Московский авиационный институт

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа № 2
по дисциплине
"Численные методы"

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

Москва, 2016

Метод 1 — Метод простой итерации/Метод Ньютона

Задание

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Замечание: В данном отчете описана теория только по решению систем нелинейных уравнений, а варианты из задания 2.1 считаются как система уравнений просто с одним уравнением.

Вариант: 3

Система 1

$$\begin{cases} \sqrt{1-x^2} - \exp^x + 0.1 = 0 \end{cases}$$

Система 2

$$\begin{cases} (x_1^2 + a^2)x_2 - a^3 = 0 \\ (x_1 - \frac{a}{2})^2 + (x_2 - \frac{a}{2})^2 - a^2 = 0 \\ a = 4 \end{cases}$$

Описание алгоритма

Систему нелинейных уравнений с n неизвестными можно записать в виде

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0 \\ f_2(x_1, x_2, ..., x_n) = 0 \\ ... \\ f_n(x_1, x_2, ..., x_n) = 0 \end{cases}$$

или, более коротко, в векторной форме

$$f(x) = 0$$

где x — вектор неизвестных величин, f — вектор–функция

$$x = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}, f = \begin{pmatrix} f_1(x) \\ f_2(x) \\ ... \\ f_n(x) \end{pmatrix}, 0 = \begin{pmatrix} 0 \\ 0 \\ ... \\ 0 \end{pmatrix}$$

Метод Ньютона

Если определено начальное приближение $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, ..., x_n^{(0)})^T$, итерационный процесс нахождения решения системы методом Ньютона можно представить в виде

$$\begin{cases} x_1^{(k+1)} = x_1^{(k)} + \Delta x_1^{(k)} \\ x_2^{(k+1)} = x_2^{(k)} + \Delta x_2^{(k)} \\ ... \\ x_n^{(k+1)} = x_n^{(k)} + \Delta x_n^{(k)} \end{cases}, k = 0, 1, 2, ...$$

где значения приращений $\Delta x_1^{(k)}, \Delta x_2^{(k)}, ..., x_n^{(k)}$ определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)})$

$$\begin{cases} f_1(x^{(k)}) + \frac{\partial f_1(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_1(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_1(x^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \\ f_2(x^{(k)}) + \frac{\partial f_2(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_2(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_2(x^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \\ ... \\ f_n(x^{(k)}) + \frac{\partial f_n(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_n(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_n(x^{(k)})}{\partial x_n}\Delta x_n^{(k)} = 0 \end{cases}$$

В векторно–матричной форме расчетные формулы имеют вид

$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)}, k = 0, 1, 2, ...$$

где вектор приращений $\Delta x^{(k)} = \begin{pmatrix} \Delta x_1^{(k)} \\ \Delta x_2^{(k)} \\ ... \\ \Delta x_n^{(k)} \end{pmatrix}$ находится из решения уравнения

$$f(x^{(k)}) = x^{(k)} + \Delta x^{(k)}, k = 0, 1, 2, ...$$

Здесь $J(x) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \dots & \frac{\partial f_2(x)}{\partial x_n} \\ ... & & & \\ \frac{\partial f_n(x)}{\partial x_1} & \frac{\partial f_n(x)}{\partial x_2} & \dots & \frac{\partial f_n(x)}{\partial x_n} \end{pmatrix}$ — матрица Якоби первых производных вектор–функции $f(x)$.

Условие окончания итераций:

$$\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon$$

где ε — заданная точность.

Метод простой итерации

При использовании метода простой итерации система уравнений приводится к эквивалентной системе специального вида

$$\begin{cases} x_1 = \phi_1(x_1, x_2, ..., x_n) \\ x_2 = \phi_2(x_1, x_2, ..., x_n) \\ ... \\ x_n = \phi_n(x_1, x_2, ..., x_n) \end{cases}$$

или, в векторной форме

$$x = \phi(x), \quad \phi(x) = \begin{pmatrix} \phi_1(x) \\ \phi_2(x) \\ ... \\ \phi_n(x) \end{pmatrix}$$

Если последовательность векторов $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)})^T$ сходится, то она сходится к решению $x^{(*)} = (x_1^{(*)}, x_2^{(*)}, ..., x_n^{(*)})$.

