**Лабораторная работа №1**

**Метод Галёркина построения разностных схем**

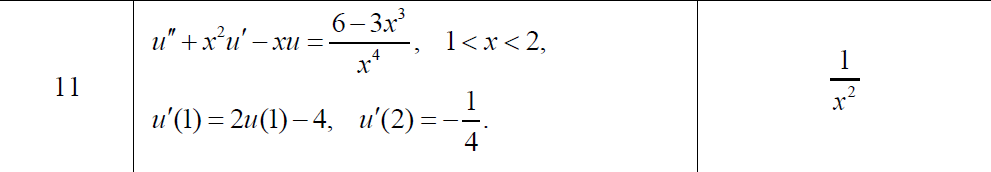
**Вариант 11**

**Чеботаревский Никита**

**3 курс, 8 группа**

**Постановка задачи**

Найти численное решение y граничной задачи из лабораторной работы №1 на равномерной сетке с шагом h=0.01 с помощью разностной схемы второго порядка аппроксимации, построенной методом Галёркина. Интегралы в коэффициентах разностной схемы вычислять приближённо. Сравнить найденное решение y с точным решением u(x), т.е найти . В одной системе координат построить график функции u(x) и график полученного численного решения y.



Оформить отчёт, который будет содержать следующую информацию:

* Постановка задачи;
* Краткие теоретические сведения (выписать разностную схему, прогоночные коэффициенты);
* Результаты решение поставленной задачи;
* Выводы;
* Листинг программы с комментариями;

**Построение разностной схемы**



Тогда поставленная задача имеет вид:

Выпишем все необходимые нам функции и коэффициенты, необходимые для решения нашей задачи:

Необходимо построить разностную схему второго порядка аппроксимации вида:

Коэффициенты считаются по следующим формулам:



Выпишем нашу задачу в индексной форме:

В первом уравнении распишем левую разностную производную, потом во всех уравнениях сгруппируем коэффициенты и в результате получим:

Тогда систему можно представить в виде:

Откуда прогоночные коэффициенты имеют вид:

**Метод прогонки**

Полученную систему линейных алгебраических уравнений с трёхдиагональной матрицей будем решать методом прогонки:

Проверка условий устойчивости метода прогонки:

Третье условие у нас выполняется строго так как в нашем случае , а h>0, а , поэтому можно говорить об устойчивости метода прогонки для данной системы.

**Листинг программы**

*import* matplotlib.pyplot *as* plt  
  
  
*class* Solver:  
 *def* \_\_init\_\_(*self*, a, b, h):  
 *self*.a = a  
 *self*.b = b  
 *self*.h = h  
  
 *def* create\_grid(*self*):  
 n = int((*self*.b - *self*.a) / *self*.h)  
 *return* [*self*.a + i \* *self*.h *for* i *in* range(n + 1)]  
  
 @staticmethod  
 *def* r(x):  
 *return* x \*\* 2  
  
 @staticmethod  
 *def* q(x):  
 *return* x  
  
 @staticmethod  
 *def* f(x):  
 *return* (3 \* x \*\* 3 - 6) / x \*\* 4  
  
 *def* bi(*self*, x):  
 *return self*.r(x) / 2  
  
 *def* d(*self*, x):  
 *return* (*self*.q(x + *self*.h / 2) + *self*.q(x - *self*.h / 2)) / 2  
  
 *def* d\_0\_n(*self*, x):  
 *return self*.q(x)  
  
 *def* phi(*self*, x):  
 *return* (*self*.f(x + *self*.h / 2) + *self*.f(x - *self*.h / 2)) / 2  
  
 *def* ai(*self*, x):  
 *return* 1 - *self*.h \*\* 2 / 6 \* *self*.q(x - *self*.h / 2)  
  
 *def* ai\_coef(*self*, x):  
 *return self*.ai(x) / *self*.h \*\* 2 - *self*.bi(x - *self*.h / 2) / *self*.h  
  
 *def* bi\_coef(*self*, x):  
 *return self*.ai(x + *self*.h) / *self*.h \*\* 2 + *self*.bi(x + *self*.h / 2) / *self*.h  
  
