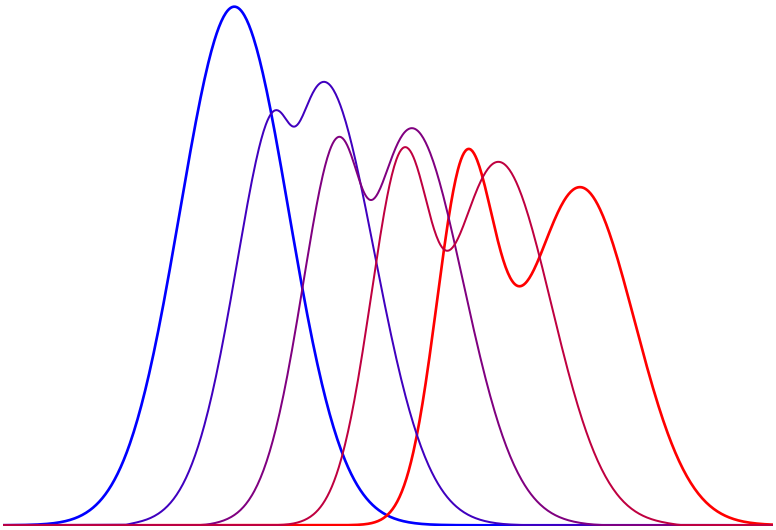
École des Ponts ParisTech
Centre d'Enseignement et de Recherche en Environnement Atmosphérique

Optimal transport problem solver

User guide

Alban Farchi



March 2016

Contents

1 Introduction

The transportation theory is the study of optimal transportation and allocation. The so-called optimal transport problem was first introduced by [4] and formalized by [3], leading to the Monge-Kantorovich transportation problem.

The goal is to look for a transport map transforming a probability density function into an other while minimizing the cost of transport.

1.1 Brief description of the problem

Let $\mathbb{E}$ be an euclidean space of dimension d and let f_0 and f_1 be two probability density functions over $\mathbb{E}$. We are looking for a couple density-velocity $(f, \mathbf{v})$ defined over $\mathbb{E} \times [0,1]$ satisfying a continuity constraint:

$$\forall (\mathbf{x}, t) \in \mathbb{E} \times [0,1], \quad \frac{\partial f}{\partial t}(\mathbf{x}, t) + \operatorname{div}(f \cdot \mathbf{v})(\mathbf{x}, t) = 0, \tag{1}$$

a boundary condition:

$$\forall \mathbf{x} \in \mathbb{E}, \quad f(\mathbf{x}, t = 0) = f_0 \quad \text{and} \quad f(\mathbf{x}, t = 1) = f_1, \tag{2}$$

a null-flux condition for mass conservation:

$$\forall (\mathbf{x}, t) \in \partial\mathbb{E} \times [0,1], \quad \mathbf{v}(\mathbf{x}, t) = 0, \tag{3}$$

and minimizing the cost of transport:

$$J(f, \mathbf{v}) = \int_{\mathbb{E} \times [0,1]} f(\mathbf{x}, t) \cdot \|\mathbf{v}(\mathbf{x}, t)\|^2 \, \mathrm{d}\mathbf{x}\mathrm{d}t. \tag{4}$$

The minimum of this cost function is defined as the Wasserstein distance between f_0 and f_1. The argument f^* of the minimum defines a non-trivial interpolation between f_0 and f_1. For more details on this problem, see [1, 6, 2].

1.2 About the solver

The optimal transport problem is a minimization problem under constraint. Following [5], we propose iterative solvers that rely on the use of proximal operators. To be able to numerically represent the fields, we have to assume that $\mathbb{E}$ is compact, and in particular we take $\mathbb{E} = [0,1]^d$. Moreover, to simplify the computation, we will work with the variables $(f, \mathbf{m})$, where $\mathbf{m} = f\mathbf{v}$ represents the momentum.

In the *python* module we implemented the Douglas-Rachford (DR) and the Primal-Dual (PD) algorithms in order to compute the density and momentum fields satisfying (1) and (2) and minimizing (4) given two tables representing discretized versions of the probability density functions f_0 and f_1. Our *python* code can handle the one-dimensional and two-dimensional cases (*i.e.* $d = 1$ or 2), is object-oriented, heavily relies on the standard *numpy* and *scipy* modules (*e.g.* linalg, fftpack, tensordot...) and is not parallelized. It is also able to solve the optimal transport problem with reservoir boundaries as presented in [2].

