Генетический алгоритм в задачах газодинамики

Теория по газодинамике:

Генетический алгоритм – алгоритм оптимизации, использующийся во многих областях физики, включая газодинамику. Он может использоваться для оптимизации физических характеристик объекта, оптимизации газовых потоков и т. д. Для проекта была выбрана задача о разработке оптимальных стратегий управления турбулентными потоками. Для начала введем основные термины, которые будут использоваться на протяжении всего проекта:

- Турбулентный поток – поток, образующий вихревые потоки, т. е. хаотичные потоки, которые не имеют общего описания до сих пор
- Ламинарный поток – поток, при котором все газовые слои движутся по направлению к основному направлению движения потока
- Число Рейнольдса (далее Re) – Число, созданное английским механиком, физиком и инженером Осборном Рейнольдсем (1842 – 1912 г.г.) в 1883 году.
  $Re = \frac{v*d}{\mu}$, где v – скорость потока (м/с), d – диаметр трубы (м), μ – кинематическая вязкость ($м^2/с$).

Число Рейнольдса является безразмерной величиной, характеризующая турбулентность потока. Для своего экспериментального доказательства Рейнольдс запускал потоки жидкости с разными скоросями, разными диаметрами трубы и разными типами жидкости, и после чего смог установить закономерность, выроденной в формуле. Также Рейнольдс менял множество характеристик, и есть разные интерпритации данной фомрулы, но в проекте была выбрана формула со скоростью, диаметром и кинематической вязкостью.

Рейнольдс экспериментально доказал, что при уменьшении числа Re поток становится ламинарнее. В алгоритме мы будем считать, что если Re < 1000, то поток ламинарный.

Теория по генетическому алгоритму:

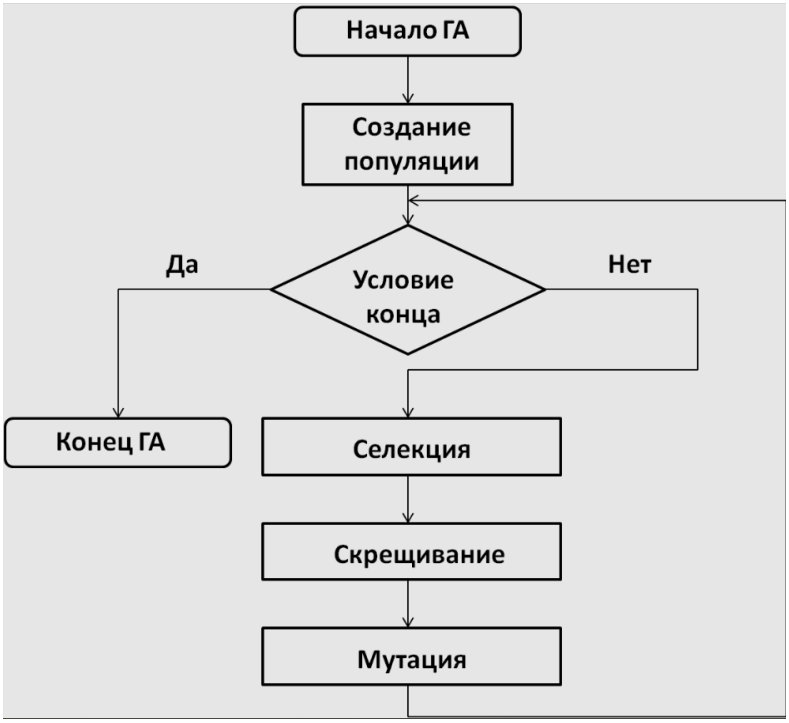


Рис. 1. Схема работы генетического алгоритма

Генетический алгоритм (далее ГА) – алгоритм оптимизации, основанный на принципах естественного отбора и генетики. Это эвристический алгоритм, то есть в нем нас будет интересовать не множество решений, а хотя бы 1 решение, которое удовлетворяет нашим условиям. Как уже было сказано, данный алгоритм основан на принципе естественного отбора в природе, и поэтому мы также введем необходимые определения, которые будем использовать для реализации ГА:

- Ген – одно из множества характеристик индивидуума, участвующего в ГА.
- Хромосома – набор генов.
- Начальная популяция – набор хромосом, участвующих в ГА
- Семья – некоторое множество хромосом

В нашей задаче у нас будут 3 типа генов: скорость, диаметр и кинематическая вязкость. 1 хромосома – набор из гена скорости, диаметра и кинематической вязкости.

