# <Bottom-up Formulation of Water Management Systems as a Reinforcement Learning Problem>

## <Generalisation of Water Management in the Context of Reinforcement Learning>

< Jorian Faber[1] >
Supervisor(s): <P.K. Murukannaiah[1]>, <Z. Osika[1]>
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: <Jorian Faber>
Final project course: CSE3000 Research Project
Thesis committee: <P.K. Murukannaiah>, <Z. Osika>, <L. Miranda da Cruz>

# Bottom-up Formulation of Water Management Systems as a Reinforcement Learning Problem

**Jorian Faber**
BSc Computer Science and Engineering
Delft University of Technology
Delft, 2628CD
`j.e.c.faber@student.tudelft.nl`

## Abstract

Water management systems (WMSs) are complex systems in which often multiple conflicting objectives are at stake. Reinforcement Learning (RL), where an agent learns through punishments and rewards, can find trade-offs between these objectives. This research studies three case studies of WMS simulations in the context of RL problems and notes their similarities and differences. Based on these, core properties of WMSs are defined and used to formulate a general WMS as an RL problem. This bottom-up approach uses Gymnasium to implement the RL problem. The result is compared to a simulation from one of the case studies and produces the same results. While maintaining this level of accuracy, it is applicable to a much wider range of WMSs. It thereby contributes to generalisation of WMSs in the context of RL, and removes the need to rewrite simulations each time.

## 1 Introduction

Water management is a complicated problem, even more so with a rapidly changing climate. More extreme climate conditions emphasise the need for sustainable water management practices, Badika et al. (2024) argues. Policymakers have the difficult task of fairly managing water. Many WMSs are defined by multiple objectives such as hydropower production, irrigation demands, water supply, or environmental protection. When water is controlled to satisfy one of the objectives, the others are often disadvantaged as a consequence. These conflicting objectives add to the complexity of WMSs.

RL is particularly well-suited for tackling such problems. RL is a Machine Learning (ML) technique that trains an agent to make optimal decisions. For every decision the agent makes, it receives a reward or punishment. Through trail-and-error, the agent learns to make more decisions that lead to a reward and less that lead to a punishment. RL problems are characterised by sequential decision-making in order to maximise a reward. A WMS fits this description very well: in sequence, decisions are made about how much water is released, affecting future states and resulting rewards such as hydropower production. Few general WMSs modelled as RL problems exist, if any. Hence, research on optimal WMS policies requires writing a simulation from scratch, or finding one that is similar enough to the WMS at hand. Even if one is found, understanding and reproducing it can be a complex and time-consuming task. A general model that can simulate general WMSs provides a basis for research on fair water management policies, without the overhead of modelling WMSs from the ground up.

To better understand their mechanics, this research studies three distinct WMSs:

- Nile River in Ethiopia, Sudan and Egypt
- Lower Volta River in Ghana
- Susquehanna River in the United States, in Pennsylvania and Maryland

The case studies, with a wide range of characteristics, will be used to define the core properties of a WMS. Based on these core properties, an abstract RL problem is modelled to simulate general WMSs. Gymnasium, an open-source Python[1] library that provides an (Application Programming Interface) API for modelling RL problems, is a common choice in the field for such tasks and is therefore used in this research (Towers et al. (2023)). This paper describes the formulation of a general WMS as an RL problem, and how its Gymnasium implementation can be applied to a wide range of WMSs. It provides an answer to the question 'What are core properties of a WMSs based on three case studies, and how can they be modelled as an RL problem?'. The result of the research is a generalised formulation and implementation of WMSs in the form of a custom Gymnasium Environment. It is compared to the simulation from the Nile River case study by Sari (2022) and produces results that are just as accurate. Additionally, it is compared to a top-down approach by Muniak (2024). His framework is a more out-of-the-box solution and is easier to apply to WMSs similar to the Nile River. The implementation described in this paper, however, offers greater flexibility and can be applied to a wider range of WMS scenarios.

The Background will provide further explanation on RL and how problems are formulated, and discuss the three WMSs. The Methodology goes over a more precise explanation of the problem, and explains how research was conducted to answer the research question. The final RL problem is described and evaluated in the Results. Relevant ethical aspects and reproducibility of the research are discussed in the section on Responsible Research. The Discussion elaborates on possible improvements and provides some recommendations for future research. Lastly, the Conclusion provides a summary and concludes the research.

## 2 Background

This section will provide background information to help understand what aspects of WMSs need to be analysed to be able to define and implement a general definiton of a WMS as an RL problem. Firstly, the notion of an RL problem will be introduced, along with relevant RL concepts that are important in the context of this research. Consequently, Gymnasium, the Python library that is used to implement the general WMS as RL problem, is explained. Lastly, this section will discuss the three case studies of the Nile, Lower Volta and Susquehanna rivers. It gives an overview of the general problem of WMSs that these case studies describe, and provides a brief description of each of the case studies.

### 2.1 RL problem definition

A RL problem is defined by an **agent** that lives in and interacts with an **environment**, a set of **states**. Based on its **observation** of the *environment*, a partial or complete interpretation of the current *state*, it makes a decision in the form of an **action**. The *environment* changes because of this *action* as the *agent* transitions to the next *state*. It receives a **reward** as a result, which indicates the quality of the current *state*. A positive *reward* indicates that the decision was good. A negative *reward*, or punishment, indicates the opposite. The agent tries to maximise the cumulative *reward* it receives, and should therefore 'learn' to make less decisions that lead to a punishment and more that lead to a reward. Figure 1 shows a diagram of this agent-environment interaction.