The code has been originally written for *python* 2.7. An other version is available for *python* 3.4 which only introduces minor modifications to match the new *python* 3.x syntax, however this version is slightly slower. This is due to the lower performance of the *numpy* and *scipy* modules in *python* 3.4. For more details about the code see [2].

2 Main objects defined

2.1 Discretization

In file `OT/OTObjects*D/grid/grid.py` are defined all the necessary classes to handle the discretized versions of the density and momentum fields on centered and staggered grids as defined in section 3.1 of [2] and in section 3 of [5].

2.2 Proximal operators

The proximal operators are defined in repertory `OT/OTObjects*D/proximals/`. Their computation mainly rely on the functions defined in the grid classes. For more details about the discretized versions of these operators, see section 4 of [5].

Four different proximal operators are defined and that are referred in the code as `proxJ`, `proxCdiv`, `proxCsc` and `proxCb`. In table 1 we describe the contraint applied for each proximal operator and the state space of the input and output variable, *i.e.* the grid on which the input and output variables must be defined to apply the proximal operator.

Proximal operator	Sub-repertory	Main constraint	State space of input and output
`proxJ`	`.`	Cost function of transport (4)	Centered grid
`proxCdiv`	`div/`	Continuity equation (1)	Staggered grid
`proxCsc`	`sc/`	Interpolation constraint	Couple staggered grid, centered grid
`proxCb`	`bound/`	Boundary conditions (2) and (3)	Staggered grid

Table 1: Details about the proximal operators.

For each constraint, there are variations of the projection operators that add (partial) boundary conditions to the main constraint. For example, in `div/`, `proxCdiv.py` defines the projection on the continuity constraint (1), `proxCdivb.py` defines the projection on the continuity constraint (1) with the boundary conditions (2) and (3) and `proxCdivtb.py` defines the projection on the continuity constraint (1) with temporal boundary conditions (2). See table 2 for more details.

Class	Boundary conditions	State space of input and output
Variations of the divergence constraint		
`ProxCdiv`	None	Staggered grid
`ProxCdivb`	(2) and (3)	Staggered grid
`ProxCdivtb`	(2)	Staggered grid
Variations of the interpolation constraint		
`ProxCsc`	None	Couple staggered grid, centered grid
`ProxCscb`	(2) and (3)	Couple staggered grid, centered grid
`ProxCscrb`	(2), only at $t = 0$ and (3)	Couple staggered grid, centered grid
`ProxCsctb`	(2)	Couple staggered grid, centered grid
Variations of the boundary conditions		
`ProxCb`	(2) and (3)	Staggered grid
`ProxCrb`	(2), only at $t = 0$ and (3)	Staggered grid
`ProxCtb`	(2)	Staggered grid

Table 2: Details about the variations of the proximal operators.

Choosing the correct version for the proximal operators is a critical step. See section 5.6 for more details about the way proximal operators are chosen.

2.3 Algorithms

The algorithms are implemented in repertory `OT/OTObjects*D/algorithms/`. When using this module to solve an optimal transport problem, there are two steps: one has to

construct an algorithm object, with the correct set of parameters, and then one just need to call the `run(self)` member of the object. The construction of the algorithm object requires a configuration object, which is in fact just a container object that contains the set of parameters.

Three algorithms are implemented: the PD algorithm in `pd/`, the DR algorithm for the minimization of two functionals in `adr/` and the DR algorithm for the minimization of three functionals in `adr3/`. More details about the algorithms are available in section 3.

2.4 Configuration

The class for the configuration object is defined in `OT/OTObjects*D/configuration.py`. It inherits from a default configuration class that handles the interface with a configuration file. It also defines a member `algorithm(self)` that constructs the algorithm for a given configuration object.

3 Algorithms

3.1 Primal-Dual algorithm

The PD algorithm allows one to minimize a functional $F = F_1 + F_2 \circ A$ where A is a linear operator and F_1 and F_2 are simple functions.

