Инициализация используемых параметров и констант представлена в файле fields_calc.py:

1. Зададим количество направлений дискретизации и размер решетки Больцмана:

```
Q = 9

Nx = Nx1 - 1
Ny = Ny1 - 1
```

2. cx и cy задают элементарные направления дискретизации для каждого $i = \overline{0,8}$; w – это вектор весовых коэффициентов, используемых при нахождении равновесной функции распределения; rc содержит индексы $\bar{\imath}$ направлений векторов, противоположно направленных векторам с индексами i; cs_2 это квадрат скорости звука, под скоростью звука понимают величину $c_s = \sqrt{RT} = 1/\sqrt{3}$.

```
cx = np.array([0, 1, 0, -1, 0, 1, -1, -1, 1])
cy = np.array([0, 0, 1, 0, -1, 1, 1, -1, -1])
w = np.array([4.0 / 9, 1.0 / 9, 1.0 / 9, 1.0 / 9, 1.0 / 9, 1.0 / 36, 1.0 /
36, 1.0 / 36, 1.0 / 36])
rc = np.array([0, 3, 4, 1, 2, 7, 8, 5, 6])  # index of reversed velocity for
BB scheme
cs_2 = 1 / 3 #cs^2
cs_minus2 = 3 #cs^(-2)
```

3. Далее зададим начальные значения: rho0, ux0 и uy0 – значение массы жидкости, скорости по оси OX и оси OY в момент t0; f и f_post – функции распределения, f_post хранит значения после столкновения молекул.

```
#########################   Initialization  ######################
rho0 = 4.0
ux0 = 0
uy0 = -0.2

rho = np.array([[rho0] * Nx1] * Ny1)
ux = np.array([[ux0] * Nx1] * Ny1)
uy = np.array([[uy0] * Nx1] * Ny1)
f = np.array([[[0.0] * Nx1] * Ny1] * Q)
f_post = np.array([[[0.0] * Nx1] * Ny1] * Q)
```

4. Зададим для TRT-оператора магический параметр Λ, ω^+ и ω^-; v0 и u0 – скорости по осям OY и OX, с которыми жидкость поступает сверху в рассматриваемую область.

```
############################   TRT vars:  ######################
magic_parameter = 1 / 6
omega_plus = 1.1
omega_min = 1.0 / (magic_parameter / (1.0 / omega_plus - 0.5) + 0.5)

v0 = np.array([-0.2] * Nx1)  # top inlet velocity y
u0 = np.array([0.0] * Nx1)  # top inlet velocity x
```

5. Для инициализации функции плотности распределения f в начальный момент времени используется равновесная функция распределения feq:

```
def init_eq(ux, uy, rho):
    for k in range(Q):
        f[k] = feq(rho, ux, uy, k)
    return f
```

В файле solid_nodes.py зададим матрицу epsilon, которая будет содержать информацию о доли твердого вещества в узле.

solid_nodes_image.py создает матрицу epsilon по входному изображению Porous1.jpg.

Основные функции алгоритма

В файле fields_calc:

Равновесная функция распределения feq():

```
def feq(rho1, u, v, k):
    cu = cx[k] * u + cy[k] * v
    u2 = u * u + v * v
    return w[k] * rho1 * (1.0 + 3.0 * cu + 4.5 * cu * cu - 1.5 * u2)


def coll_trt(rho, ux, uy):
    f_post[0] = f[0] - omega_plus * (f[0] - feq(rho, ux, uy, 0))
    for k in range(1, Q):
        f_post[k] = f[k] - omega_plus * (
                0.5 * (f[k] + f[rc[k]]) - 0.5 * (feq(rho, ux, uy, k) +
feq(rho, ux, uy, rc[k]))) - omega_min * (
                                0.5 * (f[k] - f[rc[k]]) - 0.5 * (feq(rho, ux, uy,
k) - feq(rho, ux, uy, rc[k])))
    return f_post
```

coll_trt() возвращает функцию распределения после этапа столкновения молекул жидкости.

