**Лабораторная работа №3**

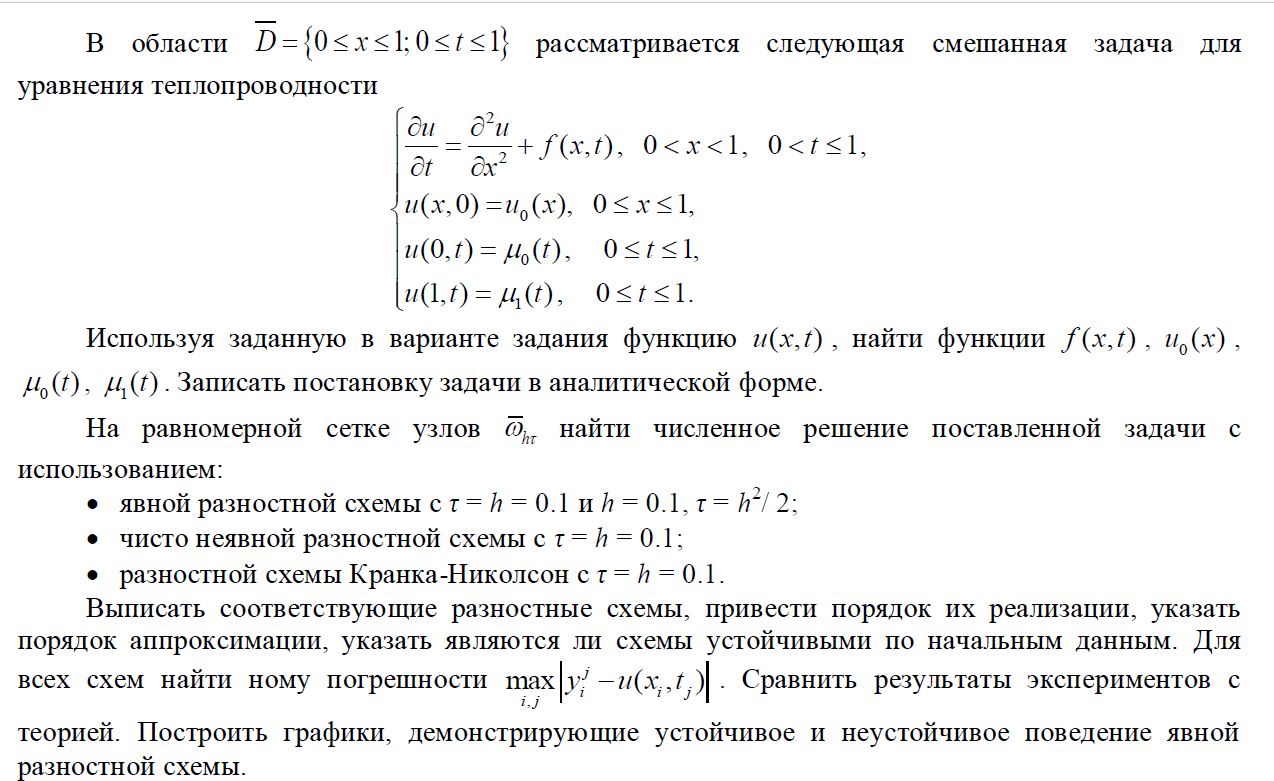
**Численное решение смешанной задачи для уравнения теплопроводности**

**Вариант 11**

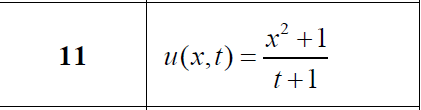
**Чеботаревский Никита**

**3 курс, 8 группа**

**Постановка задачи**







По результатам необходимо оформить отчёт, содержащий следующую информацию:

* Титульный лист
* Постановка задачи
* Краткие теоретические сведения
* Результаты
* Выводы
* Листинг программы с комментариями

**Теоретические сведения**

Смешанная задача для уравнения теплопроводности, в которой , имеет вид:

Разностная схема имеет вид:

Есть несколько видов разностных схем для решения смешанной задачи для уравнения теплопроводности, которые зависят от

, тогда имеем явную разностную схему,

, получаем чисто неявную разностную схему,

получаем неявную разностную схему.