Here we apply this method by considering the minimization space to be the set of discretized couples $(f, \mathbf{m})$ defined on staggered grids. A is the linear operator that interpolates a couple $(f, \mathbf{m})$ defined on staggered grids into a couple $(f, \mathbf{m})$ defined on centered grids. It is implemented by the `interpolation(self)` method of class `StaggeredField`.

F_1 is the cost function related to the continuity constraint (1) and the boundary conditions (2) and (3) – therefore when using this algorithm, one needs to select the `ProxCdivb` version of the `proxCdiv` proximal operator. See section 5.6 for more details about the selection of the versions of the proximal operators.

F_2 is the cost function related to the cost of tranport (4).

One then defines three recurrent sequences u_n, y_n and v_n that are sequences of couples $(f, \mathbf{m})$ defined on staggered grids for u_n and y_n and on centered grids for v_n by:

$$v_{n+1} = \sigma \cdot \mathrm{Prox}_{F_2/\sigma}\left(\frac{1}{\sigma} \cdot v_n + A\left(y_n\right)\right), \tag{5}$$

$$u_{n+1} = \mathrm{Prox}_{\tau F_1}\left(u_n - \tau \cdot A\left(v_{n+1}\right)\right), \tag{6}$$

$$y_{n+1} = (1+\theta) \cdot u_{n+1} - \theta \cdot u_n. \tag{7}$$

If $\theta \in [0,1]$ and $\sigma\tau < 1$ then u_n converge towards the solution of the discretized minimization problem.

3.2 Douglas-Rachford algorithm for three cost functions

The DR allows one to minimize a functional Fthat is the sum of a finite number of simple functionals. Here we use it to minimize the sum of 3 functionals: $F = F_1 + F_2 + F_3$.

We consider here the minimization space to be the space of the couples of discretized couples $(f, \mathbf{m})$, the first couple defined on the staggered gris and the second couple defined on the centered grids.

F_1 is the cost of transport (4) and the continuity constraint (1). The continuity constraint only applies to the first couple $(f, \mathbf{m})$ (defined on the staggered grid) whereas the cost of transport only applies to the second couple $(f, \mathbf{m})$ (defined on the centered grid).

F_2 is the interpolation constraint, that mixes both couples $(f, \mathbf{m})$.

F_3 is the boundary conditions constraint (2) and (3). It only applies to the first couple $(f, \mathbf{m})$ (defined on the staggered grid).

Note that, as long as each boundary condition required for the optimal transport problem appear at least in one functional, one can freely choose any variation of the proximal operators for the constraints. See section 5.6 for more details about the way proximal operators are selected.

One then defines four recurrent sequences u_n^1, u_n^2, u_n^3 and x_n that are in the minimization space (*i.e.* that are couples of couples $(f, \mathbf{m})$) by:

$$p^1 = \mathrm{Prox}_{\gamma F_1}\left(u_n^1\right), \tag{8}$$
$$p^2 = \mathrm{Prox}_{F_2}\left(u_n^2\right), \tag{9}$$
$$p^3 = \mathrm{Prox}_{F_3}\left(u_n^3\right), \tag{10}$$
$$p = \omega_1 \cdot p^1 + \omega_2 \cdot p^2 + \omega_3 \cdot p^3, \tag{11}$$
$$u_{n+1}^1 = u_n^1 + \alpha \cdot \left(2 \cdot p - x_n - p^1\right), \tag{12}$$
$$u_{n+1}^2 = u_n^2 + \alpha \cdot \left(2 \cdot p - x_n - p^2\right), \tag{13}$$
$$u_{n+1}^3 = u_n^3 + \alpha \cdot \left(2 \cdot p - x_n - p^3\right), \tag{14}$$
$$x_{n+1} = (1 - \alpha) \cdot x_n + \alpha \cdot p, \tag{15}$$

where p, p^1, p^2 and p^3 are working variables in the minimization space.

If $\alpha \in [0, 1]$, $\gamma > 0$, $\omega_1 > 0$, $\omega_2 > 0$, $\omega_3 > 0$ and $\omega_1 + \omega_2 + \omega_3 = 1$ then x_n converge towards the solution of the discretized minimization problem.