Достаточное условие сходимости итерационного процесса формулируется следующим образом:

Пусть вектор–функция $\phi(x)$ непрерывна, вместе со своей производной

$$\phi'(x) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \dots & \frac{\partial f_2(x)}{\partial x_n} \\ ... & & & \\ \frac{\partial f_n(x)}{\partial x_1} & \frac{\partial f_n(x)}{\partial x_2} & \dots & \frac{\partial f_n(x)}{\partial x_n} \end{pmatrix}$$

в ограниченной выпуклой замкнутой области G и

$$\max_{x \in G} \|\phi'(x)\| \leq q < 1,$$

где q — постоянная. Если $x^{(0)} \in G$ и все последовательные приближения

$$x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2, ...$$

также содержатся в G, то процесс итерации сходится к единственному решению уравнения

$$x = \phi(x)$$

в области G и справедливы оценки погрешности ($\forall k \in N$):

$$\|x^{(*)} - x^{(k+1)}\| \leq \frac{q^{k+1}}{1-q}\|x^{(1)} - x^{(0)}\|$$

$$\|x^{(*)} - x^{(k+1)}\| \leq \frac{q}{1-q}\|x^{(k+1)} - x^{(k)}\|$$

Реализация

2.1:

```
void ToSolveBySimpleIter() {
    ofstream log("solveSimpleIter.log",
       ios::out);
    _a = 0.0;
    _b = 0.5;
    double x_0 = (_a + _b) / 2;
    double q = max(abs(Dfi(_b)), abs(Dfi(_a)));
       // as function monotonously
    double eps_k = EPS;
    double solve = _b + 1.0;
    for (int iter = 0; iter < 1000; ++iter) {
        solve = fi(x_0);
        log << "x_" << iter << " = " << x_0 <<
           "; ";
        log << "x_" << iter + 1 << " = " <<
           solve << "; ";
        eps_k = q / (1.0 - q) * abs(solve -
           x_0);
        log << "eps_" << iter << " = " << eps_k
           << endl;
        if (eps_k < EPS)
            break;
        x_0 = solve;
    }
    log.close();
    out << "solve = " << solve << endl;
}
```

```
    void ToSolveByNewtoon() {
        ofstream log("solveNewtoon.log", ios::out);
        _a = 0.0;
        _b = 1.0;
        double x_0 = (_a + _b) / 2;
        if (F(x_0) * DDF(x_0) <= 0) { // must check
           this
            x_0 = _a;
            if (F(x_0) * DDF(x_0) <= 0)
                x_0 = _b;
        }
        log << "x_0 = " << x_0 << endl;
        double solve = _b + 1.0;
        for (size_t iter = 0; iter < 1000; iter++) {
            double x_new = x_0 - F(x_0) / DF(x_0);
            log << "x_" << iter + 1 << " = " <<
               x_new << endl;
            if (abs(F(x_new)) < EPS && abs(x_new -
               x_0) < EPS) {
                solve = x_new;
                break;
            }
            else {
                x_0 = x_new;
            }
        }
        log.close();
        out << "solve: " << solve << endl;
    }
```