Граничные условия

Определим граничные условия по периметру рассматриваемой области:

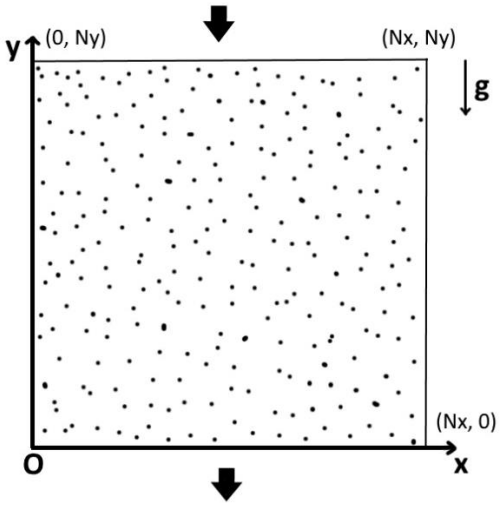

Рис. 1 – Исследуемый образец

Отрезки (0, 0)-(0, Ny) и (Nx, Ny)-(Nx, 0) представляют собой стенки некоторого сосуда; применим для них периодические граничные условия. В связи с простотой реализации включим их в этап распространения молекул (streaming step).

Будем считать, что жидкость поступает сверху, просачивается через образец грунта и снизу свободно его покидает. Тогда на отрезках (0, Ny)-(Nx, Ny) применим открытые граничные условия (inlet boundary conditions) [22]; а на (0, 0)-(Nx, 0) условия нулевого градиента по функции распределения:

```
def top_inlet_bottom_outlet(v0, u0):
    v0 = v0 * solids[Ny, :]
   u0 = u0 * solids[Ny, :]
    rho_north = (1 / (1 + v0)) * (f[0][Ny, :] + f[1][Ny, :] + f[3][Ny, :] +
                                  2 * (f[5][Ny, :] + f[2][Ny, :] + f[6][Ny,
:]))

    f[4][Ny, :] = f[2][Ny, :] - 2 / 3 * v0 * rho_north
    f[7][Ny, 1:Nx1] = f[5][Ny, 1:Nx1] + 0.5 * (f[1][Ny, 1:Nx1] - f[3][Ny,
1:Nx1]) -\
                      (v0[1:Nx1] / 6 + u0[1:Nx1] / 2) * rho_north[1:Nx1]
    f[8][Ny, 0:Nx] = f[6][Ny, 0:Nx] - 0.5 * (f[1][Ny, 0:Nx] - f[3][Ny, 0:Nx])
-\
                     (v0[0:Nx] / 6 + u0[0:Nx] / 2) * rho_north[0:Nx]

    f[6][0, 1:Nx] = f[6][1, 1:Nx] * solids[0, 1:Nx]
    f[2][0, :] = f[2][1, :] * solids[0, :]
    f[5][0, 1:Nx] = f[5][1, 1:Nx] * solids[0, 1:Nx]
    return f
```

Введение фиктивной силы тяжести

$$f_i(x + c_i\Delta t, t + \Delta t) - f_i(x, t) = \Omega_{TRT}(f) + \Delta t g_i$$

$$\text{и } g_i = \omega_i \rho \frac{g c_{iy}}{c_s^2}$$

```
def force_term(f_post, rho):
    f_g = f_post
    for k in range(1, Q):
        f_g[k] = f_g[k] + gi(rho, k)
    return f_g


def gi(rho, k):
    g = 1.102 * 10**(-3)
    force = w[k] * rho * g * cy[k] / cs_2
    return force
```

Этап распространения молекул

```
def streaming():
    for k in range(1, Q):
        for jd in range(Ny1):
            for id in range(Nx1):
                jc = jd - cy[k]
                ic = id - cx[k]
                if 0 <= jc <= Ny:
                    if ic == Nx + 1:
                        ic = 0
                    eps_jdid = epsilon[jd, id]
                    eps_jcic = epsilon[jc, ic]
```

```
                    f[rc[k]][jd, id] = eps_jcic * f_post[k][jd][id] + (1 -
eps_jdid) * f_post[rc[k]][jc, ic]

    return f
```