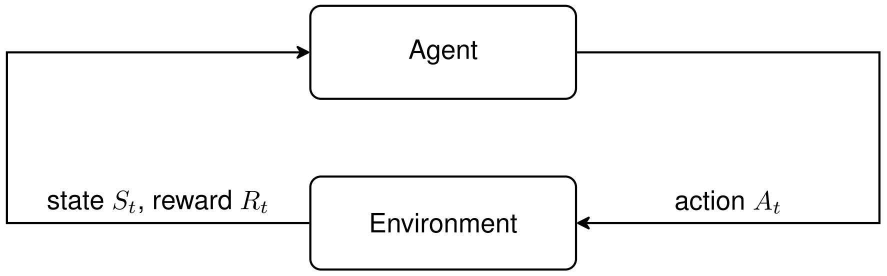

Figure 1: Agent-environment interaction

[1]https://www.python.org/

Following this, it is crucial to understand two terms when formulating RL problems: **observation space** and **action space**. *Observation space* represents all the possible *states* of an *environment* that an *agent* can observe. For a WMS, that might include water levels, flow rates or weather conditions, which help the *agent* make informed decisions. The *action space* encompasses all the possible *actions* an *agent* can take to influence the *environment* and transition to a different *state*. For example, in a 2D grid where an *agent* can only move in straight lines, its *action space* contains the *actions* of going up, down, right and left. Formulating and implementing an RL problem requires clearly defining the *observation space* and *action space*.

Usually, RL is used to optimise a single objective, by maximising the aforementioned cumulative reward. When dealing with multiple objectives, it is often assumed that they can be handled by linear combinations such as a sum of all rewards. As a linear combination does not make clear which rewards are maximised at the cost of others, this may oversimplify the problem. A different and more optimal approach is to return a vector of rewards Hayes et al. (2022). This approach aligns well with the needs of WMSs, which also often deal with multiple objectives and for which understanding trade-offs is very valuable.

## 2.2 Gymnasium

Gymnasium, developed by Towers et al. (2023), is open-source Python library that provides an API for developing RL algorithms and implementing RL problems. It offers access to standard RL environments, which users can easily interact with, simply by knowing a couple of API methods . Gymnasium makes it easy to create custom environments by implementing their standard API methods, which makes it a common choice in the field of RL. Standard Gymnasium is used for single-objective RL. As WMSs deal with multiple objectives, this research will use its multi-objective version called MO-Gymnasium, developed by Felten et al. (2023). It is very similar to Gymnasium and follows the same API standards. However, instead of returning a single reward, it returns a vector of rewards. Apart from the aforementioned *observation space* and *action space*, it therefore also requires specifying a **reward range** along with reward dimensions, to make the learning process more efficient.

Table 1: Gymnasium Environment methods and attributes to implement

| Name | Description |
|---|---|
| `step()` | Most of the logic: takes an `action` and computes the environment. |
| `reset()` | Resets the environment. Called before starting a new episode. |
| `observation_space` | Space of all valid observations. |
| `action_space` | Space of all valid actions. |
| `reward_range` | Defines the shape of the vector rewards. |
| `reward_dim` | Defines the size of the reward vector. |

### 2.2.1 Custom Gymnasium environment

Creating a custom Gymnasium environment requires implementing some standard API methods. Table 1 gives an overview of the methods and attributes that have to be implemented to create a custom Gymnasium environment, for multi-objective RL. Methods and attributes that are not relevant to this paper or do not have to be implemented, such as `render()` for rendering an environment, are disregarded.

The most important method to discuss is the `step()` function. It takes an `action`, computes the environment by applying that `action` and returns a tuple (`observation`, `reward`, `terminated`, `truncated`, `info`). Each time the *agent* takes an *action*, it calls this method. It then transitions to a new *state* defined by the *observation* that is returned, and receives a *reward*. To compute `observation` and `info` each time `step()` is called, Gymnasium recommends implementing private `_get_obs()` and `get_info()` methods. `observation` is a subset of the information available about the current *state*, which the agent can use to make decisions. `info` contains extra information about the current *state* that is not in observation but can be useful for, for instance, debugging.

The attributes inform the *agent* within which bounds it can observe, act and receive rewards. Gymnasium provides a `Space` class for `observation_space` and `action_space`. A `Space` describes a mathematical set. The most 'simple' subclasses of `Space` are `Discrete` and `Box`. They describe mathematical sets in discrete and continuous space respectively, and allow for setting lower and upper bounds, as well as its shape.

## 2.3 Studied Water Management Systems

This section first explains the choice for the three case studies. It proceeds to give a high-level overview of the general problem of WMSs that all the case studies deal with. It then provides a brief description of each of the case studies.

The three case studies are selected because they have varying characteristics and for each, a computational simulation exists. These simulations, all written in Python, allow for more thorough analysis of the system's mechanics, focused on their computational implementation. With access to a Python simulation, translating WMS mechanics to a general Gymnasium Environment, also written in Python, is made easier.

All three studies study a WMS, in which one or more dams control the flow of water and thereby its allocation to certain places. The control of this water flow provides economic, environmental and other benefits. The studies acknowledge that these benefits require making trade-offs: when water is allocated to one place, other places receive less. In other words, the objectives are in conflict. The three studies apply Evolutionary Multi-Objective Direct Policy Search (EMODPS), a framework for optimising multiple, conflicting objectives. They explore how it can be used to create different policies and what trade-offs are made to reach certain objectives.

### 2.3.1 Nile River (Ethiopia, Sudan, Egypt)

Sari (2022) describes a model of the Nile River that crosses through Ethiopia, Sudan and Egypt, with four dams on its way. Each of the countries have individual objectives, with a total of four. The objectives entail minimising irrigation demand deficits, maximising hydropower production or minimising months below minimum power generation. Sari explores creating fair policies using EMODPS. The simulation models several different facilities: reservoirs, irrigation systems, power plants and catchments.

