Project 2

FYS4460 - Disordered systems and percolation

Anders Johansson
14th April 2018

a)

I reused the argon script from project 1 but changed the size of the system and the parameter of the lattice command. According to the project description, the lattice spacing should be $a = 5.72$ Å $= 1.68\sigma$, giving a reduced density (the parameter of the `lattice` command) of

$$\rho^* = \rho\sigma^3 = \frac{4}{(a/\sigma)^3} = \frac{4}{1.68^3} = 0.84. \tag{1}$$

b)

The system was first thermalised for a certain number of timesteps, determined by looking a the visualisations in the previous exercise. A cylinder was then created with the given dimensions, and the atoms inside were selected as a group. The remaining atoms were then given zero velocity for visualisation purposes and excluded from the integration. Visual inspection in Ovito shows that this approach works.

c)

I considered many options for making spheres with random positions and radii and ended up using the `python` command in LAMMPs. This runs a Python function during the execution of the LAMMPs script. The function receives a pointer to a LAMMPs object, which it uses to extract variables and run commands.

The important part of the LAMMPs script is

```
fix 1 all nve
run 1000 # thermalise

python make_spheres input 1 SELF format p file make_spheres.py
python make_spheres invoke

group moving subtract all matrix
velocity matrix set 0 0 0
unfix 1

fix 1 moving nve
run 2000
```

The first `python` command defines the function `make_spheres`, found in the `file make_spheres.py`. This function takes `1` argument, `SELF`, which is a pointer (`p`) to the current LAMMPs object. The second `python` command runs the Python function. The `matrix` group is defined by the Python function as the atoms inside the spheres. The content of `make_spheres.py` is the simple function

```
def make_spheres(lmpptr):
    import traceback
    try:
        from lammps import lammps
        import numpy as np

        num_spheres = 20
```

```
        lmp = lammps(ptr=lmpptr)
        sigma = lmp.extract_variable("sigma", "all", 0)
        rmin = 20 / sigma
        rmax = 30 / sigma

        lx = lmp.get_thermo("lx")
        ly = lmp.get_thermo("ly")
        lz = lmp.get_thermo("lz")

        xs = np.random.uniform(0, lx, size=num_spheres)
        ys = np.random.uniform(0, ly, size=num_spheres)
        zs = np.random.uniform(0, lz, size=num_spheres)
        rs = np.random.uniform(rmin, rmax, size=num_spheres)

        for sphere in range(num_spheres):
            for i in [-1, 0, 1]:
                for j in [-1, 0, 1]:
                    for k in [-1, 0, 1]:
                        x = xs[sphere] + i * lx
                        y = ys[sphere] + j * ly
                        z = zs[sphere] + k * lz
                        r = rs[sphere]
                        cmds = [
                            "region mysphere sphere %g %g %g %g units box" %
                            (x, y, z, r), "group matrix region mysphere",
                            "region mysphere delete"
                        ]
                        lmp.commands_list(cmds)
    except Exception:
        traceback.print_exc()
        import sys
        sys.exit(1)
```

Periodic boundary conditions are fully enforced by creating all relevant periodic images of each sphere. Unfortunately, the approach of using the `python` command inside LAMMPs to run LAMMPS commands does not work well with MPI, so the simulation is run serially.

Visualisation of the result was done by colour coding the atoms according to the magnitude of their velocities, where the lower bound is zero and the upper bound is a very small number. See figure 1 on the following page.

d)

See the above exercise.

e)

Armed with the `python` command discovered in the previous exercise, I used the LAMMPs commands

```
variable num_moving equal count(moving)
print ${num_moving} file data/beforedeletion.dat

variable is_moving atom gmask(moving)
```

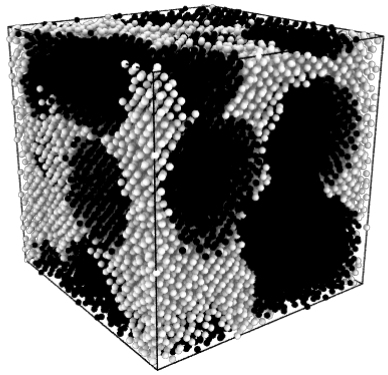

Figure 1: Visualisation of the randomly placed spheres.

```
python delete_half_of_moving input 1 SELF format p file delete_half_of_moving.py
python delete_half_of_moving invoke

print ${num_moving} file data/afterdeletion.dat
```

