Simulation of Two-Phase Flow in Porous Media using a Two-Dimensional Network Model

Shabbir K.*

March 15, 2023

1 Introduction

The algorithms and methods used to simulate two-phase flow in porous media has many practical applications in oil recovery, hydrology, electricity production where pressurized water is passed through heated pipes and is transformed into steam, etc. Our algorithm presented here is used to find the saturation of a phase with respect to time and model imbibition.

2 Volumetric Flow Rate through a cylinder

Viscous forces act according to:

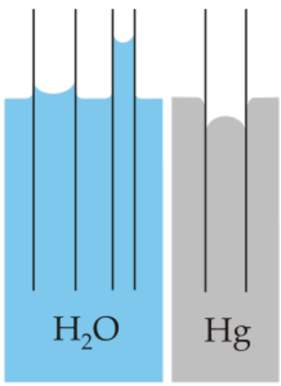
$$\frac{F}{A} = \mu \left| \frac{dv}{dr} \right|$$

Where F is the total shearing force on a surface of the layer, this force acts parallel to the surface plane, A is the area of the surface, μ is the coefficient of viscosity, v velocity of the flow parallel to the plane, and r is the coordinates perpendicular to the plane.

For a cylinder, the volumetric flow rate is given by

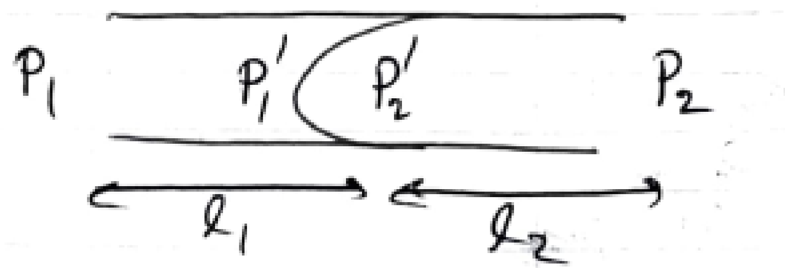
$$\boxed{Q = \frac{\pi}{8\mu} \frac{\Delta P}{l} R^4} \tag{1}$$

3 Flow rate in a tube containing meniscus



*Moscow Institute of Physics and Technology

Figure showing capillary action of water (polar) compared to mercury (non-polar), with respect to a polar surface such as glass (Si–OH). Let us apply this to our case, where the first node is filled with a fluid like water and the second node is filled with a fluid like air, and the our tube is similar to glass. Hence the meniscus will be oriented in a manner shown below.



Let there be a higher pressure in P_1 than P_2, the fluid in node which has P_1 produces a meniscus whose tends move towards the second node. We can break it down into two separate tubes of lengths l_1 and l_2, containing fluid of viscosites μ_1 and μ_2. Then the flow rates for each of the tubes are given by:

$$Q_1 = \frac{\pi}{8\mu_1} \frac{P_1 - P_1^{'}}{l_1} R_1^4 \tag{2}$$

$$Q_2 = \frac{\pi}{8\mu_2} \frac{P_2^{'} - P_2}{l_2} R_2^4 \tag{3}$$

Multiplying equations 2 and 3 by $\mu_i l_i$

$$Q_1 \mu_1 l_1 = \frac{\pi}{8}(P_1 - P_1^{'}) R_1^4 \tag{4}$$

$$Q_2 \mu_2 l_2 = \frac{\pi}{8}(P_2^{'} - P_2) R_2^4 \tag{5}$$

Due to continuity, which means no vacuum or fluid can be created, $Q_1 = Q_2$. Since it is the same tube, $R_1 = R_2$. Adding equation 4 and 5, we get:

$$Q(\mu_1 l_1 + \mu_2 l_2) = \frac{\pi}{8} R^4 (P_1 - P_2 + P_2^{'} - P_1^{'}) \tag{6}$$

In figure 2 the water rises because there is a pressure jump at the meniscus, the pressure is lower on the side of the water. Therefore in our case $P_2^{'} - P_1^{'}$ will have a positive value. Equation 6 becomes:

$$Q = \frac{\pi R^4}{8(\mu_1 l_1 + \mu_2 l_2)} (\Delta P + \frac{2\sigma}{R}) \tag{7}$$

Let the node on which we are generating linear equations be N_i and the node connected by a tube be N_j, if the concave side of the meniscus points towards N_j from N_i, then let us say that the meniscus 'points away from N_i' or simply 'points away' and in the case of opposite orientation 'points towards'. Let the sign due to the orientation of meniscus be decided by a function called $sigmns(ort, n)$, where ort is the orientation, and n is the number of meniscus in the tube:

$$sigmns(ort, n) = \begin{cases} -1, & \text{ort points towards, n = 1} \\ 0, & \text{n = 0, 2} \\ +1, & \text{ort points away, n = 1} \end{cases} \tag{8}$$

Finally we get:

$$Q_{ij} = \frac{\pi R^3}{8(\mu_1 l_1 + \mu_2 l_2)} (R \Delta P_{ij} + 2 sigmns(ort, n) \sigma) \tag{9}$$

It can be extended to the case when there are more than 1 meniscus, and using s_{ij} for $sigmns_{ij}(ort, n)$:

$$\boxed{Q_{ij} = \frac{\pi R_{ij}^3}{8lM_{ij}}(R_{ij}\Delta P_{ij} + 2s_{ij}\sigma)} \tag{10}$$

Here

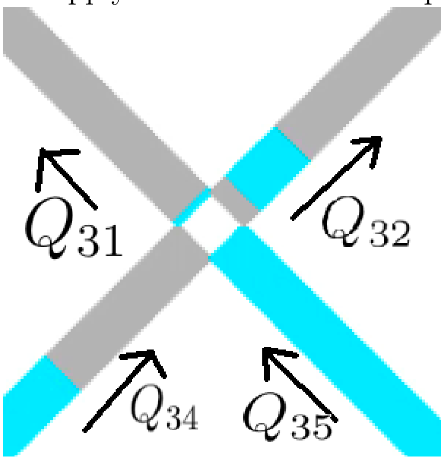
$$M_{ij} = \sum_k \mu_{ijk}\frac{l_{ijk}}{l}$$

Q_{ij} is the flow from N_i to N_j and $\Delta P_{ij} = P_i - P_j$. In case, when the flow is in the opposite direction Q_{ij} will take the opposite sign.

$$Q_{ij} = -Q_{ji} \tag{11}$$

4 The linear equations

Let us apply our method on a simple system consisting of only 5 nodes.



