Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №1

Дисциплина Моделирование

Тема Программная реализация приближенного аналитического метода и простейших численных алгоритмов первого порядка точности при решении задач Коши для ОДУ

Студент Набиев Ф.М.

Группа ИУ7–63Б

Оценка (баллы)

Преподаватель Градов В.М.

Москва, 2020 г.

ВВЕДЕНИЕ

Цель работы: Изучить методы решения задачи Коши для ОДУ, применив приближенный аналитический метод Пикара и численный метод Эйлера в явном и неявном вариантах.

Задание: Решить уравнение, не имеющее аналитического решения.

1 Теоретическая часть

Имеем систему

$$\begin{cases} u'(x) = f(x,u) \\ u(\xi) = y \end{cases} \tag{1.1}$$

ОДУ здесь можно решить **методом Пикара** (формула 1.2).

$$\begin{gathered} y^{(s)}(x) = \eta + \int_0^x f(t, y^{(s-1)}(t))dt \\ y^{(0)} = \eta \end{gathered} \tag{1.2}$$

Для задачи $u'(x) = u^2 + x^2$, $\eta = 0$ получим 4 приближения (формулы 1.3, 1.4, 1.5, 1.6).

$$y^{(1)} = \frac{x^3}{3} \tag{1.3}$$

$$y^{(2)} = \frac{x^3}{3} + \frac{x^7}{63} \tag{1.4}$$

$$y^{(3)} = \frac{x^3}{3} + \frac{x^7}{63} + \frac{2x^{11}}{2079} + \frac{x^{15}}{59535} \tag{1.5}$$

$$\begin{aligned} y^{(4)} = \frac{x^3}{3} + \frac{x^7}{63} + \frac{2x^{11}}{2079} + \frac{13x^{15}}{218295} + \\ \frac{82x^{19}}{37328445} + \frac{662x^{23}}{10438212015} + \\ \frac{4x^{27}}{3341878155} + \frac{x^{31}}{109876902975} \end{aligned} \tag{1.6}$$

Также уравнение решается **численным методом Эйлера**.
Явная схема (формула 1.7)

$$y_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{1.7}$$

Неявная схема (формула 1.8)

$$y_{n+1} = y_n + h \cdot (f(x_{n+1}, y_{n+1})) \tag{1.8}$$

2 Практическая часть

Для реализации аналитического решения ОДУ методом Пикара в контексте данного задания, необходимо определить представление полинома в программе.

Запишем полином в виде массива, где для $i \in \{0\} \cup \mathbb{N}$ коэффициент одночлена степени i записывается в i-ой позиции этого массива. Такая форма хранения многочлена предоставляет быстрый доступ к составляющим его одночленам по их степени.

Обратим внимание, что в рассматриваемой задаче полином каждого последующего приближения содержит одночлен вида $r \cdot x^{p+4}$, где r — некоторый коэффициент, p — максимальная степень среди всех одночленов полинома предыдущего приближения. Кроме того, минимальная степень одночлена в любом возможном для данной задачи полиноме равна 3. Следовательно, можно не хранить в массиве коэффициенты тех одночленов, которые никогда не будут содержаться в многочлене. Тогда для вычисления степени p i-того одночлена следует использовать следующую формулу:

$$p = 3 + i \cdot 4 \tag{2.1}$$

Теперь необходимо рассмотреть задачи интегрирования и произведения полиномов.

2.1 Интегрирование многочлена

Задача интегрирования полинома это задача интегрирования каждого входящего в его состав одночлена. Имеем:

$$\int r \cdot x^n dx = r \cdot \int x^n dx = r \cdot \frac{x^{n+1}}{n+1}, \quad n \neq -1 \tag{2.2}$$

2.2 Произведение многочленов

Традиционный алгоритм произведение полиномов имеет $O(n^2)$ асимптотическую трудоёмкость. Скорость такого умножения можно повысить двумя способами:

- отбросить одночлены с нулевыми коэффициентами;
- производить параллельные вычисления.

Описание первого улучшения было приведено выше. Для достижения второго необходимо разбить множество одночленов, из которых состоит, например, первый полином, на m непересекающихся подмножеств, где m — количество потоков.

