Московский авиационный институт

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа № 4
по дисциплине
"Численные методы"

Студент: Балес А.И.
Преподаватель: Ревизников Д.Л.
Дата:
Оценка:
Подпись:

Москва, 2016

Метод 1 — Метод Эйлера/Метод Рунге-Кутты/Метод Адамса

Задание
Реализовать методы Эйлера, Рунге-Кутты и Адамса 4–го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2–го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге-Ромберга и путем сравнения с точным решением.

Вариант: 3
Задача Коши:
$y'' - 2y - 4x^2 \exp^{x^2} = 0$
$y(0) = 3$
$y'(0) = 0$
$x \in [0, 1], h = 0.1$

Точное решение:
$y = \exp^{x^2} + \exp^{x\sqrt{2}} + \exp^{-x\sqrt{2}}$

Описание алгоритма
Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

$y' = f(x, y)$
$y(x_0) = y_0$

Требуется найти решение на отрезке $[a, b]$, где $x_0 = a$.
Введем разностную сетку на отрезке $[a, b]$ $\Omega^{(k)} = x_k = x_0 + hk, k = 0, 1, ..., N, h = |b - a|/N$.

Точки x_k — называются *узлами* разностной сетки, расстояния между узлами — *шагом* разностной сетки (h), а совокупность значений какой–либо величины заданных в узлах сетки называется *сеточной функцией* $y^{(h)} = y_k, k = 0, 1, ..., N$.
Приближенное решение задачи Коши будем искать численно в виде сеточной функции $y^{(h)}$. Для оценки погрешности приближенного численного решения $y^{(h)}$ будем рассматривать это решение как элемент $N + 1$–мерного линейного векторного пространства с какой либо нормой. В качестве погрешности решения принимается норма элемента этого пространства $\delta^{(h)} = y^{(h)} - [y]^{(h)}$, где $[y]^{(h)}$ — точное решение задачи в узлах

расчетной сетки. Таким образом $\varepsilon_h = \|\delta^{(h)}\|$.

Метод Эйлера

$y_{k+1} = hf(x_k, y_k, z_k)$
$z_{k+1} = hg(x_k, y_k, z_k)$

Метод Рунге-Кутты

Семейство явных методов Рунге-Кутты p–го порядка записывается в виде совокупности формул:

$$y_{k+1} = y_k + \Delta y_k$$
$$\Delta y_k = \sum_{i=1}^{p} c_i K_i^k \qquad (1)$$
$$K_i^k = hf(x_k + a_i h, y_k + h \sum_{j=1}^{i-1} b_{ij} K_j^k)$$
$$i = 2, 3, ..., p$$

Параметры a_i, b_{ij}, c_i подбираются так, чтобы значение y_{k+1}, рассчитанное по соотношению (1) совпадало со значением разложения в точке x_{k+1} точного решения в ряд Тейлора с погрешностью $O(h^{p+1})$.

Метод Рунге-Кутты 4–го порядка точности

$(p = 4, a_1 = 0, a_2 = \frac{1}{2}, a_3 = \frac{1}{2}, a_4 = 1, b_{21} = \frac{1}{2}, b_{31} = 0, b_{32} = \frac{1}{2}, b_{41} = 0, b_{42} = 0, b_{43} = \frac{1}{2}, c_1 = \frac{1}{6}, c_2 = \frac{1}{3}, c_3 = \frac{1}{3}, c_4 = \frac{1}{6})$

$y_{k+1} = y_k + \Delta y_k$

$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k)$

$K_1^k = hf(x_k, y_k)$

$K_2^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k)$

$K_3^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k)$

$K_4^k = hf(x_k + h, y_k + K_3^k)$

Метод Адамса

При использовании интерполяционного многочлена 3–ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

$$y_{k+1} = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3})$$

где f_k значение подынтегральной функции в узле x_k.
Метод Адамса как и все многошаговые методы не является самостартующим, т.е. для того, чтобы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле x_0 решение y_0 известно из начальных условий, а в других трех узлах x_1, x_2, x_3 решения y_1, y_2, y_3 можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