Проверим разностную схему на устойчивость с помощью метода гармоник:

где при неравенство выполняется для любого , а при

получаем следующее:

**1) Явная разностная схема,**

1. Заполняем нулевой слой

**2) Чисто неявная разностная схема, :**

1. Заполняем нулевой слой

Таким образом на каждой итерации *j* можно записать в систему вида:

Где

**3) Разностная схема Кранка-Николсон, :**

1. Заполняем нулевой слой

Таким образом 2) на каждой итерации *j* можно записать в систему вида:

Где

Системы полученные, в пунктах 2 и 3 будем решать методом прогонки.

**Метод прогонки**

Полученную трехдиагональную матрицу будем решать методом прогонки по формулам:

Проверка условий устойчивости метода прогонки:

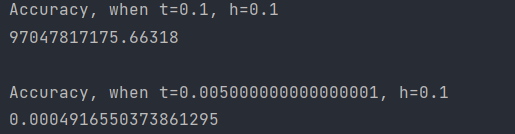
Все неравенства выполняются строго, поэтому метод прогонки будет устойчив

**Листинг программы**

*import* matplotlib.pyplot *as* plt  
  
  
*class* Solver:  
 *def* \_\_init\_\_(*self*, h, t, a=0, b=1):  
 *self*.h = h  
 *self*.t = t  
 *self*.a = a  
 *self*.b = b  
  
 *def* \_\_get\_size(*self*):  
 n1 = int((*self*.b - *self*.a) / *self*.h)  
 n2 = int((*self*.b - *self*.a) / *self*.t)  
  
 *return* n1, n2  
  
 *def* create\_grid(*self*):  
 n1, n2 = *self*.\_\_get\_size()  
 xi = [0] \* (n1 + 1)  
 ti = [0] \* (n2 + 1)  
  
 *for* i *in* range(len(xi)):  
 xi[i] = *self*.a + i \* *self*.h  
  
 *for* i *in* range(len(ti)):  
 ti[i] = *self*.a + i \* *self*.t  
  
 *return* xi, ti  
  
 @staticmethod  
 *def* u(x, t):  
 *return* (x \*\* 2 + 1) / (t + 1)  
  
 *def* phi(*self*, x, t, sigma=1.0):  
 *if* sigma == 1:  
 *return* (-x \*\* 2 - 1) / (t \*\* 2 + 2 \* t + 1) - 2 / (t + 1)  
 *return* (-x \*\* 2 - 1) / ((t + *self*.t / 2) \*\* 2 + 2 \* (t + *self*.t / 2) + 1) - 2 / ((t + *self*.t / 2) + 1)  
  
 *def* solution(*self*):  
 xi, ti = *self*.create\_grid()  
 sol = [[0] \* len(xi) *for* \_ *in* range(len(ti))]  
  
 *for* j *in* range(len(ti)):  
 *for* i *in* range(len(xi)):  
 sol[j][i] = *self*.u(xi[i], ti[j])  
  
 *return* sol  
  
 *def* explicit\_task(*self*):  
 xi, ti = *self*.create\_grid()  
 u = [[0] \* len(xi) *for* \_ *in* range(len(ti))]  
  
 *for* i *in* range(len(xi)):  
 u[0][i] = *self*.u(xi[i], 0)  
  
 *for* j *in* range(len(ti) - 1):  
 u[j + 1][0] = *self*.u(0, ti[j + 1])  
  
 *for* j *in* range(len(ti) - 1):  
 u[j + 1][len(xi) - 1] = *self*.u(1, ti[j + 1])  
  
 *for* j *in* range(len(ti) - 1):  
 *for* i *in* range(1, len(xi) - 1):  
 u[j + 1][i] = u[j][i] + (*self*.t / *self*.h \*\* 2) \* (u[j][i + 1] - 2 \* u[j][i] + u[j][i - 1]) + \  
 *self*.t \* *self*.phi(xi[i], ti[j])  
  