Пусть каждый j-ый поток вычисляет произведение второго полинома на k-ый одночлен первого многочлена, где $j = \overline{0, m-1}$, $k = i + z \cdot m$, $z \in \{0\} \cup \mathbb{N}$.

■

Для более быстрого вычисления произведения двух многочленов можно применить метод умножения, основанный на дискретном преобразовании Фурье. Данный метод обычно применяется к целым числам, так как имеет высокую погрешность для чисел с плавающей запятой. Чтобы сделать вывод о допустимости применения такого решения поставленной задачи, реализуем его.

Итак, $DFT(\overrightarrow{X})$ — операция, производящая дискретное преобразование Фурье над вектором чисел $\overrightarrow{X}$. $InverseDFT(\overrightarrow{X}_{complex})$ — обратная операция. Так же $A(\overrightarrow{X})$, $B(\overrightarrow{X})$ — некоторые полиномы. Тогда:

$$(A \cdot B)(\overrightarrow{X}) = A(\overrightarrow{X}) \cdot B(\overrightarrow{X})$$
$$DFT((A \cdot B)(\overrightarrow{X})) = DFT(A(\overrightarrow{X})) \cdot DFT(B(\overrightarrow{X}))$$
$$(A \cdot B)(\overrightarrow{X}) = InverseDFT(DFT(A(\overrightarrow{X})) \cdot DFT(B(\overrightarrow{X})))$$

Важно уточнить, что векторы $A(\overrightarrow{X})$, $B(\overrightarrow{X})$ нужно дополнить нулями так, чтобы $len((A \cdot B)(\overrightarrow{X})) = len(A(\overrightarrow{X})) = len(B(\overrightarrow{X}))$, где $len(\vec{v})$ —

операция получения длины некоторого вектора $\vec{v}$.

2.3 Инструменты реализации

Для реализации рассматриваемой задачи используется язык программирования C++.

В целях проведения параллельных вычислений используется библиотека pthread.

Задача выполнения метода Пикара при высоких приближениях требует тип данных, реализующий числа с плавающей точкой с множественной точностью. Для этого применяется библиотека mpfr.

Для вычисления DFT используется быстрое преобразование Фурье. Данный метод реализован в библиотеке Eigen.

2.4 Реализация

В листингах 2.1, 2.2, 2.3 приведён текст различных реализаций методов Пикара.

Листинг 2.1 – Метод Пикара

```
1  class Picard : public APicard
2  {
3  public:
4      Picard();
5      ~Picard();
6
7      void computePol(int approx) override;
8      double operator()(double x) override;
9      double operator()(double x, int approx) override;
10
11     static const int precision = 200;
12
13     using Real = mpfr::real<precision>;
14
15 protected:
16     void allocatePols(int maxApprox);
17
18     static inline Real pow(const Real &r, int p)
19     { return mpfr::pow(r, p); }
```

```
20
21      int approx;
22      long long *polLens;
23      Real **polynomials;
24  };
25
26  // Вычисление полинома
27  void Picard::computePol(int approx)
28  {
29      Real *squared;
30      Real *polynomial;
31      long long curLen = 1;
32      long long sqrLen;
33
34      allocatePols(approx);
35
36      squared = new Real[polLens[approx - 1]];
37      polynomial = new Real[polLens[approx - 1]];
38      polynomial[0] = 1.0 / 3;
39
40      for (int idx = 0; idx < approx; ++idx)
41      {
42          for (long long i = 0; i < curLen; ++i)
43              polynomials[idx][i] = polynomial[i];
44
45          sqrLen = curLen << 1;
46          for (long long i = 0; i < sqrLen; i++)
47              squared[i] = 0;
48
49          for (long long i = 0; i < curLen; i++)
50              for (long long j = 0; j < curLen; j++)
51                  squared[i + j + 1] += polynomial[i] *
52                                        polynomial[j] /
53                                        ((int)(i + j + 1) * 4 + 3);
54          squared[0] = 1.0 / 3;
55
56          std::swap(polynomial, squared);
57          curLen = sqrLen;
58      }
59
60      delete[] squared;
61      delete[] polynomial;
62  }
63
64  // Подстановка значения в полином
65  double Picard::operator()(double x, int approx)
66  {
67      if (!polynomials)
```

```
68          return 0;
69      if (approx > this->approx)
70          return 0;
71
72      Real res = 0;
73
74      for (int i = 0; i < polLens[approx - 1]; i++)
75          res += polynomials[approx - 1][i] * pow(Real(x), i * 4 + 3);
76
77      return res;
78  }
```