3.3 Douglas-Rachford algorithm for two cost functions

When using the standard boundary conditions (*i.e.* no reservoirs), one can use the `ProxCdivb` version of the `proxCdiv` in the DR algorithm. The boundary constraint in functional F_3 are therefore redundant with those in functional F_1. Hence, one can adapt the previous algorithm to use only the first two proximal operators. This is a clever way to reduce the computation complexity per iteration.

4 Launching an algorithm

4.1 With the default launchers

The easiest way to launch an algorithm is to use the default launchers provided in the top repertory. More precisely, the launcher to use is the *python* file `launchSimulation*D.py` with the command:

```
$ ./launchSimulation*D.py CONFIG_FILE=file.cgf PRINT_IO=True
```

where `file.cfg` is your configuration file and `PRINT_IO` is an optional parameter that enables output on the screen. The launcher then reads your configuration file, construct the adequate configuration object, then constructs the algorithm object with the `algorithm(self)` method and make it run with the `run(self)` method. At the end, it

save the results in the specified output repertory (see section 5.9) and make an analyse of the run (see section 5.3).

An example of configuration file is given by `OT/OTObjects*D/OT*D.cfg.example`. It must define the following parameters:

1. `EPSILON`: variable used as tolerance value for some testings.
2. `outputDir`: path to the repertory for output[1]. See section 5.9 for more details about the output.
3. `M`, `N` and `P`: discretization resolution along the dimensions in space for `M` (only for the $d = 2$ case) and `N` and in time for `P`.
4. `dynamics`: parameter used to select the correct variations of the proximal operators. See section 5.6 for more details about the way proximal operators are selected.
5. `boundaryType`: parameter used to select the correct boundary conditions. See section 5.7 for more information about the boundary conditions.
6. `normType`: parameter used to select the correct normalization of the fields. See section 5.11 for more details about normalization.
7. `file**`: names of the files used to define the boundary conditions when needed. See section 5.7 and section 5.8 for more information about the boundary conditions and input.
8. `algoName`: name of the algorithm to use. Use:
   - `pd` for the PD algorithm;
   - `adr3` for the DR algorithm with three proximal operators;
   - `adr` for the DR with two proximal operators.
9. `iterTarget`: number of algorithm iterations to run.
10. `nModPrint`: while running the algorithm, display information every `nModPrint` algorithm iteration. Information include the number of iterations run, the time elapsed and the current value of the cost of transport (4).
11. `nModWrite`: while running the algorithm, every `nModWrite` algorithm iteration, the current value of the converging variable will be written in the output directory. See section 5.9 for more details about output.
12. `initial`: parameter used to select the correct method to compute the initial condition. See section 5.5 for more details about initialization.
13. `initialInputDir`: when the initial condition is obtained from a previous run, this variable must contain the repertory in which the results have been saved. See section 5.5 and section 5.8 for more informations about initialization and input.

Also the algorithm parameters must be defined. See table 3 for more details about the parameters specific to each algorithm.

[1] *N.B.*: although *python* supports relative paths, it is strongly recommended to use absolute paths.

Parameter to define	Variable name	Default value
For the PD algorithm		
σ	`theta`	85
τ	`sigma`	1/85
θ	`tau`	1
For the DR with two proximal operators		
γ	`gamma`	1/75
α	`alpha`	1.998
For the DR with three proximal operators		
γ	`gamma3`	1/75
α	`alpha3`	1.998
ω_1	`omega1`	0.33
ω_2	`omega2`	0.33
ω_3	`omega3`	0.34

Table 3: Parameters to define in the config file.