Since there are 4 tubes, we can write 4 equations according to 10

$$Q_{31} = \frac{\pi R_{31}^3}{8lM_{11}}(R_{31}\Delta P_{31} + 2s_{31}\sigma)$$

$$Q_{32} = \frac{\pi R_{32}^3}{8lM_{12}}(R_{32}\Delta P_{32} + 2s_{32}\sigma)$$

$$Q_{34} = \frac{\pi R_{34}^3}{8lM_{14}}(R_{34}\Delta P_{34} + 2s_{34}\sigma)$$

$$Q_{35} = \frac{\pi R_{35}^3}{8lM_{15}}(R_{35}\Delta P_{35} + 2s_{35}\sigma)$$

Since the sum of all Q_{ij} is equal to zero. Then as we iterate for each tube connected to the node, in our case we have 4 tubes, it is sufficient to do the following three operations. Here let $K_{ij} = R_{ij}^3/M_{ij}$.

$$[P_i] + R_{ij}K_{ij}$$

$$[P_j] - R_{ij}K_{ij}$$

$$[const] - 2s_{ij}\sigma K_{ij}$$

The matrix for Gaussian elimination will be

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & P_{up} \\
0 & 1 & 0 & 0 & 0 & P_{up} \\
-R_{31}K_{31} & -R_{32}K_{32} & (R_{3k}K_{3k} + ...) & -R_{34}K_{34} & -R_{35}K_{35} & -2\sigma(s_{3k}K_{3k} + ...) \\
0 & 0 & 0 & 1 & 0 & P_{down} \\
0 & 0 & 0 & 0 & 1 & P_{down}
\end{pmatrix}
$$

It can be proven that this matrix always has a solution. Once the solution is determined the flow rate can be calculated using equation 10, and the velocity of flow in each tube is given by

$$
\boxed{v_{ij} = \frac{R_{ij}}{8lM_{ij}}(R_{ij}\Delta P_{ij} + 2s_{ij}\sigma)} \tag{12}
$$

5 Description of the Model

The algorithms and methods used to simulate two-phase flow in porous media has many practical applications in oil recovery, hydrology, electricity production where pressurized water is passed through heated pipes and is transformed into steam, etc. Our algorithm presented here is used to find the saturation of a phase with respect to time, the hysteresis curve when the pressure across the porous body is reversed, total capillary pressure as a function of saturation[4], and determination of permeability which appears in Darcy's law.

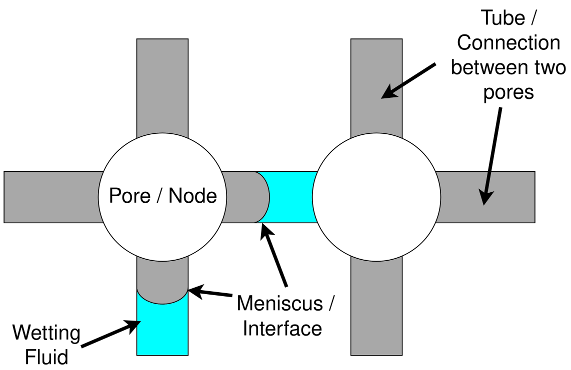


Figure showing two nodes from the network where the size of the node is much larger than the radius such that the capillary force tends to zero when the meniscus enters a node.

6 Algorithm steps

1. **Input Files:** read input files, radius.txt and mns.txt, mns.txt contains the initial setup of the meniscus

2. **Random radius:** add very small random values to the radius, this is done in order to remove the case of two equal radius for simplicity, can be removed later

3. **Loop time:** do until a certain proportion of invading fluid is reached for example 0.90:

   (a) **Pressure:** determine the pressure at each node using the linear equations given in section 3.

   (b) **Velocity:** Calculate the velocity using equation 12

(c) **time step:** determine the time step, the time step is calculated such that, it is the minimum of the time taken for a meniscus to reach a node it is heading towards. It is calculated by iterating through all the tubes, for a tube the time is determined using the $t_{ij} = x_{ij}/v_{ij}$, here x_{ij} is the distance between the node and and the meniscus closest to it, if the fluid is traveling upwards then it is the node located on the top of the tube, if the velocity is downwards then the node which is at the bottom is used. In case there is no meniscus present, $x_{ij} = l_{ij}/4$ is used, it is because a meniscus can enter the tube during the time step and this meniscus must not reach the next node, it can happen if the velocity is high and the radius is small. Also if $x_{ij} > l_{ij}/4$, then $x_{ij} = l_{ij}/4$ is used. This is done in order to smoothen the integration. In the video $l_{ij}/4$ is used, the number can be increased.

(d) **volume:** The volume displaced in each tube is determined by iterating through all the tubes, $V_{ij} = v_{ij} * t_{min}$.

(e) **integration:**

   i. **Store insertion:** create a matrix to store how much of which fluid to insert in each of these tubes.

   ii. **Loop nodes:** Iterate through all the nodes, and for each of the nodes.

      A. divide the tubes into two categories, flow-in-tube - here the fluid from these tubes flow into the nodes, flow-out-tubes here we insert the fluid into the tube from the node

      B. Find out the total of fluid1, fluid2, which is the total of each fliud from all flow-in-tubes.

      C. Start filling the each of the flow-out-tubes where the flow will go into in ascending order of the radius of the tube. This will be done simply be adding the quantity of fluid1 and fluid2 to the matrix created above.