Реализация

```
void TMethodRungeKutta::ToSolve() {
  string separator =
     "************************************************************
  double x0 = this->_a, y0 = this->_funcY0, z0 =
     this->_funcZ0;
  size_t N = (this->_b - this->_a) / this->_h;
  double K1, K2, K3, K4, L1, L2, L3, L4;
  log << "h = " << this->_h << "; N = " << N <<
     endl;
  double y_k = y0;
  double x_k = x0;
  double z_k = z0;
  double deltaZ, deltaY;
  out.precision(5);
  log.precision(5);
  out << "\tx\ty" << endl;
  log <<
     "\tx\ty\tz\t\td(y)\t\td(z)\t\ty(true)\t\teps\t\terr"
     << endl;
  for (size_t k = 0; k < N; ++k) {
    L1 = this->_h * this->_FuncExpression(x_k, y_k);
    K1 = this->_h * z_k;
```

```
    //log << fixed << k << "/" << 1 << "\t" << x_k
       << "\t" << y_k << "\t" << K1 << "\t\t\t\t"
    //  << this->_FuncY(x_k) << "\t" <<
       fabs(this->_FuncY(x_k) - y_k) << endl;
    K2 = this->_h * (z_k + 0.5 * L1);
    L2 = this->_h * this->_FuncExpression(x_k + 0.5
       * this->_h, y_k + 0.5 * K1);
    //log << fixed << k << "/" << 2 << "\t" << x_k
       + 0.5 * this->_h << "\t" << y_k + 0.5 * K1
       << "\t" << K2 << endl;
    K3 = this->_h * (z_k + 0.5 * L2);
    L3 = this->_h * this->_FuncExpression(x_k + 0.5
       * this->_h, y_k + 0.5 * K2);
    //log << fixed << k << "/" << 3 << "\t" << x_k
       + 0.5 * this->_h << "\t" << y_k + 0.5 * K2
       << "\t" << K3 << endl;
    K4 = this->_h * (z_k + L3);
    L4 = this->_h * this->_FuncExpression(x_k +
       this->_h, y_k + K3);
    deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
       + K4);
    deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
       + L4);
    //log << fixed << k << "/" << 4 << "\t" << x_k
       + this->_h << "\t" << y_k + K3 << "\t" << K4
    //  << "\t" << delta << "\t" << fabs((K2 - K3)
       / (K1 - K2)) << endl;
    out << fixed << k << "\t" << x_k << "\t" << y_k
       << endl;
    log << fixed << k << "\t" << x_k << "\t" << y_k
       << "\t" << z_k
      << "\t\t" << deltaY << "\t\t" << deltaZ <<
         "\t\t" << this->_FuncY(x_k)
      << "\t\t" << fabs(this->_FuncY(x_k) - y_k) <<
         "\t\t" << fabs((K2 - K3) / (K1 - K2)) <<
         endl;
    x_k += this->_h;
    z_k += deltaZ;
    y_k += deltaY;
    log << separator << endl;
}
```

```
  out << fixed << N << "\t" << x_k << "\t" << y_k
     << endl;
  log << fixed << N << "\t" << x_k << "\t" << y_k
     << "\t"
     << z_k << "\t\t\t\t\t" << this->_FuncY(x_k) <<
        "\t\t"
     << fabs(this->_FuncY(x_k) - y_k) << "\t\t"
     << fabs((K2 - K3) / (K1 - K2)) << endl;
}

void TMethodEuler::ToSolve() {
  double x0 = this->_a, y0 = this->_funcY0, z0 =
     this->_funcZ0;
  size_t N = (this->_b - this->_a) / this->_h;
  log << "h = " << this->_h << "; N = " << N <<
     endl;
  double y_k = y0;
  double x_k = x0;
  double z_k = z0;
  double deltaZ = this->_h *
     this->_FuncExpression(x_k, y_k);
  double deltaY = this->_h * z_k;
  out.precision(5);
  log.precision(5);
  out << "\tx\ty\tz" << endl;
  out << 0 << fixed << "\t" << x0 << "\t" << y0 <<
     "\t" << z0 << endl;
  log <<
     "\tx\ty\tz\t\td(z)\t\td(y)\t\ty(true)\t\teps"
     << endl;
  log << 0 << fixed << "\t" << x0 << "\t" << y0 <<
     "\t" << z0 << "\t\t" << deltaZ
     << "\t\t" << deltaY << "\t\t" <<
        this->_FuncY(x_k) << "\t\t" <<
        fabs(this->_FuncY(x_k) - y_k) << endl;
  for (size_t k = 1; k < N; ++k) {
    deltaY = this->_h * z_k;
    z_k = z_k + deltaZ;
    y_k = y_k + deltaY;
    x_k += this->_h;
    deltaZ = this->_h * this->_FuncExpression(x_k,
```

```
        y_k);
      out << fixed << k << "\t" << x_k << "\t" << y_k
         << "\t" << z_k << endl;
      log << k << fixed << "\t" << x_k << "\t" << y_k
        << "\t" << z_k << "\t\t" << deltaZ << "\t\t"
           << deltaY << "\t\t" << this->_FuncY(x_k)
        << "\t\t" << fabs(this->_FuncY(x_k) - y_k) <<
           endl;
  }
  deltaY = this->_h * z_k;
  z_k = z_k + deltaZ;
  y_k = y_k + deltaY;
  x_k += this->_h;
  deltaZ = this->_h * this->_FuncExpression(x_k,
     y_k);
  out << fixed << N << "\t" << x_k << "\t" << y_k
     << "\t" << z_k << endl;
  log << N << fixed << "\t" << x_k << "\t" << y_k
      << "\t" << z_k << "\t\t\t\t\t\t" <<
         this->_FuncY(x_k)
      << "\t\t" << fabs(this->_FuncY(x_k) - y_k) <<
         endl;
}

void TMethodEuler::RungeRomberg() {
  double x0 = this->_a, y0 = this->_funcY0, z0 =
     this->_funcZ0;
  double h1 = this->_h;
  double h2 = h1 / 2;
  size_t N1 = (this->_b - this->_a) / h1;
  size_t N2 = (this->_b - this->_a) / h2;
  log << "h1 = " << h1 << "; N1 = " << N1 << endl;
  log << "h2 = " << h2 << "; N2 = " << N2 << endl;
  double y_k = y0;
  double x_k = x0;
  double z_k = z0;
  double deltaZ = h1 * this->_FuncExpression(x_k,
     y_k);
  double deltaY = h1 * z_k;
  out.precision(5);
  log.precision(5);
```

```
vector<double> X, Y1, Y2;
for (size_t k = 1; k < N1; ++k) {
  X.push_back(x_k);
  Y1.push_back(y_k);
  deltaY = h1 * z_k;
  z_k = z_k + deltaZ;
  y_k = y_k + deltaY;
  x_k += h1;
  deltaZ = h1 * this->_FuncExpression(x_k, y_k);
}
deltaY = h1 * z_k;
z_k = z_k + deltaZ;
y_k = y_k + deltaY;
x_k += h1;
X.push_back(x_k);
Y1.push_back(y_k);
deltaZ = h1 * this->_FuncExpression(x_k, y_k);
y_k = y0;
x_k = x0;
z_k = z0;
deltaZ = h2 * this->_FuncExpression(x_k, y_k);
deltaY = h2 * z_k;
for (size_t k = 1; k < N2; ++k) {
  if (!(k & 1)) {
    Y2.push_back(y_k);
  }
  deltaY = h2 * z_k;
  z_k = z_k + deltaZ;
  y_k = y_k + deltaY;
  x_k += h2;
  deltaZ = h2 * this->_FuncExpression(x_k, y_k);
}
deltaY = h2 * z_k;
z_k = z_k + deltaZ;
y_k = y_k + deltaY;
x_k += h2;
Y2.push_back(y_k);
deltaZ = h1 * this->_FuncExpression(x_k, y_k);
log << "Runge-Romberg" << endl;
log << "\tx\t\ty\t\ty*\t\terr" << endl;
y_k = y0;
```

```
  for (size_t i = 0; i < min(Y1.size(), Y2.size());
     ++i) {
    y_k = Y1[i] + (Y1[i] - Y2[i]);
    log << i << "\t" << X[i] << "\t\t" << Y1[i] <<
       "\t\t"
      << y_k << "\t\t" << fabs(Y1[i] - Y2[i]) <<
         endl;
  }
}

void TMethodAdams::ToSolve() {
  double x0 = this->_a, y0 = this->_funcY0, z0 =
     this->_funcZ0;
  size_t N = (this->_b - this->_a) / this->_h;
  double K1, K2, K3, K4, L1, L2, L3, L4;