4.2 Alternatives

If one doesn't want to use the provided *python* launchers, the best way is probably to define an (almost) empty configuration class and to make it contain every variable needed for the construction of the algorithm. We provide here a minimal example to solve the optimal transport between two one-dimensional Gaussians using the PD algorithm.

```
#!/usr/bin/env python

import numpy as np
from OT.OTObjects1D.algorithms.pd.pdAlgorithm import PdAlgorithm
from OT.OTObjects1D.grid                      import grid

# Main parameters
outputDir  = '/wherever/you/want/'
N          = 31
P          = 31
sigma      = 85.0
tau        = 1.0 / 85.0
theta      = 1.0
iterTarget = 10000
nModPrint  = 1000
nModWrite  = 1000

# Empty class configuration
class Configuration:
    # You just need this function for the run of the algorithm
    def printConfig(self):
        print('Print the message you want')

# Boundary conditions
# f0 ( x )  = A0 exp ( - alphaX0 * ( x - x0 ) ^ 2 )
# f1 ( x )  = A1 exp ( - alphaX1 * ( x - x1 ) ^ 2 )
A0      = 1.0
alphaX0 = 60.0
x0      = 0.375
A1      = 1.0
alphaX1 = 60.0
x1      = 0.625
X       = np.linspace ( 0.0 , 1.0 , N + 1 )
f0      = A0 * np.exp ( - alphaX0 * np.power ( X - x0 ,  2.0 )  )
f1      = A1 * np.exp ( - alphaX1 * np.power ( X - x1 ,  2.0 )  )

tBounds = grid.TemporalBoundaries( N , P , f0 , f1 )
sBounds = grid.SpatialBoundaries( N , P )
bounds  = grid.Boundaries( N , P , tlBounds, sBounds )
```

```
40
# Fill configuration with the parameters
config             = Configuration() ;
config.N           = N
config.P           = P
45 config.iterTarget = iterTarget
config.nModPrint   = nModPrint
config.nModWrite   = nModWrite
config.dynamics    = 0
config.boundaries  = bounds
50 config.outputDir   = outputDir
config.initial     = 0
config.sigma       = sigma
config.tau         = tau
config.theta       = theta
55
# Constructs the algorithm and run it
algo = PdAlgorithm(config)
algo.run()
```

5 Specific aspects

5.1 Relaunch simulations

The *python* file `reLaunchSimulation*D.py` can be used to relaunch a simulation. This script will construct the configuration from the previous configuration file if provided by the keyword argument `CONFIG_FILE` or directly from the configuration object saved in the output directory. In the latter case, one has to specify the previous output directory with the keyword argument `OUTPUT_DIR`. Optional arguments include `NEW_ITER_TARGET`, which specify a new value for the number of algorithm iterations to perform, and `PRINT_IO`.

Use one of the following commands.

```
$ ./reLaunchSimulation*D.py CONFIG_FILE=configFile.cgf NEW_ITER_TARGET=1000
    PRINT_IO=True
```

```
$ ./reLaunchSimulation*D.py OUTPUT_DIR=/previous/output/dir/
   NEW_ITER_TARGET=1000 PRINT_IO=True
```

Also note that using this script, one necessary relaunch the simulation with the same algorithm. There are also ways the relaunch a simulation with a different algorithm, for example see section 5.5.

5.2 Anamorphosis

For $d = 1$, the solution of the optimal transport problem can be obtained with an analytical formula, and this is known as the anamorphosis transform. For more details about this transformation, see section 2.4 of farchi-2016. An implementation of these formula is available with this solver. It has been written in the exact same way as the DR and the PD algorithms in the file `OTObjects1D/algorithms/anamorph/anamorphAlgorithm.py`.

To use this "algorithm", one must set `algoName=ana` in the configuration file and define the parameter `PDFError` that will be used as a relative tolerance value to compute the inverse of cumulative density functions. Unlike the other algorithms, it will ignore the parameter `iterTarget` since it is not an iterative algorithm.

5.3 Analyse

After performing a simulation, one may want to analyse the convergence rate of the algorithm. Some tools are implemented to this end in the repertory `OTObjects*D/analyse/`. First, the output files will be opened. As described in section 4, it contains the value of the converging variable every `nModeWrite` algorithm iteration. It also contains the execution time at the current iteration – in order to perform time analysis. Then some operators will be applied to the variable.

In file `operators1.py` are defined a first set of operators: the L^∞ norm of the $(d+1)$-divergence of the field (*i.e.* the error in the continuity constraint), the minimum value of the density field and some numerical variations of the cost function (4). In file `operators2.py` the operators defined compare the current value of the field to the last value of the field (which is therefore assumed to be the solution of the minimization problem): the L^∞ norm of the difference.

The results are stored in the file `analyse.bin` in `outputDir` in binary format using the *cPickle* module as following:

1. the *numpy* array of iteration numbers;
2. the *numpy* array of iteration times;
3. a list of string representing the operator names;
4. the *numpy* two-dimensional array of the operation values, whose shape is: number of iterations $\times$ number of operators.