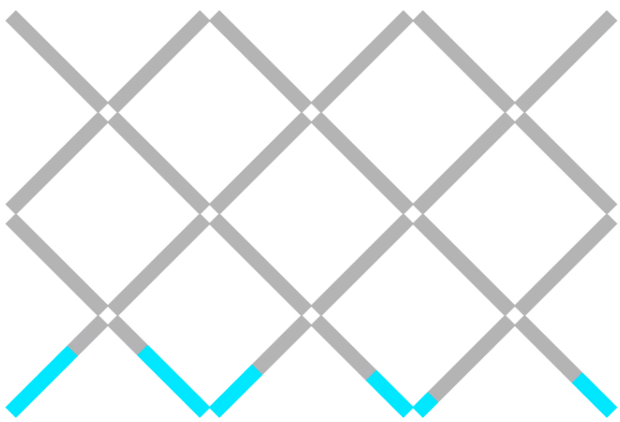
      D. while filling fist use fluid1, once fluid1 is used up then start using fluid2, which means if in a tube we have to insert two fluids, then fluid1 will go in first.

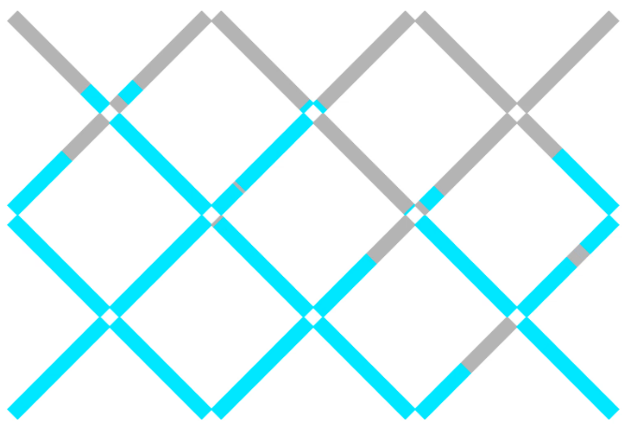
   iii. **Fluid addition:** For each of the tubes, add the volume of fluid determined to be added. After addition if there are more than 2 meniscus, then merge them retaining their center of masses.

(f) **Picture:** Save a picture of the current configuration.

4. Video: Make a video file from the pictures.

7 Results





Our model is initially set up such that the wetting fluid is low in saturation and is confined to the bottom of our network. A higher pressure is fixed for all nodes at the bottom layer, while a low pressure is fixed for the top row. In all nodes, law of conservation of volume is applied, since mass is conserved and the phases are non-compressible. However for the bottom layer of nodes, the wetting fluid is injected as much required according to the sum of flow rates determined in the tubes connected to those nodes, while from the top layer of nodes a fluid is removed.

8 Conclusion

This algorithm can be extended to the case where there are more than 4 tube connections to a node, since for two phase flow into a node case, we distribute in an ascending order of radii, in our model it is distributed to a maximum number to two tubes, but for hexagonal model it can be 4. We only need to update the function which produces the connections. The same model can be used for a 3-dimensional case[2], where one surface has higher pressure than the opposite surface which has a lower pressure, it is to be used in order to more accurately represent the porous body.

9 References

1. Aker, E., Måløy, K.J., Hansen, A., Batrouni, G.G. A two-dimensional network simulator for two-phase flow in porous media // Transp. Porous Med. 1998 V. 32 P. 163

2. Raoof A., Hassanizadeh S. A new method for generating pore-network models of porous media // Transp. Porous Med. 2010. V. 81. P. 391

3. S. Sinha et al. Effective rheology of two-phase flow in three-dimensional porous media: experiment and simulation // Transp. Porous Med. 2017. V. 119. P. 77

4. Fatt I. The network model of porous media 3, dynamic properties of networks with tube radius distribution Petroleum Trans. AIME 1956. V. 207. P. 164

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
#include <utility>
#include <limits>
#include <list>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include "drw.h"

// Type Definitions
typedef float real; //float can be changed to double or long double to increase the precision, float is used to make the program faster
class Cmns; //Class of Meniscus
typedef std::vector<std::vector<real>> Treal; //Table of real numbers
typedef std::vector<std::vector<Cmns>> Tmns;

//GENERAL CONSTANTS
const real PI = std::acos(-1);
const real HUGE = std::numeric_limits<real>::max();

//Physical Characterstics of the experiment
const real PRESSURE_BOTTOM = 100;
const real PRESSURE_TOP = 2;
const real SIGMA = 0; //7.56e-2; // FOR water at 20C in SI units, produces 75Pa pressure difference for tube of radius 1mm
const real TUBE_LENGTH = 1;//0.1;
const real MU1 = 1;//1e-3; // viscosity of the invading liquid: water
const real MU2 = 1;//1e-5; // viscosity of defending liquid: air

//Parameters of simulation
const real MAX_WETTING_PROPORTION = 0.98;
const real THRESHOLD_FILL = 0.001; //if any meniscus is smaller than this proportion, then it is destroyed
const real TIME_DIV = 4; // if the nearest meniscus by time is further, then L / TIME_DIV is prefered
```

```cpp
const int IMAGE_SIZE = 1000;
const real FINE_RADIUS_RANDOMNESS = 1e4;

//Input Output of File names
const std::string FILE_NAME_RADIUS = "radius.txt";
const std::string FILE_NAME_MNS = "fill.txt";
const std::string FOLDER_SAVE_PIC = "pic/";


class Cmns
{
    /* FUNCTION DESCRIPTION - scontb
     *
     * when we write the equation, about the volume
     * q = dV/dt = PI / 8 / MU * r ^ 4 / L * [(P[i] - P[j]) + 2 * SIGMA / r)
     * If there is high pressure and mescible fluid in the [i],
     * a positive sign before 2 * SIGMA / r means
     * the pressure difference is made higher by the interface meniscus
     * 1) blue: 0
     * 2) grey: 0
     * 3) blue|grey: +1
     * 4) grey|blue: -1
     * 5) blue|grey|blue: 0
     * 6) grey|blue|grey: 0
     *
     * sign going down means direction is 2 or 3,
     * for which type 0 means the meniscus is oriented away from [i],
     * giving a negative contribution of cappilary pressure
     */

    static real _scontb_sig(bool condition) //scontb's sign function
    {
        return condition ? -1 : 1;
    }

