Ecole Polytechnique Fédérale de Lausanne

LA Semester Project

Data Driven Supervisory Control of Microgrids

Author:
Maxime Gautier

Professor:
Giancarlo Ferrari Trecate
Supervisor:
Mustafa Sahin Turan

January 22, 2020

Contents

List of Figures

1 Introduction

Microgrids are localized groups of electrical sources and loads that can both operate connected to the traditional wide area synchronous grid, but can also disconnect to islanded mode. They then function autonomously depending on the physical and desired requirements. In order to safely operate in islanded mode, Microgrids rely on a smart controller to monitor and maintain voltages and currents in a safe range whilst ensuring correct load distribution. Different levels of control can be implemented : primary and supervisory. Primary control consists in controlling individual units whereas supervisory control consists in controlling several individuals to work together. The purpose of this project is therefore two-fold. First off, it is necessary to develop a simulator environment for microgrids in Python which implements primary control of its elements. With this simulator, Reinforcement Learning algorithms can then be applied in order to train a supervisory controller for the microgrid. Obtaining a high level controller able to keep the whole system stable whilst satisfying constraints is the second goal of the project.

More specifically, we are focusing on microgrids composed a few Distributed Generation Units (DGU) modeled as one of two types : Solar Panels and Batteries. Low-level control of DGUs similar to what can be found in [1] and [3] is implemented in simulation. The theory behind DGUs, their modes of operation and their interactions is first presented, as well as the implementation of the simulator. The assumptions and the adaptations made to the simulator to make it compatible with RL techniques are also detailed.

The concept of Reinforcement Learning is then quickly presented, followed by what adaptations were made to apply RL techniques to the microgrid simulator. The different RL algorithms used to train the agent and the results obtained are shown and explained. What follows is are comments concerning the implementation of the project, designed to inform anyone continuing this project of the working procedures.

All systems are implemented in python, all code and results can be found in the joint folder named $Final\_Model$.

2 Simulation

The first goal of this project is to build a simple but complete model of a microgrid in order to correctly simulate its behavior. The simulation consists of a network of several DGUs which interact based on the equations that can be found in the paper 'Passivity-Based Approach to voltage Stabilization in DC Microgrids with ZIP loads' [4]. The equations used for this implementation are described in detail in section 2.2. DGU and line objects with variables parameters were created, which are linked together in a network object, which can be run, reset and plotted. The network then constitutes the environment which is used to teach the RL agent. This section will present the theory used to construct the model as well as the assumptions made for the implementation.

2.1 DGU

A Distributed Generation Unit is the main element of a microgrid network and high level control of their behavior is the goal of this project, so a correct although simplified implementation is essential. Two different types of Distributed Generation Units are studied : Batteries and Solar Panels. They can each function in two different modes, which are presented in the next section. First presented here are the types of DGUs implemented, as well as the simplifications made in the model.

A DGU consists of a voltage source which represents either a PhotoVoltaic Solar Panel (PV) or a Battery, an RLC circuit and a load. Figure 1 shows the representation of DGUs the model is based on and equations 1 and 2 show how they interact.

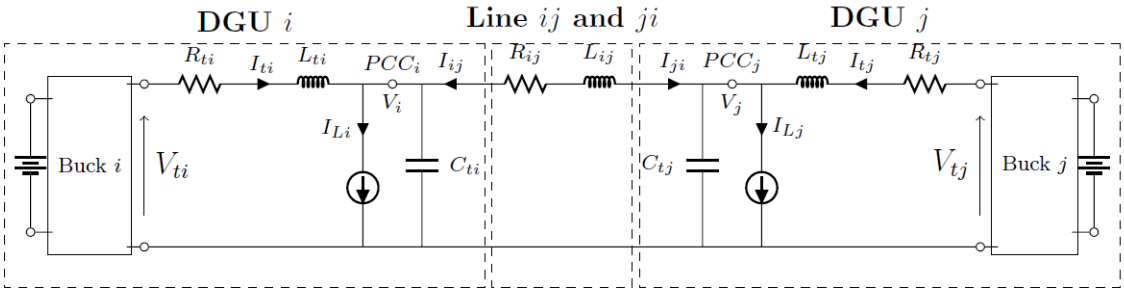


Figure 1 – ***Electrical scheme of a DC ImG composed of two radially connected DGUs with unmodeled loads.***

$$
\begin{aligned}
\text{DGU}i : & \begin{cases} \frac{dV_i}{dt} = \frac{1}{C_{ti}} I_{ti} + \frac{1}{C_{ti}} I_{ij} - \frac{1}{C_{ti}} I_{Li} \\ \frac{dI_{ti}}{dt} = -\frac{R_{ti}}{L_{ti}} I_{ti} - \frac{1}{L_{ti}} V_i + \frac{1}{L_{ti}} V_{ti} \end{cases} \\
\text{Line } ij : & \begin{cases} \frac{dI_{ij}}{dt} = \frac{1}{L_{ij}} V_j - \frac{R_{ij}}{L_{ij}} I_{ij} - \frac{1}{L_{ij}} V_i \end{cases} \\
\text{Line } ji : & \begin{cases} \frac{dI_{ji}}{dt} = \frac{1}{L_j} V_i - \frac{R_{ji}}{L_{ji}} I_{ji} - \frac{1}{L_{ji}} V_j \end{cases}
\end{aligned}
\tag{1}
$$

$$\text{DGU}j : \begin{cases} \frac{dV_j}{dt} = \frac{1}{C_{tj}} I_{tj} + \frac{1}{C_{tj}} I_{ji} - \frac{1}{C_{tj}} I_{Lj} \\ \frac{dI_{tj}}{dt} = -\frac{R_{tj}}{L_{tj}} I_{tj} - \frac{1}{L_{tj}} V_j + \frac{1}{L_{tj}} V_{tj} \end{cases} \tag{2}$$

Several simplifications were made in this model since a simple model is sufficient to use as an environment in RL settings. The implementation is however adjustable and a more realistic model can easily be build atop this simpler one. One simplification is the use of constant loads in DGUs instead of variable ones. Resistance, Inductance and Capacity values for DGUs are also constant and each DGU was initialized with random values for those parameters in the following ranges :

Elements	PV	Battery	Line
Resitance R_i $[\Omega]$	$[0.1; 1]$	$[0.1; 1]$	$[4; 8] \times 10^{-2}$
Inductance L_i $[mH]$	$[1.5; 2.5]$	$[1.5; 2.5]$	$[1.5; 2.5] \times 10^{-3}$
Capacity C_i $[mF]$	$[2; 3]$	$[2; 3]$	–
Load I_l $[A]$	$[5; 10]$	$[5; 10]$	–
Capacity C_{bat} $[Ah]$	-	3.5	–

Table 1 – ***Ranges of DGU Components***

As can be seen in Figure 1, DGUs in previous theses[1],[3] used a buck converter in order to regulate the voltage coming from the supply source. In order to simplify our model, we modeled the supply source and buck converter as a single voltage source which provides voltage and current labelled in this report as V_s and I_s.

The way the voltage Source is modeled depends on if a DGU is a PV module or a Battery. For PV modules, the voltage source is constant when it is imposed and otherwise adjusts to the rest of the network. In a more realistic model, it should vary with time to represent solar irradiance variation, but this is not necessary for a first iteration of the model.

For Batteries, the source voltage V_s varies with the State of Charge (SoC). The SoC is measured in percentage and indicates the level of charge of a battery, from 0 to 100. Batteries have a longer life cycle when the SoC is kept in a certain range, between 20% and 80%. In order to preserve the microgrid and heighten its lifetime, one of the main purposes of a high-level controller is therefore to keep the SoC in this range. SoC variation depends on the battery's nominal capacity : the bigger the capacity, the smaller the SoC variation. The model used here for the SoC variation is the Coulomb Counting method [2] :

$$SoC(t) = SoC(t-1) + \frac{I(t)}{Q} \Delta t \tag{3}$$

Where $I(t)$ is the discharging current, Q the nominal capacity of the battery and Δt the timestep of the model. This is an approximation of the real SoC variation and does not take

into account temperature, battery history and cycle life.

Another important simplification was made : the nominal capacity of the batteries is set to 3.5 Ah, which is only slightly bigger than an AA battery. This is to avoid having to simulate long periods of time to observe an SoC variation. Indeed, the timestep of the model is quite small and therefore the network simulation takes about one second of computation for one second of simulation. A bigger battery capacity would mean having to simulate several hours to see any significant SoC variation. In order to train well, an RL agent needs to explore as many different states as possible and therefore observe and test many different combinations of actions on different SoCs. This is not possible in reasonable time with a large capacity and its size does not affect the behavior of the agent, only the speed at which it can explore and therefore learn. Since the purpose of the model here is not simply to be realistic, but to train an RL agent in reasonable time, a tradeoff must be made between a large, more accurate battery capacity and a small battery, more efficient to train a high level controller. The value corresponding to the best observed behavior in this implementation is therefore 3.5 Ah, although this could be changed were the Network implementation updated, as detailed in section 4.

Evidently, the SoC influences the behavior of the battery and it stops supplying power when it reaches 0%. The best representation of the influence of the battery's SoC on its output voltage is the Sheperd Model [6][7], which can be seen in Figure 2. Keep in mind that the voltage shown here is the output voltage of the battery, and therefore the source voltage of the DGU : V_s.

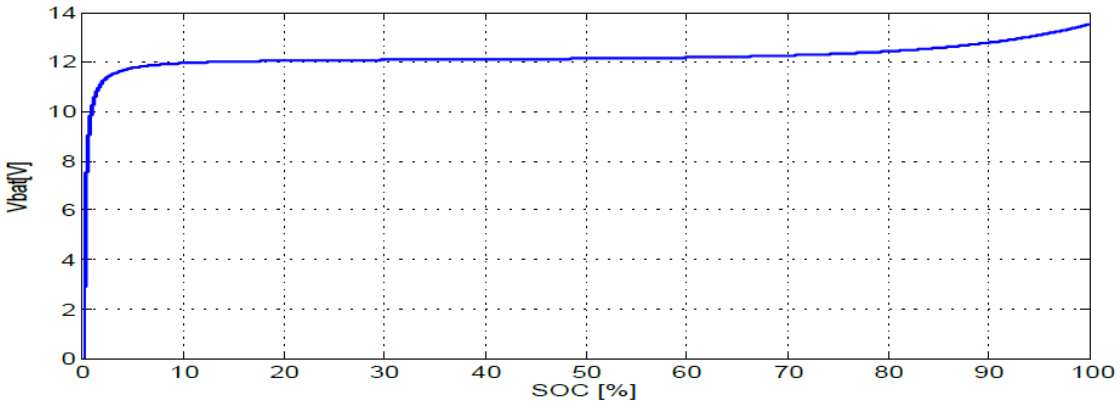


Figure 2 – ***Variation of Voltage across Battery depending on State of Charge following Sheperd's model***

Due to the non-linear nature of this model, another simplification was made to implement this more easily. The variation of $Vs(SoC)$ in the model therefore follows a function decomposed in 3 linear sections as shown in Figure 3. The threshold settings are set to mimick Sheperd's model, but they can be easily changed if necessary.

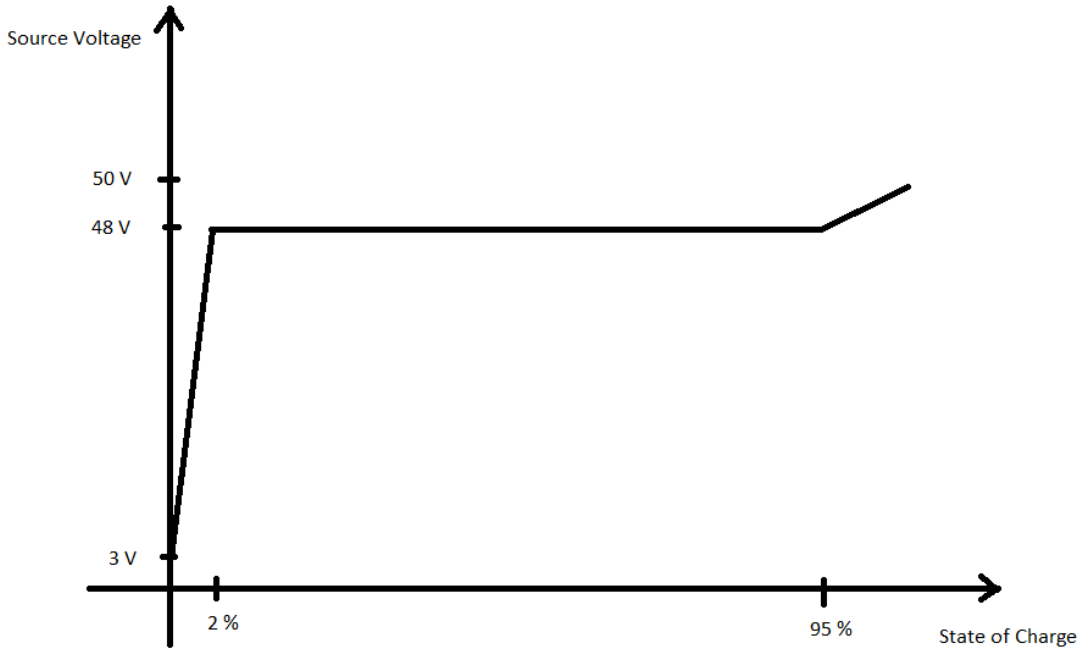


Figure 3 – ***Implemented variation of Voltage across Battery depending on State of Charge***

DGUs are connected to each others through current lines as can be seen in Figure 1 and physics therefore dictates their behaviors and interactions. In the following section, the equations used to compute the behavior of interacting DGUs are presented.

