Water distribution network

Dr. Jean Auriol

September 2019

In this project[1] we solve the equations describing the steady state of a **drinking water distribution network**. This kind of problem naturally arises when considering the development of an urban water network, as the one described in Figure 1. The network which is described in this project is an extremely simplified version of such an urban water network. This problem

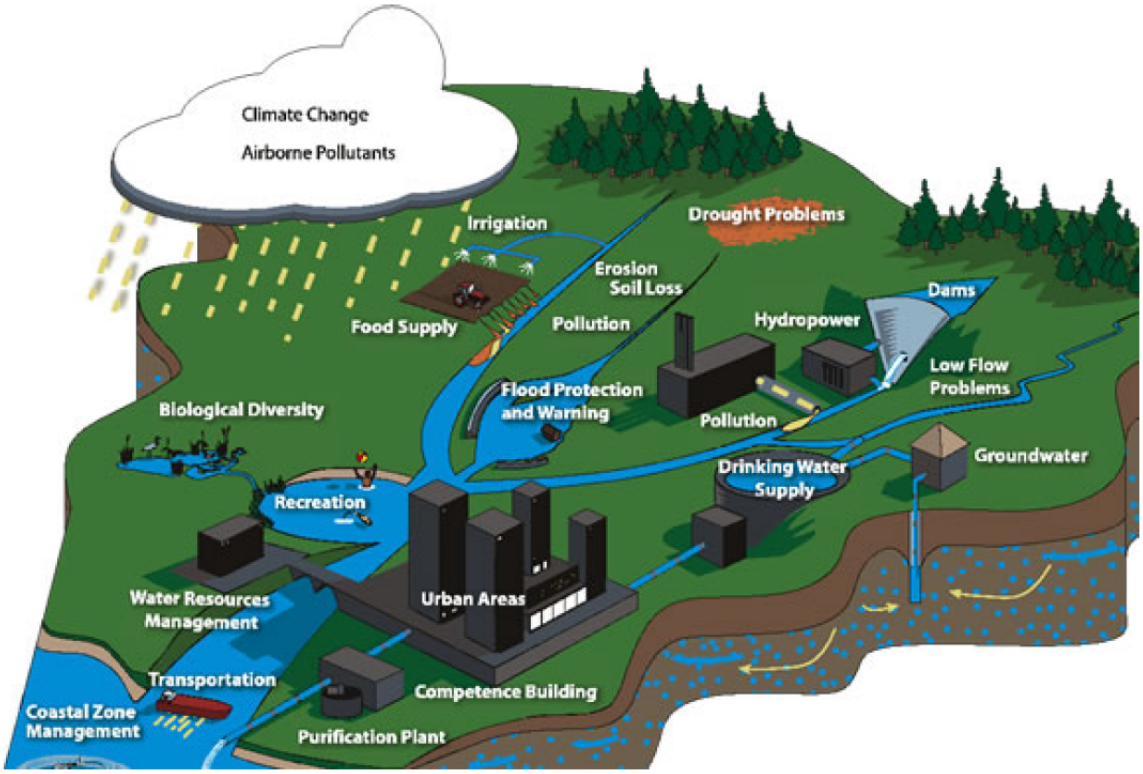


Figure 1: Example of an urban water network.

consists in determining the steady state of such a network. It can be rewritten as an **optimization problem** for which we want to minimize the energy of the network under some linear constraint (first Kirchhoff's law). Such an optimization problem can then be simplified into a simple unconstrained optimization problem. The objective of this project is to code the different optimization algorithms introduced during the lecture and to compare their performance. The simplified model under consideration is given in Section 1. It is given for curiosity purpose and you do not need to understand it completely. The optimization problem you will have to solve is described in Section 2.

[1] The content of this document is inspired from the one written by Pierre Carpentier for the optimization class in Mines ParisTech.

1 Presentation of the problem

A drinking water network can be described by an **oriented graph** $\mathcal{G}$ with n arcs and m nodes. Among the m nodes, there are m_r nodes that correspond to the water tank and m_d nodes that correspond to the nodes where the water is consumed. We have a total of $m_r + m_d = m$ nodes. We assume that the arcs (respectively the nodes) of the graph are numbered from 1 to n (respectively from 1 to m) and that the nodes that correspond to the reservoirs are the m_r first ones. We also assume that the graph is **connex**, which means that there exist at least one path between two arbitrary nodes of the graph. This implies

$$n \geq m - 1.$$

A schematic example of such a graph is given in Figure 2.