Листинг 2.2 – Параллелизированный метод Пикара

```
1   class MTPicard : public Picard
2   {
3   public:
4       void computePol(int approx) override;
5   };
6
7   void MTPicard::computePol(int approx)
8   {
9       Real *squared;
10      Real *polynomial;
11      long long curLen = 1;
12      long long sqrLen;
13
14      allocatePols(approx);
15      squared = new Real[polLens[approx - 1]];
16      polynomial = new Real[polLens[approx - 1]];
17      polynomial[0] = 1.0 / 3;
18
19      std::vector<std::thread> threads;
20
21      for (int idx = 0; idx < approx; ++idx)
22      {
23          for (long long i = 0; i < curLen; ++i)
24              polynomials[idx][i] = polynomial[i];
25
26          sqrLen = curLen << 1;
27          for (long long i = 0; i < sqrLen; i++)
28              squared[i] = 0;
29
30          for (int i = 1; i <= THREADS_QTY; i++)
31              threads.push_back(std::thread(
32                          &::squarePolPart,
33                          squared, i, sqrLen, THREADS_QTY,
34                          polynomial));
```

```
35
36          for (std::thread &thread: threads)
37              thread.join();
38
39          squared[0] = 1.0 / 3;
40
41          std::swap(polynomial, squared);
42          curLen = sqrLen;
43          threads.clear();
44      }
45
46      delete[] squared;
47      delete[] polynomial;
48  }
49
50  static void squarePolPart(
51          mmlabs::Picard::Real *squared,
52          long long begin, long long end, long long step,
53          mmlabs::Picard::Real *polynomial
54          )
55  {
56      for (long long i, idx = begin; idx < end; idx += step)
57      {
58          i = idx - 1;
59
60          do
61              squared[idx] += polynomial[i] *
62                              polynomial[idx - 1 - i] /
63                              ((int)idx * 4 + 3);
64          while (i--);
65      }
66  }
```

Листинг 2.3 – Метод Пикара, основанный на быстром преобразовании Фурье

```
1  template<typename TComplex>
2  class FFT;
3
4  class FFTPicard : public Picard
5  {
6  public:
7      FFTPicard();
8      ~FFTPicard();
9
10     void computePol(int approx) override;
11
12     using Complex = std::complex<Real>;
```

```
13
14 private:
15     FFT<Complex> *fft;
16 };
17
18 void FFTPicard::computePol(int approx)
19 {
20     int sqrLen = 1;
21     Complex *polynomialComplex;
22     Complex *integratedComplex;
23
24     allocatePols(approx);
25     polynomialComplex = new Complex[polLens[approx - 1]];
26     integratedComplex = new Complex[polLens[approx - 1]];
27     polynomialComplex[0] = 1.0 / 3;
28
29     fft->setMaxVectorSize(polLens[approx - 1]);
30
31     for (int idx = 0; idx < approx; ++idx)
32     {
33         for (long long i = 0; i < polLens[idx]; ++i)
34             polynomials[idx][i] = polynomialComplex[i].real();
35
36         sqrLen <<= 1;
37
38         (*fft)(polynomialComplex, sqrLen, false);
39         for (long long i = 0; i < sqrLen; ++i)
40             polynomialComplex[i] *= polynomialComplex[i];
41         (*fft)(polynomialComplex, sqrLen, true);
42
43         for (long long i = 0; i < sqrLen - 1; ++i)
44             integratedComplex[i + 1] = polynomialComplex[i] /
45                                        Complex((int)(i + 1) * 4 + 3);
46
47         integratedComplex[0] = 1.0 / 3;
48         std::swap(integratedComplex, polynomialComplex);
49     }
50
51     delete[] polynomialComplex;
52     delete[] integratedComplex;
53 }
```

В листинге 2.4 приведён текст реализации явного метода Эйлера.