### 2.3.2 Lower Volta River (Ghana)

Owusu et al. (2023) explain how the construction of the Akosombo and Kpong dams in the Lower Volta river basin have provided economic benefits to the Republic of Ghana. However, these benefits from hydropower, irrigation schemes and lake tourism have come at the cost of the ecosystem and downstream communities. EMODPS is used to explore different balances between economic benefits and the livelihood of the ecosystem and downstream communities. The objectives include maximising annual hydropower, maximising irrigation demand satisfaction, minimising flood occurrences and adhering to so-called "e-flow recommendations". E-flows recommendations are ranges of water flows with certain goals, such as supporting the ecosystem. The simulation models two facilities: dams and their hydropower generation.

### 2.3.3 Susquehanna River (United States of America)

Witvliet (2022) uses the Susquehanna River to answer questions about the characteristics of EMODPS in relation to multi-objective water management problems, and thereby positions the river less at the centre of his research. Nonetheless, he describes how the Susquehanna River passes through Pennsylvania and Maryland, with the Conowingo dam regulating a large part of the water flow of the river basin. The hydroelectric dam supplies water to two cities, Chester and Baltimore, as well as cooling water for a nuclear power station. The releases of the dam are subject to minimum flow requirements to conserve fishing resources. A pumped storage hydroelectric facility is connected to the reservoir and acts as a battery, in that it can cycle water back and forth from the Conowingo reservoir when needed for additional power generation. The research describes objectives that maximise hydropower revenue, water supply and -storage reliability and one that minimises environmental shortage. The simulation models a few facilities: a dam, hydropower facility and pumped storage.

## 3 Methodology

This section will explain the procedure that was used to formulate a general WMS as an RL problem, implement it in Gymnasium and provide an answer to the research question.

At the start of the research, the author, together with peers, worked on rewriting the simulation of the Nile river to a multi-objective RL simulation. This provided some initial understanding of the mechanics of WMSs and the problems it deals with. Subsequently, the following steps were taken to implement a general WMS as RL problem:

1. Define **elements of RL problem** formulation
2. List **required methods and attributes** for implementation
3. **Analyse three case studies**, note down their characteristics for each with RL elements in mind
4. Compare case studies and find **key similarities and differences**
5. Define **core properties** based on similarities and use generalisation to handle differences
6. Choose **data structure** for WMS
7. Translate core properties to **RL problem formulation**, start with **most basic** concept of WMS
8. **Implement** this most basic WMS as RL problem
9. With basic WMS as starting point, study core properties and **work out implementations** of **additional features** step by step

During analysis of the three case studies, the elements of RL problem formulations as described in 2.1 are carefully considered. This ensures that relevant characteristics and features necessary for defining the RL problem—such as *environment* and *state*—are accurately identified and prioritized. Consequently, the analysis focuses on essential aspects that shape the RL formulation. The result is an abstract RL problem formulation of the most basic definition of a general WMS, along with a more advanced general definition of WMSs as RL problem based on the three case studies. The RL problem formulations are implemented as custom Gymnasium Environments.

By starting from scratch with an analysis of similarities and differences, this approach can be considered to be bottom-up. Muniak (2024) instead took a top-down approach. He considered the Nile River case study and its simulation and generalised it with each feature that he removed. The two different approaches are compared to evaluate which one is easier to use and which one can be applied to a wider range of systems. The result of this research is also evaluated by verifying if it can simulate the Nile river, by comparing it to the results from Sari (2022). The paper also discusses to what extent it can simulate the Susquehanna and Lower Volta rivers.

## 4 Results

This section will present the results of the research. First, 4.1 discusses the similarities and differences between the analysed case studies. In 4.2 the core properties of a basic, general WMS are defined, and its implementation as an RL problem is explained. 4.3 goes over additional features of the implementation which define the final RL problem, and how differences are handled by generalising them. Lastly, the final result is evaluated in 4.4 by showing the results of comparison with the Nile river case study from Sari (2022).

### 4.1 Similarities and differences

To determine what characteristics should be simulated in the general RL problem, the similarities are listed. Key differences are also highlighted, to help decide how they can be handled in defining the general model. This comparison disregards many differences that are deemed irrelevant, such as the varying number of objectives. This analysis merely serves as a means to determine what WMS mechanics should be taken into account when formulating a general multi-objective RL problem, aspects that do not serve this purpose are ignored.

### 4.1.1 Important similarities

**Dams** All case studies have dams, or 'reservoirs' as some call it, at the centre of their simulation, with water flowing in and out. It is these points in the WMS that control water flow and thereby control its allocation. They are the points where decisions are made in the simulation. Each dam has an inflow, an outflow, storage volume, volume-level-surface relationships and release constraints. Volume-level-surface relationships are used to convert storage volume to other metrics, e.g. when calculating evaporation based on surface area of the stored water. Release constraints are used when calculating actual release flows, to adhere to physical constraints.

**Other water management facilities** The simulations also describe facilities other than dams such as irrigation systems and power plants. Although they might seem different, they are quite similar on an abstract level: they have water flowing in and out, as well as unique attributes that are used in reward functions.

**Water volume: mass-balance equations** Although the three case studies have slightly unique mass-balance equations to calculate water release and remaining storage volume at a facility, they are all very similar. As is demonstrated below, storage volume at the next step is calculated by taking the current storage volume, adding inflow and subtracting released water along with some other variables such as evaporation. Surprisingly, Witvliet (2022) and Owusu et al. (2023) both subtract a constant leakage value in the mass-balance equations in their simulations, while this is not mentioned in their papers. The complete mass-balance equations of all three case studies are shown below: equation 1 for the Nile river and equation 2 for the Susquehanna and Lower Volta rivers as they are both the same. $V_t$ represents volume, $\sum IF_t$ the sum of inflows, $\sum OF_t$ the sum of outflows, $E_t$ the evaporation and $L_t$ the leakage at time $t$. To accurately calculate changes in water flow over time, calculations are made over smaller integration steps, as is explained in 4.3.4.

$$V_{t+1} = V_t + \sum IF_{t+1} - \sum OF_{t+1} - E_{t+1} \tag{1}$$

$$V_{t+1} = V_t + \sum IF_{t+1} - \sum OF_{t+1} - E_{t+1} - L_{t+1} \tag{2}$$

**Release constraints** To realistically simulate physical constraints and adhere to environmental, regulatory or other forms of constraints, the case studies apply release constraints. Even though the factors that dictate release constraints differ slightly, they are similar quite similar: they can be defined as a release range with a minimum and maximum release that depends on the current state of a water facility.