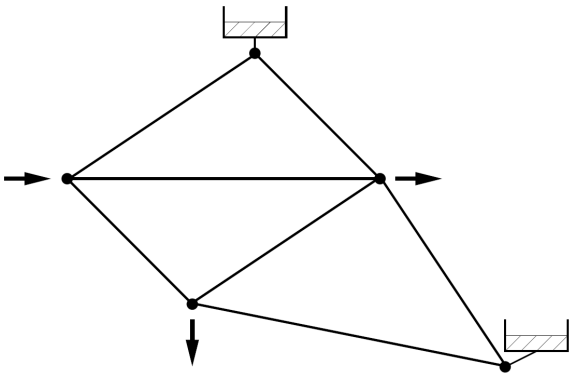


Figure 2: Example of a graph $\mathcal{G}$

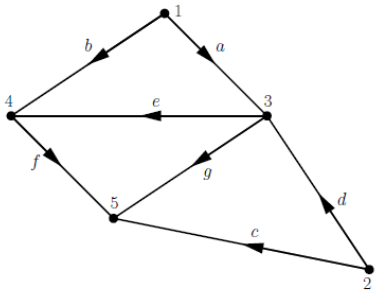
The topology of such a network can be described by an **incidence matrix** that will be denoted A. This matrix has m rows (the number of rows correspond to the number of nodes) and n columns (the number of columns correspond to the number of arcs). If we choose an arc α and a node i of the graph, the element $a_{i,\alpha}$ of the matrix A is defined by

$$a_{i,\alpha} = \begin{cases} -1 & \text{if } i \text{ is the original node of the arc } \alpha, \\ +1 & \text{if } i \text{ is the final node of the arc } \alpha, \\ 0 & \text{else.} \end{cases} \tag{1}$$

An example of such a matrix A for a simple graph is given in Figure 3.

We define the following notations

- f is the **flow vector for the nodes of the graph**; the values of the flow for the last m_d nodes are known (they correspond to the demand of the consumers expressed in $m^3 s^{-1}$). They are positive if the water is consumed and negative if the water is injected. The flow for the first m_r nodes has to be calculated. They correspond to the flows entering or exiting the different reservoirs.
- p is the **pressure vector for the nodes of the graph**; the values of the pressure for the first m_r nodes is known (they correspond to the height of the different water columns). The pressure for the other nodes has to be calculated.
- r is the **resistance vector for the arcs of the graph**; the values of the resistance are known. They depend of the length, the radius and the roughness of the pipes.



$$A = \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{pmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ +1 & 0 & 0 & +1 & -1 & 0 & -1 \\ 0 & +1 & 0 & 0 & +1 & -1 & 0 \\ 0 & 0 & +1 & 0 & 0 & +1 & +1 \end{pmatrix} \end{array} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}$$

Figure 3: Example of an incidence matrix A

- q is the **flow vector for the arcs of the graph**. It has to be computed.

The steady state of the network implies two set of equations

- The first set of equations expresses the fact that there cannot be any accumulation in the nodes of the graph (**first Kirchhof's law**). Thus, we have

$$Aq - f = 0. \tag{2}$$

- The second set of equations expresses the fact that the loss of charge along an arc derives from a potential (**second Kirchhof's law**) and that this loss of charge is a function of the flow in the arc (**non linear Ohm's law**). Using Colebrooks' formula and denoting $r \bullet q \bullet |q|$ the vector whose components are $r_\alpha q_\alpha |q_\alpha|$ (Hadamard's product), we obtain

$$A^T p + r \bullet q \bullet |q| = 0. \tag{3}$$

2 An optimization problem

2.1 Mathematical notations

We use the following definition for the Hadamard product: considering two vectors with n lines, v_1 and v_2, the Hadamard product $v_1 \bullet v_2$ is the vector whose i^{th} line is the product of the i^{th} components of v_1 and v_2. For instance, we have,

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} \bullet \begin{pmatrix} 2 \\ 5 \end{pmatrix} = \begin{pmatrix} 2 \\ 15 \end{pmatrix}.$$

We use the following definition for the scalar product: for two vectors $v_1 \in \mathbb{R}^n$ and $v_2 \in \mathbb{R}^n$, we have

$$\langle v_1, v_2 \rangle = \sum_{i=1}^{n} (v_1)_i (v_2)_i = v_1^T v_2 = v_2^T v_1.$$

2.2 Optimization problem

The objective of this project is to determine the steady state of the network. It can be proved (with lengthy computations) that this problem amounts to minimizing the energy of the network while respecting the first Kirchhoff law. More precisely, we are lead to the following optimization problem

$$\min_{q_c \in \mathbb{R}^{n-m_d}} \frac{1}{3} \langle q^{(0)} + Bq_c, r \bullet (q^0 + Bq_c) \bullet |(q^0 + Bq_c)| \rangle + \langle p_r, A_r(q^0 + Bq_c) \rangle. \tag{4}$$

where q_c is a vector $\in \mathbb{R}^{n-m_d}$ that corresponds to the last $n-m_d$ components of q, where B and A_r are known matrices that only depend on A, where q^0 is a vector $\in \mathbb{R}^n$ that depends on the system parameters (but not on q_c) and where p_r is a vector that only depends on p. The explicit expressions of all these parameters is not required in this project as they will be given numerically. More precisely, I have coded two scripts in Python called `Problem_R` and `Structures_R`. In the first script, I define the different parameters of the network (number of arcs, nodes, reservoirs), the structure of the graph, the resistance of the different arcs while the second script computes the incidence matrix and the other matrices required to compute the cost function. **You have to run these two scripts before running your own code**.