  log << "h = " << this->_h << "; N = " << N <<
     endl;
  double y_k = y0;
  double x_k = x0;
  double z_k = z0;
  double deltaZ, deltaY;
  out.precision(5);
  log.precision(5);
  log.width(10);
  vector<double> X, Y, Z;
  out << "\tx\ty" << endl;
  log <<
     "\tx\t\ty\t\tz\t\td(y)\t\td(z)\t\ty(true)\t\teps\t\terr"
     << endl;
  size_t sz = min((size_t)4, N);
  for (size_t k = 0; k < sz; ++k) {
    L1 = this->_h * this->_FuncExpression(x_k, y_k);
    K1 = this->_h * z_k;
    //log << fixed << k << "/" << 1 << "\t" << x_k
       << "\t" << y_k << "\t" << K1 << "\t\t\t\t"
    //  << this->_FuncY(x_k) << "\t" <<
       fabs(this->_FuncY(x_k) - y_k) << endl;
    K2 = this->_h * (z_k + 0.5 * L1);
    L2 = this->_h * this->_FuncExpression(x_k + 0.5
       * this->_h, y_k + 0.5 * K1);
    //log << fixed << k << "/" << 2 << "\t" << x_k
```

```
      + 0.5 * this->_h << "\t" << y_k + 0.5 * K1
      << "\t" << K2 << endl;
    K3 = this->_h * (z_k + 0.5 * L2);
    L3 = this->_h * this->_FuncExpression(x_k + 0.5
      * this->_h, y_k + 0.5 * K2);
    //log << fixed << k << "/" << 3 << "\t" << x_k
      + 0.5 * this->_h << "\t" << y_k + 0.5 * K2
      << "\t" << K3 << endl;
    K4 = this->_h * (z_k + L3);
    L4 = this->_h * this->_FuncExpression(x_k +
      this->_h, y_k + K3);
    deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
      + K4);
    deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
      + L4);
    //log << fixed << k << "/" << 4 << "\t" << x_k
      + this->_h << "\t" << y_k + K3 << "\t" << K4
    //  << "\t" << delta << "\t" << fabs((K2 - K3)
      / (K1 - K2)) << endl;
    out << fixed << k << "\t" << x_k << "\t" << y_k
      << endl;
    log << fixed << k << "\t" << x_k << "\t\t" <<
      y_k << "\t\t" << z_k
      << "\t\t" << deltaY << "\t\t" << deltaZ <<
        "\t\t" << this->_FuncY(x_k)
      << "\t\t" << fabs(this->_FuncY(x_k) - y_k) <<
        "\t\t" << fabs((K2 - K3) / (K1 - K2)) <<
        endl;
    Y.push_back(y_k);
    X.push_back(x_k);
    Z.push_back(z_k);
    x_k += this->_h;
    z_k += deltaZ;
    y_k += deltaY;
  }
  x_k -= this->_h;
  y_k -= deltaY;
  z_k -= deltaZ;
  for (size_t k = sz - 1; k < N; ++k) {
    z_k = z_k + this->_h / 24.0 * (55.0 *
      this->_FuncExpression(X[k], Y[k])
```

```
            - 59.0 * this->_FuncExpression(X[k - 1],
               Y[k - 1]) + 37.0 *
               this->_FuncExpression(X[k - 2], Y[k - 2])
            - 9.0 * this->_FuncExpression(X[k - 3], Y[k
               - 3]));
      y_k = y_k + this->_h * z_k;
      x_k += this->_h;
      log << fixed << k + 1 << "\t" << x_k << "\t\t"
         << y_k << "\t\t" << z_k
        << "\t\t" << deltaY << "\t\t" << deltaZ <<
           "\t\t" << this->_FuncY(x_k)
        << "\t\t" << fabs(this->_FuncY(x_k) - y_k) <<
           endl;
      out << fixed << "\t" << x_k << "\t" << y_k <<
         endl;
      Y.push_back(y_k);
      X.push_back(x_k);
      Z.push_back(z_k);
    }
}

void TMethodAdams::RungeRomberg() {
    double x0 = this->_a, y0 = this->_funcY0, z0 =
       this->_funcZ0;
    double h1 = this->_h;
    double h2 = h1 / 2;
    size_t N1 = (this->_b - this->_a) / h1;
    size_t N2 = (this->_b - this->_a) / h2;
    double K1, K2, K3, K4, L1, L2, L3, L4;
    log << "h1 = " << h1 << "; N1 = " << N1 << endl;
    log << "h2 = " << h2 << "; N2 = " << N2 << endl;
    double y_k = y0;
    double x_k = x0;
    double z_k = z0;
    double deltaZ, deltaY;
    out.precision(5);
    log.precision(5);
    log.width(10);
    vector<double> X, Y, Z;
    size_t sz = min((size_t)4, N1);
    for (size_t k = 0; k < sz; ++k) {
```

```
  L1 = h1 * this->_FuncExpression(x_k, y_k);
  K1 = h1 * z_k;
  K2 = h1 * (z_k + 0.5 * L1);
  L2 = h1 * this->_FuncExpression(x_k + 0.5 * h1,
     y_k + 0.5 * K1);
  K3 = h1 * (z_k + 0.5 * L2);
  L3 = h1 * this->_FuncExpression(x_k + 0.5 * h1,
     y_k + 0.5 * K2);
  K4 = h1 * (z_k + L3);
  L4 = h1 * this->_FuncExpression(x_k + h1, y_k +
     K3);
  deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
     + K4);
  deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
     + L4);
  Y.push_back(y_k);
  X.push_back(x_k);
  Z.push_back(z_k);
  x_k += h1;
  z_k += deltaZ;
  y_k += deltaY;
}
x_k -= h1;
y_k -= deltaY;
z_k -= deltaZ;
for (size_t k = sz - 1; k < N1; ++k) {
  z_k = z_k + h1 / 24.0 * (55.0 *
     this->_FuncExpression(X[k], Y[k])
       - 59.0 * this->_FuncExpression(X[k - 1],
          Y[k - 1]) + 37.0 *
          this->_FuncExpression(X[k - 2], Y[k - 2])
       - 9.0 * this->_FuncExpression(X[k - 3], Y[k
          - 3]));
  y_k = y_k + h1 * z_k;
  x_k += h1;
  Y.push_back(y_k);
  X.push_back(x_k);
  Z.push_back(z_k);
}

vector<double> X2, Y2;
```

```
y_k = y0;
x_k = x0;
z_k = z0;
sz = min((size_t)4, N2);
for (size_t k = 0; k < sz; ++k) {
  L1 = h2 * this->_FuncExpression(x_k, y_k);
  K1 = h2 * z_k;
  K2 = h2 * (z_k + 0.5 * L1);
  L2 = h2 * this->_FuncExpression(x_k + 0.5 * h2,
     y_k + 0.5 * K1);
  K3 = h2 * (z_k + 0.5 * L2);
  L3 = h2 * this->_FuncExpression(x_k + 0.5 * h2,
     y_k + 0.5 * K2);
  K4 = h2 * (z_k + L3);
  L4 = h2 * this->_FuncExpression(x_k + h2, y_k +
     K3);
  deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
     + K4);
  deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
     + L4);
  Y2.push_back(y_k);
  X2.push_back(x_k);
  x_k += h2;
  z_k += deltaZ;
  y_k += deltaY;
}
x_k -= h2;
y_k -= deltaY;
z_k -= deltaZ;
for (size_t k = sz - 1; k < N2; ++k) {
  z_k = z_k + h2 / 24.0 * (55.0 *
     this->_FuncExpression(X2[k], Y2[k])
       - 59.0 * this->_FuncExpression(X2[k - 1],
          Y2[k - 1]) + 37.0 *
          this->_FuncExpression(X2[k - 2], Y2[k -
          2])
       - 9.0 * this->_FuncExpression(X2[k - 3],
          Y2[k - 3]));
  y_k = y_k + h2 * z_k;
  x_k += h2;
  Y2.push_back(y_k);
```

```
    X2.push_back(x_k);
  }
  log << "Runge-Romberg" << endl;
  log << "\tx\t\ty\t\ty*\t\terr" << endl;
  for (size_t i = 0, j = 0; i < Y.size() && j <
     Y2.size(); ++i, j += 2) {
    y_k = Y[i] + (Y[i] - Y2[j]) / 15;
    log << i << "\t" << X[i] << "\t\t" << Y[i] <<
       "\t\t"
      << y_k << "\t\t" << fabs(Y[i] - Y2[j]) / 15
         << endl;

  }

}
```