 *return* u  
  
 *def* get\_coeff(*self*, y, index, size, sigma=1.0):  
 xi, ti = *self*.create\_grid()  
 a, b, c, f = [], [], [], []  
  
 *for* i *in* range(size + 1):  
 *if* i == 0:  
 b.append(0)  
 c.append(1)  
 f.append(*self*.u(0, ti[index + 1]))  
 *elif* i == size:  
 c.append(1)  
 a.append(0)  
 f.append(*self*.u(1, ti[index + 1]))  
 *else*:  
 a.append(sigma / *self*.h \*\* 2)  
 b.append(sigma / *self*.h \*\* 2)  
 c.append(1 / *self*.t + (2 \* sigma) / *self*.h \*\* 2)  
 f.append(y[i] / *self*.t + *self*.phi(xi[i], ti[index], sigma) +  
 ((1 - sigma) / *self*.h \*\* 2) \* (y[i + 1] - 2 \* y[i] + y[i - 1]))  
  
 *return* a, c, b, f  
  
 *def* sweep\_method(*self*, yj, index, size, sigma=1.0):  
 a, c, b, f = *self*.get\_coeff(yj, index, size, sigma)  
 xi, ti = *self*.create\_grid()  
  
 alpha = [b[0] / c[0]]  
 betta = [f[0] / c[0]]  
 y = [0] \* len(xi)  
  
 *for* i *in* range(1, len(xi) - 1):  
 alpha.append(b[i] / (c[i] - a[i - 1] \* alpha[i - 1]))  
  
 *for* i *in* range(1, len(xi)):  
 betta.append((f[i] + a[i - 1] \* betta[i - 1]) / (c[i] - a[i - 1] \* alpha[i - 1]))  
  
 y[len(xi) - 1] = betta[-1]  
 *for* i *in* range(len(xi) - 2, -1, -1):  
 y[i] = alpha[i] \* y[i + 1] + betta[i]  
  
 *return* y  
  
 *def* implicit\_task(*self*, sigma=1.0):  
 xi, ti = *self*.create\_grid()  
 u = [[0] \* (len(xi)) *for* \_ *in* range(len(ti))]  
  
 *for* i *in* range(len(xi)):  
 u[0][i] = *self*.u(xi[i], 0)  
  
 *for* j *in* range(len(ti) - 1):  
 u[j + 1][0] = *self*.u(0, ti[j + 1])  
 u[j + 1][-1] = *self*.u(1, ti[j + 1])  
 tmp = *self*.sweep\_method(u[j], j, len(xi) - 1, sigma)  
  
 *for* k *in* range(1, len(xi) - 1):  
 u[j + 1][k] = tmp[k]  
  
 *return* u  
  
 @staticmethod  
 *def* count\_accuracy(u, y):  
 accuracy = []  
  
 *for* j *in* range(len(u)):  
 *for* i *in* range(len(u[0])):  
 accuracy.append(abs(u[j][i] - y[j][i]))  
  
 *return* max(accuracy)  
  
  
*def* draw\_plot(x, y, u):  
 plt.plot(x, y)  
 plt.scatter(x, u, c="r")  
 plt.show()  
  
  
sol1 = Solver(0.1, 0.1)  
y1 = sol1.explicit\_task()  
u1 = sol1.solution()  
  
draw\_plot(sol1.create\_grid()[0], y1[-2], u1[-2])  
  
print(f'Accuracy, when t=0.1, h=0.1')  
print(sol1.count\_accuracy(u1, y1), end="\n" \* 2)  
  
sol2 = Solver(0.1, 0.1 \*\* 2 / 2)  
y2 = sol2.explicit\_task()  
u2 = sol2.solution()  
  
draw\_plot(sol2.create\_grid()[0], y2[-2], u2[-2])  
  
print(f'Accuracy, when t=0.1, h={0.1 \*\* 2 / 2}')  
print(sol2.count\_accuracy(u2, y2), end="\n" \* 2)  
  
sol3 = Solver(0.1, 0.1)  
y3 = sol3.implicit\_task()  
u3 = sol3.solution()  
print(f'Accuracy, when t=0.1, h=0.1 and sigma=1')  
print(sol3.count\_accuracy(u3, y3), end="\n" \* 2)  
  
sol4 = Solver(0.1, 0.1)  
y4 = sol4.implicit\_task(0.5)  
u4 = sol4.solution()  
print(f'Accuracy, when t=0.1, h=0.1 and sigma=0.5')  
print(sol4.count\_accuracy(u4, y4))