Let us now define the cost function F as

$$F : q_c \in \mathbb{R}^{n-m_d} \mapsto \frac{1}{3}\langle q^{(0)} + Bq_c, r \bullet (q^0 + Bq_c) \bullet |(q^0 + Bq_c)|\rangle + \langle p_r, A_r(q^0 + Bq_c)\rangle. \quad (5)$$

The problem (4) can be rewritten as

$$\min_{q_c \in \mathbb{R}^{n-m_d}} F(q_c). \quad (6)$$

It can be proved that

$$\nabla F(q_c) = B^T(r \bullet (q^0 + Bq_c) \bullet |(q^0 + Bq_c)|) + (A_r B)^T p_r, \quad (7)$$

$$\nabla^2 F(q_c) = 2B^T \begin{pmatrix} 2r_1(q^0 + Bq_c)_1 & 0 & \cdots & 0 \\ 0 & 2r_2(q^0 + Bq_c)_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 2r_n(q^0 + Bq_c)_n \end{pmatrix} B \quad (8)$$

If you feel mathematically confident, you can prove it, but do not waste your time on it.

3 Implementation

The objective of this project is to code the different optimization algorithms presented during the lecture to solve the problem (4). We will compare their performance in terms of **Number of iterations** and **computational time** (I recall that in Python, you need to use the command `process_time()` to get the CPU current time). I have already coded two scripts (the scripts `Problem_R` and `Structures_R` that you will have to run to get the numerical values of the different variables (do not change the notations)) and a function `Visualg` that allows you to plot the results of your optimization algorithms (this function plots the value of the cost function, the norm of the gradient and the step length for each iteration of your optimization algorithm). I have also partially coded some of the algorithms you will have to code in this project.

Question 1

Using Python, code an oracle `OraclePG`(q_c, ind) that returns the value of the function F and of its gradient $G = \nabla F$ for a given point q_c. More precisely, your function should return the vector $[F, G, ind]$ where

q_c : corresponds to the reduced flow vector.

F is the value of the criterion (the function we want to minimize) in q_c.

G is the value of the gradient of F in q_c.

ind is an integer that corresponds to the computation you wan to do: if $ind = 2$, you should only compute F (G is set to 0), $if ind = 3$, you should only compute G (F is set to 0) and if $ind = 4$, you should compute F and G. The objective is to avoid useless computations (it is not necessary to spend time computing the gradient if you don't use it).

In Python the Hadamard product is obtained using the command np.product.

Question 2

Write a gradient algorithm with a fixed step to solve the optimization problem (4). For each iteration, you should store the values of the function F and of $\log_{10}(G)$ (rather than G). Thus, you will be able to plot the performance of your algorithm. All the optimization algorithms you will code should return

q_opt : optimal solution.

F_opt : value of the criterion for the optimal point.

G_opt : value of the gradient for the optimal point.

F_tot : the vector where the values of the cost function are stored.

G_tot : the vector where the values of the gradient norm are stored.

G_step : the vector where the values of the gradient step are stored.

nb_iter : the number of iterations.

Do not forget to put a stop condition: the algorithm should stop if the norm of the difference between two consecutive iterations is less than a given tolerance. Moreover, to avoid an infinite loop we stop the algorithm is the number of iterations reaches a given threshold (10000 here). I have written the skeleton of the algorithm in the function `GradientFixedStep`.

Question 3

Test your algorithm with a fixed step of 0.0005 and a tolerance of 0.000001. The initial condition should be a vector with 9 rows whose component are chosen arbitrarily (namely 0). Compute the number of iterations and the computational time.

Question 4

Read the Fletcher's algorithm `Wolfe` that returns the optimal step that satisfies the Wolfe's conditions. Adjust your gradient algorithm to use a step that satisfies Wolfe's conditions. Test your new algorithm and compare the number of iterations, the computational time with the ones your previously obtained.

Question 5

Code the BFGS algorithm. Test your algorithm and compare the number of iterations, the computational time with the ones of the previously written algorithms.

Question 6

Code a new oracle `OraclePH`(q_c, ind) that now also returns the value of the Hessian.

Question 7

Code the Newton's algorithm with a constant step. Test your algorithm and compare the number of iterations, the computational time with the ones of the previously written algorithms. Change your algorithm to replace the constant step with a Wolfe's step that satisfies the Wolfe's conditions (using `Wolfe`)

Question 8

Compare your results (BFGS and Newton) with the ones obtained using Python optimization functions. To understand the syntax used in Python, run the file **TestPython-Rosenbrock** in which I present the different algorithms on the Rosenbrock's function $f : x = (x_1, x_2) \in \mathbb{R}^2 \mapsto 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. For instance, the BFGS algorithm can be implemented as follows

```
res = minimize(Fun, x0, method='BFGS', jac=Der, options='disp': True),
```

where $x0$ is the initial guess, Fun is the oracle that returns the value of the cost function and Der is the oracle that returns the value of the gradient of the cost function.