**Time-dependent data** All three case studies use time-dependent data in their simulations, either stochastic or historical. Time-dependent data is used to accurately represent real-life values changing over time. For example, inflows might differ per month or evaporation rates might differ per day.

### 4.1.2 Key differences

**Time per step** The Nile river case study simulates steps of one month, whereas the Lower Volta river case study simulates steps of one day. The Susquehanna takes even smaller steps, namely four hour ones.

**Reward functions** The most challenging difference is the varying reward functions. With reward functions at the core of the 'learning' aspect of RL, it is no surprise that this difference poses a challenge when trying to create a general model that is applicable to a side variety of WMSs. Even with two objectives that focus on demand, for example, their difference can be sufficiently large to ignore. The Nile river case study tries to minimise demand deficit, calculated with equation 3. The Susquehanna river case study, on the other hand, tries to maximise volumetric reliability, calculated using equation 4. In the two equations, $R_t$ represents the reward at step $t$, $D_t$ the demand and $I_t$ the inflow, or amount of water that is actually received, at time $t$. Similar to this example, however, most reward functions are very similar at their core, in that they use similar state information to calculate rewards.

$$R_t = D_t - IF_t \tag{3}$$

$$R_t = IF_t / D_t \tag{4}$$

**In- and out-degree of water points** The Susquehanna River simulation is unique as it contains a dam with an out-degree larger than one. In other words, this single point in the WMS has multiple outgoing flows. This makes the WMS more difficult to model, because the multiple outgoing flows share one storage volume, from which they might receive varying release flows.

**Delay** The simulation of the Nile river implements a delay of water flow from one point to another, to account for long travel distance. The delay has a duration of 1 step, which means the destination water facility handles that flow's water one step later.

**Water flow direction** Naturally, water flows from upstream to downstream, which is the case in all three case studies. However, as 2.3.3 describes, the Susquehanna river adds some complexity to its simulation with a pumped storage hydroelectric facililty, which is able to pump water back and forth from the dam to the powerplant if needed.

**Metrics** The Nile simulation uses the metric system, whereas the Lower Volta and Susquehanna simulations use the imperial system.

## 4.2 Core properties of WMSs

At its core, a WMS is simply a volume of water moving from one place to another. One water point is connected to another via a flow. As this water flows from and towards several different directions, a simple list does not suffice. A data structure that fits this description is a graph, where nodes present water points and directed edges present the flows between them. Figure 2 shows a graph representation of this most basic formulation.

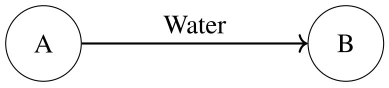

Figure 2: Graph representation of a basic WMS

### 4.2.1 Nodes and edges

To represent any basic water facility or point, each node should have four basic properties: `id`, `volume`, `capacity` and `demand`. `id` identifies the node. `volume` indicates how much water is currently at the water point. `capacity` limits how much water can be at the water point. `demand` is the most basic component of reward functions and defines how much water the water point requires. Since it can be set to zero to remove its effect, it forms a good starting point for reward functions in the RL problem.

Edges connect nodes and represent water flowing from one node to another. Apart from properties that indicate source and destination of the edge, they have one property: `flow`, the amount of water that flows from source to destination. Because an algorithm should be able to traverse through the graph structure from upstream to downstream, the connection between nodes should be defined. With `incoming_flow` and `outgoing_flow` as node properties, an algorithm can traverse from node to node. To allow for nodes that have an inflow without a source or outflow without a destination, `inflow` and `outflow`, how much water flows and flows out, are also properties of a node. The resulting graph representation is presented in Figure 3.

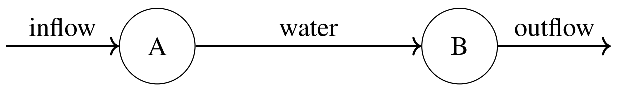

Figure 3: Graph representation of a basic WMS with inflow and outflow

### 4.2.2 Implementation as an RL problem

Turning this formulation of a very basic WMS into an RL problem and creating a custom Gymnasium Environment still requires some steps. As discussed in 2.1, an RL problem is defined by an environment with a set of states. An agent can observe this state and take an action based on this.

The *environment* consists of all nodes and edges. A *state* is defined by all information there is about these nodes and edges, such as `volume`, `capacity` and `flow`. An *agent* can take an *action* by releasing an amount of water from a node (to an outgoing edge). To make its decision, the *agent* can make an *observation* of the `volume` that is currently at a node. Therefore, `_get_obs()` returns the `volume` of a node. Now that actions and observations are determined, their spaces can be defined: *action space* is a continuous range from 0 to a node's `capacity`, since an action that is larger than a node's capacity is never possible. The *observation space* is also defined as a range from 0 to a node's `capacity`, as the observed `volume` can never be larger than a node's `capacity`. Based on the action, the agent transitions to a new state. At each step, `_get_info()` returns a dictionary with info about the current state: `id`, `volume`, `inflow`, `outflow`, `capacity`, and `demand`.