Тестирование

Выходной файл

```
Метод 1: Метод Эйлера
x y z
0 0.00000 3.00000 0.00000
1 0.10000 3.00000 0.60000
2 0.20000 3.06000 1.20404
3 0.30000 3.18040 1.83269
4 0.40000 3.36367 2.50816
5 0.50000 3.61449 3.25600
6 0.60000 3.94009 4.10730
7 0.70000 4.35082 5.10172
8 0.80000 4.86099 6.29182
9 0.90000 5.49017 7.74952
10 1.00000 6.26513 9.57587

h = 0.1; N = 10
x y z d(z) d(y) y(true) eps
0 0.00000 3.00000 0.00000 0.60000 0.00000 3.00000 0.00000
1 0.10000 3.00000 0.60000 0.60404 0.00000 3.03008 0.03008
2 0.20000 3.06000 1.20404 0.62865 0.06000 3.12135 0.06135
3 0.30000 3.18040 1.83269 0.67547 0.12040 3.27689 0.09649
4 0.40000 3.36367 2.50816 0.74784 0.18327 3.50214 0.13846
```

```
5 0.50000 3.61449 3.25600 0.85130 0.25082 3.80521 0.19072
6 0.60000 3.94009 4.10730 0.99442 0.32560 4.19758 0.25749
7 0.70000 4.35082 5.10172 1.19010 0.41073 4.69501 0.34419
8 0.80000 4.86099 6.29182 1.45770 0.51017 5.31897 0.45798
9 0.90000 5.49017 7.74952 1.82636 0.62918 6.09877 0.60859
10 1.00000 6.26513 9.57587 7.07465 0.80952
h1 = 0.10000; N1 = 10
h2 = 0.05000; N2 = 20
Runge-Romberg
x y y* err
0 0.00000 3.00000 3.00000 0.00000
1 0.10000 3.00000 2.95497 0.04503
2 0.20000 3.06000 2.96912 0.09088
3 0.30000 3.18040 3.04046 0.13995
4 0.40000 3.36367 3.16888 0.19479
5 0.50000 3.61449 3.35608 0.25841
6 0.60000 3.94009 3.60564 0.33445
7 0.70000 4.35082 3.92324 0.42758
8 0.80000 4.86099 4.31702 0.54397
9 1.00000 6.26513 5.89139 0.37374

Метод 2: Метод Рунге-Кутты
x y
0 0.00000 3.00000
1 0.10000 3.03008
2 0.20000 3.12134
3 0.30000 3.27689
4 0.40000 3.50213
5 0.50000 3.80520
6 0.60000 4.19757
7 0.70000 4.69499
8 0.80000 5.31895
9 0.90000 6.09873
10 1.00000 7.07459

h = 0.1; N = 10
x y z d(y) d(z) y(true) eps err
0 0.00000 3.00000 0.00000 0.03008 0.60334 3.00000 0.00000 0.00167
********************************************************************************
1 0.10000 3.03008 0.60334 0.09126 0.62369 3.03008 0.00000 0.01836
********************************************************************************
```

```
2 0.20000 3.12134 1.22703 0.15554 0.66579 3.12135 0.00000 0.03468
*******************************************************************************
3 0.30000 3.27689 1.89283 0.22524 0.73269 3.27689 0.00000 0.05027
*******************************************************************************
4 0.40000 3.50213 2.62551 0.30307 0.82939 3.50214 0.00000 0.06490
*******************************************************************************
5 0.50000 3.80520 3.45490 0.39237 0.96363 3.80521 0.00001 0.07858
*******************************************************************************
6 0.60000 4.19757 4.41853 0.49742 1.14697 4.19758 0.00001 0.09144
*******************************************************************************
7 0.70000 4.69499 5.56550 0.62396 1.39656 4.69501 0.00002 0.10375
*******************************************************************************
8 0.80000 5.31895 6.96207 0.77978 1.73799 5.31897 0.00003 0.11576
*******************************************************************************
9 0.90000 6.09873 8.70006 0.97587 2.20969 6.09877 0.00004 0.12773
*******************************************************************************
10 1.00000 7.07459 10.90974 7.07465 0.00006 0.12773

Метод 3: Метод Адамса
x y
0 0.00000 3.00000
1 0.10000 3.03008
2 0.20000 3.12134
3 0.30000 3.27689
4 0.40000 3.53938
5 0.50000 3.88644
6 0.60000 4.33164
7 0.70000 4.89469
8 0.80000 5.60194
9 0.90000 6.48933
10 1.00000 7.60628

h = 0.1; N = 10
x y z d(y) d(z) y(true) eps err
0 0.00000 3.00000 0.00000 0.03008 0.60334 3.00000 0.00000 0.00167
1 0.10000 3.03008 0.60334 0.09126 0.62369 3.03008 0.00000 0.01836
2 0.20000 3.12134 1.22703 0.15554 0.66579 3.12135 0.00000 0.03468
3 0.30000 3.27689 1.89283 0.22524 0.73269 3.27689 0.00000 0.05027
4 0.40000 3.53938 2.62491 0.22524 0.73269 3.50214 0.03724
5 0.50000 3.88644 3.47058 0.22524 0.73269 3.80521 0.08123
6 0.60000 4.33164 4.45205 0.22524 0.73269 4.19758 0.13406
```

```
7 0.70000 4.89469 5.63045 0.22524 0.73269 4.69501 0.19968
8 0.80000 5.60194 7.07257 0.22524 0.73269 5.31897 0.28297
9 0.90000 6.48933 8.87382 0.22524 0.73269 6.09877 0.39056
10 1.00000 7.60628 11.16951 0.22524 0.73269 7.07465 0.53163
h1 = 0.10000; N1 = 10
h2 = 0.05000; N2 = 20
Runge-Romberg
x y y* err
0 0.00000 3.00000 3.00000 0.00000
1 0.10000 3.03008 3.03008 0.00000
2 0.20000 3.12134 3.12082 0.00053
3 0.30000 3.27689 3.27522 0.00166
4 0.40000 3.53938 3.53891 0.00047
5 0.50000 3.88644 3.88738 0.00094
6 0.60000 4.33164 4.33426 0.00262
7 0.70000 4.89469 4.89938 0.00469
8 0.80000 5.60194 5.60928 0.00734
9 0.90000 6.48933 6.50009 0.01077
10 1.00000 7.60628 7.62157 0.01530
```