Для начала алгоритма мы должны создать начальную популяцию, с фиксированным числом хромосом и уже заданными генами у каждого из хромосом.

В программе мы будем рандомно создавать гены хромосом, при этом количество хромосом клиент вводит самостоятельно.

Далее, после задания начальной популяции мы начинаем основной цикл генетического алгоритма

Сначала происходит отбор. Отбор возможно реализовывать по-разному. В проекте мы будем производить турнирный отбор. Всех хромосом мы будем разбивать на семьи, с которыми будем в дальнейшем работать. Семьи будут состоять из трех хромосом. Остаточные хромосомы не будут меняться в процессе ГА.

В семье мы будем отбирать двух из трех наиболее приспособленных хромосом и называть их родителями. Третья хромосома умрет в процессе естественного отбора.

Далее происходит скрещивание. В семье мы оставшихся родителей скрещиваем, и получаем новую хромосому с наилучшими генами от родителей. И так мы проделываем со всеми семьями, не меняя размер популяции.

Далее происходит мутация. Мутация улучшает 1 ген хромосомы.

Решением ГА является нахождение хотя бы одной приспособленной хромосомы.

Также важно, что ГА проходит конечное число итераций, заданное клиентом.

Также важно учесть, что скрещивание и мутация имеют вероятность происхождения. То есть хромосом не всегда будет мутировать, а родители не всегда будут скрещиваться.

Генетический алгоритм в газодинамике:

В проекте отбор будет происходить по наилучшему Re. То есть в семье умрет хромосом с наибольшим Re. Данный отбор не учитывает, что в процессе отбора мы могли потерять хромосома, который имел наибольшее Re, но также имел ценный ген, который он мог передать при скрещивании как родитель.

При скрещивании мы даем ребенку наименьший ген скорости двух родителей, наименьший ген диаметра двух родителей и наибольший ген кинематической вязкости двух родителей.

При мутации мы будем уменьшать в 2 раза ген скорости и диаметра и увеличивать в 2 раза ген кинематической вязкости. Для одного хромосома при одной итерации рандомно мутирует 1 ген.

Если после этого мы найдем хромосому с $Re < 1000$, то задача будет решена

Если такого хромосома не найдется после заданного числа итераций, то популяция вымрет, и задача не будет решена.

Описание структуры коды и отдельных его частей

```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>

class Chromosome
{
public:
    Chromosome()
    {
        speed = diameter = (1 + rand() % 100) * 0.01;
        viscosity = (1 + rand() % 100) * 0.0000000001;
        countRe();
    }
    Chromosome(double speed, double diameter, double viscosity) : speed(speed), diameter(diameter), viscosity(viscosity)
    {
        countRe();
    }
    Chromosome(const Chromosome& src)
    {
        this->speed = src.speed;
        this->diameter = src.diameter;
        this->viscosity = src.viscosity;
        this->Re = src.Re;
        countRe();
    }
    ~Chromosome() {}
```

Рис. 2. Класс функции, конструкторы

Для проекта мы будем использовать 3 библиотеки. Одна основная: <iostram>, а две другие для использования функций rand() и srand()

Объявляется класс Chromosome, каждый элемент которого хранит 3 переменные – скорость, диаметр, вязкость, то есть в ГА является хромосомом.

Первый конструктор нужен для инициализации переменных. Наш конструктор создает рандомные значения для переменных и считает Re для них. Вызывается сразу, когда мы объявляем переменную нашего класса в main. Используется в 88 строке при объявлении массива

Второй конструктор нужен для описания состояния полей класса. Он нужен для того, чтобы мы в main смогли работать с полями класса, вносить туда переменные тех типов, которые описаны в конструкторе, столько переменных, сколько описаны в конструкторе. Используется в 99 строке при присваивании элемента массива нового элемента класса Chromosome. Также испольуем countRe, так как мы всегда должны обновлять или считать Re при изменении генов.