 *def* ci\_coef(*self*, xi):  
 *return* (*self*.ai(xi + *self*.h) + *self*.ai(xi)) / *self*.h \*\* 2 + *self*.bi(xi + *self*.h / 2) / *self*.h - \  
 *self*.bi(xi - *self*.h / 2) / *self*.h + *self*.d(xi)  
  
 *def* c0\_coef(*self*, x):  
 *return* (*self*.ai(x + *self*.h) + *self*.bi(x + *self*.h / 2) \* *self*.h) / *self*.h + 2 + \  
 *self*.h / 2 \* *self*.d\_0\_n(x + *self*.h / 2)  
  
 *def* b0\_coef(*self*, x):  
 *return* (*self*.ai(x + *self*.h) + *self*.bi(x + *self*.h / 2) \* *self*.h) / *self*.h  
  
 *def* f0\_coef(*self*, x):  
 *return* 4 + *self*.h / 2 \* *self*.f(x + *self*.h / 2)  
  
 *def* an\_coef(*self*, x):  
 *return* (*self*.ai(x) - *self*.bi(x - *self*.h / 2) \* *self*.h) / *self*.h  
  
 *def* cn\_coef(*self*, x):  
 *return* (*self*.ai(x) - *self*.bi(x - *self*.h / 2) \* *self*.h) / *self*.h + *self*.h / 2 \* *self*.d\_0\_n(x - *self*.h / 2)  
  
 *def* fn\_coef(*self*, x):  
 *return* -1 / 4 + *self*.h / 2 \* *self*.f(x - *self*.h / 2)  
  
 @staticmethod  
 *def* u(x):  
 *return* 1 / x \*\* 2  
  
 *def* count\_coefficients(*self*):  
 a, c, b, f = [], [], [], []  
 grid = *self*.create\_grid()  
  
 *for* i *in* range(len(grid)):  
 *if* i == 0:  
 c.append(*self*.c0\_coef(grid[i]))  
 b.append(*self*.b0\_coef(grid[i]))  
 f.append(*self*.f0\_coef(grid[i]))  
 *elif* i == len(grid) - 1:  
 a.append(*self*.an\_coef(grid[i]))  
 c.append(*self*.cn\_coef(grid[i]))  
 f.append(*self*.fn\_coef(grid[i]))  
 *else*:  
 a.append(*self*.ai\_coef(grid[i]))  
 c.append(*self*.ci\_coef(grid[i]))  
 b.append(*self*.bi\_coef(grid[i]))  
 f.append(*self*.phi(grid[i]))  
  
 *return* a, c, b, f  
  
 *def* sweep\_method(*self*):  
 a, c, b, f = *self*.count\_coefficients()  
 grid = *self*.create\_grid()  
  
 alpha = [b[0] / c[0]]  
 betta = [f[0] / c[0]]  
 y = [0] \* len(grid)  
  
 *for* i *in* range(1, len(grid) - 1):  
 alpha.append(b[i] / (c[i] - a[i - 1] \* alpha[i - 1]))  
  
 *for* i *in* range(1, len(grid)):  
 betta.append((f[i] + a[i - 1] \* betta[i - 1]) / (c[i] - a[i - 1] \* alpha[i - 1]))  
  
 y[len(grid) - 1] = betta[-1]  
 *for* i *in* range(len(grid) - 2, -1, -1):  
 y[i] = alpha[i] \* y[i + 1] + betta[i]  
  
 *return* y  
  
 @staticmethod  
 *def* count\_accuracy(u, y):  
 *return* max([abs(u[i] - y[i]) *for* i *in* range(len(y))])  
  
  
*def* draw\_plots(x, u, y):  
 plt.plot(x[:2], y[:2])  
 plt.plot(x[:2], u[:2])  
 plt.xlim(1.0, 1.00005)  
 plt.ylim(0.9999, 1.000)  
 plt.show()  
  
  
sol1 = Solver(1, 2, 0.01)  
  
y1 = sol1.sweep\_method()  
print(y1)  
  
u1 = [sol1.u(x) *for* x *in* sol1.create\_grid()]  
print(u1)  
  
print(f'Accuracy is {sol1.count\_accuracy(u1, y1)}', end='\n' \* 2)  
draw\_plots(sol1.create\_grid(), u1, y1)  
  
plt.plot(sol1.create\_grid(), u1)  
plt.show()

**Графики**