Метод 4 — Метод Стрельбы/Метод конечных разностей

Задание
Реализовать метод стрельбы и конечно–разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2–го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге-Ромберга и путем сравнения с точным решением.

Вариант: 3
Краевая задача:
$x^2(x+1)y'' - 2y = 0$
$y'(1) = -1$
$2y(2) - 4'(2) = 4$

Точное решение:
$y(x) = \frac{1}{x} + 1$

Описание алгоритма
Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка.

$y'' = f(x, y, y')$

с граничными условиями первого порядка, заданными на концах отрезка $[a, b]$.

$y(a) = y_0$
$y(b) = y_1$

Следует найти такое решение $y(x)$ на этом отрезке, которое принимает на концах отрезка значения y_0, y_1. Если функция $f(x, y, y')$ линейна по аргументам y, y', то задача является линейной краевой задачей, в противном случае — нелинейной.

Граничные условия второго порядка:

$y'(a) = \hat{y_0}$
$y'(b) = \hat{y_1}$

Граничные условия третьего порядка:

$\alpha y(a) + \beta y'(a) = \hat{y_0}$
$\delta y(b) + \gamma y'(b) = \hat{y_1}$

где $\alpha, \beta, \delta, \gamma$ — такие, что $|\alpha| + |\beta| \neq 0, |\delta| + |\gamma| \neq 0$

Возможно на разных концах отрезка использовать условия различных типов.

Метод стрельбы

Суть метода заключается в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

В случае $\beta \neq 0$ задается $y(a) = \eta$, а $y'(a) = \frac{\hat{y_0} - \alpha\eta}{\beta}$.
В случае $\beta = 0$ задается $y'(a) = \eta$, а $y(a) = \frac{\hat{y_0}}{\alpha}$

В данном случае η — некоторое значение либо высоты, либо угла наклона, которое нужно найти.
При использовании метода деления пополам или метода секущих берутся произвольные η_0, η_1 такие, чтобы значения функций $\Phi(\eta_0), \Phi(\eta_1)$ отличались по знаку.
Другими словами решение исходной задачи эквивалентно нахождению корня уравнения:

$\Phi(\eta) = 0 \qquad (1)$
где $\Phi(\eta) = y(b, y_0, \eta) - y_1$

Уравнение (1) является алгоритмическим уравнением, т.к. левая часть его задается с помощью алгоритма численного решения соответствующей задачи Коши. Следующее значение искомого корня определяется по соотношению:

$\eta_{j+2} = \eta_{j+1} - \frac{\eta_{j+1} - \eta_j}{\Phi(\eta_{j+1}) - \Phi(\eta_j)} \Phi(\eta_{j+1})$

Условие окончания итераций:

$|\eta_{j+1} - \eta_j| \leq \varepsilon$

Метод конечных разностей

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке $[a, b]$

$y'' + p(x)y' + q(x)y = f(x)$
$y(a) = y_0, y(b) = y_1$

Введем разностную сетку на отрезке $[a, b]$ $\quad \Omega^{(h)} = x_k = x_0 + hk, k = 0, 1, ..., N, h = |b - a|/N$.
Решение задачи будем искать в виде сеточной функции $y^{(k)} = y_k, k = 0, 1, ..., N$, предлагая, что решение существует и единственно. Введем разностную аппроксимацию производных следующим образом:

$y'_k = \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2)$

$y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2)$

Подставляя аппроксимации производных получим систему уравнений для нахождения y_k:

$$\begin{cases} (\alpha - \frac{\beta}{h})y_0 + (\frac{\beta}{h})y_1 = \hat{y_0} \\ (\frac{1}{h^2} - \frac{p(x_k)}{2h})y_{k-1} + (-\frac{2}{h^2} + q(x_k))y_k + (\frac{1}{h^2} + \frac{p(x_k)}{2h})y_{k+1} = f(x_k), k = 1, ..., N-1 \\ (-\frac{\gamma}{h})y_{N-1} + (\delta + \frac{\gamma}{h})y_N = \hat{y_1} \end{cases}$$

Для системы при достаточно малых шагах сетки h и $q(x) < 0$ выполнены условия преобладания диагональных элементов

$| - 2 + h^2 q(x_k)| > |1 - \frac{p(x_k)h}{2}| + |1 + \frac{p(x_k)h}{2}|$

что гарантирует устойчивость счета и корректность применения метода прогонки для решения этой системы.
В случае использования граничных условий второго и третьего рода аппроксимация производных проводится с помощью односторонних разностей первого и второго порядков.

$y'_0 = \frac{y_1 - y_0}{h} + O(h)$

$y'_N = \frac{y_N - y_{N-1}}{h} + O(h) \qquad (1)$

$$y_0'' = \frac{-3y_0+4y_1-y_2}{2h} + O(h^2)$$

$$y_N'' = \frac{y_{N-2}-4y_{N-1}+3y_N}{2h} + O(h^2) \qquad (2)$$

В случае использования формул (1) линейная алгебраическая система аппроксимирует дифференциальную задачу в целом только с первым порядком (из-за аппроксимации в граничных точках), однако сохраняется трехдиагональная структура матрицы коэффициентов. В случае использования формул (2) второй порядок аппроксимации сохраняется везде, но матрица линейной системы не трехдиагональная.