2.2 DGU Modes

Two modes of operations for each DGU were implemented. They are loosely based on the modes used in Davide Riccardi [1] and Giuseppe Tagliaferri's [3] theses but adapted to this more simplified model and so will be presented in detail here. The equations used for DGU behavior are taken from the paper 'Passivity-Based Approach to voltage Stabilization in DC Microgrids with ZIP loads' [4] and adapted to a discrete system using Euler's explicit forward method with a timestep T of $2 * 10^{-5}$ seconds. Any higher timesteps resulted in an unstable system. The explicit Euler method calculates each next step as such :

$$y_{n+1} = y_n + T * \frac{dy}{dt} \tag{4}$$

2.2.1 Free mode

In free mode, no control algorithm is applied to the DGU but the source voltage V_s is imposed. Although different, this mode is similar to the Maximum Power Point Tracking (MPPT)

and Charge modes used in Davide Riccardi[1] and Giuseppe Tagliaferri's[3] theses and so those names are used as labels. The idea of those modes is to control the power input/output of the DGU through voltage imposition. The same principle is used here but simplified.

The idea of the MPPT is that an imposition of the voltage drop across the PV source implies a specific current flow and therefore a specific power. The Maximum Power Point is the voltage for which the power is maximum and it varies with solar irradiance. However since our model does not take solar irradiance variation into account, the voltage used for MPPT mode is constant.

The same principle applies in Charge mode : apply a voltage to the terminals of the battery to have it charge at a constant power. These modes are usually imposed through the buck converter, but since in our model, the supply source and buck converter are represented by a voltage source, we simply change the voltage of the source (V_s).

Since our bus voltage reference is set at 48V, the voltage imposed for MPPT mode is 48.5 V to ensure power transmission from the PV module to the network and is labeled V_{MPPT}. The voltage for Charge mode is 40V to ensure power flow to the battery and is labeled V_{Charge}. When simply imposing the source voltage, the components therefore interact under the following equations :

$$\sum_{[i]}^{DGU} : \begin{cases} V_{i,n+1} = V_{i,n} + T * (I_{i,n} - I_{load} + I_{net,n}) \dfrac{1}{C_i} \\ \\ I_{i,n+1} = I_{i,n} + T * (-V_i - R_i I_i + V_s) \dfrac{1}{L_i} \end{cases} \tag{5}$$

with

$$I_{net,n} = \sum_{k}^{lines} \alpha_k \times I_{line,k,n}$$

I_{net} is the net current injected into the DGU and therefore the sum of the line currents attached to the DGU. The parameter α depends on the arbitrary direction of current : if the current is injected in the DGU $\alpha = 1$, else if the current is exiting the DGU $\alpha = -1$.

I_{load} is the current load of the DGU, V_i is the voltage at the node connecting the DGU to the network (PCC), I_s is the current at the supply source and V_s is the imposed source voltage.

2.2.2 PnP mode

The other mode is based on the Plug-and-Play mode of Davide Riccardi[1] and Giuseppe Tagliaferri's[3] theses, which imposes the bus reference voltage to the PCC of the DGU. Controlling the voltage at the PCC ensures it does not increase beyond a critical level, damaging the con-

nected loads. In order to control the voltage at PCC, we steer the error $e_{[i]}(t) = V_{ref,i}(t) - V_i(t)$ to 0 as $t \to \infty$. For this purpose each DGU is augmented with an integrator

$$\frac{dv_i}{dt} = e_{[i]}(t) = V_{ref,i}(t) - V_i(t) \tag{6}$$

and equipped with a state-feedback controller

$$C_{[i]} : V_s(t) = K_{[i]} x_{[i]}(t) \tag{7}$$

where $x_{[i]}(t) = [V_i \ I_s \ v_i]^T \in \Re^3$ is the augmented state of the DGU and $K_{[i]} = [k_{1,i} \ k_{2,i} \ k_{3,i}] \in \Re^{1\times 3}$ is the feedback gain. Controller $C_{[i]}$ therefore defines a PI regulator as can be seen in Figure 4.

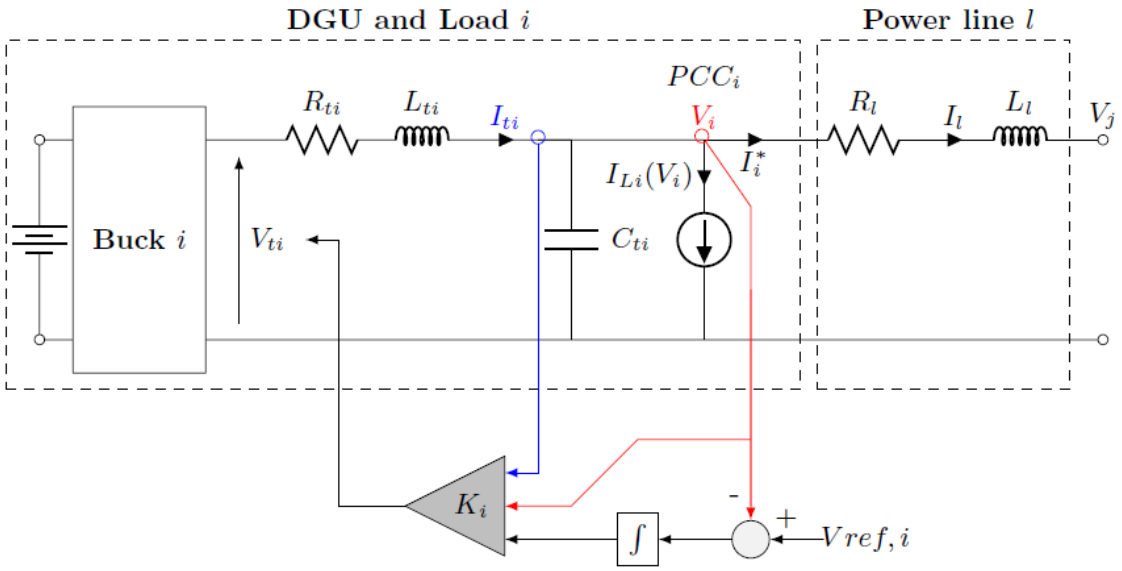


Figure 4 – ***Electric Scheme of ith DGU along with load, connecting line(s), and local PnP voltage controller.***

A DGU in PnP mode's behavior is therefore modeled using the following equations, derived from (5)

$$\sum_{[i]}^{DGU} : \begin{cases} V_{i,n+1} = V_{i,n} + T * (I_{i,n} - I_{load} + I_{net,n}) \dfrac{1}{C_i} \\ I_{i,n+1} = I_{i,n} + T * (\alpha_i V_{i,n} + \beta_i R_i I_{i,n} + \gamma_i V_{s,n}) \\ v_{i,n+1} = v_{i,n} + T * (-V_{i,n} + V_{ref,i,n}) \end{cases} \tag{8}$$

where

$$\alpha_i = \frac{k_{1,i} - 1}{L_i}, \beta_i = \frac{k_{2,i} - R_i}{L_i}, \gamma_i = \frac{k_{3,i}}{L_i}, \tag{9}$$

In order to ensure local passivity as demonstrated in the paper 'A Passivity-Based Approach to voltage Stabilization in DC Microgrids with ZIP loads'[4] the feedback gains $k_{1,i}k_{2,i}k_{3,i}$ belong to the set

$$Z_{[i]} = \left\{ \begin{array}{l} k_{[1,i]} < 1, \\ k_{[2,i]} < R_i, \\ 0 < k_{[3,i]} < \dfrac{1}{L_i}(k_{1,i} - 1)(k_{2,i} - R_i) \end{array} \right\} \tag{10}$$

In order fine tune the gains so the voltage goes directly to the V_{ref} value with the smallest overshoot, the correct gains must be determined. To easily find the correct gains for all DGUs simultaneously, coefficients were applied to the positive range of each gain as defined by (10) in order to determine the gain values. This sets equivalent gains for all DGUs, even if their intrinsic RLC values differ. The coefficients applied to each gain respectively are $c_1 = -1.8$, $c_2 = -15$ and $c_3 = 0.08$. These coefficients are used to determine the gain values a seen in equations 11. For a DGU with a resistance and inductance of respectively 0.5 Ω and $2e^{-3}$ H, this makes the gains $K_1 = -1.8$, $K_2 = -7.5$ and $K_3 = 896$.

$$\begin{array}{l} k_{1,i} = c_1 * 1 \\ k_{2,i} = c_2 * R_i \\ k_{3,i} = c_3 * \dfrac{1}{L_i}(k_{1,i} - 1)(k_{2,i} - R_i) \end{array} \tag{11}$$

2.3 Lines

In order to connect DGUs, lines were also implemented. They are composed of a resistance and inductance within the ranges shown in Table 1 and interact under the following equations as seen in [4] :

$$\sum_{[l]}^{Line} : I_{l,n+1} = I_{l,n} + T * (-R_l I_{l,n} + V_{in,n} - V_{out,n}) \frac{1}{L_l} \tag{12}$$

$$\tag{13}$$

where I_l is the line current, Rl and L_l the line components and V_{in} and V_{out} the voltage at the PCC of the line. The distinction between the voltage going in and out is only necessary in order to set the sign of said voltage. The same is true for the computing of I_{net} in DGUs, where the sign of the current added to I_{net} depends on the direction of the current (injected is positive, outgoing is negative)

In order to compute all systems correctly, an arbitrary direction of current is set and kept during the simulation for each line, thus determining the signs for I_{net} and I_l computations. The

arbitrary direction of current chosen is that all current flows from smaller index DGUs to larger index DGUs, as can be seen in Figure 6.

2.3.1 Bumpless Transfer

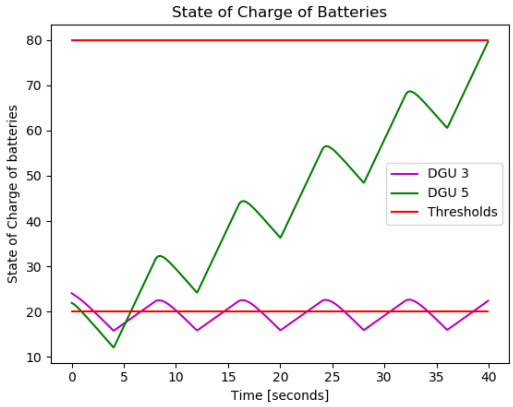
In order to have smooth transfer between modes, a strategy based on Bumpless Transfer was implemented. It consists in using values from the previous mode to determine the new mode's setpoint when switching modes, thus avoiding a jump in values. In PnP mode, the setpoint is tracked through the integrator, and so in order to avoid a jump in values, when switching to PnP mode, the integrator is set as follows

$$v_i = (V_s - (k_1 * V + k_2 * I_i)) * \frac{1}{k_3} \tag{14}$$

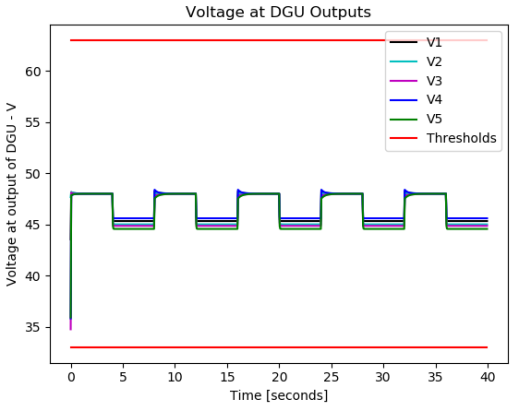
where V_s is the previous source voltage, V is the voltage at the PCC of the DGU, I_i the current at the source and k_i the PI gains.

In free mode, there is no setpoint to track, however the source voltage is imposed to the system. The imposed source voltages are constant values that depend on the type of the DGU. In order to avoid a jump in this value when switching from PnP mode, we first use the V_s value set by the PI controller when calculating the source current. Then, over several iterations, the V_s gradually moves to the desired constant value, ensuring a smooth transition. In this manner the voltage does not abruptly jump from one value to the other when changing modes.

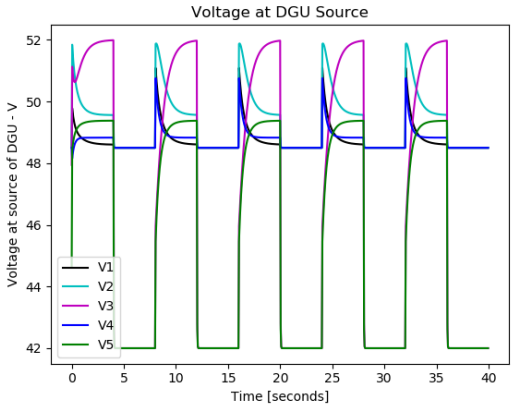
An example of bumpless transfer is shown in Figure 5 where the the mode of all DGUs is changed every two seconds. This figure shows the various outputs of the network, of which several things can be noted. Firstly, the red lines in Figures 5a and 5b represent the thresholds of the range those values should be kept in. We can see in figure 5f when the modes are switched and corresponding voltages at PCC, which transitions smoothly between modes.