Третий конструктор предназначен для копирования элементов при вызове методов по значению. Данный конструктор нужен, когда мы пишем метод, в котором принимается значения из класса. В описании данного конструктора мы присваиваем полям класса ссылку на копируемый элемент.

```
28      void setSpeed(double s)
29      {
30          speed = s;
31          countRe();
32      }
33
34      void setDiameter(double d)
35      {
36          diameter = d;
37          countRe();
38      }
39      void setViscosity(double v)
40      {
41          viscosity = v;
42          countRe();
43      }
44
45      double getSpeed() { return speed; }
46      double getDiameter() { return diameter; }
47      double getViscosity() { return viscosity; }
48      double getRe() { return Re; }
49
50
51      void countRe()
52      {
53          Re = speed * diameter / viscosity;
54      }
```

Рис. 3. Геттеры и сеттеры

Реализованы геттеры для всех четырех полей класса для получения отдельного гена хромосома. Все значения типа double.

Сеттеры реализованы для генов. Для Re мы не можем написать set, так как не можем менять Re, но при этом можем подглядывать, используя метод getRe. В реализации сеттеров мы обязательно должны обновлять значения Re, так как мы меняем какой то ген.

```
55
56      bool operator> (const Chromosome& other)
57      {
58          return Re > other.Re;
59      }
60      bool operator< (const Chromosome& other)
61      {
62          return Re < other.Re;
63      }
64      friend std::ostream& operator<< (std::ostream& stream, const Chromosome& population);
65      friend std::istream& operator>> (std::istream& stream, Chromosome& population);
66      Chromosome& operator =(const Chromosome& other)
67      {
68          this->speed = other.speed;
69          this->diameter = other.diameter;
70          this->viscosity = other.viscosity;
71          this->Re = other.Re;
72          return *this;
73      }
74  private:
75      double speed, diameter, viscosity, Re;
76  };
```

Рис. 4. Операторы

Операторы больше меньше имеют тип bool, то есть возвращают true или false. Принимают константную переменную, так как они не должны меняться. Передаем значение по ссылке. Возвращаем результат сравнения.

Операторы ввода вывода принимают на вход два аргумента: ссылку на объект и значение для вывода. Возвращает уже новую ссылку на объект.

Оператор присваивания по реализации схож с конструктором копирования, только в начале нужно очистить память, так как в конструкторе копирования этого делать не надо, потому что конструктор вызывается сразу при создании нового объекта, а в операторе присваивания у нас может быть уже заполненный объект, и тогда сможет произойти утечка памяти, но так как внутри класса нет работы с динамической памятью, то реализация аналогична.

```
80  int main()
81  {
82      srand(time(0));
83      const double RE_INF = 10e8;
84      double bestRe = RE_INF;
85      int popSize, maxIterations;
86      double probCrossover, probMutation;
87      std::cin >> popSize >> maxIterations >> probCrossover >> probMutation;
88      Chromosome* chr = new Chromosome[popSize];
89
90      for (int i = 0; bestRe > 1000 && i < maxIterations; i++)
91      {
92
93          std::cout << "Iteration number: " << i << std::endl << std::endl;
94          std::cout << "Crossed Family: " << std::endl;
95
96          for (int j = 0; j < popSize - 2; j += 3)
97          {
98              if ((rand() % 1000) * 0.001 < probCrossover)
99              {
100                 sortFamily(chr[j], chr[j + 1], chr[j + 2]);
101                 chr[j + 2] = Chromosome(std::min(chr[j].getSpeed(), chr[j + 1].getSpeed()),
102                     std::min(chr[j].getDiameter(), chr[j + 1].getDiameter()),
103                     std::max(chr[j].getViscosity(), chr[j + 1].getViscosity()));
104                 std::cout << j << " ";
105             }
106         }
```

Рис. 5. Main. Скрещивание

Считываем размер популяции, максимальное количество итераций, вероятность скрещивания и мутации. Динамически создаем массив типа Chromosome, в котором имеются хромосомы, у которых по нашему первому конструктору уже заполнены все гены.