The variable `is_moving` is set to 1 for all atoms in the group `moving` and 0 for all atoms in the matrix. The Python function is

```
def delete_half_of_moving(lmpptr):
    import traceback
    try:
        from lammps import lammps
        import numpy as np

        lmp = lammps(ptr=lmpptr)
        N = lmp.get_natoms()

        def tointarray(x):
            return np.ctypeslib.as_array(x, shape=(N, ))

        ids = tointarray(lmp.extract_atom("id", 1))
        is_moving = tointarray(lmp.extract_variable("is_moving", "all", 1))

        moving_ids = ids[is_moving == 1]
        delete_ids = moving_ids[::2]

        cmd = "group deletethese id " + " ".join(map(str, delete_ids))
        lmp.command(cmd)

        cmd = "delete_atoms group deletethese"
        lmp.command(cmd)

    except Exception:
        traceback.print_exc()
        import sys
        sys.exit(1)
```

Halfway through writing this, I discovered that LAMMPs actually has built-in commands for deleting a certain fraction of the particles in a region, but at that point, there was no turning back. One difficulty of extracting values from `lammps` objects is the type of the values returned, which for arrays is a pointer to the underlying C++ structures. This must be converted using `numpy.ctypeslib.as_array`.

The files written by the `print` commands in the LAMMPs snippet above contain 12 127 and 6063, showing that the number of atoms not in the matrix is indeed halved when the Python function is executed. Figure 2 shows the time evolution of the temperature in the pores.

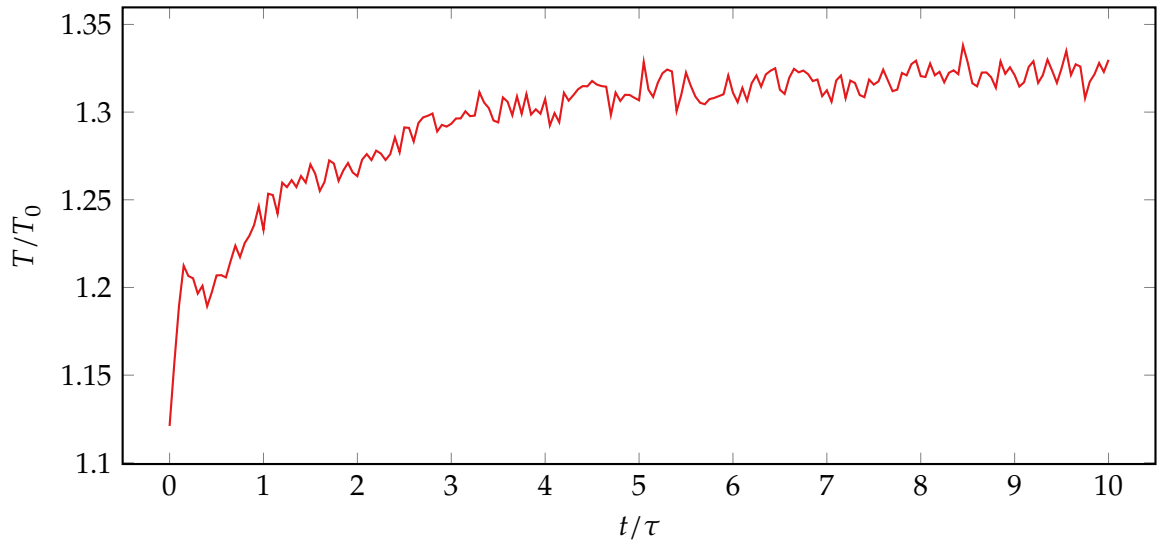

Figure 2: Temperature in the pores of the material as a function of time.

f)

The mean squared displacement of the atoms in the pores was calculated using the `compute msd` command in LAMMPs.