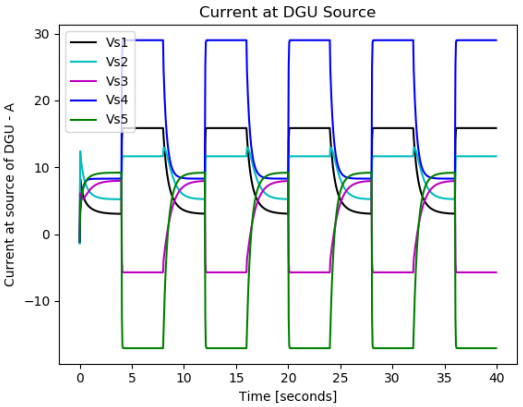
(a) States of Charge



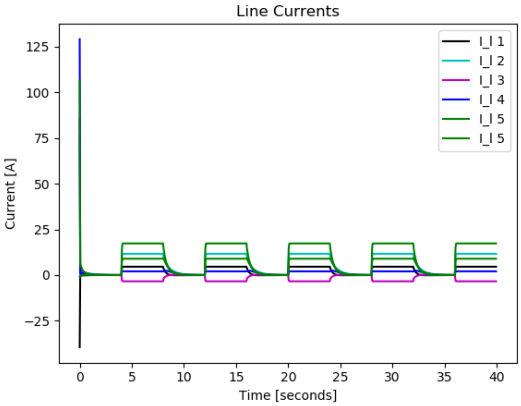
(b) Voltages at PCC



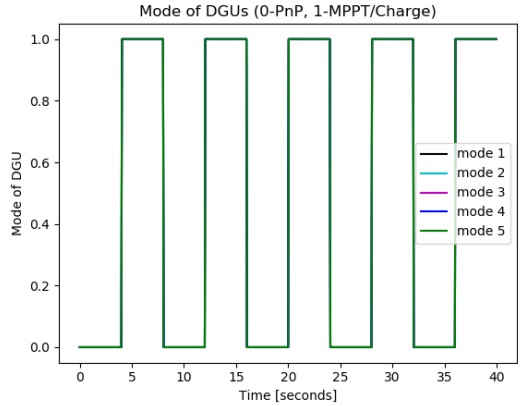
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) DGU Modes

Figure 5 – ***Demonstration of Bumpless Transfer when switching between modes***

2.4 Network implementation

A microgrid is a communicating network of DGUs and so an object called Network was implemented to connect the lines and DGUs and observe their behavior. The network that was implemented and used in all simulations can be seen in Figure 6. As mentioned previously, an arbitrary direction of current is required for the system to work correctly. The chosen direction can be seen in Figure 6 and corresponds to current flowing from small index DGU to higher index DGUs.

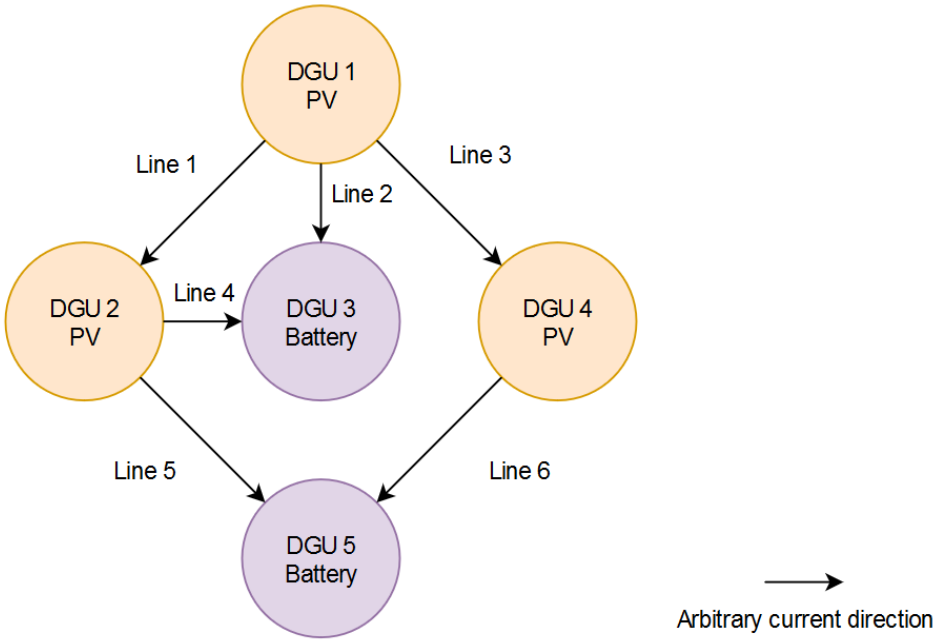


Figure 6 – ***Network Configuration***

The network object was updated and adapted to an openAI gym environment. This is a library designed to test out various environments and algorithms for Reinforcement Learning. Many RL python libraries use openAI gym environments or at least environments structured in a similar fashion. Therefore, in order to test new RL algorithms more easily, the network was adapted to be used as a gym environment. The simulation is run using the *step* function, which executes one iteration of the system. The network can also be reset to random values using the *reset* function. *render* and *close* functions were not implemented as they are not necessary to run a simulation and in fact slow down computation.

The Network function was also re-worked to be able to create any networks. It can now be initialized simply by calling the object and specifying the type of each DGU in one list and connections between each DGU and the connections in another. However due to the addition of for loops to handle any network type, the simulation is considerably slowed, taking about 5 minutes for 1 second of computation, instead of the previous 1:1 ratio. A recommendation

for the creation of different networks would be to either use the older version and change the network manually or improve computations in the latest version. Furthermore, most of the other functions were hard coded specifically for the network configuration presented in Figure 6 and it is therefore recommended to use them with the previous network implementation. This new function can however easily be used for new algorithm implementations.

3 Reinforcement Learning

The second goal of this project is to create a supervisory controller for microgrids using Reinforcement Learning techniques. For this purpose, a model of a microgrid as described in section 2 was implemented in python. Several RL algorithms were then tested to obtain a policy capable of maintaining the microgrid in the desired operating conditions. In this section, the concept of RL is presented, followed by the design of several parameters necessary to apply RL algorithms to our model. Finally, the different algorithms and their results are presented.

3.1 Basic Concept

Reinforcement Learning is an area of machine learning concerned with what actions an agent should take in an environment in order to maximize its reward. At each step, an agent chooses an action which modifies its state and receives a corresponding reward, as is represented in Figure 7.

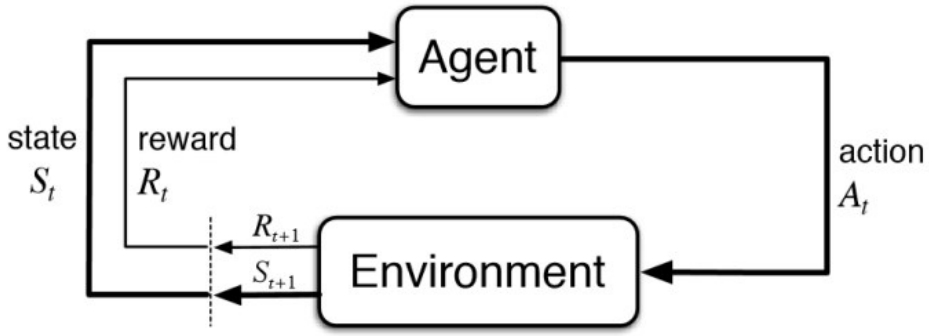


Figure 7 – ***Reinforcement Learning Concept***

Reinforcement Learning[1] environments are usually defined in the form of a Markov Decision Process (MDP). An MDP is a 4-tuple (S, A, P_a, R_a) where :

- S is a finite set of states
- A is a finite set of actions
- $P_a(s, s')$ is the probability that action a in state s will lead to state s'

[1]Explanation sampled from [8]

- $R_a(s, s')$ is the immediate reward after transitioning from state s to state s' due to action a

The core problem of an MDP is to find a policy for the controller, a function π which specifies the action $\pi(s)$ for each state s. To create an efficient controller, the idea is to have a policy which maximizes some cumulative function of the immediate reward for each state, typically some discounted sum over a potentially infinite horizon. This is called the value function $V_\pi(s)$ and is defined as the expected return starting with state s_0 and following the policy π. Both the Value Function (VF) and Policy are usually updated at each step of the RL algorithm when training, until it either stops updating or the maximum amount of training steps as been reached. The updates differ with the algorithms, but it usually follows rules similar to equations 15 and 16 .

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi(s)}(s, s') + \gamma * V(s')) \tag{15}$$

$$\pi(s) := \text{argmax}_a \left\{ \sum_{s'} P(s'|s, a)(R(s'|s, a) + \gamma * V(s')) \right\} \tag{16}$$

where γ is the discount factor, a fixed parameter which determines the importance of future rewards. Value Functions can also be labelled using both state and action and is then referred to as a Q-function. Values are then defined for each state-action pair. Although this is the type of function implemented, the term VF will be used throughout this report for clarity.

The principle of RL is to explore as many different states as possible in an environment in order to update the Q-Function of the agent. This function determines a value from the reward of each state/action combination. This must be done for as many states as possible, ideally all of them, to obtain an optimal VF. Once the VF for an agent is determined for many states, it can be used to create a policy. The policy tells the agent which action is best for a certain state. To create a policy from a value function, one simply compares the value of different actions for a given state and the resulting best action corresponds to the policy, as can be seen in (16).

3.2 Application to Microgrid Environment

In order to adapt RL techniques to our model, several parameters need to be defined. In our case, the environment is defined by the microgrid model presented in section 2. The agent is the high level controller operating the network and its goal is to keep the State of Charge and voltages inside the desired range. Its state is defined by the some outputs from the Network and its action is the choice of the low-level control mode of each DGU. In our model, there are no uncertainties concerning the outcome of an action, therefore all probabilities as defined in an MDP are set to 1 when transitioning to a new state, and 0 otherwise, as seen in (17). More explanations of the parameters modified for the Microgrid environment are detailed below.

$$\begin{aligned} P_{\pi(s)}(s, s') &= 1 \quad for\ s' = s_{next} \\ P_{\pi(s)}(s, s') &= 0 \quad for\ s' = s_{next} \end{aligned} \tag{17}$$

3.2.1 State/action Representation

RL problems usually operate in a discrete state-space environment, and limiting the number of states greatly reduces the computation time necessary to explore all states and compute a correct value function. Our state-space is however continuous and should ideally include all varying values from each DGU in order to create a well-informed controller. To simplify matters for the first iteration of RL algorithms, a reduced state form was chosen. It was designed to be sufficient whilst staying as simple as possible. The state chosen consists of the voltage at PCC (V) values for each DGU as well as the state of charge (SoC) of the two batteries. This state is used both to teach the agent and compute the reward. Since our main goal is to ensure the microgrid stays in a safe voltage range and that batteries stay in an optimal SoC range, this state should be sufficient to teach an agent.

The action representation chosen is binary, since only two options are available for each DGU. The action is simply an array where the chosen mode for each DGU is listed in the form of a 0 (PnP mode) or 1 (free mode).

A summary of the representations used for RL can be seen in (18)

$$\begin{aligned} \text{state} &:= [V_1, V_2, ..., V_n, SoC_1, SoC_2, ..., SoC_b] \\ \text{action} &:= [mode_1, mode_2, ..., mode_n] \end{aligned} \tag{18}$$

where n is the number of DGU and b the number of batteries.