    /* generate positions long version
     *
     * the generated std::vector<real> can be of 3 types:
     * 1) [0, 1]
```

```
     * 2) [0, pos1, 1]
     * 3) [0, pos1, pos2, 1]
     *
     * it is distinguished from the short version which is of the form
     * 1) [], n = 0
     * 2) [a], n = 1
     * 3) [a, b], n = 2
     */

    std::vector<real> gen_pos_long_after_dspl(real vel, real l) const
    {
        auto pos_long_after_dslp = gen_pos_long();
        if(vel < 0)
        {
            pos_long_after_dslp.front() = l;
        }
        else
        {
            pos_long_after_dslp.back() -= l;
        }

        return pos_long_after_dslp;
    }

    //generate compartments of the configuration which exists
    typedef std::list<std::pair<bool, real>> Ccmprt;
    Ccmprt gen_cmprt_existing(real vel, real l) const
    {
        const auto pos_long_after_dspl = gen_pos_long_after_dspl(vel, l);
        Ccmprt cmprt_existing;
        for(int i = 1; i < pos_long_after_dspl.size(); ++ i)
        {
            cmprt_existing.push_back({(i + type - 1) % 2, pos_long_after_dspl[i] - pos_long_after_dspl[i - 1]});
        }

        return cmprt_existing;
    }

    static Ccmprt merge_existing_and_new_cmprts(Ccmprt& cmprt_existing,
```

```cpp
    const Ccmprt& cmprt_new, real vel)
    {
        if(vel > 0)
        {
            cmprt_existing.insert(cmprt_existing.begin(), cmprt_new.crbegin(), cmprt_new.crend());
        }
        else
        {
            cmprt_existing.insert(cmprt_existing.end(), cmprt_new.begin(), cmprt_new.end());
        }
        return cmprt_existing;
    }

    struct CmnsAfterDspl
    {
        bool type;
        std::vector<real> v;
    };

    static std::vector<real> gen_pos_from_segmented(std::vector<real> pos_segmented)
    {
        pos_segmented.pop_back();
        for(int i = 1; i < pos_segmented.size(); ++ i)
        {
            pos_segmented[i] += pos_segmented[i - 1];
        }

        return pos_segmented;
    }

    CmnsAfterDspl gen_pos_new_and_type(const Ccmprt& cmprt_new, const real threshold_fill) const
    {
        std::vector<std::pair<int, real>> cmprt_new_temp_vector;
        for(const auto& x: cmprt_new) // Step-1 Filter out anything smaller than threshold_fill
        {
```

```
            if(x.second >= threshold_fill)
            {
                cmprt_new_temp_vector.push_back({x.first, x.second});
            }
        }

        for(int i = 1; i < cmprt_new_temp_vector.size(); ++ i) // Step-2 Merge any two compartments of the smae fluid type
        {
            if(cmprt_new_temp_vector[i - 1].first == cmprt_new_temp_vector[i].first)
            {
                cmprt_new_temp_vector[i - 1].first = -1;
                cmprt_new_temp_vector[i].second += cmprt_new_temp_vector[i - 1].second;
            }
        }

        int type_begin_temp = -1;
        std::vector<real> pos_segmented;
        for(const auto& x: cmprt_new_temp_vector)
        {
            if(x.first != -1)
            {
                if(type_begin_temp == -1)
                {
                    type_begin_temp = x.first;
                }
                pos_segmented.push_back(x.second);
            }
        }

        const bool type_begin = type_begin_temp;
        const auto pos_new = gen_pos_from_segmented(pos_segmented);

        if(pos_new.size() < 3) // Depending on the number of meniscus, recombinate or preenet as it is
        {
            return {type_begin, pos_new};
        }
```

```
        if(pos_new.size() == 3)
        {
            const real l1 = 0;
            const real l2 = pos_new[0];
            const real l3 = pos_new[1];
            const real l4 = pos_new[2];

            const real d1 = l2 - l1;
            const real d2 = l4 - l3;
            const real d = d1 + d2;
            const real c1 = (l1 + l2) / 2;
            const real c2 = (l3 + l4) / 2;

            const real L1 = (c1 * d1 + c2 * d2) / d - d / 2;
            const real L2 = L1 + d;

            return {!type_begin, {L1, L2}};
        }
        if(pos_new.size() == 4)
        {
            const real l1 = pos_new[0];
            const real l2 = pos_new[1];
            const real l3 = pos_new[2];
            const real l4 = pos_new[3];

            const real d1 = l2 - l1;
            const real d2 = l4 - l3;
            const real d = d1 + d2;
            const real c1 = (l1 + l2) / 2;
            const real c2 = (l3 + l4) / 2;

            const real L1 = (c1 * d1 + c2 * d2) / d - d / 2;
            const real L2 = L1 + d;

            return {type_begin, {L1, L2}};
        }

        std::cout << "ER3-oversized decompartalization" << std::endl;
        return {type_begin, {-1, -1}};
    }
```

```

public:
    int n; //number of meniscus present
    bool type; // 0 - corresponds to blue fluid - which is invading
    std::vector<real> pos; // positions of meniscus

    Cmns(): n(0), type(1), pos(2) {} //by default everything is the defending fluid
    Cmns(int n, bool type, real p1, real p2): n(n), type(type), pos{p1, p2} {}

    real mu(const real mu1, const real mu2) const
    {
        std::vector<real> muv{mu1, mu2};
        const auto pos_long = gen_pos_long();

        real sum = 0;
        for(int i = 1; i < pos_long.size(); ++ i)
        {
            sum += muv[(i - 1 + type) % muv.size()] * (pos_long[i] - pos_long[i - 1]);
        }

        return sum;
    }

    real time(const real velocity, const real length, const real time_div) const
    {
        if(n == 0)
        {
            return length / velocity / time_div;
        }

        real dspl = (velocity >= 0 ? (1 - pos[n - 1]): pos.front());
        dspl = std::min(1.0f / time_div, dspl);
        return length * dspl / velocity;
    }

    void update(const real vel, const real r, const std::vector<real>&
```

```
    add, const real threshold_fill)
    {
        const real area = PI * std::pow(r, 2);
        const real l1 = add.front() / area;
        const real l2 = add.back() / area;
        const real l = l1 + l2;

        auto cmprt_existing = gen_cmprt_existing(vel, l);
        Ccmprt cmprt_new{{0, l1}, {1, l2}};
        auto cmprt = merge_existing_and_new_cmprts(cmprt_existing, cmprt_new, vel);
        auto pos_new_and_type = gen_pos_new_and_type(cmprt, threshold_fill);
        n = pos_new_and_type.v.size();
        type = pos_new_and_type.type;
        pos = pos_new_and_type.v;
        pos.resize(2);
    }

    real scontb(int direction) const
    {
        return _scontb_sig(direction > 1) * _scontb_sig(type) * (n % 2);
    }