**Reward functions** To calculate and keep track of *rewards*, a custom data type is defined. It offers a generalised and flexible way to define rewards, to allow for custom reward functions. Each `Reward` object has an `id`, a `reward_function` and a `value`. `reward_function` is the most interesting property of this data type: it is a function that takes the `info` dictionary as its argument, and returns a float value. This is the first feature that demonstrates how widely applicable the RL problem is: the `reward_function` can be any lambda that uses the current state's information in the `info` dictionary to calculate a reward. Pseudocode 1 shows two examples of such reward functions: a simple division operation that can easily be rewritten as a lambda and a more complicated one, demonstrating how widely applicable they are. State information can be used, constants such as `efficiency` can be set within functions, and external methods such as `volume_to_level()` can be called. Every node can be assigned one or multiple of these rewards, and they can even be shared across multiple different nodes because of its `value` property.

```
Function volumetric_reliability(info)
    return $info["inflow"]/info["demand"]$
```

```
Function power_production(info)
    $efficiency \leftarrow 0.80$
    $production\_coefficient \leftarrow 200$
    $waterlevel \leftarrow volume\_to\_level(info["volume"])$
    return $efficiency * production\_coefficient * water\_level * info["outflow"]$
```

Pseudocode 1: Two examples of different reward functions

The Gymnasium Environment is initialised with a list of nodes. It traverses the graph by iterating over the list, calling each node's `step()` function. The `step()` function, that which actually executes the steps of the RL problem, does the following things to a node when passed an action:

```
if $incoming\_flow \neq$ null then
    $inflow \leftarrow inflow + incoming\_flow.get\_current\_flow()$
end if
$volume \leftarrow volume + inflow$
$outflow \leftarrow min(action, volume)$
$outgoing\_flow.set\_flow(outflow)$
$volume \leftarrow volume - outflow$
for $reward$ in $rewards$ do
    $reward.calculate\_reward()$
end for
if $volume > capacity$ then
    $terminated \leftarrow True$
end if
$observation \leftarrow \_get\_obs()$
$info \leftarrow \_get\_info()$
return $(observation, rewards, terminated, False, info)$
```

Pseudocode 2: Basic `step()` function

First, it adds the `flow` from `incoming_flow` to `inflow` if it is defined and updates `volume`. It then makes sure the `outflow` determined by `action` is not more than `volume`. It updates `volume` again before calculating rewards and checking for termination. It finally gets `observation` and `info` and returns the tuple as explained in 2.2. This most basic formulation of a WMS forms a good basis for the final RL problem formulation. It can simulate water flowing from point to point, perform water release actions and return custom defined rewards as a result. However, it is far from able to simulate any of the three case studies. Many features have to be added first.

## 4.3 Features of final RL problem

### 4.3.1 Timestep and time-dependent data

As explained in 4.1.1, each case study uses time-dependent data in their simulation. Hence, at every step, the environment should be influenced by diffent time-dependent data. In RL, timesteps are commonly used to represent discrete units of time at which the agent interacts with the environment. At each timestep, the agent is in a certain state and makes a decision that leads to transition to a different state, progressing to the next timestep. By using timesteps to index time-dependent data, any data can be turned into time-dependent data. Instead of defining `inflow` as a float, it can be defined as a list of floats. The value of time-dependent data at a certain timestep can be found at `index=timestep%data.length`. Taking the modulo allows for time-dependent data of any length and simulations of any number of timesteps. Monthly data can be represented by a list of length 12, daily data by a list of length 365. For data that is not time-dependent, a list with only a single element is used. Pseudocode 3 illustrates how this can be implemented.

```
Function current_timely_value(data, timestep)
    $index \leftarrow timestep \bmod list.length$
    return $data[index]$
```

Pseudocode 3: Indexing time-dependent data

### 4.3.2 Water update functions

Apart from inflow and outflow, the volume's mass-balance equation contains other variables such as evaporation and leakage. In two of the case studies, leakage is hard-coded into the simulation. While this may work for specific scenarios, it limits the model's flexibility and does not fit the requirements of a flexible, widely applicable model. In all of the case studies, evaporation is calculated by converting storage volume to surface area and multiplying it by an evaporation rate. Although this suffices for the three case studies, other WMSs might very well have a different method for calculating evaporation. The term "water update functions" serves as an umbrella term to generalize various factors affecting water volume, such as evaporation and leakage. When going back to the mass-balance equations discussed in 4.1.1, it makes sense to generalise to equation 5, with $\sum WUV_t$ representing the sum of all water update functions, calculated by at time $t$.

$$V_{t+1} = V_t + \sum IF_{t+1} - \sum OF_{t+1} + \sum WUFt + 1 \tag{5}$$

Nodes can be assigned multiple water update functions, which, similar to the reward functions in 4.2.2, take the current state's `info` dictionary and return a float. When updating volume, the node calculates the sum of all water update functions and adds it in the volume's mass-balance equation, as equation 5 demonstrates. Therefore, a negative value is subtracted, as should be the case for evaporation. Because the `info` provides the value of `timestep`, time-dependent water updates are also possible. This way, water update functions provide flexibility to implement any water update, such as heavy rainfall that only occurs specific months.