Листинг 2.4 – Явный метод Эйлера

```
1 double explicitMethod(double x, double h)
2 {
3     double res = 0;
```

```
4       double x0 = h;
5
6       while (x0 < x + h)
7       {
8           res += h * (x0 * x0 + res * res);
9           x0 += h;
10      }
11
12      return res;
13  }
```

В листинге 2.5 приведён текст реализации неявного метода Эйлера.

Листинг 2.5 – Неявный метод Эйлера

```
1   double implicitMethod(double x, double h)
2   {
3       double D;
4       double res = 0;
5       double x0 = h;
6
7       while (x0 < x + h)
8       {
9           D = 1 - 4 * h * (h * x0 * x0 + res);
10
11          if (D >= 0)
12              res = (1 - ::sqrt(D)) / 2 / h;
13
14          x0 += h;
15      }
16
17      return res;
18  }
```

2.5 Примеры работы

На рисунках 2.1, 2.2 изображены примеры реализованной программы. В первом случае используется параллеллизированный метод Пикара, во втором — метод Пикара, основанный на быстром преобразовании Фурье.

Рис. 2.1 – Параллелизированный метод Пикара

```
 # time ./mmlab01 -mt
Введите приближение метода Пикара: 17
Введите шаг метода Эйлера: 0.01
Введите шаг таблицы: 0.1

    x | 1-ое приближение | 2-ое приближение | 3-ое приближение | 4-ое приближение | 17-ое приближение | Явный метод | Неявный метод
------+------------------+------------------+------------------+------------------+-------------------+-------------+--------------
 0.00 |           0.0000 |           0.0000 |           0.0000 |           0.0000 |            0.0000 |      0.0000 |       0.0000
 0.10 |           0.0003 |           0.0003 |           0.0003 |           0.0003 |            0.0003 |      0.0004 |       0.0005
 0.20 |           0.0027 |           0.0027 |           0.0027 |           0.0027 |            0.0027 |      0.0029 |       0.0029
 0.30 |           0.0090 |           0.0090 |           0.0090 |           0.0090 |            0.0090 |      0.0095 |       0.0095
 0.40 |           0.0213 |           0.0214 |           0.0214 |           0.0214 |            0.0214 |      0.0222 |       0.0222
 0.50 |           0.0417 |           0.0418 |           0.0418 |           0.0418 |            0.0418 |      0.0430 |       0.0431
 0.60 |           0.0720 |           0.0724 |           0.0724 |           0.0724 |            0.0724 |      0.0743 |       0.0743
 0.70 |           0.1143 |           0.1156 |           0.1157 |           0.1157 |            0.1157 |      0.1181 |       0.1183
 0.80 |           0.1707 |           0.1740 |           0.1741 |           0.1741 |            0.1741 |      0.1773 |       0.1776
 0.90 |           0.2430 |           0.2506 |           0.2509 |           0.2509 |            0.2509 |      0.2550 |       0.2556
 1.00 |           0.3333 |           0.3492 |           0.3502 |           0.3502 |            0.3502 |      0.3552 |       0.3566
 1.10 |           0.4437 |           0.4746 |           0.4774 |           0.4776 |            0.4776 |      0.4836 |       0.4862
 1.20 |           0.5760 |           0.6329 |           0.6403 |           0.6410 |            0.6411 |      0.6480 |       0.6529
 1.30 |           0.7323 |           0.8319 |           0.8500 |           0.8526 |            0.8529 |      0.8605 |       0.8697
 1.40 |           0.9147 |           1.0820 |           1.1236 |           1.1317 |            1.1331 |      1.1409 |       1.1584
 2.00 |           2.6667 |           4.6984 |           7.2190 |          10.4839 |          232.5727 |     36.4939 |      25.1333
./mmlab01 -mt  3601.75s user 0.44s system 772% cpu 7:46.55 total
```