3.2.2 Reward

The reward is computed using the state at each step, it is therefore based on the values of all output voltages V_i and the batteries' state of charges SoC_i. The desired range for those two prameters is shown in (19). It is calculated in the same manner for all values, but the weight for the SoCs is higher since the voltages usually stay in a safe range thanks to the low-level controller, and therefore the focus is on teaching the agent to maintain the SoC in their desired range.

$$\begin{aligned} \text{State of Charge range} &:= [\,20\,\%; 80\,\%\,] \\ \text{Voltage range} &:= [\,33\,V; 63\,V\,] \end{aligned} \tag{19}$$

For this purpose, the reward implemented is negative outside of the desired range and positive inside. Outside the range, it grows linearly bigger as the distance from the range increases. Inside

the range, it is inversely proportional to the distance to an 'optimal' point (60% for SoC and 48V for V). This is not an exactly correct implementation, as there is no optimal point for the state of charge, but this allows for a varying reward inside the desired range and should guide the agent to aim for the center of the range and stay stable. A representation of the reward for both State of Charge and voltage can be seen in Figure 8

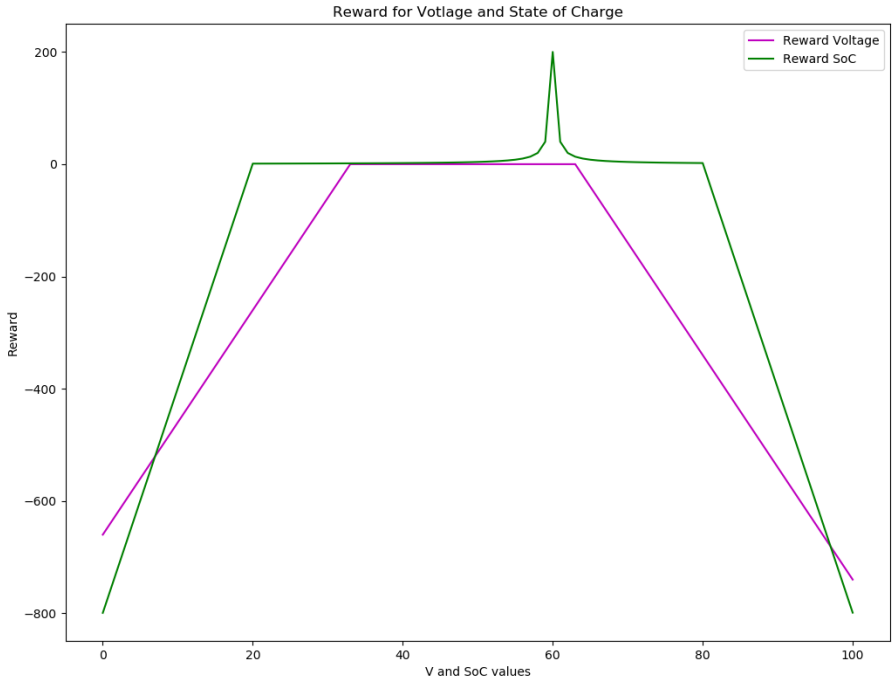


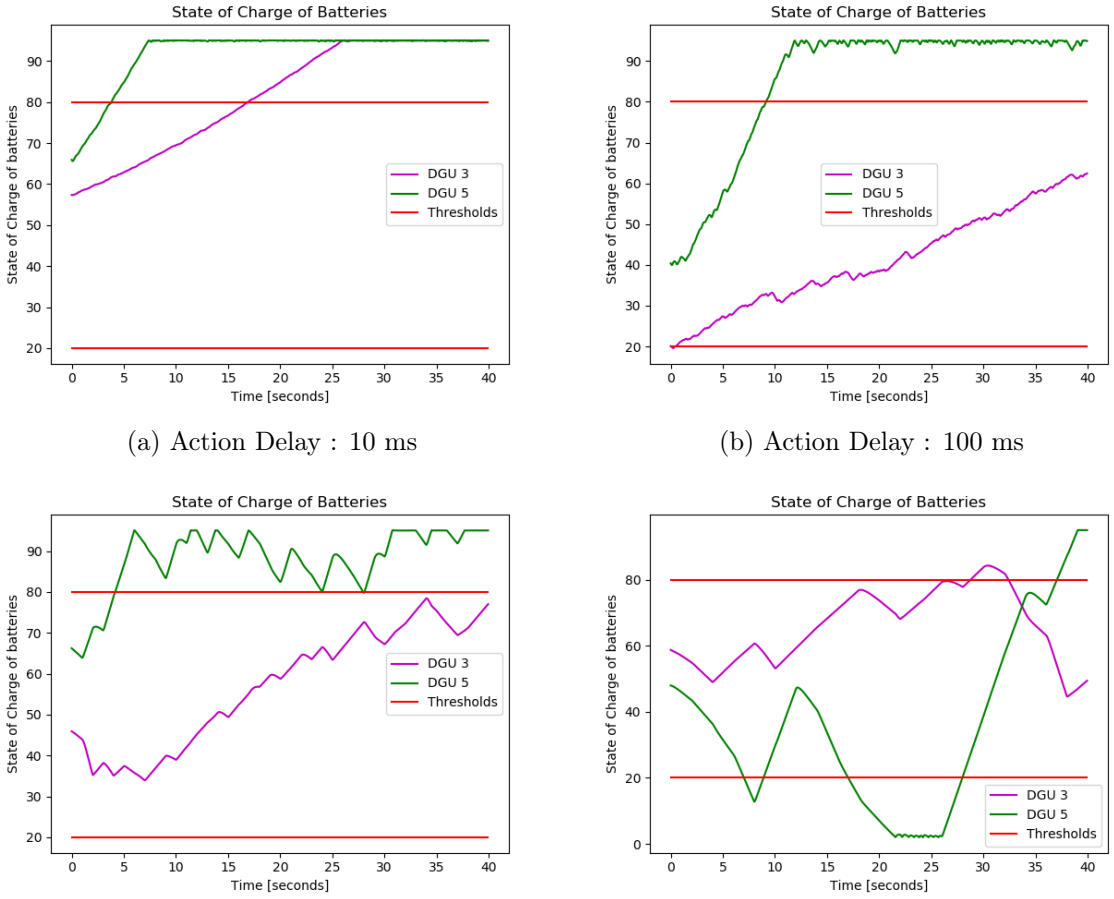
Figure 8 – ***Reward for Voltage and State of Charge***

3.2.3 Action Delay

Another key parameter that requires tuning before applying RL to the microgrid environment is the action delay. This defines the amount of time between each action/input by the high-level controller. This parameter should ideally be as low as possible so the agent can update its control constantly. However, many experiments were run and it was noticed that agents learned better when the action delay wasn't too low. This can be seen in Figure 9 where the network was run without any controller but instead by setting random modes (aka choosing random actions) at different frequencies, thus simulating an exploring RL agent. It becomes evident seeing the figures that when changing modes too often, the SoC of the batteries tend to fill up and stay at 100% . This is counter productive to train an RL agent : when training, the agent tries new actions as often as possible and if every set of actions guide him to the same state, it will never learn anything.

To learn correctly, the agent must be able to execute an action, observe the new and different state it is in and compute its reward accordingly. Two already set parameters play against this : the small timestep and the batteries' nominal capacities. Due to those parameters, when running

one step, the state the agent observes is barely different from the previous one. Therefore, in order to compute a appropriate reward and correctly estimate the impact of a set of modes on the network, the agent must keep the same action for many steps in a row. This delay between each new action is called the action delay. Its value varies depending on the RL algorithm used, since they do not all explore in the same way, but should be kept above 5000 steps (100 ms).



(a) Action Delay : 10 ms

(b) Action Delay : 100 ms

(c) Action Delay : 1 s

(d) Action Delay : 2 s

Figure 9 – ***State of Charge variation for different action frequencies***

3.3 Application to Continuous state-space

As mentioned previously, RL is usually applied to discreet environments with a discreet state-space. That way, the Value Function is simply represented as an array giving a value for each state. The smaller the state-space, the quicker the Value Function is filled and the quicker the agent will learn and behave correctly.

In our case, the state-space is both large and continuous as it is defined by the voltages and state of charges of each DGUs. Therefore, in order to run RL algorithms, we require an approximation for the value function. The goal of a value function approximation is to obtain values for as many states as possible without having to actually explore all the states, which would be

impossible in our scenario. This way, a value function can be computed in a reasonable amount of time even for a very large state-space. Several methods for value function approximation exist and are presented below.

3.3.1 State aggregation

The simplest way to approximate states is to aggregate them so neighbor states are characterized by the same values. This technique can drastically reduce the number of states present in a state-space, but not without creating drawbacks. Indeed, aggregating states in this way creates less states but also a less precise value function, since it will be impossible to know to which exact state belongs each value. Values used will be approximate and this would cause issues in situations where neighboring states have different rewards, for example at the boundary of the desired range. Furthermore, due to the large range of distinct possible states present in our scenario, a state aggregation that correctly trades off precision and computational efficiency is difficult to implement.

For those reasons, although this method was first applied with no success, it was quickly dropped in favor of more efficient methods.

3.3.2 Tile Coding

The Tile Coding approximation is an extended form of state aggregation, much more efficient in terms of both mapping and computational efficiency. In tile coding the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a tiling, and each element of the partition is called a tile. An example of this can be seen in Figure 10, where the white point will be represented with exactly one value per tile[2]. A more detailed explanation of tile coding can be found online or in Richard Sutton's book 'Reinforcement Learning - An introduction'[8].

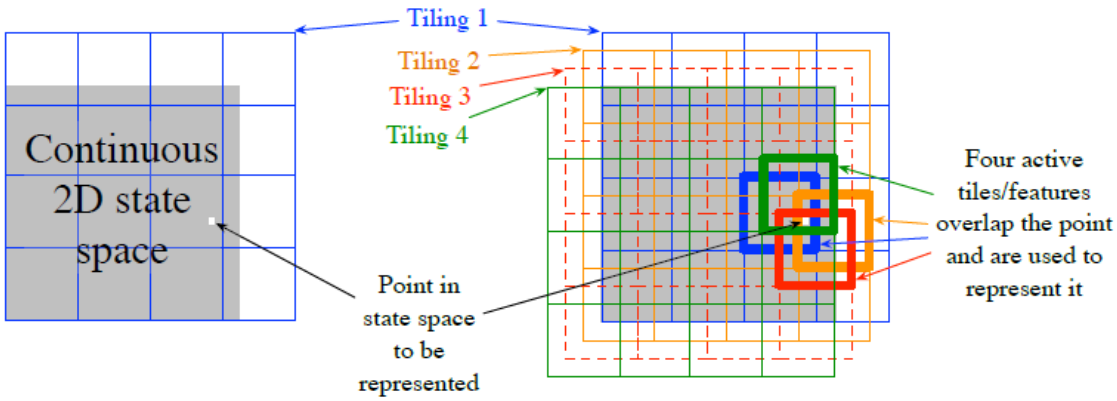


Figure 10 – ***Tile Coding Example***

[2]tile coding requires *distinct* integers for action mapping, be careful to convert action list to an action list of distinct ints before calling tiles

The parameters used for the results presented in section 3.5 are the following :

- **MaxSize** $= 32'768$ is the maximum number of tiles
- **numOfTilings** $= 4$ is the number of tilings per state value
- **SoC range** $= [0; 100]$ the range of possible SoC values
- **V range** $= [0; 120]$ the range of possible V values
- **Tiles** $= 5$ the number of tiles per tiling
- **Width of a Tile** $= 20$ for States of Charge and 24 for voltages
- **Resolution** $= 5\%$ for States of Charge and 6V for voltages

The resolution is given by $\frac{width\ of\ a\ tile}{number\ of\ tilings}$, with the width of a tile being $\frac{value\ range}{number\ of\ tiles}$.

3.3.3 Artificial Neural Networks

Another popular method for value function approximation is to use an Artificial Neural Network (ANN). This a powerful type of non-linear function approximator which has been frequently used in RL development. Of course, the efficiency and complexity of an ANN is determined by its number of hidden layers, input units and output units. A representation of a simple feedforward artificial neural network can be seen in Figure 11.

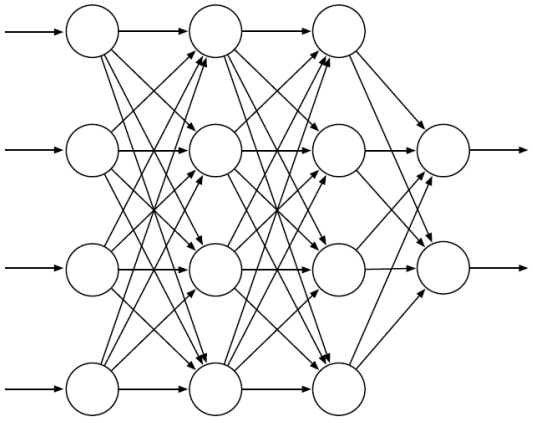


Figure 11 – ***Generic Feedforward ANN with 4 input units, two output units and two hidden layers***

Each unit (circle in Figure [8] computes a weighted sum of its input signals and applies to the result a non-linear function called the activation function. Several activation functions can be used, such as a sigmoid, step function or variants of a linear function. The parameters of the neural net determines its efficiency, both in term of computation time and results. Neural Networks have been increasingly used as function approximators in a branch of RL called Deep Reinforcement Learning which can yield powerful controllers for continuous state-space.

3.4 Algorithms

Several algorithms were tested in order to try and train an agent for autonomous high-level control of a microgrid. Our situation here is non-episodic, with a continuous action and therefore no termination state. The first algorithm for this type of task presented in 'Reinforcement Learning - An introduction'[8] is the differential semi-gradient SARSA, which is the first algorithm implemented. The second is the Least-squares approximate Q-iteration for deterministic MDPs, which is the first model-based algorithm using a parametric function approximator presented in 'Reinforcement Learning and Dynamic Programming using Function Approximators'[9] . Both those algorithms were adapted for the microgrid environment and use the Tile Coding for Value Function approximation.

The last algorithm implemented is the Normalized Advantage Function, from the paper 'Continuous Deep Q-Learning with Model-based Acceleration'[5]. It was designed as a deep Q-learning algorithm for continuing tasks, which ideally suits this project. The implementation was directly taken from the keras-rl repository and therefore uses keras neural networks for value function approximation.

3.4.1 Differential Semi-gradient SARSA

This algorithm, labelled SARSA, is presented in Figure 12. The goal of a SARSA algorithm is to tradeoff between exploration and exploitation using the ϵ parameter. The agent will explore by choosing a random action with a probability ϵ and exploit the already constructed VF with a probability of $(1 - \epsilon)$. This is the ϵ-greedy policy.

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Figure 12 – ***Differential Semi-gradient SARSA***

This algorithm is designed for continuous action and therefore does not use a discount factor. As proven in Sutton's book[8] (p 254), a discount factor is useless when approximating continuing tasks. Instead, this algorithm uses an average reward to compute the reward at each step, thus

ensuring data collected from all previous states is accounted for when calculating new values. The features of the values function approximation are then updated based on how far from the average the current reward is. This algorithm only resets to a random state at the start of an episode, so in order to explore as much of the state space as possible and therefore fill the Value Function, it is better to run many short episodes of a few minutes rather than few long episodes.