Расчет макропараметров

Определим макроскопические параметры: скорость потока жидкости в каждой точке решетки Больцмана и плотность ρ жидкости по формулам $\rho = \sum_i f_i$ и $u = \frac{1}{\rho}\sum_i c_i f_i$:

```
def den_vel():
    rho = np.array([[0.0] * Nx1] * Ny1)
    ux1 = np.array([[0.0] * Nx1] * Ny1)
    uy1 = np.array([[0.0] * Nx1] * Ny1)

    for k in range(Q):
        rho += f[k]
        ux1 += cx[k] * f[k]
        uy1 += cy[k] * f[k]

    for j in range(Ny1):
        for i in range(Nx1):
            if rho[j, i] != 0:
                ux1[j, i] = ux1[j, i] / rho[j, i]
                uy1[j, i] = uy1[j, i] / rho[j, i]
            else:
                ux1[j, i] = 0
                uy1[j, i] = 0
    return ux1, uy1, rho
```

Расчет коэффициента проницаемости

Определим коэффициент проницаемости и падение давления Δp по формулам (permeability_calc.py):

$$k = K_l\,(\delta x)^2, \quad K_l = -\nu_l \frac{(u_{avg})_l}{\nabla p_l}$$

$$\Delta p_l = \Delta \rho_l c_s^2, \quad \Delta p = \Delta p_l \cdot \rho^{FR} \left(\frac{\delta x}{\delta t}\right)^2$$

```
def non_zero_average(vector):
    vec2 = vector * vector
    vect_non_zero = (vec2 > 0.0) * 1.0
    count_non_zero = np.sum(vect_non_zero)
    return np.sum(vector) / count_non_zero


def new_Kl(rho, start, end):
    delta_rho = non_zero_average(rho[end, :]) - non_zero_average(rho[start,
:])
    delta_p = cs_2 * delta_rho
    k = -viscosity * non_zero_average(uy[end, :]) / delta_p
    return k


def calc_k(rho):
    Kl_dinamics = np.array([0.0] * Ny)
```

```
    for i in range(Ny, 0, -1):
        Kl_dinamics[i - 1] = new_Kl(rho, i, i - 1)

    K_dinamics = Kl_dinamics * dx * dx
    K_dinamics_out = K_dinamics / Darcy

    summa = sum(K_dinamics_out) / len(K_dinamics_out)
    return summa
```

Файлы визуализации

visual_laws_stability.py – контроль законов сохранения массы и импульса.

visual_vel_field.py – визуальное представление поля скоростей жидкости.

Алгоритм работы программы

Приведем алгоритм, в соответствии с которым разрабатывалась модель нахождения коэффициента проницаемости порового пространства (рис. 2):

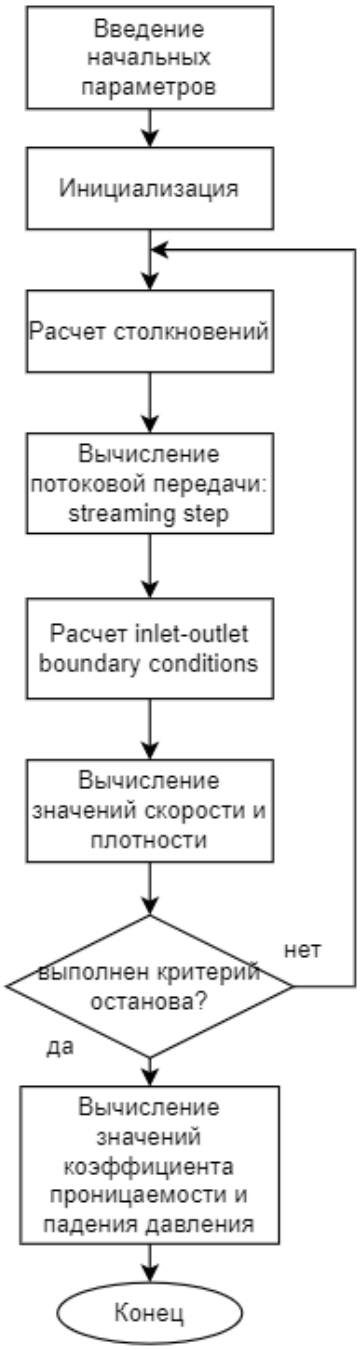

Рис. 2 – Алгоритм работы программы