    /*      vel | [true]     | [false]    |
     * drec     | above(<2)  | below(>=2)|
     * [true]+1 | out-0      | in-1       |
     * [flase]-1| in-1       | out-0      |
     */

    bool is_flow_into_node(const int direction, const real velocity) const
    {
        return (direction < 2) ^ (velocity >= 0);
    }

    bool type_fluid_into_node(int direction) const
    {
        if(direction < 2) // if fluid is coming from the above
        {
            return type; // whatever is at the lowest part is what gets
```

```
                into the node
        }
        /*
         * What is on the top part?
         *
         * n    | type=0 | type=1 |
         * 0    | 0      | 1      |
         * 1    | 1      | 0      |
         * 2    | 0      | 1      |
         */

        return type ^ (n % 2);
    }

    real sum_type_first() const
    {
        const auto pos_long = gen_pos_long();
        real sum = 0;
        for(int i = 1 + type; i < pos_long.size(); i += 2)
        {
            sum += pos_long[i] - pos_long[i - 1];
        }

        return sum;
    }

    std::vector<real> gen_pos_long() const
    {
        std::vector<real> v(n + 2);
        for(int i = 0; i < n; ++ i)
        {
            v[i + 1] = pos[i];
        }
        v.back() = 1;

        return v;
    }
};

std::ifstream& operator>> (std::ifstream& fin, Cmns& val)
```

```
{
    fin >> val.n >> val.type >> val.pos.front() >> val.pos.back();
    return fin;
}

std::ofstream& operator<< (std::ofstream& fout, const Cmns& val)
{
    fout << '\n' << val.n << ' ' << val.type << ' ' << val.pos.front() << ' ' << val.pos.back();
    return fout;
}

struct Coordinate
{
    real x;
    real y;
};

template <class T>
class FTable
{
public:
    int nrows;
    int ncols;
    std::vector<std::vector<T>> v;

    FTable() = default;
    FTable(int nrows, int ncols, const T& val = T()): nrows(nrows), ncols(ncols), v(nrows, std::vector<T>(ncols, val)) {}

    bool read(const std::string& file_name)
    {
        std::ifstream fin(file_name);
        if(!(fin >> nrows >> ncols))
        {
            std::cout << "-ER2-" << file_name << " is corrupted!" << '\n';
            return false;
        }

        std::vector<T> w;
```

```cpp
        T val;

        while(fin >> val)
        {
            w.push_back(val);
        }

        if(nrows * ncols != w.size())
        {
            std::cout << "-ER2-" << file_name << " has incorrect diamensions." << '\n';
            return false;
        }

        v.resize(nrows, std::vector<T>(ncols));
        for(int i = 0; i < w.size(); ++ i)
        {
            v[i / ncols][i % ncols] = w[i];
        }

        return true;
    }

    Coordinate _coordinate (int row, int col) const
    {
        return {0.5f + col, -0.5f + nrows - row};
    }

    void write(const std::string& file_name) const
    {
        std::ofstream fout(file_name);
        fout << nrows << ' ' << ncols << "\n\n";
        for(const auto& row: v)
        {
            for(const auto& val: row)
            {
                fout << val << ' ';
            }

            fout << '\n';
```

```cpp
        }
    }

    bool between(real x, real a, real b) const
    {
        return x >= a && x <= b;
    }

    bool inside(real x1, real x2, real y1, real y2, const Coordinate& coordinate) const
    {
        return between(coordinate.x, x1, x2) && between(coordinate.y, y1, y2);
    }

    void update(real x1, real x2, real y1, real y2, const T& val)
    {
        real xmin = std::min(x1, x2);
        real xmax = std::max(x1, x2);

        real ymin = std::min(y1, y2);
        real ymax = std::max(y1, y2);
        for(int i = 0; i < nrows; ++ i)
        {
            for(int j = 0; j < ncols; ++ j)
            {
                if(inside(xmin, xmax, ymin, ymax, _coordinate(i, j)))
                {
                    v[i][j] = val;
                }
            }
        }
    }

    void print() const
    {
        for(const auto& row: v)
        {
            for(const auto& val: row)
            {
```

```
                    std::cout << val << ' ';
                }
                std::cout << '\n';
            }
        }
};

typedef FTable<real> FTradius;
typedef FTable<Cmns> FTmns;

bool FCheckValidity()
{
    FTradius radius;
    FTmns mns;

    bool validity = true;
    if(radius.read(FILE_NAME_RADIUS))
    {
        std::cout << "-FDK-" << FILE_NAME_RADIUS << " is valid" << '\n';
    }
    else
    {
        validity = false;
    }

    if(mns.read(FILE_NAME_MNS))
    {
        std::cout << "-FDK-" << FILE_NAME_MNS << " is valid, " << '\n';
    }
    else
    {
        validity = false;
    }

    if(validity)
    {
        if((radius.nrows == mns.nrows) && (radius.ncols == mns.ncols))
        {
            std::cout << "-FDK-" << "diamensions of " << 
            FILE_NAME_RADIUS << " and " << FILE_NAME_MNS << " match" << 
```

```
            '\n';
        }
        else
        {
            std::cout << "-ERR-" << "diamensions of " << FILE_NAME_RADIUS << " and " << FILE_NAME_MNS << " do not match!" << '\n';
            validity = false;
        }
    }

    return validity;
}

void FPrintValidityStatus()
{
    if(FCheckValidity())
    {
        std::cout << "-FDK-" << FILE_NAME_RADIUS << ", " << FILE_NAME_MNS << " is okay" << '\n';
    }
    else
    {
        std::cout << "-ERR-" << std::string(30, '!') << '\n';
    }
}