Рис. 2.2 – Метод Пикара, основанный на быстром преобразовании Фурье

```
 # time ./mmlab01 -fft
Введите приближение метода Пикара: 17
Введите шаг метода Эйлера: 0.01
Введите шаг таблицы: 0.1

    x | 1-ое приближение | 2-ое приближение | 3-ое приближение | 4-ое приближение | 17-ое приближение | Явный метод | Неявный метод
------+------------------+------------------+------------------+------------------+-------------------+-------------+--------------
 0.00 |           0.0000 |           0.0000 |           0.0000 |           0.0000 |            0.0000 |      0.0000 |       0.0000
 0.10 |           0.0003 |           0.0003 |           0.0003 |           0.0003 |            0.0003 |      0.0004 |       0.0005
 0.20 |           0.0027 |           0.0027 |           0.0027 |           0.0027 |            0.0027 |      0.0029 |       0.0029
 0.30 |           0.0090 |           0.0090 |           0.0090 |           0.0090 |            0.0090 |      0.0095 |       0.0095
 0.40 |           0.0213 |           0.0214 |           0.0214 |           0.0214 |            0.0214 |      0.0222 |       0.0222
 0.50 |           0.0417 |           0.0418 |           0.0418 |           0.0418 |            0.0418 |      0.0430 |       0.0431
 0.60 |           0.0720 |           0.0724 |           0.0724 |           0.0724 |            0.0724 |      0.0743 |       0.0743
 0.70 |           0.1143 |           0.1156 |           0.1157 |           0.1157 |            0.1157 |      0.1181 |       0.1183
 0.80 |           0.1707 |           0.1740 |           0.1741 |           0.1741 |            0.1741 |      0.1773 |       0.1776
 0.90 |           0.2430 |           0.2506 |           0.2509 |           0.2509 |            0.2509 |      0.2550 |       0.2556
 1.00 |           0.3333 |           0.3492 |           0.3502 |           0.3502 |            0.3502 |      0.3552 |       0.3566
 1.10 |           0.4437 |           0.4746 |           0.4774 |           0.4776 |              -inf |      0.4836 |       0.4862
 1.20 |           0.5760 |           0.6329 |           0.6403 |           0.6410 |              -inf |      0.6480 |       0.6529
 1.30 |           0.7323 |           0.8319 |           0.8500 |           0.8526 |              -inf |      0.8605 |       0.8697
 1.40 |           0.9147 |           1.0820 |           1.1236 |           1.1317 |              -inf |      1.1409 |       1.1584
 2.00 |           2.6667 |           4.6984 |           7.2190 |          10.4839 |              -inf |     36.4939 |      25.1333
./mmlab01 -fft  7.71s user 0.04s system 26% cpu 29.565 total
```

Как видно, высокая погрешность быстрого преобразования Фурье не позволяет использовать этот метод для реализации метода Пикара.

2.6 Ответы на контрольные вопросы

2.6.1 Укажите интервалы значений аргумента, в которых можно считать решением заданного уравнения каждое из первых 4-х приближений Пикара. Точность результата оценивать до второй цифры после запятой. Объяснить свой ответ.

Чтобы проверить, сходится ли n-ое приближение Пикара к точному, нужно вычислить следующее. Если y_n и y_{n+1} равны с точностью до некоторого знака (в данном случае до второго), то n-ое приближение можно считать решением заданного уравнения.

Необходимое условие можно задать таким образом:

$$y_{n+1}(x) - y_n(x) < 0.01$$

Для первых четырёх приближений имеем:

$$y_2(x) - y_1(x) < 0.01 \tag{2.3}$$

$$y_3(x) - y_2(x) < 0.01 \tag{2.4}$$

$$y_4(x) - y_3(x) < 0.01 \tag{2.5}$$

$$y_5(x) - y_4(x) < 0.01 \tag{2.6}$$

Для решения этих неравенств была реализована программа, текст который представлен в листинге 2.6.