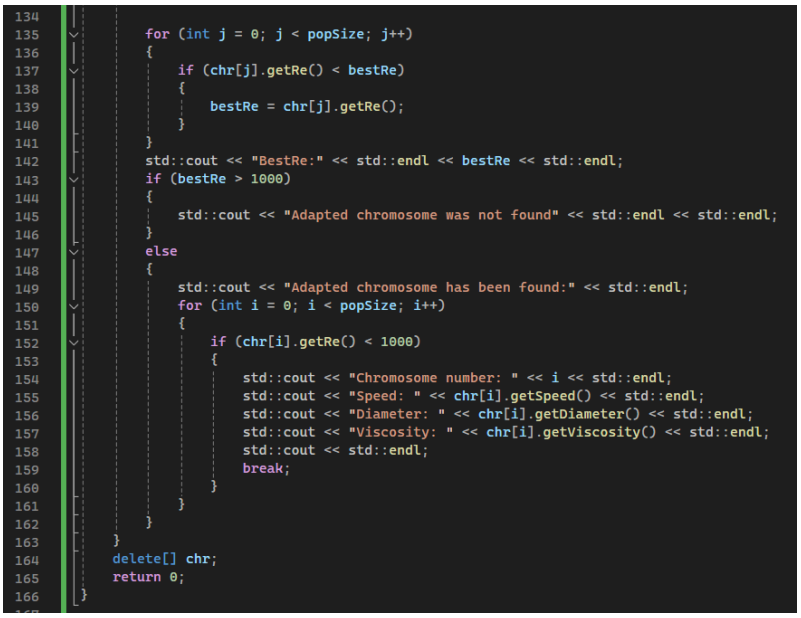
Пишем основной цикл генетического алгоритма, внутри которого цикл скрещивания. В нем, если сработала вероятность скрещивания, делаем отбор элементов с помощью метода sortFamily, который из тройки сортирует так, чтобы правый элемент был наихудшим и производим с помощью 101 строки скрещивание.

В описании ГА говорилось, что у нас умирает хромосом, а за место него рождается новый. Это то же самое, если просто худшему хромосому дать новые гены.

```
109         std::cout << std::endl << "Mutating chromosomes: " << std::endl;
110
111         for (int j = 0; j < popSize; j++)
112         {
113             if ((rand() % 1000) * 0.001 < probMutation)
114             {
115                 int mutGen = rand() % 3;
116
117                 if (mutGen == 0)
118                 {
119                     chr[j].setSpeed(chr[j].getSpeed() / 2);
120                 }
121                 else if (mutGen == 1)
122                 {
123                     chr[j].setDiameter(chr[j].getSpeed() / 2);
124                 }
125                 else if (mutGen == 2)
126                 {
127                     chr[j].setViscosity(chr[j].getViscosity() * 2);
128                 }
129                 std::cout << j << " ";
130             }
131         }
132
133         std::cout << std::endl;
```

Рис. 6. Мутация

Заводим новый цикл. В нем с вероятностью мутации мутируем рандомный ген. Мы с помощью метода set вносим новый ген, равный исходному, деленный пополам, или умноженный на 2.



```
        for (int j = 0; j < popSize; j++)
        {
            if (chr[j].getRe() < bestRe)
            {
                bestRe = chr[j].getRe();
            }
        }
        std::cout << "BestRe:" << std::endl << bestRe << std::endl;
        if (bestRe > 1000)
        {
            std::cout << "Adapted chromosome was not found" << std::endl << std::endl;
        }
        else
        {
            std::cout << "Adapted chromosome has been found:" << std::endl;
            for (int i = 0; i < popSize; i++)
            {
                if (chr[i].getRe() < 1000)
                {
                    std::cout << "Chromosome number: " << i << std::endl;
                    std::cout << "Speed: " << chr[i].getSpeed() << std::endl;
                    std::cout << "Diameter: " << chr[i].getDiameter() << std::endl;
                    std::cout << "Viscosity: " << chr[i].getViscosity() << std::endl;
                    std::cout << std::endl;
                    break;
                }
            }
        }
    }
    delete[] chr;
    return 0;
}
```

Рис. 7. Вывод

Сначала обновляем bestRe в цикле. Если он больше 1000, то идем дальше

Если меньше, то выводим всю информацию про хромосому, которую мы нашли.