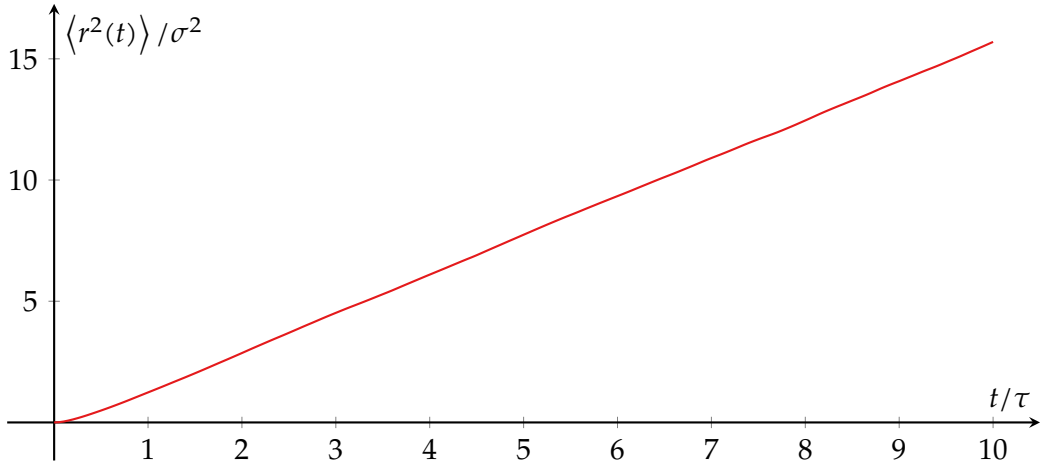

Figure 3: Mean squared displacement in the pores of the material as a function of time. As expected, the result is linear.

g)

Darcy's law is

$$U = \frac{k}{\mu}(\nabla P - \rho g), \tag{2}$$

where U is the volume flux density, k is the permeability, ∇P is the gradient of the external pressure (non-existent in this project), ρ is the mass density and g is the net acceleration due to externally applied forces. The applied force points in the x-direction, and F_x is the force applied to each atom. Newton's second law for a system of N atoms in a volume V with the force F_x applied to each is thus

$$NF_x = Nmg. \tag{3}$$

This can be divided by the volume, giving

$$\frac{Nm}{V}g = \rho g = \frac{N}{V}F_x = nF_x, \tag{4}$$

where n is the number density.

h)

(I choose to redo the analytical solution for the velocity field with an applied force instead of an applied pressure gradient for my own reference and understanding.)

I here assume that we are dealing with a Newtonian fluid, defined as a fluid in which the viscous forces due to a difference in the $\hat{e}_i$ component of the velocity creates a force in the $\hat{e}_i$ direction on a surface with normal $\hat{e}_j$ that is proportional to the derivative of the velocity when going in the $\hat{e}_j$ direction,

$$\sigma_{ij}\left(\equiv \frac{F_i}{A_j}\right) = \mu\frac{\partial v_i}{\partial x_j}. \tag{5}$$

The quantity μ is the viscosity, while σ_{ij} is the stress tensor. In a cylindrical pore orientated in the x direction with radius a, the velocity field should be independent of the angle with the x axis and only depend on the distance to the centre of the cylinder.

A small volume element in cylindrical coordinates has volume $r\,\mathrm{d}\phi\,\mathrm{d}r\,\mathrm{d}x$. It is acted upon by two forces in the x direction:

- The externally applied force. With the number density n, the number of particles in the volume element is $nr\,\mathrm{d}\phi\,\mathrm{d}r\,\mathrm{d}x$, giving a total force

$$F_{\text{ext}} = nF_x r\,\mathrm{d}\phi\,\mathrm{d}r\,\mathrm{d}x\,. \tag{6}$$

- Viscous forces due to the radial dependence of the velocity. According to equation (5), the force on the "outer" surface of the volume element, whose normal is directed radially outwards, is

$$F_{\text{outer}} = \mu\left.\frac{\partial v_x}{\partial r}\right|_{r=r+\mathrm{d}r}\overbrace{(r+\mathrm{d}r)\,\mathrm{d}\phi\,\mathrm{d}x}^{A_r}, \tag{7}$$

while the force on the "inner" surface, whose normal is directed radially inwards, is

$$F_{\text{inner}} = -\mu \left.\frac{\partial v_x}{\partial r}\right|_{r=r} r \,\mathrm{d}\phi \,\mathrm{d}x\,. \tag{8}$$

The minus sign is due to the normal of the surface being $-\hat{e}_r$. Without the minus, it would be the force from "our" volume element, $r \in [r, r + \mathrm{d}r]$, on the volume element with $r \in [r - \mathrm{d}r\,, r]$. Newton's third law says that the required correction is a minus sign. Due to the assumption of rotational invariance there are no other viscous forces.