Реализация

```
#include <iostream>
#include <functional>
#include <vector>
#include <utility>
#include <cmath>
#include <fstream>
#include "dependences/TSolve.h"

using namespace std;

// TODO: write method Runge-Romberg

pair<double, double> ToSolveRungeKutta(double a,
                     double b,
                     double h,
                     double funcY0,
                     double funcZ0,
                     function<double(double,
                        double)> FuncExpression,
                     const char* fName) {
  /* Runge-Kutta method */
  bool flag = false;
  ofstream out;
  if (!!fName) {
    flag = true;
    out.open(fName, ios::out | ios::app);
  }
```

```
vector<double> X, Y, Z;
double x0 = a, y0 = funcY0, z0 = funcZ0;
size_t N = (b - a) / h;
double K1, K2, K3, K4, L1, L2, L3, L4;
double y_k = y0;
double x_k = x0;
double z_k = z0;
double deltaZ, deltaY;
for (size_t k = 0; k < N; ++k) {
  L1 = h * FuncExpression(x_k, y_k);
  K1 = h * z_k;

  K2 = h * (z_k + 0.5 * L1);
  L2 = h * FuncExpression(x_k + 0.5 * h, y_k +
     0.5 * K1);

  K3 = h * (z_k + 0.5 * L2);
  L3 = h * FuncExpression(x_k + 0.5 * h, y_k +
     0.5 * K2);

  K4 = h * (z_k + L3);
  L4 = h * FuncExpression(x_k + h, y_k + K3);

  deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
     + K4);
  deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
     + L4);

  X.push_back(x_k);
  Y.push_back(y_k);
  Z.push_back(z_k);

  x_k += h;
  z_k += deltaZ;
  y_k += deltaY;
}
X.push_back(x_k);
Y.push_back(y_k);
Z.push_back(z_k);
if (flag) {
  auto FuncY = [](double x) -> double {
```

```
      return 1.0 / x + 1.0;
    };
    vector <double> tmp;
    out.precision(5);
    out << "x : ";
    for (size_t i = 0; i < X.size(); ++i) {
      out << fixed << X[i] << "\t";
      tmp.push_back(FuncY(X[i]));
    }
    out << endl;
    out << "y : ";
    for (size_t i = 0; i < Y.size(); ++i) {
      out << fixed << Y[i] << "\t";
    }
    out << endl;
    out << "y*: ";
    for (size_t i = 0; i < Y.size(); ++i) {
      out << fixed << tmp[i] << "\t";
    }
    out << endl;
    out.close();
  }
  return make_pair(Y.back(), Z.back());
}

void ShooterMethod(const char* f) {
  double a = 1.0, b = 2.0, h = 0.1;
  double n1 = b, n2 = b - h, n;
  double eps = 0.0001;
  double funcZ0 = -1.0;
  size_t j = 0;
  bool flag = false;
  vector<double> X, Y;
  auto FuncExpression = [](double x, double y) ->
     double {
    return 2.0 * y / (x * x * (x + 1.0));
  };
  auto calc = [](double y, double dy) -> double {
    return 2.0 * y - 4.0 * dy;
  };
  auto calcFi = [&calc](const pair<double, double>&
```

```
    p) -> double {
    return calc(p.first, p.second) - 4.0;
  };
  auto checkRightBorder = [&calcFi](const
    pair<double, double>& p, double eps) -> bool {
    return fabs(calcFi(p)) < eps;
  };

  string fn = f;
  fn += "ShooterMethod";
  ofstream out(fn, ios::out);
  out.precision(5);

  out << "\tn\ty\tFi" << endl;
  for (; !flag; j += 2) {
    auto tmp1 = ToSolveRungeKutta(a, b, h, n1,
      funcZ0, FuncExpression, NULL);
    auto tmp2 = ToSolveRungeKutta(a, b, h, n2,
      funcZ0, FuncExpression, NULL);
    out << fixed << j << "\t" << n1 << "\t" <<
      tmp1.first << "\t" << calcFi(tmp1) << endl;
    out << fixed << j + 1 << "\t" << n2 << "\t" <<
      tmp2.first << "\t" << calcFi(tmp2) << endl;
    n = n2 - (n2 - n1)
      / (calcFi(tmp2)
        - calcFi(tmp1))
      * calcFi(tmp2);
    n1 = n2;
    n2 = n;
    auto tmp3 = ToSolveRungeKutta(a, b, h, n,
      funcZ0, FuncExpression, NULL);
    flag = checkRightBorder(tmp3, eps);
  }
  auto tmp = ToSolveRungeKutta(a, b, h, n2, funcZ0,
    FuncExpression, NULL);
  out << fixed << j << "\t" << n2 << "\t" <<
    tmp.first << "\t" << calcFi(tmp) << endl;
  out.close();
  ToSolveRungeKutta(a, b, h, n2, funcZ0,
    FuncExpression, fn.c_str());
}
```

```
void FiniteDifferenceMethod (const char* f) {
  double a = 1.0, b = 2.0, h = 0.1, ya, yb;
  size_t N = fabs(b - a) / h;
  vector<double> X, Y;
  for (size_t i = 0; i <= N; ++i) {
    X.push_back(a + h * i);
  }
  auto qx = [](double x) -> double {
    return -2.0 / (x * x * (x + 1.0));
  };
  auto funcY = [](double x) -> double {
    return 1.0 / x + 1;
  };

  string fn = f;
  string inFile = "dependences/inputData";
  string outFile = "dependences/outputData";

  ofstream out(fn + "FiniteDifferenceMethod",
     ios::out);
  ofstream output(outFile, ios::out);

  output << N - 1 << endl;
  output << -1.0 + h * h * qx(X[0]) << " " << 1.0
     << " " << 0.0 << " " << -h << endl;
  for (size_t i = 1; i < N - 2; ++i) {
    output << 1.0 << " " << -2.0 + h * h * qx(X[i])
       << " " << 1.0 << " " << 0.0 << endl;
  }
  output << 1.0 << " " << -2.0 + h * h * qx(X[N -
     1]) - 4.0 / (2.0 * h - 4.0) << " " << 0.0 << "
     " << -4.0 * h / (2.0 * h - 4.0) << endl;
  output.close();

  TSolve solution(outFile, inFile);
  if (!!solution.ToSolveByTripleDiagMatrix()) {
    cerr << "Error: Troubles with method
       TripleDiagMatrix!" << endl;
    exit(-1);
  }
```

```
  ifstream in(inFile, ios::in);
  double temp;
  while (in >> temp) {
    Y.push_back(temp);
  }
  in.close();
  ya = Y[0] + h;
  yb = 4.0 * (h - Y.back()) / (2.0 * h - 4.0);
  out << "\tx\ty\ty*" << endl;
  out.precision(5);
  out << fixed << 0 << "\t" << X[0] << "\t" << ya
     << "\t" << funcY(X[0]) << endl;
  for (size_t i = 0; i < Y.size(); ++i) {
    out << fixed << i + 1 << "\t" << X[i + 1] <<
       "\t" << Y[i] << "\t" << funcY(X[i + 1]) <<
       endl;
  }
  out << fixed << N << "\t" << X[N] << "\t" << yb
     << "\t" << funcY(X[N]) << endl;
  out.close();
  output.close();
}

pair<double, double> tempSolutionRK(double a,
                     double b,