With these informations, everything is available to plot analyses of the run as a function of time or as a function of iteration number. See section 5.4 for more information about plots. Also note that there is a *python* launcher dedicated to this analyse process: `analyseSimulation*D.py`, to use with one of the following commands.

```
$ ./analyseSimulation*D.py CONFIG_FILE=configFile.cgf
```

```
$ ./analyseSimulation*D.py OUTPUT_DIR=/output/dir/
```

5.4 Plotting and animating

With this module, two submodules are provided for plotting and animating purpose in the `OTObjects*D/plotting/` and `OTObjects*D/animating` repertories. The operation of these submodules is approximately the same as the main module. Two specific configuration classes are implemented. As usual, these submodules come with the dedicated *python* launchers `plotSimulation*D.py` and `animateSimulation*D.py` and their usage require a configuration file. The configuration files are specific to each submodule, examples are available: `OTObjects*D/plotting/plotting.cfg.example` and `OTObjects*D/animating/animating.cfg.examp`
Use the launchers with the following commands.

```
$ ./plotSimulation*D.py CONFIG_FILE=plotting.cgf PRINT_IO=True
```

```
$ ./animateSimulation*D.py CONFIG_FILE=animating.cgf PRINT_IO=True
```

Note that these submodules rely on *matplotlib* to draw plot and on movie encoders (*e.g. ffmpeg* or *mencoder*) to save the animations in files.

5.5 Initialization

The algorithm presented in section 3 converge, no matter which initial condition is chosen for the algorithm state. However, choosing a clever initial condition is a good strategy to achieve a better convergence without increasing the number of algorithm iterations. Yet, there has been no study about the influence of the initial condition in this particular optimal transport problem.

In our solver, all algorithm use the same initialization function `initialStaggeredField(config)` defined in the file `OTObjects*D/init/initialFields.py`. Basically, it just constructs the linear interpolation between the temporal boundary conditions f_0 and f_1.

Alternatively the initial condition can be obtained from the output of a previous simulation. This is driven by parameter `initial` of the configuration object:

- if `initial` is 1, then the algorithm will look for the result of a simulation whose result is located in `outputDir` to initialize its state;
- if `initial` is 2, then it will look in the repertory `initialInputDir`;
- if `initial` is 3, it will first look in `outputDir` then in `initialInputDir`.

If no previous run is found, the initialization method will fall back to the `initialStaggeredField(config)` function. With this method, an algorithm can catch the state of an other algorithm. In that case, it uses the converging variable to define its initial state.

5.6 Proximal variations selection

Choosing the correct versions for the proximal operators is a critical step while constructing the algorithms. The proximal used by each algorithm is detailed in section 3. When choosing the proximal operators, one must insure that each constraint appear at least in one proximal operator that will be used by the algorithm.

In file `OTObjects*D/proximals/defineProximals.py`, the function `proximalForConfig(config)` defines the correct proximal operators according to the parameter `dynamics` of the configuration.

- If `dynamics` is 0, then boundary conditions in space and time are added to all constraint.
- It is also the case if `dynamics` is 1, but this time the spatial boundary conditions are supposed to be zero – while they can be non-zero with `dynamics` equal to 0, as presented in section 5.7.
- If `dynamics` is 2, then the temporal boundary conditions are added to all constraints but no spatial boundary condition is applied. For this reason, the convergence of the algorithm with this method is questionable, as the mathematical problem is not well defined.

The 3 and 4 values for `dynamics` are dedicated to the version of the problem with reservoirs (see section 5.7). The divergence constraint never include the boundary conditions (because it would break the symmetry of the problem and prevent one to use the Fourier transformation methods). They are included in the interpolation constraint if `dynamics` is equal to 3 but not if it is equal to 4.

As a consequence, the PD algorithm must be used with `dynamics` equals to 1 or 2. The DR algorithm with two proximal operators can be used with `dynamics` equals to 1 or 2 for the classical optimal transport problem and `dynamics` equals to 3 for the optimal transport problem with reservoirs. The DR algorithm with three proximal operators can be used with `dynamics` equals to 1 or 2 for the classical optimal transport problem and `dynamics` equals to 3 or 4 for the optimal transport problem with reservoirs. Using `dynamics` equals to 2 is not recommanded.