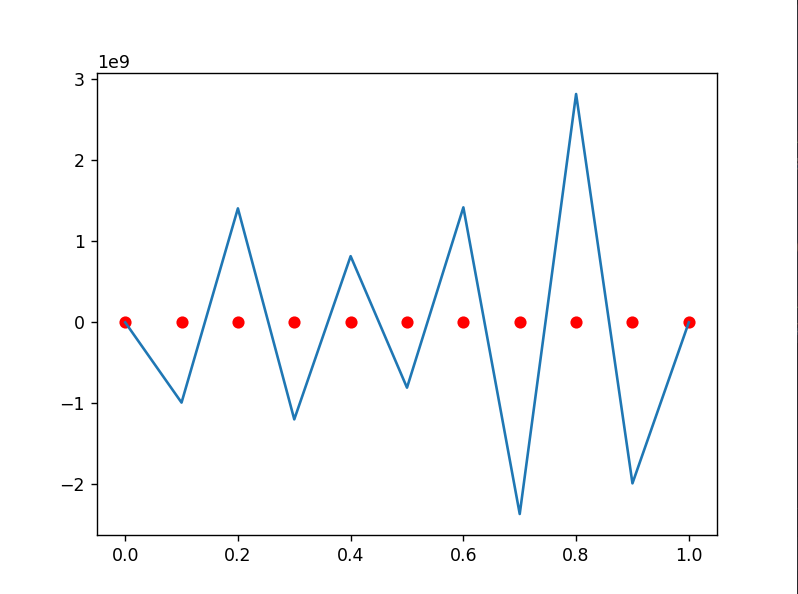
**Результаты**

**1) Для явной разностной схемы:**

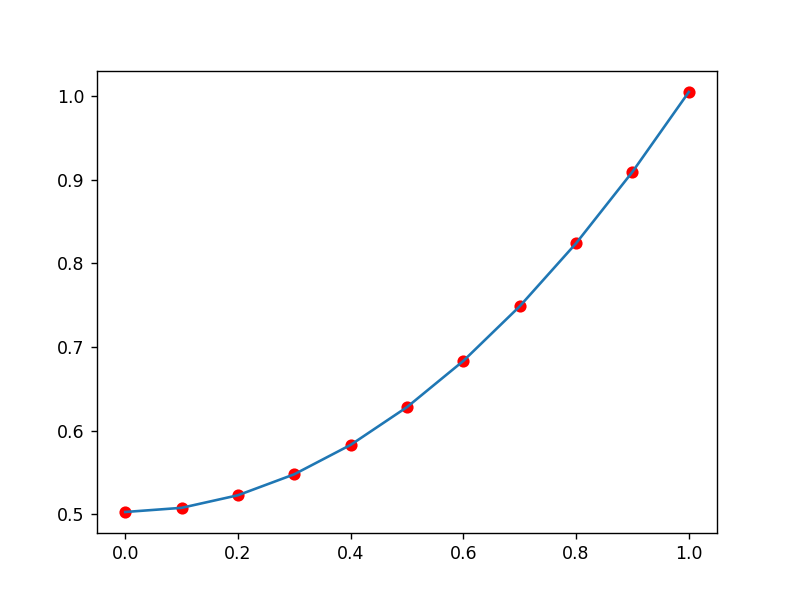
****

**Графики, отражающие устойчивость и неустойчивость явной разностной схемы:**

**1)**



**2)**



**2**) **Чисто неявная разностная схема:**

****

**3) Схема Кранка-Николсон:**

****

**Выводы**

Результаты при решении смешанной задачи с помощью явной разностной схемы соответствует условию устойчивости, полученному ранее. Схема Кранка-Николсона дала более точные значения, чем чисто неявная разностная схема. Это связано с тем, что первая имеет порядок , а вторая -