In equilibrium, the sum of these forces must be zero from Newton's first law. This gives

$$\mu\left(\left.\frac{\partial v_x}{\partial r}\right|_{r+\mathrm{d}r} (r + \mathrm{d}r) - \left.\frac{\partial v_x}{\partial r}\right|_{r} r \right) \mathrm{d}\phi \,\mathrm{d}x + nF_x r \,\mathrm{d}\phi \,\mathrm{d}r \,\mathrm{d}x = 0. \tag{9}$$

I now divide by $\mathrm{d}\phi \,\mathrm{d}r \,\mathrm{d}x$, recognize the resulting first term as the derivative of $r\,\partial v_x/\partial r$ and move some terms, resulting in the differential equation

$$\frac{\partial}{\partial r}\left(r\frac{\partial v_x}{\partial r} \right) = -\frac{nF_x}{\mu} r. \tag{10}$$

Everything except r is constant on the right-hand side, so this equation can be integrated directly to yield the x component of the velocity as a function of r,

$$r\frac{\partial v_x}{\partial r} = -\frac{1}{2}\frac{nF_x}{\mu} r^2 + C \implies \frac{\partial v_x}{\partial r} = -\frac{nF_x}{2\mu} r + C/r \implies v_x(r) = -\frac{nF_x}{4\mu} r^2 + C \ln r + D. \tag{11}$$

The velocity should not be unproblematic when $r = 0$, i.e. in the centre of the cylinder, so C must be zero. The second integration constant, D, is determined by the choice of boundary conditions. A reasonable assumption for a solid-liquid interface is zero velocity at the boundary, which in this case is $r = a$. This implies $D = nF_x a^2/4\mu$, giving the velocity profile

$$v_x(r) = \frac{nF_x}{4\mu}\left(a^2 - r^2\right). \tag{12}$$

Comparing this to the result from the lectures shows that $\Delta P/L = \nabla P$ is replaced by $-nF_x$ when a force is applied instead of a pressure gradient.

Darcy's law contains the volume flux density, i.e. the amount of volume passing through a cross-section per time per area, $U = \mathrm{d}V/A\,\mathrm{d}t$. The infinitesimal volume must be $\mathrm{d}V = A\,\mathrm{d}x = A\bar{v}_x\,\mathrm{d}t$, where A is the area of the cross-section and $\bar{v}_x$ is the average velocity in the x-direction. Consequently $U = \bar{v}_x$. The mean velocity can be calculated as

$$U = \bar{v}_x = \frac{1}{A}\int v_x \,\mathrm{d}A = \frac{1}{\pi a^2}\int_0^a v(r) \cdot 2\pi r \,\mathrm{d}r \tag{13}$$

$$= \frac{nF_x}{2\mu a^2}\int_0^a \left(a^2 - r^2\right) r \,\mathrm{d}r = \frac{nF_x}{2\mu a^2}\left(\tfrac{1}{2}a^4 - \tfrac{1}{4}a^4\right) = \frac{nF_x a^2}{8\mu} = \frac{k}{\mu} nF_x, \tag{14}$$

which is just Darcy's law for an externally applied force, where $k = a^2/8$.

The viscosity can now be determined numerically by running simulations and comparing the resulting velocity profile (at equilibrium) to equation (12), while the permeability for an arbitrary geometry can be calculated from equation (14) with the viscosity found from a cylindrical pore.

I here choose to study the fluid flow of argon with half the density, using the method developed in exercise e) on page 2 to delete half of the moving atoms. The setup is otherwise as in the first exercises with a cylindrical pore.

The velocity of the centre of mass of a group can be calculated by LAMMPs. This is equal to the mean velocity of the atoms in the group. Figure 4 shows the time evolution, while figure 5 on the next page shows the radial distribution after the system has reached equilibrium.