5.7 Boundary conditions and reservoir approach

5.7.1 Boundary conditions

In the very definition of the optimal transport problem, we defined f_0 and f_1 to be probability density functions over $\mathbb{E}$. In particular, this means that:

$$\int_{\mathbb{E}} f_0(\mathbf{x})\,\mathrm{d}\mathbf{x} = \int_{\mathbb{E}} f_1(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1, \tag{16}$$

which is a necessary condition to have a couple $(f, \mathbf{m})$ satisfying (1), (2) and (3). In fact, this condition can be relaxed if one changes the null-flux condition (3) into:

$$\forall (\mathbf{x}, t) \in \partial\mathbb{E} \times [0,1], \quad \mathbf{m}(\mathbf{x}, t) = \mathbf{m}_0(\mathbf{x}, t), \tag{17}$$

with the new mass conservation condition:

$$\int_{\partial\mathbb{E}\times[0,1]} (\mathbf{m}_0(\mathbf{x}, t) \cdot \mathrm{d}\mathbf{x})\,\mathrm{d}t = \int_{\mathbb{E}} (f_1(\mathbf{x}) - f_0(\mathbf{x}))\,\mathrm{d}\mathbf{x}. \tag{18}$$

Numerically, it doesn't change anything in the computation of the proximal operators, it just changes the construction of the numerical boundary conditions. These conditions are constructed during the construction of the configuration object by the function `boundariesForConfig(config)` of file `OTObjects*D/boundaries/defineBoundaries.py`. This function relies on the parameter `boundaryType` of the configuration object. If `boundaryType` takes value 1 to 8, then default boundary conditions will be applied (see section 5.10). If `boundaryType` is 0, then the boundary conditions will be constructed with data from files. The file names for the f_0 and f_1 conditions are determined by the parameters `filef0` and `filef1` of the configuration object. If necessary – *i.e.* if the parameter `dynamics` is 0 as said in section 5.6 – spatial boundary conditions are also determined from files `filem0` and `filem1` for the $d = 1$ case and from files `filemx0`, `filemx1`, `filemy0` and `filemy1` for the $d = 2$ case. The corresponding boundary conditions are presented in table 4.

Parameter to define	Boundary condition driven
`filef0`	$t = 0$
`filef1`	$t = 1$
$d = 1$	
`filem0`	$\mathbf{x} = 0$
`filem1`	$\mathbf{x} = 1$
$d = 2$	
`filemx0`	$\mathbf{x}_1 = 0$
`filemx1`	$\mathbf{x}_1 = 1$
`filemy0`	$\mathbf{x}_2 = 0$
`filemy1`	$\mathbf{x}_2 = 1$

Table 4: Boundary conditions driven by the files.

The format of the file accepted by the algorithm is detailed in section 5.8. After loading the fields from the files, one has to check that the mass is conserved according to (18).

See section 5.11 for more information about normalization. If one is interested in solving the optimal transport problem without the mass conservation constraint, then one should consider using the reservoir method we developed and presented in [2].

5.7.2 Reservoir approach

For this method, $\mathbb{E}$ is expanded by adding a reservoir variable at each boundary location of $\mathbb{E}$. To keep control of the total mass inside the domain, we impose the null-flux condition (3) to the extended domain. Potentially inverting the role of f_0 and f_1 – this operation is tracked by the variable `swappedInitFinal` of the configuration object – one can assume that f_0 has more mass than f_1. Then we can impose the following boundary conditions:

- at $t = 0$, all reservoirs must be empty, *i.e.* all reservoir variables must be equal to zero;
- at $t = 1$, the total mass inside the reservoir must be equal to the difference of mass between f_0 and f_1;
- in the interior of $\mathbb{E}$ we impose the classical temporal boundary conditions (2).

To solve the optimal transport problem with reservoir, one must use the DR algorithm with the parameter `dynamics` equals to 3 or 4 as presented in section 5.6. Note that in this case, the boundary values of the arrays specified for f_0 and f_1 will be ignored and replaced by zero.