                     double h,
                     double funcY0,
                     double funcZ0,
                     function<double(double,
                        double)> FuncExpression,
                     vector<double>* v) {
  /* Runge-Kutta method */
  bool flag = false;
  if (!!v) {
    flag = true;
  }
  vector<double> X, Y, Z;
  double x0 = a, y0 = funcY0, z0 = funcZ0;
  size_t N = (b - a) / h;
  double K1, K2, K3, K4, L1, L2, L3, L4;
```

```
double y_k = y0;
double x_k = x0;
double z_k = z0;
double deltaZ, deltaY;
for (size_t k = 0; k < N; ++k) {
  L1 = h * FuncExpression(x_k, y_k);
  K1 = h * z_k;

  K2 = h * (z_k + 0.5 * L1);
  L2 = h * FuncExpression(x_k + 0.5 * h, y_k +
     0.5 * K1);

  K3 = h * (z_k + 0.5 * L2);
  L3 = h * FuncExpression(x_k + 0.5 * h, y_k +
     0.5 * K2);

  K4 = h * (z_k + L3);
  L4 = h * FuncExpression(x_k + h, y_k + K3);

  deltaY = 1.0 / 6.0 * (K1 + 2.0 * K2 + 2.0 * K3
     + K4);
  deltaZ = 1.0 / 6.0 * (L1 + 2.0 * L2 + 2.0 * L3
     + L4);

  X.push_back(x_k);
  Y.push_back(y_k);
  Z.push_back(z_k);

  x_k += h;
  z_k += deltaZ;
  y_k += deltaY;
}
X.push_back(x_k);
Y.push_back(y_k);
Z.push_back(z_k);
if (flag) {
  for (size_t i = 0; i < Y.size(); ++i) {
    v->push_back(Y[i]);
  }
}
return make_pair(Y.back(), Z.back());
```

```
}


vector<double> RungeRombergForShooterMethod(double
   step) {
  double a = 1.0, b = 2.0, h = step;
  double n1 = b, n2 = b - h, n;
  double eps = 0.0001;
  double funcZ0 = -1.0;
  size_t j = 0;
  bool flag = false;
  vector<double> X, Y;
  auto FuncExpression = [](double x, double y) ->
     double {
    return 2.0 * y / (x * x * (x + 1.0));
  };
  auto calc = [](double y, double dy) -> double {
    return 2.0 * y - 4.0 * dy;
  };
  auto calcFi = [&calc](const pair<double, double>&
     p) -> double {
    return calc(p.first, p.second) - 4.0;
  };
  auto checkRightBorder = [&calcFi](const
     pair<double, double>& p, double eps) -> bool {
    return fabs(calcFi(p)) < eps;
  };

  for (; !flag; j += 2) {
    auto tmp1 = tempSolutionRK(a, b, h, n1, funcZ0,
       FuncExpression, NULL);
    auto tmp2 = tempSolutionRK(a, b, h, n2, funcZ0,
       FuncExpression, NULL);
    n = n2 - (n2 - n1)
      / (calcFi(tmp2)
        - calcFi(tmp1))
      * calcFi(tmp2);
    n1 = n2;
    n2 = n;
    auto tmp3 = tempSolutionRK(a, b, h, n, funcZ0,
       FuncExpression, NULL);
```

```
    flag = checkRightBorder(tmp3, eps);
  }
  auto tmp = tempSolutionRK(a, b, h, n2, funcZ0,
     FuncExpression, NULL);
  tempSolutionRK(a, b, h, n2, funcZ0,
     FuncExpression, &Y);
  return Y;
}

vector<double>
   RungeRombergForFiniteDifferenceMethod(double
   step) {
  double a = 1.0, b = 2.0, h = step, ya, yb;
  size_t N = fabs(b - a) / h;
  vector<double> X, Y;
  for (size_t i = 0; i <= N; ++i) {
    X.push_back(a + h * i);
  }
  auto qx = [](double x) -> double {
    return -2.0 / (x * x * (x + 1.0));
  };
  auto funcY = [](double x) -> double {
    return 1.0 / x + 1;
  };

  string inFile = "dependences/inputData";
  string outFile = "dependences/outputData";

  ofstream output(outFile, ios::out);

  output << N - 1 << endl;
  output << -1.0 + h * h * qx(X[0]) << " " << 1.0
     << " " << 0.0 << " " << -h << endl;
  for (size_t i = 1; i < N - 2; ++i) {
    output << 1.0 << " " << -2.0 + h * h * qx(X[i])
       << " " << 1.0 << " " << 0.0 << endl;
  }
  output << 1.0 << " " << -2.0 + h * h * qx(X[N -
     1]) - 4.0 / (2.0 * h - 4.0) << " " << 0.0 << "
     " << -4.0 * h / (2.0 * h - 4.0) << endl;
  output.close();
```

```
  TSolve solution(outFile, inFile);
  if (!!solution.ToSolveByTripleDiagMatrix()) {
    cerr << "Error: Troubles with method
       TripleDiagMatrix!" << endl;
    exit(-1);
  }

  ifstream in(inFile, ios::in);
  double temp;
  while (in >> temp) {
    Y.push_back(temp);
  }
  in.close();
  ya = Y[0] + h;
  yb = 4.0 * (h - Y.back()) / (2.0 * h - 4.0);
  vector<double> result;
  result.push_back(ya);
  for (size_t i = 0; i < Y.size(); ++i) {
    result.push_back(Y[i]);
  }
  result.push_back(yb);
  return result;
}

int main(int argc, char* argv[]) {
  if (argc != 2) {
    cerr << "Error! Incorrect number of arguments"
       << endl;
    exit(-1);
  }
  ShooterMethod(argv[1]);
  {
    string fn = argv[1];
    fn += "Runge-RombergForShooter";
    ofstream out(fn, ios::out);

    double a = 1.0, b = 2.0, h = 0.1;
    double N = fabs(b - a) / h;
    vector<double> Y1, Y2, X;
    for (size_t i = 0; i <= N; ++i) {
```

```
    X.push_back(a + h * i);
  }
  Y1 = RungeRombergForShooterMethod(h);
  Y2 = RungeRombergForShooterMethod(h / 2);
  out.precision(5);
  out << "\tx\ty\ty*\terr" << endl;
  for (size_t i = 0, j = 0; i < X.size() && j <
     Y2.size(); ++i, j += 2) {
    out << fixed << i << "\t" << X[i] << "\t" <<
       Y1[i] << "\t" << Y2[j] + (Y2[j] - Y1[i]) /
       15.0 << "\t" << fabs(Y1[i] - Y2[j] -
       (Y2[j] - Y1[i]) / 15.0)  << endl;
  }
  out.close();
}
FiniteDifferenceMethod(argv[1]);
{
  string fn = argv[1];
  fn += "Runge-RombergForFiniteDifferenceMethod";
  ofstream out(fn, ios::out);

  double a = 1.0, b = 2.0, h = 0.1;
  double N = fabs(b - a) / h;
  vector<double> Y1, Y2, X;
  for (size_t i = 0; i <= N; ++i) {
    X.push_back(a + h * i);
  }
  Y1 = RungeRombergForFiniteDifferenceMethod(h);
  Y2 = RungeRombergForFiniteDifferenceMethod(h /
     2);
  out.precision(5);
  out << "\tx\ty\ty*\terr" << endl;
  for (size_t i = 0, j = 0; i < X.size() && j <
     Y2.size(); ++i, j += 2) {
    out << fixed << i << "\t" << X[i] << "\t" <<
       Y1[i] << "\t" << Y2[j] + (Y2[j] - Y1[i])
       << "\t" << fabs(Y1[i] - Y2[j] - (Y2[j] -
       Y1[i]))  << endl;
  }
  out.close();
}
```

```
  return 0;
}
```