Treal FReadFileRadius()
{
    FTradius radius;
    radius.read(FILE_NAME_RADIUS);

    for(auto& row: radius.v)
    {
        for(auto& cell: row)
        {
            cell += (rand() % 100) / FINE_RADIUS_RANDOMNESS;
        }
    }
    return radius.v;
```

```
}

Tmns FReadFileFill()
{
    FTmns mns;
    mns.read(FILE_NAME_MNS);
    return mns.v;
}

int FLinearLocNode(int i, int j, int m)
{
    return (i * (m + 1) + (i % 2)) / 2 + j;
}

std::pair<int, int> FConnectionEnds(int r, int c, int m)
{
    return {FLinearLocNode(r, c / 2 + (c % 2) * ((r + 1) % 2), m),
    FLinearLocNode(r + 1, c / 2 + (c % 2) * (r % 2), m)};
}

int FTotalNodes(int n, int m)
{
    return ((n + 1) * (m + 1) + 1) / 2;
}

struct Connections
{
    bool a = true;
    int r;
    int c;
    int p;
};

std::vector<Connections> FGenConnectionsEqu(int r, int c, int n, int m)
{
    const auto p = FLinearLocNode(r, c, m);
    std::vector<Connections> v
    {
        {true, r - 1, 2 * c - 1 + r % 2, p - m / 2 - 1},
        {true, r - 1, 2 * c - 0 + r % 2, p - m / 2 - 0},
```

```
        {true, r - 0, 2 * c - 0 + r % 2, p + m / 2 + 1},
        {true, r - 0, 2 * c - 1 + r % 2, p + m / 2 + 0}
    };

    if(r % 2)
    {
        return v;
    }

    if(r == 0)
    {
        v[0].a = false;
        v[1].a = false;
    }
    if(c == 0)
    {
        v[0].a = false;
        v[3].a = false;
    }
    if(2 * c == m)
    {
        v[1].a = false;
        v[2].a = false;
    }
    if(r == n)
    {
        v[2].a = false;
        v[3].a = false;
    }

    return v;
}

std::vector<real> FGaussElimination(Treal M)
{
    //std::cout << "okay-gauss gaussian eleimination" << std::endl;
    const int n = M.front().size() - 1;
    for(int i = 0; i < n; ++ i)
    {
        real divider = M[i][i];
```

```cpp
        for(int j = 0; j <= n; ++ j)
        {
            M[i][j] /= divider;
        }

        for(int j = 0; j < n; ++ j)
        {
            if(i == j)
            {
                continue;
            }

            real coeff = M[j][i];

            for(int k = 0; k <= n; ++ k)
            {
                M[j][k] -= M[i][k] * coeff;
            }
        }
    }
    std::vector<real> v;

    for(auto& row: M)
    {
        v.push_back(row.back());
    }

    return v;
}

void TFPrintMatrix(const std::string& s, const Treal& matrix)
{
    std::cout << "\n\n------------------------------------\n";
    std::cout << s << '\n';

    std::cout << std::setw(7) << -1 << " | ";
    for(int j = 0; j < matrix.front().size(); ++ j)
    {
        std::cout << std::setw(7) << (float)j << ' ';
    }
```

```
    std::cout << '\n';
    for(int i = 0; i < matrix.size(); ++ i)
    {
        std::cout << std::setw(7) << (float)i << " | ";
        for(int j = 0; j < matrix.front().size(); ++ j)
        {
            std::cout << std::setw(7) << matrix[i][j] << ' ';
        }
        std::cout << '\n';
    }
}

void TFPrintMatrix(const std::string& s, const std::vector<float>& v, const int n, const int m)
{
    std::cout << "\n\n-----------------------------------\n";
    std::cout << s << '\n';

    int count = 0;
    for(int i = 0; i <= n; ++ i)
    {
        int mt = m / 2 - i % 2;
        for(int j = 0; j <= mt; ++ j)
        {
            std::cout << std::setw(7) << v[count++] << ' ';
        }
        std::cout << '\n';
    }
}

Treal FGenEquForGauss(const Treal& radius, const Tmns& mns)
{
    //std::cout << "okay-FGenEquForGauss" << std::endl;
    const int n = radius.size();
    const int m = radius.front().size();
    const int total_nodes = FTotalNodes(n, m);
    Treal equation(total_nodes, std::vector<real>(total_nodes + 1));

    //std::cout << "okay-FGenEquForGauss" << std::endl;
    //std::cout << "total_nodes=" << total_nodes << std::endl;
```

```cpp

    for(int i = 0; i <= n; ++ i)
    {
        int mt = m / 2 - (i % 2);
        //std::cout << "n= " << n << ", m=" << m << ", mt=" << mt << std::endl;
        for(int j = 0; j <= mt; ++ j)
        {

            //std::cout << "i=" << i << ", j=" << j << std::endl;

            const int l = FLinearLocNode(i, j, m);
            auto& e = equation[l];
            if(i == 0)
            {
                e[l] = 1;
                e.back() = PRESSURE_TOP;
                continue;
            }
            if(i == n)
            {
                e[l] = 1;
                e.back() = PRESSURE_BOTTOM;
                continue;
            }

            //derection: 0-topleft, 1-topright, 2-bottomright, 3-bottomleft
            const auto connections = FGenConnectionsEqu(i, j, n, m);

            for(int i = 0; i < connections.size(); ++ i)
            {
                const auto& c = connections[i];
                //std::cout << "connection, a=" << c.a << " c=" << c.c << ", r=" << c.r << ", p=" << c.p << std::endl;
                if(c.a)
                {
                    const auto& r = radius[c.r][c.c];
                    const auto& f = mns[c.r][c.c];
                    const auto& s = f.scontb(i);
```

```cpp

                    const real K = std::pow(r, 3) / f.mu(MU1, MU2);
                    e[l] += r * K;
                    e[c.p] -= r * K;
                    e.back() -= SIGMA * 2 * s * K;
                }
            }
        }
    }