5.8 Input

5.8.1 Boundary conditions

The boundary conditions can be defined from files as presented in section 5.7. The files are read with the `arrayFromFile(fileName)` function, implemented in the file `utils/io/io.py`. It accepts either `.npy` files, loaded with with the `numpy.load(fileName)` function, or binary files, loaded with the `numpy.fromfile(fileName)` function and then reshaped with the `numpy.ndarray.reshape(newShape)` function to match the dimensions imposed by the parameters `M`, `N` and `P`.

5.8.2 Initial condition

As seen in section 5.5, the initial condition can be retrieved from a previous run. In that case, the algorithm will read the files `runCount.bin` – to check that there has been indeed a run – and `finalState.bin` that have both been written by an algorithm, according to the method presented in section 5.9.

5.9 Output

5.9.1 During the algorithm run

As mentioned in section 4, every `nModWrite` iterations, the algorithm writes the converging variable and the current time elapsed to a file. The file is named `states.bin` and is located in the `outputDir` repertory. The variable is written as a `StaggeredField` object, in binary format using the *cPickle* module. Note that the results are appended to the file `states.bin` so that one can keep the states of the previous runs.

5.9.2 At the end of the run

After the run, a full copy of the algorithm state is written in `outputDir`. More precisely, the configuration object is written in the file `config.bin` and the algorithm state in the file `finalState.bin`. Once again, the objects are written in binary format with the *cPickle* module. Note that the configuration is appended to the file in order to keep track of the configurations for the previous runs.

Finally, the number in the file `runCount.bin` is incremented if it already exists. Else the number 1 is written in file `runCount.bin`. This way, one is able to keep track of the number of runs performed.

5.10 Default examples

As seen in section 5.7, there are a few default configurations for the boundary conditions provided for testing purpose. They are all defined in the repertory `OTObjects*D/boundaries/` and the selection of the correct boundary condition happens in function `boundariesForConfig(config)` from file `defineBoundaries.py` according to the parameter `boundaryType`. The possibilities are described in table 5.

`boundaryType`	File	Description
1	`gaussian.py`	One Gaussian
2	`gaussian.py`	Sum of two Gaussians
3	`gaussianSplit.py`	Sum of two Gaussians rejoining into one Gaussian
4	`gaussianSplit.py`	One Gaussian splitting into two Gaussians
5	`gaussianSine.py`	One Gaussian with sine oscillations
6	`gaussianSine.py`	One Gaussian with cosine oscillations
7	`custom.py`	Guess what...
8	`custom.py`	Guess what...

Table 5: Default boundary conditions depending on the parameter `boundaryType`.

5.11 Normalization

After defining the correct boundary conditions, one has to check the mass conservation (if not using the reservoir method). This is performed by the `normalize(normType)` method of class `Boundaries` whose behavior only depends on the `normType` parameter of the configuration object.

- If `normType` is 0, then f_1 is rescaled using the mass of f_0.
- If `normType` is 1, then f_0 is rescaled using the mass of f_1.
- If `normType` is 2, then f_0 and f_1 are not changed but the spatial boundary conditions are adapted to compensate for the mass losses.
- If `normType` is 3, then the mass of f_0 and the mass of f_1 are set to unity.

A References

[1] J.-D. Benamou and Y. Brenier. "A computational fluid mechanics solution to the Monge-Kantorovich mass transfer problem". In: *Numerische Mathematik* 84.3 (2000), pp. 375–393.

[2] Alban Farchi, Marc Bocquet, Yelva Roustan, Anne Mathieu, and Arnaud Quérel. "Using the Wasserstein distance to compare fields of pollutants: Application to the radionuclides atmospheric dispersion of the Fukushima-Daiichi accident". In: *Tellus B* –.– (2016). in preparation,

[3] L. V. Kantorovich. "On the translocation of masses". In: *Dokl. Akad. Nauk SSSR* 37 (1942), pp. 199–201.

[4] G. Monge. "Mémoire sur la théorie des déblais et des remblais". In: *Histoire de l'Académie Royale des Sciences de Paris*. 1781, pp. 666–704.

[5] N. Papadakis, G. Peyré, and E. Oudet. "Optimal Transport with Proximal Splitting". In: *SIAM Journal on Imaging Sciences* 7.1 (2014), pp. 212–238.

[6] C. Villani. *Optimal Transport: Old and New*. Vol. 338. Springer-Verlag Berlin Heidelberg, 2008, p. 976.