### 4.3.3 Release range

An essential aspect of WMSs that the most basic formulation in 4.2 ignores, is release constraints: water release can be bounded by physical, regulatory, environmental or other forms of constraints. The three case studies differ in what defines release constraints. For example, the release range of Nile dams depends on their storage volume and some Volta dams have minimum water levels. To offer the possibility of implementing any type of release range, each node is assigned a so-called "release range function". Similar to the reward function in 4.2.2, the function takes the current state's `info` dict. It returns a release range, a tuple of two floats: `min_release` and `max_release`. Pseudocode 4 shows a very simple example of a release range function, but in principle any release range function can be defined, as long as it takes an `info` dictionary and returns a tuple of two floats. These values can then be used in `clip_action(action)` to clip the RL agent's action within these bounds. This happens to also be useful for facilities that have constant release ranges or -policies. In that case, bounds can be set to identical values to ensure a certain release, regardless of the action. For instance, irrigation facilities try to consume as much of the inflow as possible, until their demand is met. Since the `info` dictionary provides access to the current state's demand and inflow, a corresponding release range can be returned. Because this allows setting a release range regardless of action, all methods can be the same for both passive and active nodes: passive nodes can pass any action and the correct amount of water will be released.

```
1: Function release_range(info):
2: $volume \leftarrow info["volume"]$
3: if $volume \leq 500$ then
4:     return $(0, 500)$
5: else
6:     return $(500, 1000)$
7: end if
```

Pseudocode 4: Example release range function

### 4.3.4 Integration in step function

Although the case studies' papers are not clear about this, each of their simulations implement integration steps to calculate volume mass-balance equations over smaller time intervals. Instead of, for instance, calculating all variables for a month only once, smaller intervals such as half hours are used to more accurately depict real-life flow of water. Variables such as `min_release`, `max_release` and `evaporation` all depend on `volume` and vice versa. This cross-dependency means that, for instance, release ranges change when volume changes. Taking into account this volume change over smaller time intervals more accurately depicts real-life mechanics. It can easily be implemented by setting `sec_per_timestep` as well as `integration_steps_per_timestep` when initialising a node. In the `step()` function, `integration()` can then be called to loop over the mass-balance equation `integration_steps_per_timestep` times. To make sure averages are used for functions that return values for whole timesteps, such as `calculate_water_update()`, these values are divided by the number of integration steps at every integration step. Pseudocode 5 shows the integration loop. As the pseudocode demonstrates, when updating volume with the mass-balance equation, `flow` ($m^3/s$) is converted to `volume` ($m^3$) by multiplying by seconds. To keep the pseudocode concise, things like updating outgoing flow values is left out.

```
1: $seconds \leftarrow sec\_per\_timestep / integration\_steps$
2: for $i$ in range($integration\_steps$) do
3:     $min\_release, max\_release \leftarrow release\_range(info)$
4:     $actual\_action \leftarrow clip\_action(min, max, action)$
5:     $volume \leftarrow volume + (inflow * seconds) - (actual\_action * seconds) + (calculate\_water\_update()/integration\_steps)$
6: end for
```

Pseudocode 5: Integration loop

### 4.3.5 Larger in-degree and out-degree

Naturally, the case studies describe water points that have an in-degree larger than one: multiple incoming edges lead water to one point. The Susquehanna River, however, requires logic that is much less trivial: an out-degree larger than one. Where multiple incoming edges can simply be added up, multiple outgoing edges are more difficult to handle.

Figure 4 shows an example of a node with an in-degree larger than one: flows from both nodes A and B come together at node C. To get the total inflow at C, the flows AC and BC are summed and added to the value of C's `inflow`.

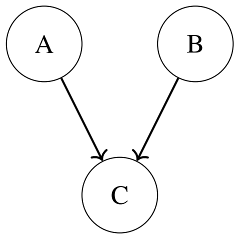

Figure 4: Node with in-degree of 2

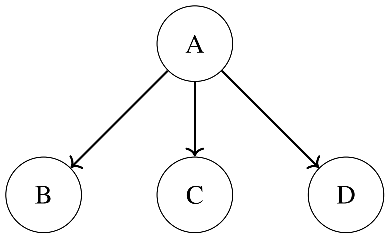

Figure 5: Node with out-degree of 3

An example of a node with an out-degree larger than one is depicted in Figure 5: three individual flows share A as source but have a unique destination. Equally dividing the outflow over all outgoing edges might seem intuitive but does not suffice. If one of three destination nodes requires half the water, it will never receive this, even if this might very well be possible in real life. Knowing that this is the case would allow for allocating constant fractions of the outflow. However, the challenge of these problems and the reason to use RL, is that this knowledge is often not available. The RL agent should be able to decide to release none or all of the water to an outflow, and the remaining water for other outflows has to be updated accordingly.

To tackle this problem, the nodes use proportional allocation to scale individual actions by their proportion to the sum of actions. First, it checks whether the sum of actions fits within the bounds of `min_release` and `max_release`. If not, it scales this action sum to clip to a bound by scaling each individual action: the proportion of an individual action is multiplied by this bound to get the scaled individual action. This way, original proportions are maintained. Pseudocode 6 shows how this can be implemented. Apart from changing how actions are handled, one more step is required: the RL agent should be able to take multiple distinct actions for one node. To achieve this, the dimension of a node's action space is increased by one when an outgoing edge is added. Increasing action space dimension will require the RL agent to take more actions.

```
1: $result \leftarrow copy(actions)$
2: $action\_sum \leftarrow sum(actions)$
3: **if** $action\_sum > max\_release$ **then**
4:     **for** $index, element$ in $actions$ **do**
5:         $proportion \leftarrow element/action\_sum$
6:         $result[index] \leftarrow proportion * max\_release$
7:     **end for**
8: **else if** $action\_sum < min\_release$ **then**
9:     **for** $index, element$ in $actions$ **do**
10:        $proportion \leftarrow element/action\_sum$
11:        $result[index] \leftarrow proportion * min\_release$
12:    **end for**
13: **end if**
14: **return** $result$
```

Pseudocode 6: Handling larger out-degree by clipping actions

#### 4.3.6 Distinction between passive and active nodes

In the case studies, there are water management facilities for which release decisions have to be made, such as dams, and ones for which that is not the case, such as irrigation districts. Nodes that do not require any decision-making require less computational overhead, as less calculations have to be made. Therefore, there has to be a clear distinction: passive and active nodes. Active nodes increase the action space of the RL problem. To ensure that passive nodes do not, the Gymnasium Environment checks each node's `passive` Boolean to decide whether to add it to its action space.