From equation (12) on the preceding page, the radial distribution should be a linear function of $nF_x(a^2 - r^2)/4$ with slope $1/\mu$. I use this relation to approximate the viscosity, resulting in

$$\mu \approx 0.62\, \varepsilon\tau/\sigma^3. \tag{15}$$

According to the internet, this is only 50 % too large. Figure 5 on the next page shows the quality of the approximation. The number density is approximated by $2/b^3$, as there are initially 4 atoms per face-centred cubic unit cell with volume b^3. A few comments on the implementation:

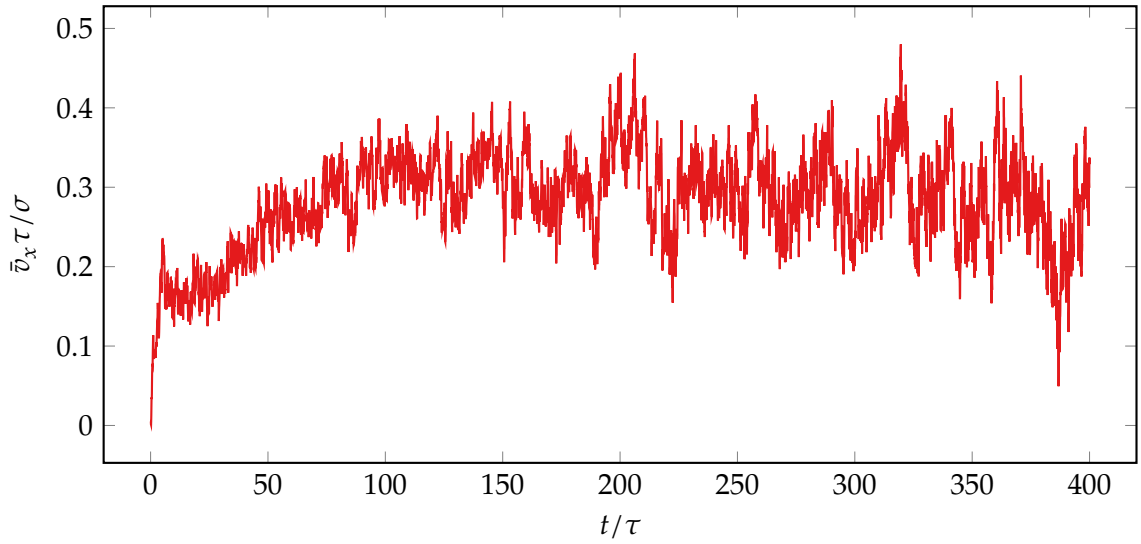

Figure 4: Mean velocity in the x-direction of atoms in the cylinder as a function of time. The system appears to reach equilibrium after approximately 100τ, or 20 000 timesteps.

- The radial distribution is made by finding all the x-components of the velocities of the atoms, binning them by the atom's distance to the centre of the cylinder, and taking the average velocity in each bin. Additionally, the result is averaged over time for all saved data sets with the system in equilibrium.
- Initially the radial bins had linearly spaced radii. This causes the number of atoms per bin to increase linearly with the bin number, which means that very few atoms are in the inner bins. As a result, the statistics were not very good. The solution was to distribute the radial bin edges as the square root of linearly spaced values from 0 to a^2, resulting in an approximately uniform distribution of particles.
- The radial bin averaging was done by `scipy.stats.binned_statistics`. Unfortunately, Scipy is not easily available inside the Ovitos interpreter, so the dump files had to be read manually. Fortunately, the LAMMPs dump format is very suitable for programmed reading.