Листинг 2.6 – Поиск интервалов

```
1  int main()
2  {
3      std::vector<double> v;
4      mmlabs::MTPicard picard;
5
6      int maxApprox = 4;
7
8      int p = 2;
9      int n = 3;
```

```
10      double eps = 1 * pow(10.0, -p);
11      double step = 1 * pow(10.0, -n);
12
13      picard.computePol(maxApprox + 1);
14
15      for (int i = 1; i <= maxApprox; ++i)
16      {
17          v.push_back(estimateMaxX(picard, i, i + 1, step, eps));
18          printf("x_%d <= %.*lf\n", i, n, *v.rbegin());
19      }
20      printf("\n");
21
22      printf("x <= %.*lf\n", n, *std::min_element(v.begin(), v.end()));
23
24      return 0;
25  }
26
27  double estimateMaxX(mmlabs::MTPicard &picard,
28                      int approx1, int approx2,
29                      double step, double eps)
30  {
31      double x = 0;
32      double y1, y2;
33
34      while (fabs((picard(x, approx2)) - (y1 = picard(x, approx1))) < eps)
35          x += step;
36
37      return x - step;
38  }
```

На рисунке 2.3 приведён результат работы этой программы.

Рис. 2.3 – Поиск $u(2)$

```
x_1 <= 0.936
x_2 <= 1.232
x_3 <= 1.418
x_4 <= 1.546

x <= 0.936
```

Таким образом, при $\underline{\overline{x \in [0, 0.936]}}$ решением заданного уравнения можно считать каждое из первых 4-х приближений Пикара.

2.6.2 Пояснить, каким образом можно доказать правильность полученного результата при фиксированном значении аргумента в численных методах.

Численные методы зависят от шага. Чем меньше шаг — тем точнее решение. Чтобы доказать правильность полученного результата с точностью до n-ой цифры после запятой, нужно уменьшать шаг до тех пор, пока результаты для двух крайних шагов не перестанут отличаться с точностью до n-ой цифры после запятой.

2.6.3 Из каких соображений выбирался корень уравнения в неявном методе?

В неявном методе выбирается минимальный корень для того, чтобы минимизировать накапливаемую ошибку и, тем самым, повысить точность.

2.6.4 Каково значение функции при $x = 2$, т.е. привести значение $u(2)$.

Для поиска $u(2)$ используем явный метод Эйлера. С учётом ответа на вопрос 2.6.2 была реализована программа, текст которой приведён в листинге 2.7.

Листинг 2.7 – Поиск $u(2)$

```
1  int main()
2  {
3      const double EPS = 1e-2;
4
5      double y;
6      double y_next;
7
8      double x = 2;
9      double h = 1e-1;
10
11     int h_len = 7;
12     int h_frac_len = 1;
13     int y_len = 8;
14     int y_frac_len = 4;
```

```
15
16      char hrule[256] = {};
17      memset(hrule, '-', sizeof(hrule) - 1);
18
19      printf("Явный метод Эйлера, x = %.1lf\n", x);
20
21      printf(" %*s | %*s \n", h_len, "h", y_len, "y");
22      printf("-%.*s-+-%.*s-\n", h_len, hrule, y_len, hrule);
23
24      y_next = mmlabs::EulerMethod::explicitMethod(x, h);
25
26      do
27      {
28          y = y_next;
29
30          printf(" %*.*e | %*.*lf \n",
31                 h_len, h_frac_len, h, y_len, y_frac_len, y);
32
33          h /= 10;
34          y_next = mmlabs::EulerMethod::explicitMethod(x, h);
35      }
36      while (fabs(y_next - y) >= EPS);
37
38      printf(" %*.*e | %*.*lf \n",
39             h_len, h_frac_len, h, y_len, y_frac_len, y_next);
40
41      return 0;
42  }
```

Результат работы этой программы приведён на рисунке 2.4.

Рис. 2.4 – Поиск интервалов

```
Явный метод Эйлера, x = 2.0
         h |         y
-----------+-----------
 1.0e-01 |   9.6768
 1.0e-02 |  36.4939
 1.0e-03 | 162.9738
 1.0e-04 | 285.0559
 1.0e-05 | 313.0351
 1.0e-06 | 317.3466
 1.0e-07 | 317.6747
 1.0e-08 | 317.7188
 1.0e-09 | 317.7154
```

В полученной таблице последний и предпоследний результаты равны с точностью до второго знака после запятой. Таким образом, $\underline{\underline{u(2) \approx 317.72}}$.