    //std::cout << "okay-FGenEquForGauss" << std::endl;
    //TFPrintMatrix("Gauss", equation);
    return equation;
}

std::vector<real> FCalcPressure(const Treal& radius, const Tmns& mns)
{
    //std::cout << "okay-gauss Fclac pres" << std::endl;
    return FGaussElimination(FGenEquForGauss(radius, mns));
}



Treal FCalcVelocity(const std::vector<real>& pressure, const Treal& radius, const Tmns& mns)
{
    const int n = radius.size();
    const int m = radius.front().size();
    auto velocity = radius;
    for(int i = 0; i < n; ++ i)
    {
        for(int j = 0; j < m; ++ j)
        {
            const auto locs = FConnectionEnds(i, j, m);
            const auto delp = pressure[locs.second] - pressure[locs.first];
            const auto& r = radius[i][j];
            const auto& mu = mns[i][j].mu(MU1, MU2);
            const auto& s = mns[i][j].scontb(0);
            velocity[i][j] = r / 8 / mu / TUBE_LENGTH * (delp * r + s *
```

```
                2 * SIGMA);
        }
    }

    return velocity;
}

Treal FCalcVolume(Treal velocity, const Treal& radius, const real time_step)
{
    for(int i = 0; i < velocity.size(); ++ i)
    {
        auto& v = velocity[i];
        for(int j = 0; j < v.size(); ++ j)
        {
            v[j] = std::abs(v[j] * std::pow(radius[i][j], 2) * PI * time_step);
        }
    }

    return velocity;
}

real FDetermineTimeStep(const Tmns& mns, const Treal& velocity)
{
    real min_time = HUGE;
    for(int i = 0; i < mns.size(); ++ i)
    {
        for(int j = 0; j < mns[i].size(); ++ j)
        {

            const real temp_time = mns[i][j].time(velocity[i][j], TUBE_LENGTH, TIME_DIV);
            min_time = std::min(temp_time, min_time);
        }
    }

    return min_time;
}

```

```
struct IntegrationResult
{
    Tmns mns;
    real fluid1in;
    real fluid1out;
    real fluid2in;
    real fluid2out;
};

int FCountConnections(const std::vector<Connections>& connections)
{
    int count = 0;

    for(const auto& connection: connections)
    {
        count += connection.a;
    }

    return count;
}

struct TubeWhereFlowOut
{
    real radius;
    int r;
    int c;
};

bool Fcomparison_outflow(const TubeWhereFlowOut& first, const TubeWhereFlowOut& second)
{
    return first.radius < second.radius;
}

std::vector<real> FAmountVolumeToBePushedIn(real volume, std::vector<real>& tank)
{
    auto v = tank;
    v.front() = std::min(tank.front(), volume);
    v.back() = volume - v.front();
```

```cpp

    for(int i = 0; i < tank.size(); ++ i)
    {
        tank[i] -= v[i];
    }

    return v;
}


Tmns FCombineFillAndAdditions(Tmns mns, const Treal& velocity, const Treal& radius, const std::vector<std::vector<std::vector<real>>>& additions)
{
    for(int i = 0; i < mns.size(); ++ i)
    {
        auto& f = mns[i];
        for(int j = 0; j < f.size(); ++ j)
        {
            f[j].update(velocity[i][j], radius[i][j], additions[i][j], THRESHOLD_FILL);
        }
    }

    return mns;
}

Tmns FPerformIntegration(const Tmns& mns, const Treal& volume, const Treal& velocity, const Treal& radius)
{
    const int n = volume.size();
    const int m = volume.front().size();

    real fluid1in = 0;
    real fluid1out = 0;
    real fluid2in = 0;
    real fluid2out = 0;

    std::vector<std::vector<std::vector<real>>> additions(n, std::vector<std::vector<real>>(m));
```

```

    for(int i = 0; i <= n; ++ i)
    {
        int mt = m / 2 - i % 2;
        for(int j = 0; j <= mt; ++ j)
        {
            //std::cout << "Performing integration i=" << i << ", j=" << j << std::endl;
            const auto connections = FGenConnectionsEqu(i, j, n, m);

            /*
            for(const auto& connection: connections)
            {
                std::cout << "connection, a=" << connection.a << " c=" << connection.c << ", r=" << connection.r << ", p=" << connection.p << std::endl;
            }
            */
            std::vector<real> vol_in(2);
            std::vector<TubeWhereFlowOut> tubes_flow_out;
            for(int direction = 0; direction < connections.size(); ++ direction)
            {
                const auto& c = connections[direction];
                if(c.a)
                {
                    const auto& f = mns[c.r][c.c];
                    const auto& vel = velocity[c.r][c.c];
                    const auto& vol = volume[c.r][c.c];
                    const auto& r = radius[c.r][c.c];
                    if(f.is_flow_into_node(direction, vel))
                    {
                        vol_in[f.type_fluid_into_node(direction)] += vol;
                    }
                    else
                    {
                        tubes_flow_out.push_back({r, c.r, c.c});
                    }
                }
            }
```

```
            //for(const auto& tpshin: tubes_flow_out)   std::cout << "tube_push_out before short: radius=" << tpshin.radius << ", r=" << tpshin.r << ", c=" << tpshin.c << std::endl;

            //std::cout << "second stage reached!" << std::endl;
            if(i == 0)
            {
                fluid1out += vol_in.front();
                fluid2out += vol_in.back();
                continue;
            }
            if(i == n) // NOTE might remove else
            {
                for(const auto& tpshin: tubes_flow_out)
                {
                    additions[tpshin.r][tpshin.c] = {volume[tpshin.r][tpshin.c], 0};
                }
                continue;

            }

            std::sort(tubes_flow_out.begin(), tubes_flow_out.end(), *Fcomparison_outflow);
            for(const auto& tpshin: tubes_flow_out)
            {
                //std::cout << "tube_push_out after sort: radius=" << tpshin.radius << ", r=" << tpshin.r << ", c=" << tpshin.c << std::endl;

                additions[tpshin.r][tpshin.c] = FAmountVolumeToBePushedIn(volume[tpshin.r][tpshin.c], vol_in);
            }
        }
    }
    //std::cout << "-------FCombineFillAndAdditions" << std::endl;
    return FCombineFillAndAdditions(mns, velocity, radius, additions);
}
```