Тестирование

Выходной файл

```
Метод 1: Метод стрельбы
n y Fi
0 2.00000 1.50001 -0.00001
1 1.90000 1.37165 -0.07564
2 2.00002 1.50003 -0.00000
x : 1.00000 1.10000 1.20000 1.30000 1.40000 1.50000 1.60000 1.70000
1.80000 1.90000 2.00000
y : 2.00002 1.90911 1.83335 1.76925 1.71431 1.66669 1.62502 1.58826
1.55558 1.52634 1.50003
y*: 2.00000 1.90909 1.83333 1.76923 1.71429 1.66667 1.62500 1.58824
1.55556 1.52632 1.50000

Рунге-Ромберг:
x y y* err
0 1.00000 2.00002 2.00000 0.00002
1 1.10000 1.90911 1.90909 0.00002
2 1.20000 1.83335 1.83333 0.00002
3 1.30000 1.76925 1.76923 0.00002
4 1.40000 1.71431 1.71429 0.00002
5 1.50000 1.66669 1.66667 0.00002
6 1.60000 1.62502 1.62500 0.00002
7 1.70000 1.58826 1.58824 0.00002
8 1.80000 1.55558 1.55556 0.00003
9 1.90000 1.52634 1.52632 0.00003
10 2.00000 1.50003 1.50000 0.00003

Метод 2: Метод конечных разностей
x y y*
0 1.00000 2.15000 2.00000
1 1.10000 2.05000 1.90909
2 1.20000 1.97000 1.83333
3 1.30000 1.90000 1.76923
4 1.40000 1.85000 1.71429
5 1.50000 1.81000 1.66667
```

```
6 1.60000 1.77000 1.62500
7 1.70000 1.75000 1.58824
8 1.80000 1.72000 1.55556
9 1.90000 1.70000 1.52632
10 2.00000 1.68421 1.50000

Рунге-Ромберг:
x y y* err
0 1.00000 2.15000 2.01000 0.14000
1 1.10000 2.05000 1.91000 0.14000
2 1.20000 1.97000 1.85000 0.12000
3 1.30000 1.90000 1.78000 0.12000
4 1.40000 1.85000 1.73000 0.12000
5 1.50000 1.81000 1.69000 0.12000
6 1.60000 1.77000 1.65000 0.12000
7 1.70000 1.75000 1.59000 0.16000
8 1.80000 1.72000 1.58000 0.14000
9 1.90000 1.70000 1.54000 0.16000
10 2.00000 1.68421 1.51579 0.16842
```

График

Рис. 1: Метод стрельбы

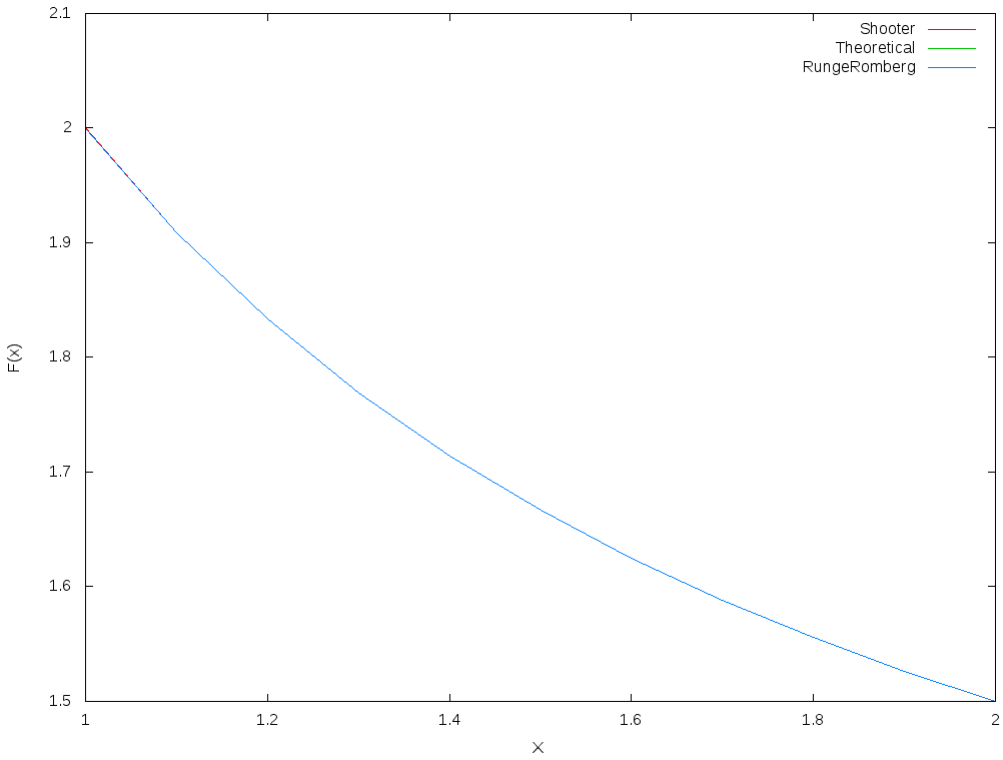

Рис. 2: Метод конечных разностей