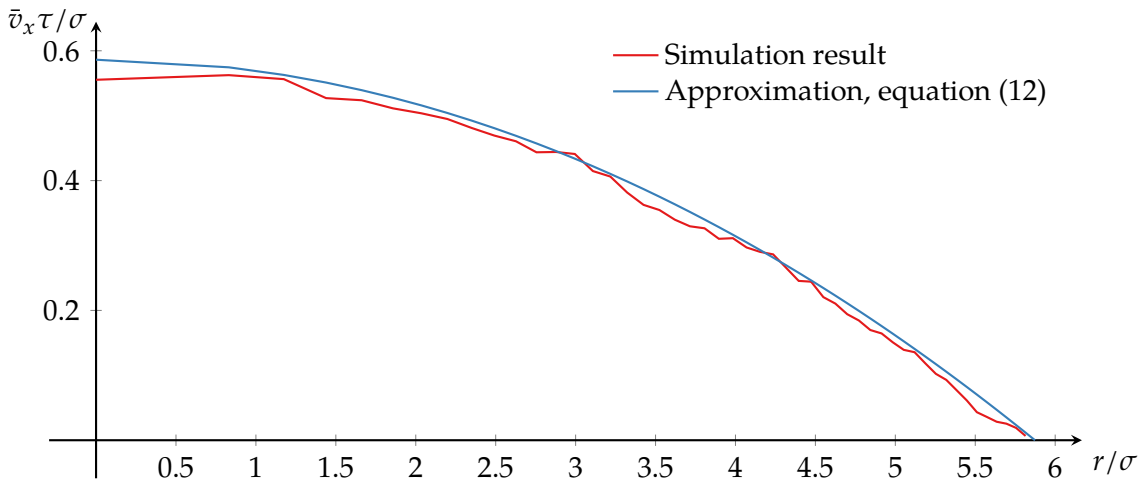

Figure 5: Radial distribution of the mean velocity in the x-direction of atoms in the cylinder in equilibrium. The result is an average over all dumped timesteps after the one determined by looking at figure 4 on the previous page.

i)

The permeability as a function of porosity can be found by using equation (14) on page 6 with the viscosity from equation (15) on the previous page for set of porosities. I here choose to vary the porosity simply by varying the number of spheres in the matrix.

A rough theoretical model for the permeability is[1]

$$k = \frac{a^2}{45}\frac{\phi^3}{(1-\phi)^2}, \tag{16}$$

which is compared with the simulation results in figure 6 on the following page. The fit is not very good, apart from a similar trend.

Since each simulation takes a non-trivial amount of time to run, I avoid redoing simulations with the following `make` rule.

```
data/porosity_%.dat: in.script make_spheres.py delete_half_of_moving.py
        lmp_mpi -var num_spheres $* -in in.script
```

A `Pool` object from `multiprocessing` is used to call a Python analysis function for each number of spheres in parallel. This function issues a `make` command, reads the resulting log file and calculates the permeability from equation (14) on page 6. The result is shown in figure 6 on the following page.

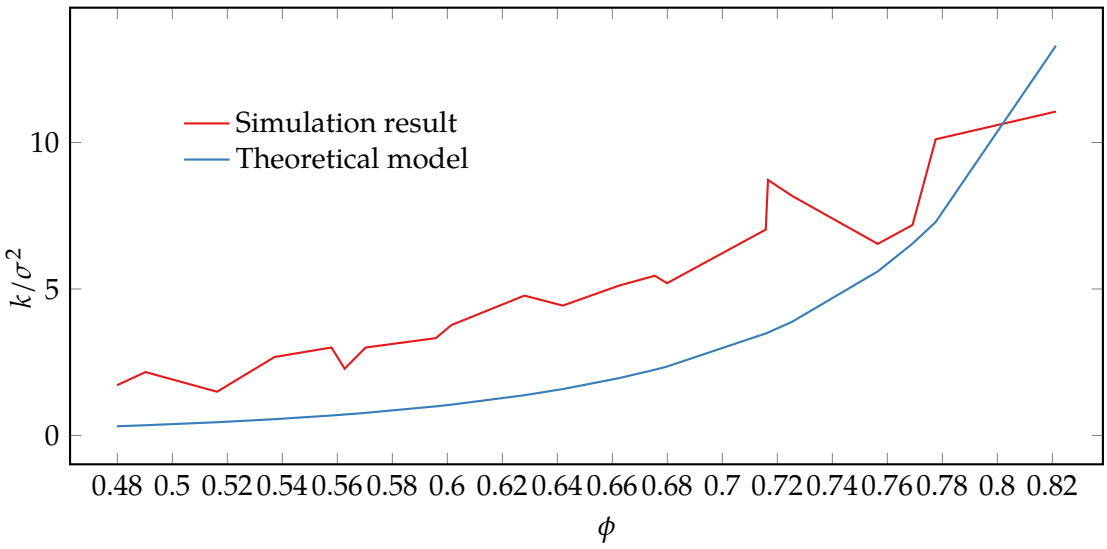

Figure 6: Permeability as a function of porosity in a system with spheres making up the matrix. Comparison with the theoretical model in equation (16) on the preceding page shows that the fit is not particularly good, but at least the trend is similar.

References

[1] Jens Feder. *Flow in Porous Media*. 2nd ed. Oslo, 1984. 270 pp. URL: https://www.uio.no/studier/emner/matnat/fys/FYS4460/v13/notes/flow.pdf (visited on 14/04/2018).