Here, the implementation of the action delay is quite simple : when taking an action, the algorithm waits for an amount of time corresponding to the action delay before calculating the average reward and updating the Value Function. In order to improve the update, the 'step' reward is actually the averaged reward over the action delay.

3.4.2 Least-Squares approximate Q-iteration for deterministic MDPS

This algorithm, labelled Qfit, is presented in 'Reinforcement Learning and Dynamic Programming using Function Approximators'[9] as the first algorithm using parametric function approximation and is shown in Figure 13. The idea of approximate Q-iteration is to explore only a subset of random states, and use the computed features of the approximate value function to approximate the rest of features through a least-squares regression. This is then repeated several times to obtain an accurate value function. In order to ensure optimal approximation of the features through regression, a regression function for VF approximation using tile coding was created. It makes a pipeline model with polynomial transformation and LASSO regression with cross-validation, ensuring a robust regression.

Another modification was the addition of a delay when starting at a random state. The network implemented is prone to instabilities in the first 0,2 seconds after a random initialization, due to values adapting to each other. Therefore, after a random state for the network is set, there is a delay of 0,2 seconds. After which, the best action for this state is chosen, executed for the duration of the action delay and only then is the VF updated.

ALGORITHM 3.1 Least-squares approximate Q-iteration for deterministic MDPs.

Input: dynamics f, reward function ρ, discount factor γ,
approximation mapping F, samples $\{(x_{l_s}, u_{l_s}) \mid l_s = 1, \ldots, n_s\}$

1: initialize parameter vector, e.g., $\theta_0 \leftarrow 0$
2: **repeat** at every iteration $\ell = 0, 1, 2, \ldots$
3: **for** $l_s = 1, \ldots, n_s$ **do**
4: $Q^{\ddagger}_{\ell+1}(x_{l_s}, u_{l_s}) \leftarrow \rho(x_{l_s}, u_{l_s}) + \gamma \max_{u'}[F(\theta_\ell)](f(x_{l_s}, u_{l_s}), u')$
5: **end for**
6: $\theta_{\ell+1} \leftarrow \theta^{\ddagger}$, where $\theta^{\ddagger} \in \arg\min_\theta \sum_{l_s=1}^{n_s} \left(Q^{\ddagger}_{\ell+1}(x_{l_s}, u_{l_s}) - [F(\theta)](x_{l_s}, u_{l_s}) \right)^2$
7: **until** $\theta_{\ell+1}$ is satisfactory

Output: $\widehat{\theta}^* = \theta_{\ell+1}$

Figure 13 – ***Least Squares Q iteration algorithm***

3.4.3 Normalized Advantages Function

This algorithm allows application of Q-learning with experience replay to continuous tasks, which corresponds perfectly to our situation. Furthermore, it was already completely implemented in the keras-rl repository library, which greatly facilitated its utilization. The algorithm as presented in the paper [5] is presented in Figure 14. The only modification made was the use of the Microgrids Network and the addition of a Processor object. This type of object comes from the keras library and is designed to handle the interaction between agent and environment. It translates data to be readable by either the agent or environment when necessary. The adjustment made here was simply to change the action from the floats provided by the Neural Network to the binaries that could be processed by the Microgrids Environment.

Algorithm 1 Continuous Q-Learning with NAF

Randomly initialize normalized Q network $Q(\boldsymbol{x}, \boldsymbol{u}|\theta^Q)$.
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$.
Initialize replay buffer $R \leftarrow \emptyset$.
for episode=1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $\boldsymbol{x}_1 \sim p(\boldsymbol{x}_1)$
  for t=1, T **do**
    Select action $\boldsymbol{u}_t = \mu(\boldsymbol{x}_t|\theta^\mu) + \mathcal{N}_t$
    Execute $\boldsymbol{u}_t$ and observe r_t and $\boldsymbol{x}_{t+1}$
    Store transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ in R
    for iteration=1, I **do**
      Sample a random minibatch of m transitions from R
      Set $y_i = r_i + \gamma V'(\boldsymbol{x}_{i+1}|\theta^{Q'})$
      Update θ^Q by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(\boldsymbol{x}_i, \boldsymbol{u}_i|\theta^Q))^2$
      Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
  end for
end for

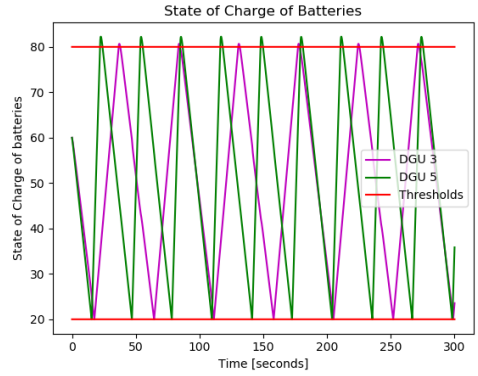
Figure 14 – ***Normalized Advantage Function Algorithm***

3.5 Results

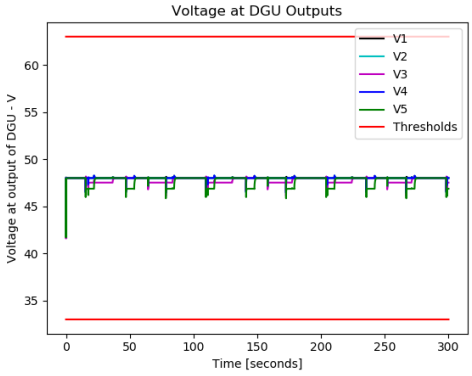
In this section, results from the different RL algorithms will be presented. In order to establish a baseline for 'good' behavior, a dummy controller was created, which will be presented first. A reminder that the purpose here is to keep the voltages at PCC and States of Charges of the batteries in their respective ranges. To test the different algorithms, the network is set in initialized to a random value then after a delay of X seconds as defined by the action delay, the best action is chosen through the value function and executed. This last step is repeated for a few minutes, the agent choosing the best policy at intervals defined by the action delay. The value functions were trained with the algorithms presented in section and both the training and testing parameters of the best versions are presented below.

3.5.1 Dummy Controller

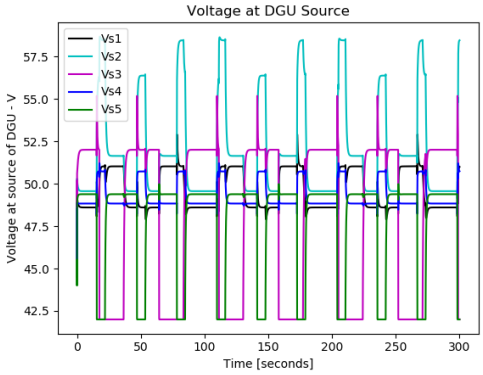
Since low-level control for each DGU is already implemented, the system stays stable and only the SoCs significantly vary. The role of the agent is therefore to simply apply the correct mode for each DGU when needed. Thus, when the state of charge of a battery is too low, it should be set to charge in order to remain in the desired range. Similarly, if the state of charge is too high, the battery should be set to PnP mode to discharge. Testing this controller for 500 seconds resulted in the behavior seen in Figure 15.



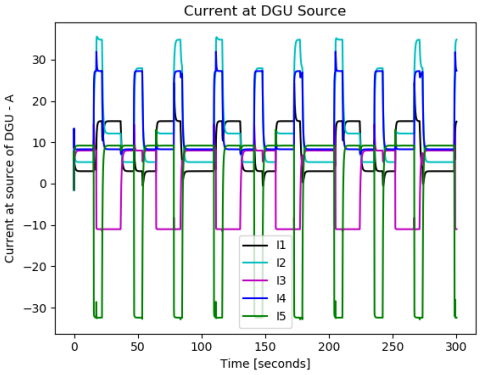
(a) States of Charge



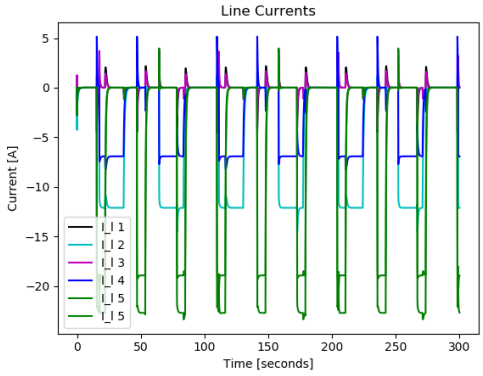
(b) Voltages at PCC



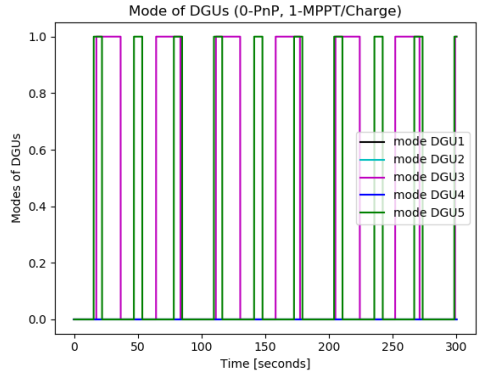
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) DGU Modes

Figure 15 – ***Test results with Dummy Controller - 500 seconds***

3.5.2 SARSA

The SARSA controller was trained with the Tile Coding function approximation. Many different parameters were tested both for the VF approximation and the RL algorithm. The parameters for the VF were presented earlier. Parameters for previous versions can be found in the $Value\_Function$ folder in note form and those for this version are presented below. The reward used for this agent and all others is the one presented in section 3.2.2. The number and length of the episode describe the amount of time the agent was trained, and the action delay is the time the agent waits before choosing a new action. α and β are the coefficients applied when updating the average reward and the weights of the value function. The exploration rate is the ratio at which a random action is chosen.

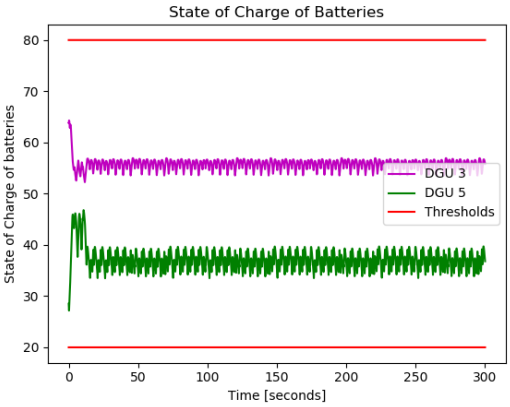
Training parameters :

- **Number of Episodes -** 40
- **Time per Episode -** 180 seconds
- **Action Delay -** 2 seconds
- α **-** 0.01
- β **-** 0.01
- **Exploration rate -** 0.12

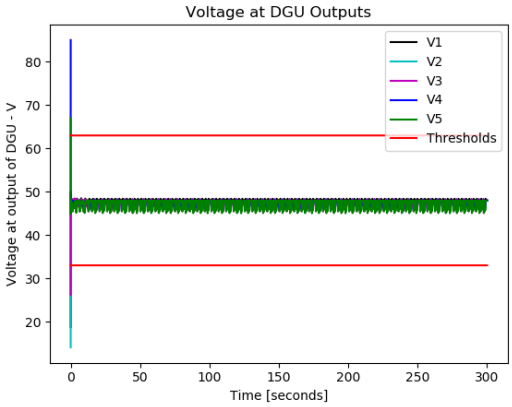
Testing parameters :

- **Time -** 300 seconds
- **Action delay -** 0.2 seconds

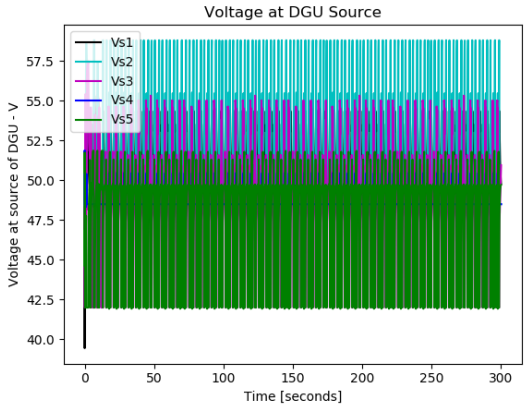
Figure 23 shows the SARSA agent is capable of keeping the SoC of the batteries inside the desired range. It must be pointed out that the action delay used for training is higher than the one used for testing. As mentioned previously, a high action delay when training allows the agent to better estimate the value of a state, but a higher frequency when being tested allows a more precise control of the microgrid. The range of motion of the SoC is therefore quite small, with constant, fast oscillations due to the lower action delay in testing. This agent and previous versions were tested with a higher delay and showed a larger range in SoC variation, these results can be seen in the Appendix. The weights of the value function are mostly 0, meaning unexplored, which either shows that not enough of the state-space as been explored, or that a smaller maximum number of tiles can be used.