```

//Tested works Correctly
void FPlot(Tmns mns, Treal radius, real clock, int count)
{
    std::reverse(mns.begin(), mns.end());
    std::reverse(radius.begin(), radius.end());

    real max_radius = -1;
    real min_radius = 1e12;

    for(const auto& x: radius)
    {
        for(auto y: x)
        {
            max_radius = std::max(max_radius, y);
            min_radius = std::min(min_radius, y);
        }
    }

    const int image_size = IMAGE_SIZE;
    const int length = mns.front().size();
    const int height = mns.size();

    const int effective_length = image_size / (std::max(length, height)
    + 2);

    drw::bmp a(image_size, image_size);

    const int start_y = effective_length;
    const int start_x = effective_length;
    const real max_thick = effective_length;
    const real min_thick = effective_length / 6.0;


    int y = start_y;
    for(int row = 0; row < mns.size(); ++ row)
    {
        const auto& w = mns[row];
        int x = start_x + effective_length * (row % 2);
        for(int col = 0; col < w.size(); ++ col)
```

```
        {
            int sign = (1 - 2 * (row % 2)) * (1 - 2 * (col % 2));
            const real r = radius[row][col];
            real thick = min_thick;
            if(max_radius != min_radius)
            {
                thick  += (r - min_radius) * (max_thick - min_thick) / (max_radius - min_radius);
            }
            a.drawVector(x, y, effective_length, thick, sign, w[col].n, w[col].pos, w[col].type);
            x += 2 * effective_length * (sign > 0);
        }

        y += effective_length;
    }

    //a.save(FOLDER_SAVE_PIC + "pic-" + std::to_string(count) + "_t-" + std::to_string(clock) + ".bmp");
    a.save(FOLDER_SAVE_PIC + "pic-" + std::to_string(count) + ".bmp");
}

void FPlotWithoutRadius(Tmns mns, int count)
{
    std::reverse(mns.begin(), mns.end());

    const int image_size = IMAGE_SIZE;
    const int n_cols = mns.front().size();
    const int n_rows = mns.size();

    const int length = image_size / (std::max(n_rows, n_cols) + 2);

    drw::bmp a(image_size, image_size);

    const int start_y = length;
    const int start_x = length;
    const int thick = length / 10;


    int y = start_y;
```

```cpp
    for(int row = 0; row < mns.size(); ++ row)
    {
        const auto& w = mns[row];
        int x = start_x + length * (row % 2);
        for(int col = 0; col < w.size(); ++ col)
        {
            int sign = ((row % 2) ^ (col % 2) ? -1 : 1);
            a.drawStrip(x, y, length, thick, sign, w[col].gen_pos_long(), w[col].type);
            x += 2 * length * (sign > 0);
        }
        y += length;
    }

    //a.save(FOLDER_SAVE_PIC + "pic-" + std::to_string(count) + "_t-" + std::to_string(clock) + ".bmp");
    a.save(FOLDER_SAVE_PIC + "stp-" + std::to_string(count) + ".bmp");
}

/*
class Diamension
{
public:
    int m;
    int n;

    Diamension(int number_cols, int number_rows): m(number_cols), n(number_rows) {}
    Diamension(const Treal& table): m(table.front().size()), n(table.size()) {}

    std::pair<int, int> FConnectionEnds(int r, int c, int m)
    {
        return {FLinearLocNode(r, c/2 + 1 - (r % 2), m), FLinearLocNode(r + 1, c/2 + (r % 2), m)};
    }

    int FTotalNodes(int n, int m)
    {
        return ((n + 1) * (m + 1) + 1) / 2;
```

```
    }

    int FLinearLocNode(int i, int j, int m)
    {
        return (i * (m + 1) + (i % 2)) / 2 + j;
    }
};
*/



real FMeasureWettingFluidProportion(const Tmns& mns, const Treal& radius)
{
    real total = 0;
    real type_first = 0;
    for(int i = 0; i < radius.size(); ++ i)
    {
        for(int j = 0; j < radius[i].size(); ++ j)
        {
            const real rsq = std::pow(radius[i][j], 2);
            type_first += mns[i][j].sum_type_first() * rsq;
            total += rsq;
        }
    }

    return type_first / total;
}



void FSimulate(const Treal& radius, Tmns& mns)
{
    TFPrintMatrix("radius", radius);

    real clock = 0;
    int count = 10000;

    FPlot(mns, radius, clock, count);
    FPlotWithoutRadius(mns, count);

```

```cpp
    real wetting_fluid_proportion;
    while((wetting_fluid_proportion = FMeasureWettingFluidProportion(mns, radius)) <= MAX_WETTING_PROPORTION)
    {
        std::cout << "PRS-" << count << ", clock=" << clock << ", proportion=" << wetting_fluid_proportion << std::endl;
        const auto pressure = FCalcPressure(radius, mns);
        //TFPrintMatrix("pressure", pressure, radius.size(), radius.front().size());

        const auto velocity = FCalcVelocity(pressure, radius, mns);
        TFPrintMatrix("velocity", velocity);

        const auto time_step = FDetermineTimeStep(mns, velocity);
        const auto volume = FCalcVolume(velocity, radius, time_step);

        TFPrintMatrix("volume", volume);

        mns = FPerformIntegration(mns, volume, velocity, radius);
        clock += time_step;
        ++ count;
        FPlot(mns, radius, clock, count);
        FPlotWithoutRadius(mns, count);
    }
}
//ffmpeg -framerate 10 -i filename-%03d.jpg output.mp4
int main()
{
    std::srand((unsigned)std::time(nullptr));
    FPrintValidityStatus();
    const auto radius = FReadFileRadius();
    auto mns = FReadFileFill();

    std::cout << std::fixed << std::setprecision(4);

    FSimulate(radius, mns);
    return 0;
}

```