放空间
    fit = NULL;
    return fitProb;
}

//产生0~1之间的随机数
double getRand(){
    return rand()/(double(RAND_MAX));
}

//二分法查找
int binarySearch(double sr,double* fitProb){
    if(sr<=fitProb[0]){
        return 0;
    }
    int m = _msize(fitProb)/sizeof(double);
    int left = 0;
    int right = m-1;
    int mid = (left+right)/2;
    while(left+1<right){
        if(sr>fitProb[mid]){
            left = mid;
        }else if (sr<fitProb[mid]){
            right = mid;
        }else{
            return mid;
        }
        mid = (left+right)/2;
    }
    return right;
}

//对数组排序
```

```
void sort(int left,int right,int* index){
    if(left>=right){
        return;
    }
    int i = left;
    int j = right;
    int base = index[left];
    int temp;
    while(i<j){
        while(index[j]>=base&&i<j){
            j--;
        }
        while(index[i]<=base&&i<j){
            i++;
        }
        if(i<j){
            temp = index[i];
            index[i] = index[j];
            index[j] = temp;
        }
    }
    index[left] = index[i];
    index[i] = base;
    sort(left,i-1,index);
    sort(i+1,right,index);
}

//删除重复元素
int* getIndex(int* index){
    int m = _msize(index)/sizeof(int);
    sort(0,m-1,index);
    int i = 1;
    int sum = 0;
    while(i<m){
        if(index[i]!=index[i-1]){
            sum++;
            index[sum] = index[i];
        }
        i++;
    }
    int* newIndex = new int[sum+1];
    for(int i = 0;i<=sum;i++){
        newIndex[i] = index[i];
    }
    delete [] index;            //释放空间
    return newIndex;
}

//选择函数，适应度高的更容易被选择
int** select(int** pop,double* fitProb){
    int m = _msize(pop)/sizeof(*pop);
    int* sIndex = new int[m];
    for(int i = 0;i < m;i++){
        double sr = getRand();              //产生随机因子
        int index = binarySearch(sr,fitProb);   //选择个体
        sIndex[i] = index;
    }
    int* index = getIndex(sIndex);
```

```
    int n = _msize(index)/sizeof(int);
    int** newPop = new int*[n];
    for(int i = 0;i < n;i++){
        newPop[i] = new int[lenOfDNA];
        for(int j = 0;j < lenOfDNA;j++){
            newPop[i][j] = pop[index[i]][j];       //深拷贝
        }
    }
     for(int i = 0;i < m;i++){
        delete [] pop[i];
        pop[i] = NULL;
    }
    delete [] pop;              //释放空间
    pop = NULL;
    delete [] fitProb;          //释放空间
    fitProb = NULL;
    return newPop;
}

//变异函数
void mutation(int* child){
    if(getRand()<mutation_rate){
        int mutation_point = rand()%lenOfDNA;        //变异点
        child[mutation_point] ^= 1;
    }
}

//繁殖，交叉函数
int** reproduce(int** pop){
    int m = _msize(pop)/sizeof(*pop);
    int** nextPop = new int*[m];
    for(int i = 0;i < m;i++){
        nextPop[i] = new int[lenOfDNA];
    }
    int* child = new int[lenOfDNA];
    int* mother = new int[lenOfDNA];
    for(int i = 0;i < m;i++){
        for(int j = 0;j < lenOfDNA;j++){
            child[j] = pop[i][j];
        }
        if(getRand()<reproduce_rate){
            int other = rand()%m;
            for(int j = 0;j < lenOfDNA;j++){
                mother[j] = pop[other][j];
            }
            int cross_point = rand()%lenOfDNA;
            for(int j = cross_point;j < lenOfDNA;j++){
                child[j] = mother[j];
            }
        }
        mutation(child);
        for(int j = 0;j < lenOfDNA;j++){
            nextPop[i][j] = child[j];
        }
    }
    delete [] child;
    child = NULL;
    delete [] mother;
```

```
    mother = NULL;
    for(int i = 0;i < m;i++){
        delete [] pop[i];
        pop[i] = NULL;
    }
    delete [] pop;
    pop = NULL;
    return nextPop;
}

//迭代N次
void iteration(int N){
    int** pop = initPopulation();          //初始化种群
    for(int i = 0;i < N;i++){
        double* fit = fitness(pop);        //适应度评价
        double* fitProb = prop(fit);       //获得累积概率
        int** newpop = select(pop,fitProb); //选择
        pop = reproduce(newpop);  //交叉变异
    }
    cout<<"last iteration:"<<endl;
    double* X = decode(pop);
    double* Y = F(X);
    int m = _msize(X)/sizeof(double);
    for(int i = 0;i < m;i++){
        cout<<"x:"<<X[i]<<endl;
    }
    int n = _msize(Y)/sizeof(double);
    for(int i = 0;i < m;i++){
        cout<<"y:"<<Y[i]<<endl;
    }
}

int main(){
    iteration(50);
    getchar();
    return 0;
}
```

迭代结果

x=1.851740,y=2.84897,这与最大值已经十分接近了。

备注

遗传算法是我最近才接触的，所以对于他的理解还是停留在很浅的层面，所以文章写的不好，请大家不要嘲笑我。

这个遗传算法的代码是我学校的数据结构与算法周上写的，老师不允许我们使用任何的工具库，所以我选择了c++作为实现语言，而我又一年多没碰过c++了，所以对于指针的一些操作可能稀烂，代码写的可读性差，希望大家谅解。