### 4.3.7 Delay

To achieve the same in the `Edge` class, it is possible to specify a `delay` and corresponding `replacement_values`. When fetching the 'current' value of `flow`, instead of indexing by `timestep`, the delayed `flow` can be found at index equal to `timestep-delay`. Replacement values are optional and are used to pass flow values when no flow data from source nodes is available yet, due to the delay. For example, at `timestep=1` with `delay=2`, index `timestep-delay` is smaller than zero and will therefore return incorrect information. A replacement value can fill that gap.

## 4.4 Verification

To verify the final implementation of a general WMS as RL problem, its results are compared to those from the Nile river case study by Sari (2022). The data from his simulation is used to model the Nile river in the implemented Gymnasium Environment. His simulation is run with actions randomly generated by a pre-set seed. The same actions are put into the Gymnasium Environment and the values of `volume` and `outflow` are compared at each timestep. Naturally, before implementing delay, the values of the dam that proceeds the delayed flow were different. Surprisingly, however, after implementing delay, the values were still different. Analysing this difference lead to an interesting conclusion: Sari not only delays the flow from Tamaniat to Hassanab, as he explains in his paper. Instead, he also delays the inflow from the Atbara catchment, which is different than the formula he provides in his paper. After incorporating this change, only a few values were slightly different. As the results in table 2 show, the differences are only as large as 0.01. This slight difference is assumed to be caused by a Floating Point Arithmetic error from Python math operations Python Software Foundation (2023).

Table 2: Differences in values for each dam at each timestep

| Year | Column | Sari's simulation | Author's simulation |
|---|---|---|---|
| 26 | HAD_volume | 70050350890.77 | 70050350890.76 |
| 72 | HAD_volume | 44545987385.75 | 44545987385.74 |
| 100 | HAD_volume | 28437697773.79 | 28437697773.8 |

Due to time constraints and lack of data for the Lower Volta river case study, the other two case studies are not subject to comparison. As this paper describes the formulation of a general WMS as an RL problem, it is widely applicable. For that reason and the fact that the features discussed in 4.3 allow for simulating all mechanics from the case studies, comparison should produce the same results.

An example is the unique larger out-degree of the Susquehanna River, which can be implemented as discussed in 4.3.5. Similarly, generalisation allows implementing unique features of the Lower Volta River: the unique e-flow rewards mentioned in 2.3.2 can simply be modelled as a node with a constant release range. The corresponding reward can be calculated by taking the `inflow` value and checking whether it fits within the range defined by the e-flow.

However, there is one exception to this: the pumped storage hydroelectric facility from the Susquehanna case study, which pumps water back, is not implemented or covered by a different generalised feature. If the decision to pump water back does not depend on a different volume than its own, and is not affected by volume changes from a different water point, the implementation described in this paper is able to simulate this mechanic. It would only require adding another edge which flows in the opposite direction. To verify this, empirical testing is required, and is recommended for future research.

# 5 Responsible Research

This section will explain how well this research adheres to responsible research standards. First, it discusses to what extent the research is reproducible. Furthermore, it explains how its accessibility contributes to future efforts of academia. Lastly, it will provide a disclaimer to avoid that the results of this research are used for the wrong cause and cause harm to people involved.

This paper gives a thorough explanation of the process and thinking that led to the final result. It elaborates on the step-by-step process: from research, analysis and formulation to implementation and the reasoning behind it. To aid with analysis, background information on relevant RL concepts is explained, to show what RL concepts should be carefully considered when doing similar research. By starting off with a more basic formulation of a WMS as RL problem, and providing its code, the more advanced steps that lead to the final result become easier to grasp. Moreover, the well-documented repository, along with an extensive README file, provide more insight and make the implementation more understandable. It thereby makes the research much more reproducible, and paves the way for continued research on the topic. Nevertheless, there are, as 6.2 explains, improvements to be made: the code lacks thorough testing and exception handling. This makes the code more prone to errors when used by others. Without deep knowledge of the code, it might be difficult to find a solution. To tackle this and give users the chance to seek support, the author's contact details are included in the README file.

The research publishes an open-source Github repository[2] which contains all the code from the research: implementations of the basic WMS as RL problem, the more advanced final result and the implementation of and comparison with the Nile river case study from Sari (2022). This greatly improves accessibility, and makes it easier to verify the results of the research. In addition, transparency is created by providing access to the Nile river data that is used to compare the paper's result with the case study's simulation.

Research on water management, where people are involved and could be affected by its results, requires a disclaimer. The results of this research should not be used to make any decisions with regards to water management in any way. The research does not argue that the provided implementation is in any way able to provide information that can be used for that cause. The research merely aims to provide an RL problem formulation of a general WMS. Its goal is to further research into water management policies, with regards to how RL can play a role in this. Wrong decisions in WMSs can have countless catastrophic effects and should be avoided at all costs.

## 6 Discussion

This section starts off by discussing a research with a slightly different approach that has the same goal. It then goes over some possible future improvements and finishes by giving some suggestions for future research contributions.

### 6.1 Comparison with top-down approach

Different from the bottom-up approach of this research, Muniak (2024) describes a top-down approach to creating an implementation of generalised WMSs as an RL problem. Instead of starting from scratch and defining core properties of WMSs, he studies the Nile river case study from Sari (2022) and removes features to further generalise his implementation. By doing so, he attempts to create a framework that is applicable to a wider range of WMSs than just the Nile river. Similar to Sari, he uses distinct classes to represent different water management facilities. For instance, `DemandDistrict`, similar to irrigation districts discussed in this paper, and `Reservoir`, similar to dams discussed in this paper, each have their own classes. These classes implement abstract classes of either `Facility` or `ControlledFacility`. Facilities are connected by a `Flow`. This is different from the formulation in 4.2.2, which describes a node as the only type of water point, with nodes connected by edges. The only distinction between nodes is whether they are active or passive, similar to facilities being controlled or uncontrolled in Muniak's framework.