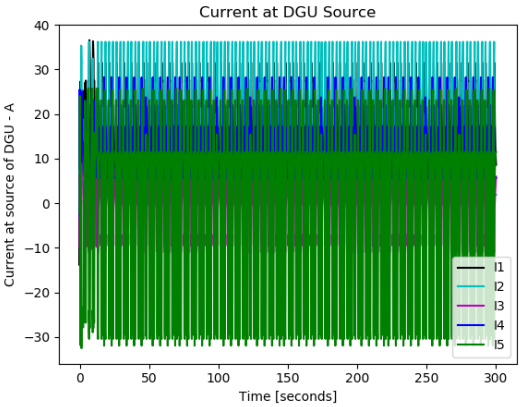
(a) States of Charge



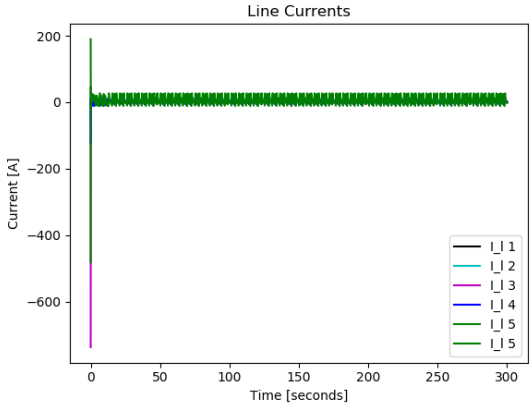
(b) Voltages at PCC



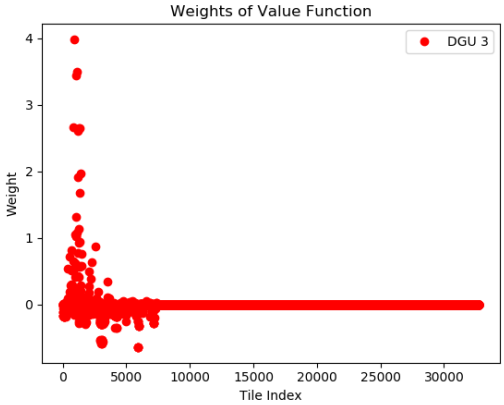
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) Value Function Weights

Figure 16 – ***Test results with SARSA v21 - 300 seconds***

3.5.3 Qfit

The Qfit algorithm was also trained with the Tile Coding approximation, with the parameters described above. This algorithm works differently and therefore has different parameters. The training sample describes the amount of random states sampled, the number of steps is the number of times of new random sample is trained. This agent was trained twice as much as the SARSA agent to obtain the results shown below. The action delay here describes the time waited to evaluate the value of each state-action pair. Gamma is the discount factor for the value of the previous value function.

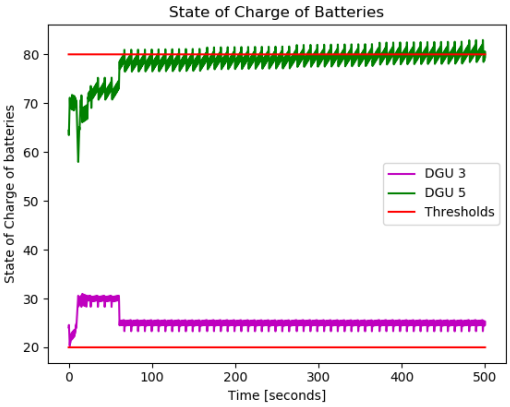
Training parameters :

- **Number of steps -** 30
- **Training Sample -** 1000 random states
- **Action Delay -** 0,2 seconds
- γ **-** 0.1

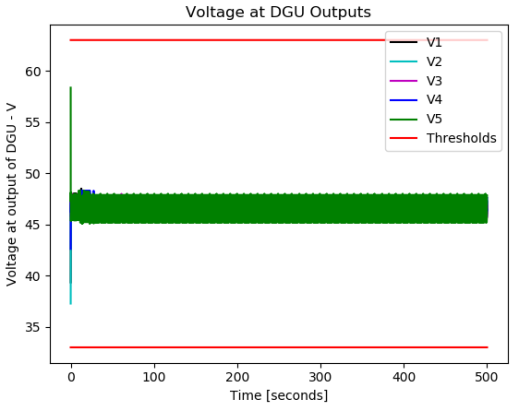
Testing parameters :

- **Time -** 300 seconds
- **Action Delay -** 0,2 seconds

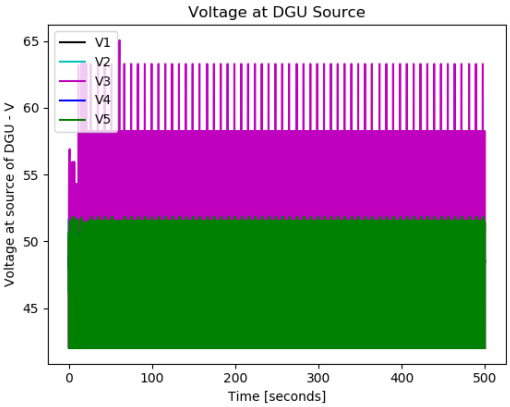
Figure 17 shows the Qfit agent can also keep the SoC of the batteries within the desired range, although the SoC of DGU 5 is noticeably growing. This might be due to the total load of the microgrids being a bit too low, although the controller should be able to adapt to this. The SoC variation range is similar to the one seen with the SARSA agent, due to the action delay. The weights of this value function are mostly negative, with the line formed by the regressions clearly visible. This might be because the agent was often outside of the positive reward range when training and therefore negative values were more often explored. Longer training, or limitation in the range of states when resetting the network could lead to a value function with better defined positive weights.



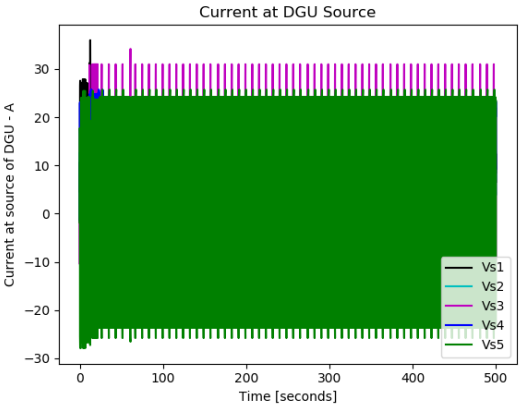
(a) States of Charge



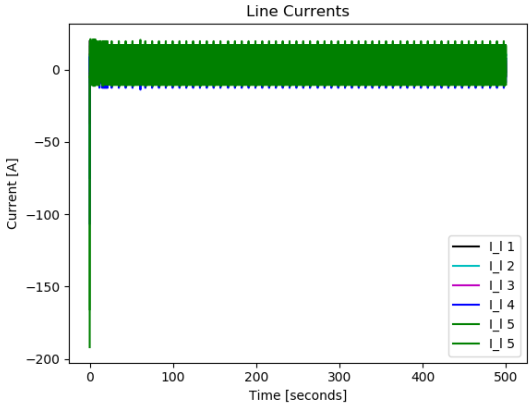
(b) Voltages at PCC



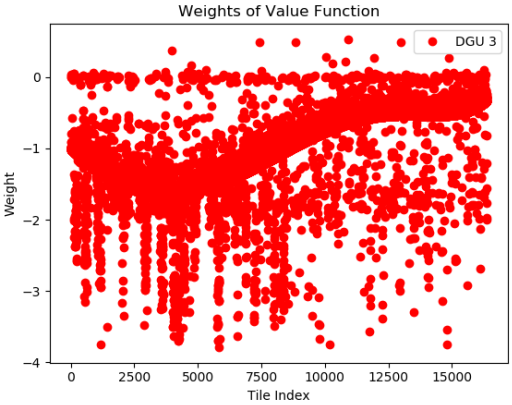
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) Value Function Weights

Figure 17 – ***Test results with Qfit v2.2 - 500 seconds***

3.5.4 NAF

The NAF algorithm implementation from keras-rl takes in several parameters, results shown in Figure 18 were obtain with an agent initialized with the default parameters and the processor mentioned in section 3.4.3. The parameters presented below are the ones set when testing and training the agent.

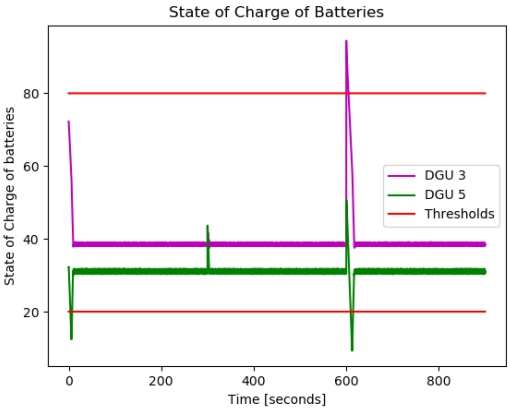
Training parameters :

- **Time-** 25 minutes
- **Action Delay -** 100 ms

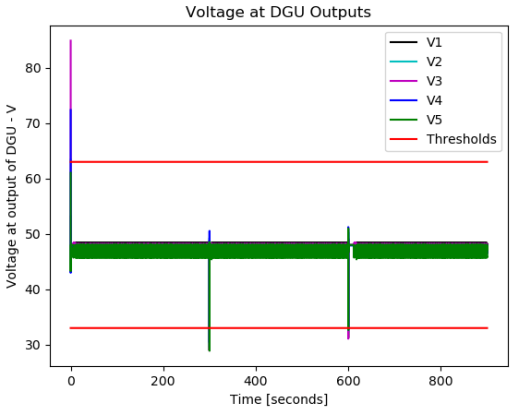
Testing parameters :

- **Number of Episodes -** 3
- **Time per Episode -** 300 seconds
- **Action Delay -** 100 ms

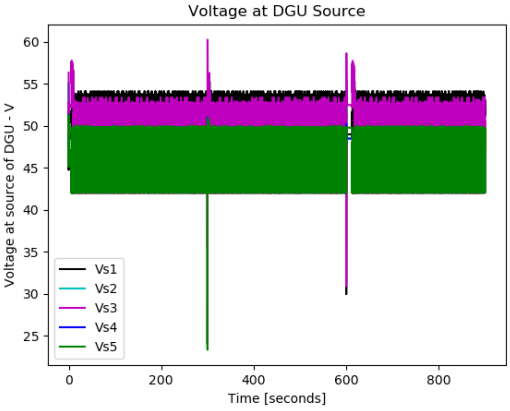
Figure 18 shows the results for the NAF agent. The SoC of the batteries are clearly more stable than in both previous agents, with the range variation again smaller due to the smaller action delay. The spikes in the SoCs are due to the initialization of each episode, and we can clearly see the agent immediately converges to a value with a positive reward. It should be noted than the best state for the agent, as defined by the reward, should actually be with SoCs at 60%. The offset from this 'ideal' state might be due to the short training time, as the agent might not have yet learned to reach it. The agent was however trained again for a longer period and showed worst results, as shown in the Appendix.



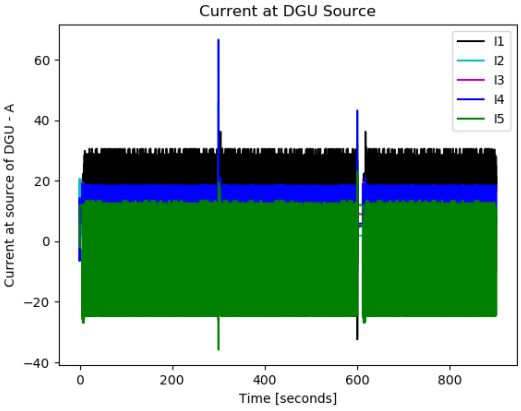
(a) States of Charge



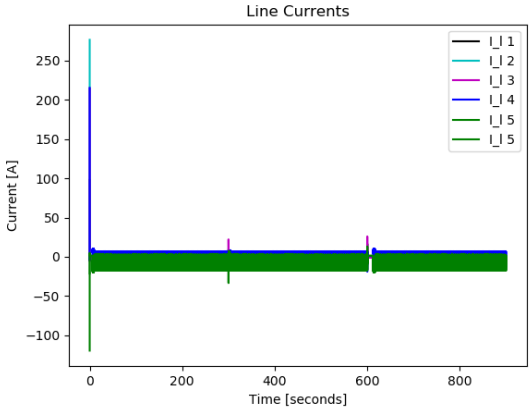
(b) Voltages at PCC



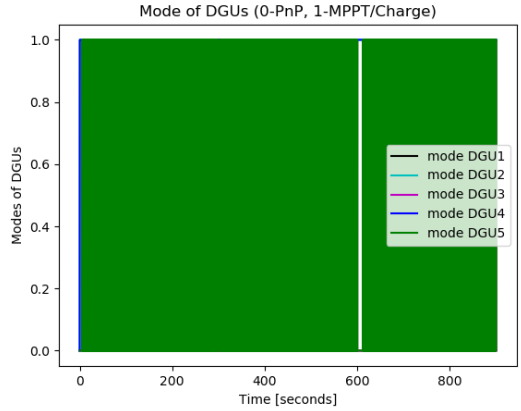
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) DGU Modes

Figure 18 – ***Test results with NAF - 3 episodes of 300 seconds***

4 Implementation and recommendations

Several RL algorithms and VF approximation were explored, all with a multitude of parameters to tune before being able to achieve any results. This section will summarize what experiments were tried and their results in order to inform anyone continuing this project of the DOs and DONTs. Said person should be familiar with Reinforcement Learning and ready to explore recent python implementations in order to adapt them to the microgrid environment. The following section will contain several notes first on the general challenges and implementation of this project, then on the limitations of the current value function approximation and some recommendations concerning what to do next.