2.2:

```
void ToSolveBySimpleIter() {
        ofstream log("solveSimpleIter.log",
           ios::out);
        TVector x_0(GetAverageVector(_borders));
        auto CalcJakobian_F = [](double x1, double
           x2, double a) -> double {
            return max(abs(2 * x2 * x1),
                       max(abs(2 * x2 - a),
                           max(abs(2 * x1 - a),
```

```
                        abs(x1 * x1 + a *
                        a))));
};
double J_0 = CalcJakobian_F(x_0[0], x_0[1],
   A);
log << "J_0 = " << J_0 << endl;
auto CalcJakobian_fi = [=](double x1,
   double x2, double a, double J) -> double
   {
    return max(abs(1.0 - (2 * x2 * x1) / J),
            max(abs((x1 * x1 + a * a) / J),
                max(abs((2 * x1 - a) / J),
                   abs(1.0 - (2 * x2 - a) /
                   J))));
};
auto PrintJakobian_fi = [=](double x1,
   double x2, double a, double J) -> double
   {
    cout << abs(1.0 - (2 * x2 * x1) / J) <<
       endl <<
            abs((x1 * x1 + a * a) / J) <<
               endl <<
                abs((2 * x1 - a) / J) <<
                   endl << abs(1.0 - (2 *
                   x2 - a) / J) << endl;
};
auto fi_1 = [&](double x1, double x2,
   double a, double J) -> double {
    return x1 - (_systemFunctions[0](x1,
       x2, A) < 0 ? -1 : 1) *
       _systemFunctions[0](x1, x2, A) / J;
};
auto fi_2 = [&](double x1, double x2,
   double a, double J) -> double {
    return x2 - (_systemFunctions[1](x1,
       x2, A) < 0 ? -1 : 1) *
       _systemFunctions[1](x1, x2, A) / J;
};
double q;
try {
    q = CalcJakobian_fi(x_0[0], x_0[1], A,
```

```
            J_0);
    }
    catch (const out_of_range& e) {
        cerr << e.what() << endl;
    }
    log << "q = " << q << endl;
if (q >= 1.0) {
  cerr << "Error: q more than 1" << endl;
  PrintJakobian_fi(x_0[0], x_0[1], A, J_0);
  exit(-1);
}
    auto norm = [](const TVector& x) -> double {
        double res = 0;
        for (int i = 0; i < x.GetSize(); ++i)
            res = max(abs(x[i]), res);
        return res;
    };
    double eps_k = EPS;
    TVector solution(x_0.GetSize());
    for (int iter = 0; iter < 1000; ++iter) {
        solution[0] = fi_1(x_0[0], x_0[1], A,
           J_0);
        solution[1] = fi_2(x_0[0], x_0[1], A,
           J_0);
        log << "x[" << iter << "] = (" <<
           x_0[0] << ", " << x_0[1] << "); ";
        log << "x[" << iter + 1 << "] = (" <<
           solution[0] << ", " << solution[1]
           << "); ";
        eps_k = q / (1.0 - q) * norm(solution -
           x_0);
        log << "eps_" << iter << " = " << eps_k
           << endl;
        if (eps_k < EPS)
            break;
        x_0 = solution;
    }
    log.close();
    out << "solution = (" << solution[0] << ",
       " << solution[1] << ")" << endl;
}
```

```
void ToSolveByNewtoon() {
    ofstream log("solveNewtoon.log", ios::out);
    TVector x_0(GetAverageVector(_borders));
TVector solution(x_0);
string inputStr = "./dependens/inputData";
string outputStr = "./dependens/outputData";
size_t szOfMatrix = 2;
auto write2File = [&] (double x1, double x2,
   double a, ofstream& o) {
  o << (2 * x2 * x1) << " " << (x1 * x1 + a *
     a) << " " << -1.0 * ((x1 * x1 + a * a) *
     x2 - a * a * a) << endl;
  o << (2 * x1 - a) << " " << (2 * x2 - a) << "
     " << -1.0 * (pow(x1 - a/2, 2.0) + pow(x2 -
     a/2, 2.0) - a * a) << endl;
};
auto findMax = [] (const vector<double>& v) ->
   double {
  double res = abs(v.back());
  for (size_t i = 0; i < v.size(); ++i)
    res = max(res, abs(v[i]));
  return res;
};
double eps_k = EPS + 1;
  for (size_t iter = 0; iter < 1000; iter++) {
    log << "x_[" << iter << "] = (" << x_0[0]
       << ", " << x_0[1] << "); ";
  ofstream o(inputStr.c_str(), ios::out);
    o << szOfMatrix << endl;
  write2File(x_0[0], x_0[1], A, o);
    o.close();
  TSolve solve(inputStr, outputStr);
  if (!solve.ToSolveByGauss()) {
    ifstream in(outputStr.c_str(), ios::in);
          vector<double> tempVec;
    double tmp;
    while (1) {
        in >> tmp;
        if (in.eof())
            break;
```

```
                tempVec.push_back(tmp);
        }
        eps_k = findMax(tempVec);
        solution = TVector(tempVec);
        in.close();
        }
        else {
             cerr << "Some troubles..." << endl;
           exit(-1);
        }
        solution = x_0 + solution;
        log << "x_[" << iter + 1 << "] = (" <<
           solution[0] << ", " << solution[1] <<
           "); ";
        log << "eps[k] = " << eps_k << endl;
          if (eps_k >= EPS) {
        x_0 = solution;
          }
          else break;
      }
        log.close();
        out << "solve: ";
    solution.Print(out);
    }
```

Тестирование

Входной файл (система 1 — метод простой итерации)

```
0.001
```

Выходной файл (система 1 — метод простой итерации)

```
x_0 = 0.25; x_1 = 0.039653; eps_0 = 0.312456
x_1 = 0.039653; x_2 = 0.0983719; eps_1 = 0.087223
x_2 = 0.0983719; x_3 = 0.0901856; eps_2 = 0.0121602
x_3 = 0.0901856; x_4 = 0.0917273; eps_3 = 0.00229011
x_4 = 0.0917273; x_5 = 0.0914467; eps_4 = 0.000416835

solve = 0.0914467
```

Выходной файл (система 1 — метод Ньютона)

```
x_0 = 0.5
x_1 = -0.137217
x_2 = 0.0793109
x_3 = 0.0935775
x_4 = 0.091104
x_5 = 0.0915607

solve: 0.0915607
```

Входной файл (система 2 — метод простой итерации)

```
0.01
5 6 1 2
```

Выходной файл (система 2 — метод простой итерации)

Входной файл (система 2 — метод Ньютона)

```
0.01
5 6 1 2
```

Выходной файл (система 2 — метод Ньютона)

```
solve: 5.92926 1.25107
```