The framework Muniak describes in his paper also produces the same results as Sari's simulation, and is therefore just as accurate as the RL problem implementation in this paper. Because he bases his framework on the Nile river simulation, it is much easier to set up a simulation for a similar WMS. For example, the right release range only requires inputting the right data in the correct format. There is no need to understand and build a custom "release range function", as explained in 4.3.3. Less knowledge and understanding is required to work with Muniak's framework. However, the implemenation described in this paper is applicable to a much wider range of WMSs. Take for example the water update function described in 4.3.2: implementing the Nile's evaporation is much

[2]https://github.com/jorianfaber/water-management-system-rl

less cumbersome in Muniak's framework, but the implemenation described in this paper provides the flexibility to simulate WMSs with any "water updates", such as heavy rainfall.

All in all, Muniak has done an outstanding job in generalising the Sari's Nile river simulation and formulating it as an RL problem. It paves the way for easier and more frictionless research on similar WMSs. The implementation described in this paper requires a more knowledge and understanding but provides more flexibility and is applicable to a much wider range of WMSs.

### 6.2 Future improvements

During a research with time constraints, priorities are set to achieve the best possible result. Through setting these priorities, some aspects are not worked on, leaving room for future improvement. Firstly, more exception handling and testing would make the code more robust. It would make the code much more robust and lower the chance that users encounter errors. Similarly, more thorough documentation would improve the user-experience. Specifically, Jupyter notebooks with step-by-step explanation of the repository, from the basic to more advanced RL problem, along with visuals of graph structures, would greatly improve comprehensibility. Finally, node ordering is now the responsibility of the user who initialises the Gymnasium Environment. Removing this responsibility and instead ordering nodes with an algorithm based on flow directions could prove to be useful.

### 6.3 Suggestions for future research

This research inspires some interesting questions for continued research on the topic. Analysis of reward functions in each of the case studies, which were different but similar, sparks curiosity as to which reward functions are most effective, and why. For example, the two different reward functions discussed in 4.2.2 are both aimed at minimising deficit, dictated by a demand. It would be interesting to research the effect of varying reward functions. Another, much more obvious, suggestion for future research is to evaluate the effectiveness of the implementation described in this paper on other WMSs. Starting with the Lower Volta and Susquehanna rivers, it would be useful to determine how well the RL problem implementation is able to simulate them. As the goal is a formulation and implementation that is applicable to a wide range of WMSs, it would be beneficial to find out to what extent this is the case, and what its caveats are. Lastly, it would make sense to test this RL problem implementation with actual multi-objective RL algorithms, to see if they are able to learn and make valuable progress.

## 7 Conclusion

The research aims to formulate a general WMS as a RL problem based on three distinct case studies: the Nile River, the Lower Volta River, and the Susquehanna River. By first identifying their similarities and differences, and defining core properties of these WMSs, this research takes a bottom-up approach. Based on these core properties, the RL problem formulation is implemented as a custom Gymnasium Environment. A graph structure is selected to represent a WMS: nodes to represent water points and edges to represent water flowing between points. In the RL formulation, the agent's action dictates how much water is released from a water point to the next. With a very basic WMS formulation as starting point, more features are added and key differences between case studies are handled through generalisation. The most illustrative and intriguing example is perhaps the flexibility of reward functions: any custom function can be added that takes an `info` dictionary, containing information about the current state, and returns a float.

The proposed implementation not only matches the accuracy of the studied Nile River simulation, but also offers greater flexibility and generalisation across different WMSs. To further evaluate the implementation, however, it should be applied to a larger number and wide variety of WMSs. Moreover, several RL algorithms should be used to assess the quality of the implementation with regards to RL. Nonetheless, the research demonstrates how an RL problem can effectively formulate complex WMSs with multiple conflicting objectives. The implementation's flexibility ensures that the model can be tailored to specific needs and objectives of different WMSs. Hence, the paper significantly contributes to research on WMSs in the context of RL, by providing a generalised formulation and implementation of WMSs as an RL problem.

## References

Badika, P., Raghuvanshi, A. S., and Agarwal, A. (2024). Climate change impact assessment on the hydrological response of the tawa basin for sustainable water management. *Groundwater for Sustainable Development*, 26:101249.

Felten, F., Alegre, L. N., Nowé, A., Bazzan, A. L. C., Talbi, E. G., Danoy, G., and Silva, B. C. da. (2023). A toolkit for reliable benchmarking and research in multi-objective reinforcement learning. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.

Hayes, C. F., Rădulescu, R., Bargiacchi, E., et al. (2022). A practical guide to multi-objective reinforcement learning and planning. *Auton Agent Multi-Agent Syst*, 36(26).

Muniak, K. (2024). Rl4water: Reinforcement learning environment for water management.

Owusu, A., Salazar, J. Z., Mul, M., van der Zaag, P., and Slinger, J. (2023). Quantifying the trade-offs in re-operating dams for the environment in the lower volta river. *Hydrology and Earth System Sciences*.

Python Software Foundation (2023). Floating point arithmetic: Issues and limitations. Accessed: 2024-06-22.

Sari, Y. (2022). Exploring trade-offs in reservoir operations through many objective optimisation: Case of nile river basin. Master's thesis, Delft University of Technology.

Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G. d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., and Younis, O. G. (2023). Gymnasium.

Witvliet, M. (2022). Multi-sector water allocation: The impact of nonlinear approximation network hyperparameters for multi-objective reservoir control. Master's thesis, Delft University of Technology.