4.1 General Notes

Several trade-offs have been made during the design of this implementation, as mentioned above. Their cause is due to one crucial initial parameter : the timestep of the simulation. In fact, $2 * 10^{-5}$ is the upper limit for which the implemented network stays stable due to the euler method implementation. A different method of implementation like backward Euler or Crank-Nicholson could be considered to improve this parameter. From this limitation is derived several others explained below :

• **Value sampling -** Due to the small timestep, values at each step cannot be saved for later plotting without causing a memory overload. A parameter called *sample_ratio* was implemented in the network environment to only sample data at every X timestep. It is currently set at 2500, which means data is sampled every 50 ms.

• **Action Delay -** Enacting a mode switch at every time step causes the system to grow exponentially. The action delay allows the simulation to update for some time to better simulate real behavior. This is crucial in order to correctly train the system when using RL. The lower limitation for the system to behave realistically is 500 steps.

• **Computing Time -** Due to the small timestep, 50'000 steps are necessary to simulate 1 second. When running the network without a supervisory controller, this takes less than a second. Unfortunately when using a controller, it builds its policy at each step using the value function approximation. This takes some time to compute and so second in simulation takes about 1 second, depending on the VF approximation used. For this reason, it is time consuming to run long simulations, which must be taken into account when training the agent.

• **Battery Capacitance -** A microgrid should be able to run for days autonomously and should therefore have batteries capable of storing enough energy to last the night. However such batteries would barely discharge in an hour of simulation and so have been sized down for training purposes. A well trained agent should have also be capable of controlling a real-sized microgrid.

In order to be able to improve the current implementation, some instructions should be read. They are located in the README available in the joint $Final\_Model$ folder. Future upgrades

of the network should look to several things to add. First a variation with time in the source voltage in free mode for PV, in order to simulate solar irradiance variation. Similarly, loads varying with time should be implemented to better train the supervisory controller. Another useful addition would also be the possibility for the agent to choose to disconnect a battery from the network in order to conserve its power supply. Of course this would be in a network where the DGU can sustain its individual load.

4.2 VF approximation Notes

The state-space that the agent needs to explore in order to gain knowledge of the environment is quite large and continuous. To adapt to this, it is necessary to properly approximate it so the agent can learn in reasonable time, particularly due to the limitations in simulation length mentioned in Section 4.1. In order to do this, a value function which uses tile coding was implemented, originally based on this article. Several parameters must be carefully set in order to correctly use this approximation for this project.

The values used when updating the weights should be kept small in order to avoid creating discrepancies when later comparing values. Indeed, some could be disproportionately high or low solely because they were explored more often. Several parameter enable this at different levels, β in the SARSA controller, α in the VF approximation and $reward\_weight$ in the Network environment. Furthermore, in order to train an agent well, the number of tiles per tiling should be kept low, even if that entails a worse resolution. A state-space approximation that is too precise takes too long to explore and the agent will need much longer training for similar behavior. However if a solution could be found to speed up simulation, a more precise approximation could be used.

Another type of function approximation are ANNs, which work extremely well for our scenario when utilized properly. Future study should focus on implementing Deep RL techniques which use ANNS for value function approximation. Several parameters determine the efficiency of ANNs and they should be studied to correctly understand how to tune a neural network to work with our simulation. For now, the default parameters were kept.

4.3 RL Notes

Many parameters must be tweaked in order to correctly train an efficient supervisory agent. These parameters were already mentioned when describing the different RL algorithms, this section will describe their implementation and their recommended values. All parameters are already set to default values in the current code implementation.

• **Episode Number and Length** These parameters can be modified directly in the main function and define the training time for the SARSA agent. Since the agent is randomly reset at the start of each episode, many episodes will ensure it explores different regions of the state-space. Episodes shouldn't be too short however, or the agent will not learn to control the system autonomously for an extended time.

• **Number of steps and Training size -** Similarly, these parameters are set in the *main* function and define the training time for the Qfit agent. The training set here is composed of random states and so its size should big enough to explore enough of the state-space to approximate a useful VF through regression. The number of steps describe the number of times a new training set is sampled, the more steps during training, the better trained the agent. Be however careful of overtraining, as this could lead to an agent operating the current network implementation perfectly, but unable to handle any variations in the network model.

• **Hyperparameters -** Several parameters are available in the *main* function to regulate the training of both the SARSA and Qfit agent. They should all be set to values between 0 and 1. Any important change to their values should be carefully monitored to determine its exact impact on the training of an agent.

• **State initialization -** When the network is reset, a random state is chosen among the state-space. In order to guide the agent to explore specific regions of the state-space, it is possible to change the initialization to limit the state-space in which the environment will be reset. This must be done directly in the *reset* function of the Network environment implementation. Doing this with an already trained agent could be useful to improve its knowledge of a specific part of the state-space.

• **Reward -** The reward is defined in the Network environment and its weighting parameters are available directly in the *init* function of the object. The reward is key to the agent's understanding of the environment it explores and should be adapted to clearly express the user's goal. The current implementation works, but several weights can be tweaked in order to potentially improve it.

Only a few algorithms have already been tested and many more remain to be adapted and used to train a supervisory controller for microgrids. Python repositories like keras-rl and garage are available online with implementations of deep RL algorithms compatible with gym environments. Deep Q Learning (DQN), Deep Sarsa and Dueling DQN for example, are available in keras-rl and could yield efficient agents if used properly. It must be noted that keras-rl is out of date and older versions of python and its modules must be used for it to work. The correct versions are noted in the README available with the code. Furthermore, careful attention should be paid to how each algorithm handles the action representation, as a binary list is not the standard. As mentioned, a processor to process the action should be implemented, an example of which is available in the $NAF\_Controller$ function.

Concerning previous results, VF weights of older versions are available in the $Value\_Function$ folder, however they might not work as previously recorded since the code was updated since. Their results are available in the appendix and their parameters in the $Notes$ in the $Value\_Function$ folder. A recommendation for anyone continuing the project would be to train new value function weights small variations to the default parameters to explore their impact on the resulting agent.

5 Conclusion

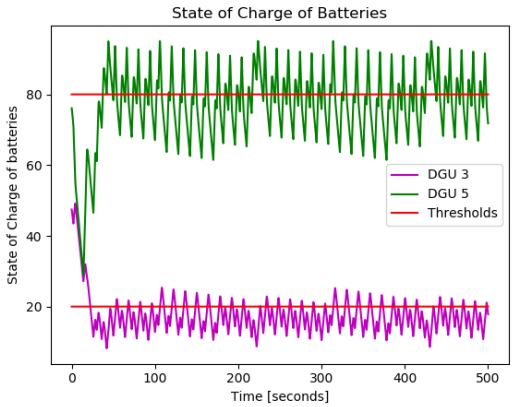
The purpose of this project was to explore solutions to train a supervisory controller for microgrids based on modeled data. To achieve this, the first crucial step was to build a microgrid environment from which realistic data could be sampled. Several assumptions were made to create this microgrid environment where Reinforcement Learning agents could be trained. The result is a basic, functioning model with two types of DGUs, each with two modes of operation. The model is also sized down in order to be computationally reasonable. Although all results presented here were computed using a single microgrid configuration, the python implementation was restructured to be modulable for future iterations. Furthermore, it was adaptated as a openAI gym environment in order to be compatible with open source RL repositories.

Once the model constructed, Reinforcement Learning algorithms were applied to train an agent to control the microgrid. Each algorithm comes with its own parameters and constraints, which must be adapted to the microgrid environment. The first algorithm implemented was a SARSA-like adapted to continuous tasks and using tile coding to approximate the value function. After some parameter tuning, it yielded an efficient agent which manages to keep the SoCs of both batteries within their desired range. The second implemented algorithm was an approximate Q iteration using Least Squares polynomial regression and once again tile coding for value function approximation. This also yielded correct results, although the SoCs tend to stabilize very close to the limits of the desired range. Finally, the NAF algorithm from the keras-rl library was used, which yielded the best results by far, with the least parameter tuning.

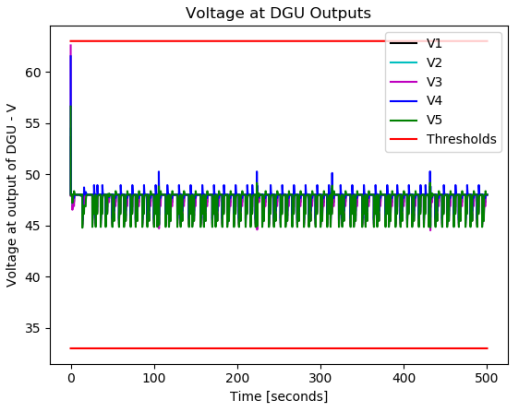
Due to the excellent results obtained with neural network function approximation, continuation of this project should aim to explore more Deep Reinforcement Learning algorithms. Many recent implementations are available online and with the use of the microgrid environment, exploration and tuning of both old and new algorithms is the next step forward.

Appendix

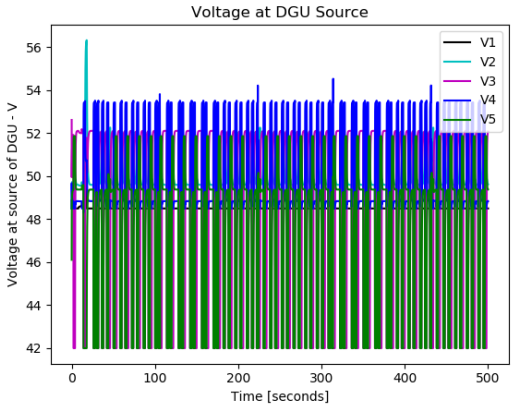
Results of different versions



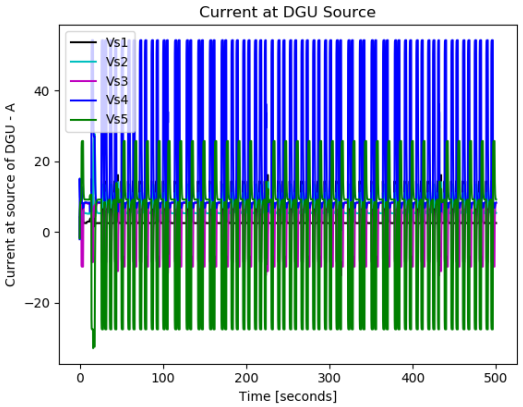
(a) States of Charge



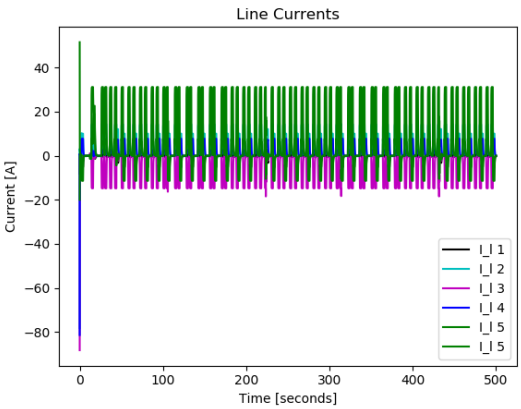
(b) Voltages at PCC



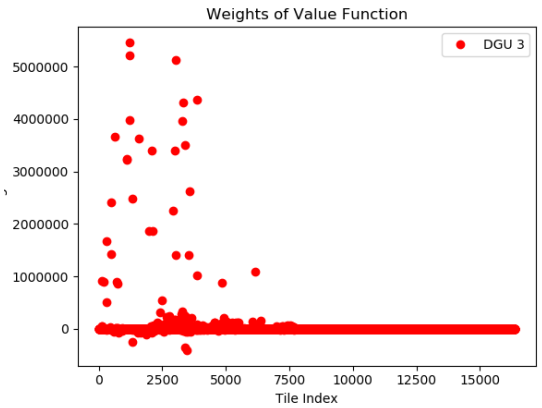
(c) Source Voltages



(d) Source Currents

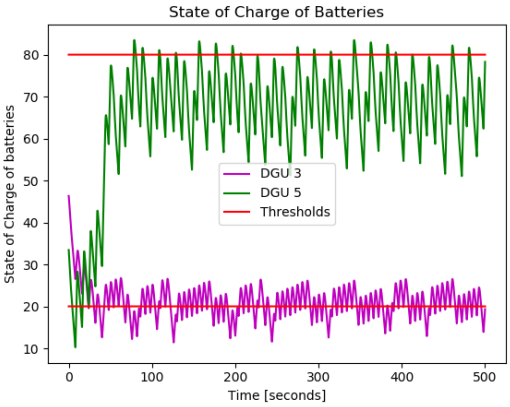


(e) Line Currents

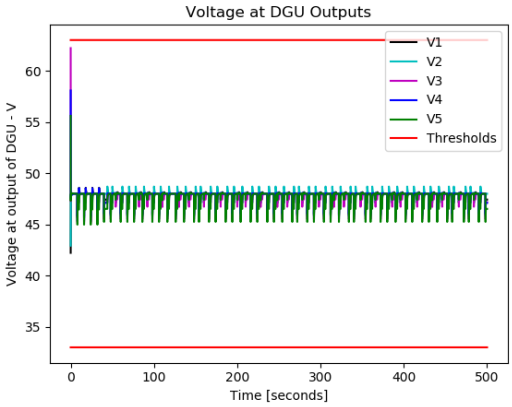


(f) Value Function Weights

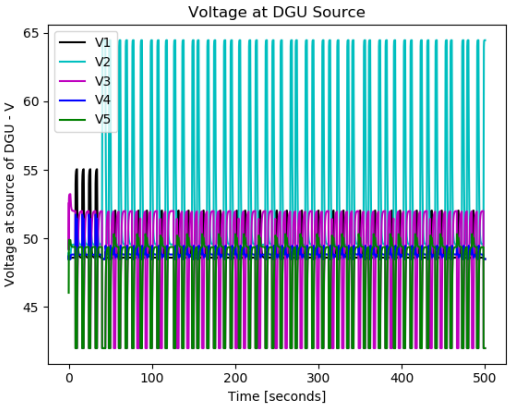
Figure 19 – ***Test results with SARSA v13 - 500 seconds - 1h of training, 2s action delay***



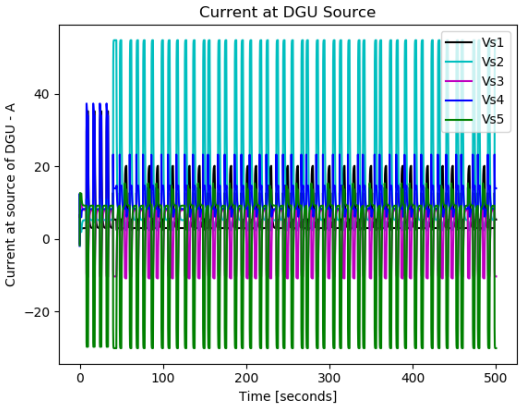
(a) States of Charge



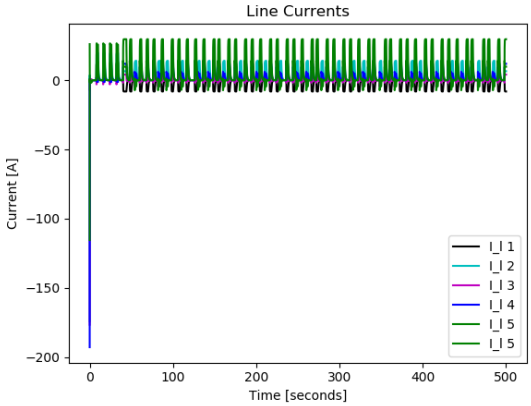
(b) Voltages at PCC



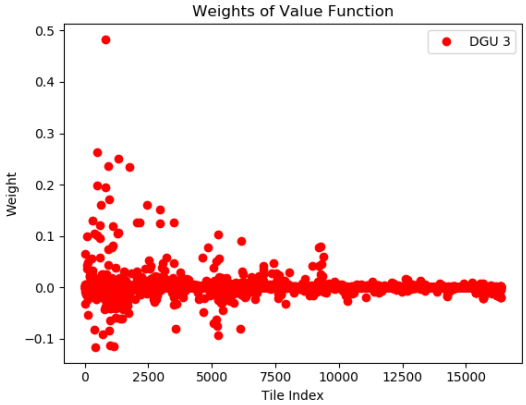
(c) Source Voltages



(d) Source Currents

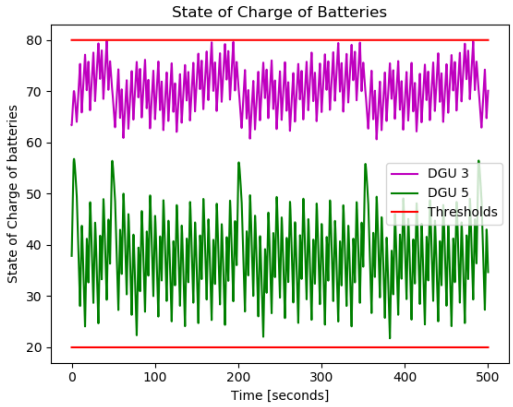


(e) Line Currents

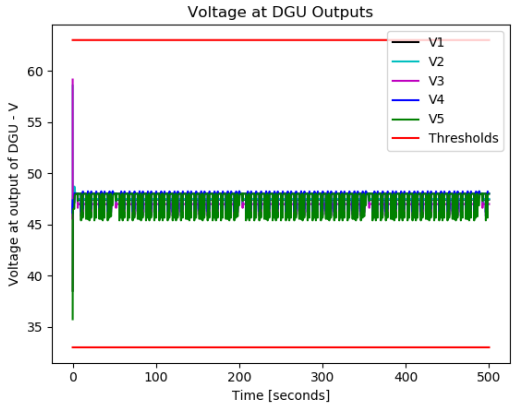


(f) Value Function Weights

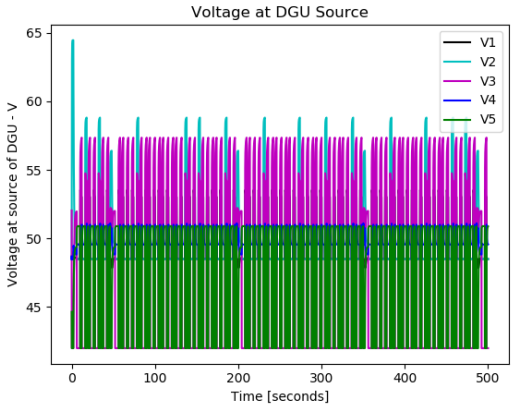
Figure 20 – ***Test results with SARSA v15 - 500 seconds - 2h of training, 2s action delay***



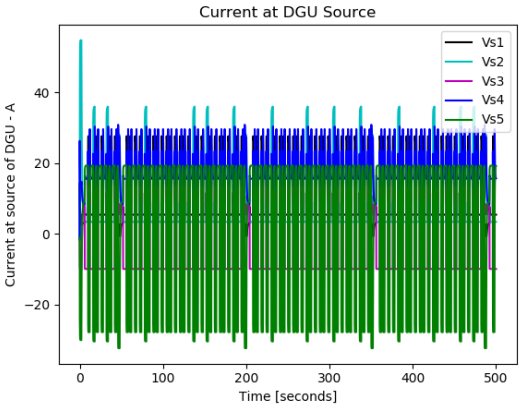
(a) States of Charge



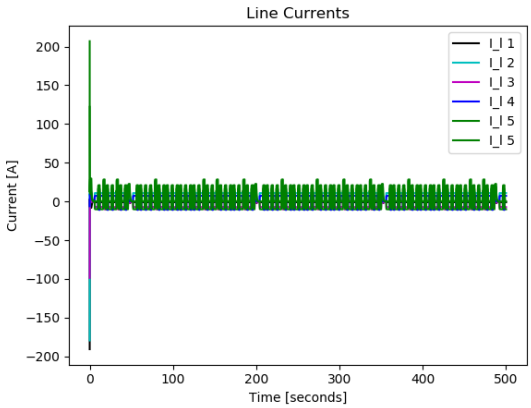
(b) Voltages at PCC



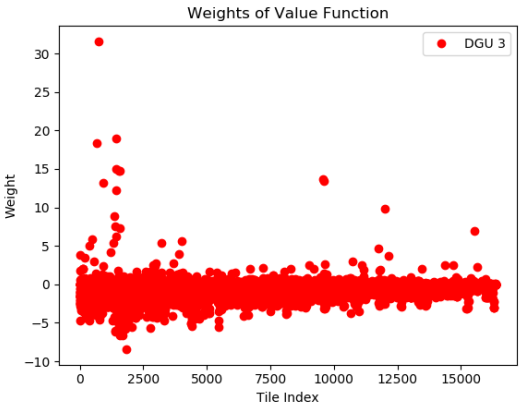
(c) Source Voltages



(d) Source Currents

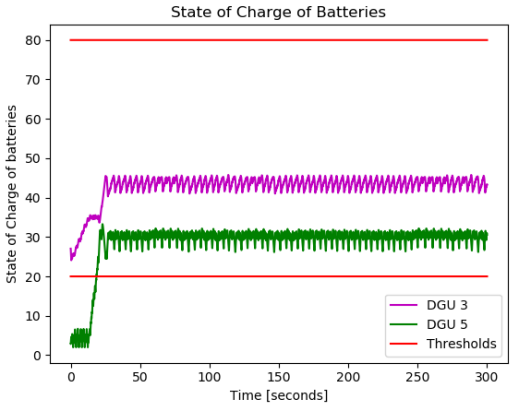


(e) Line Currents

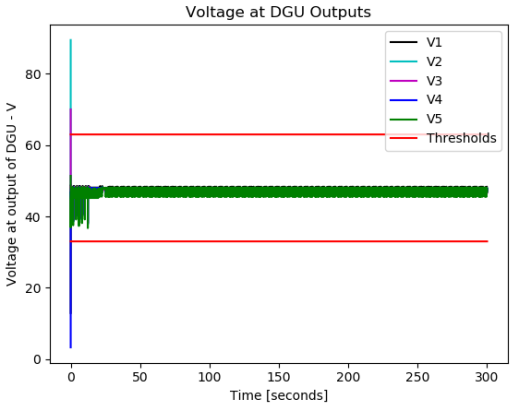


(f) Value Function Weights

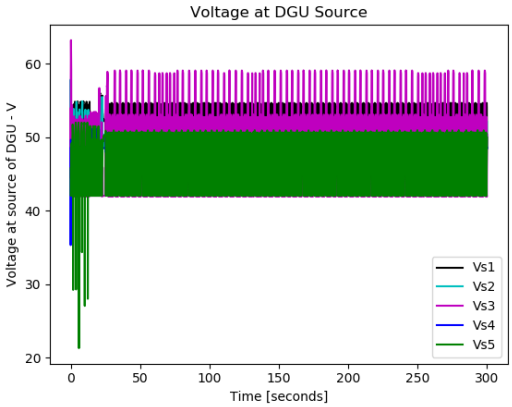
Figure 21 – ***Test results with SARSA v17 - 500 seconds - 2h of training, 2s action delay***



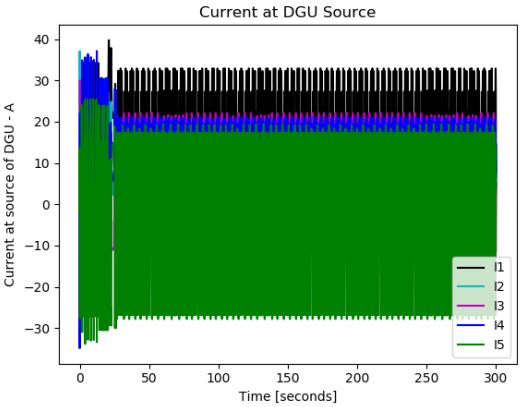
(a) States of Charge



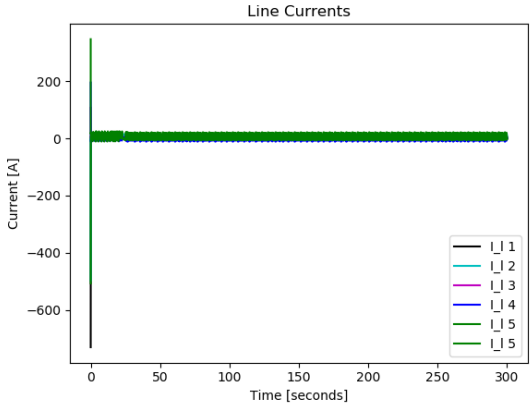
(b) Voltages at PCC



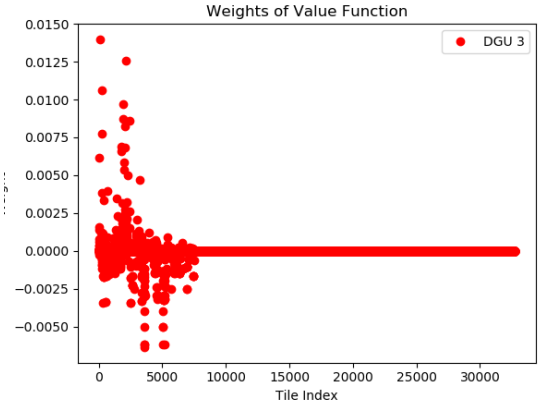
(c) Source Voltages



(d) Source Currents

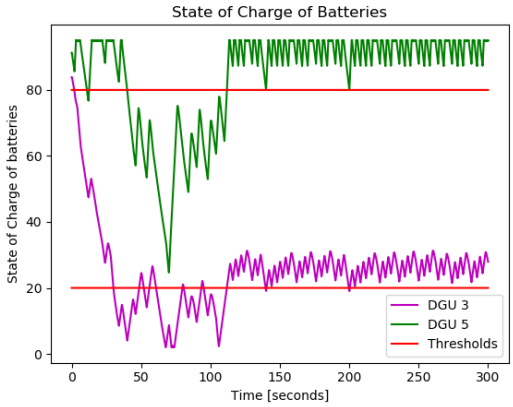


(e) Line Currents

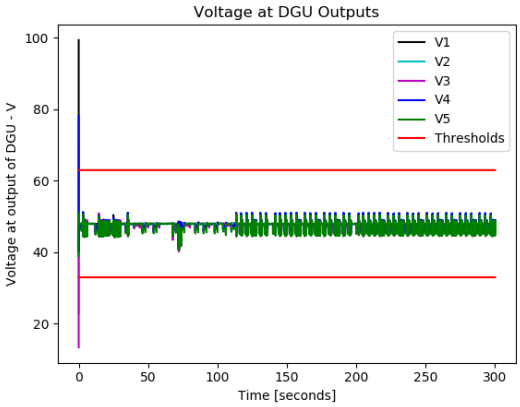


(f) Value Function Weights

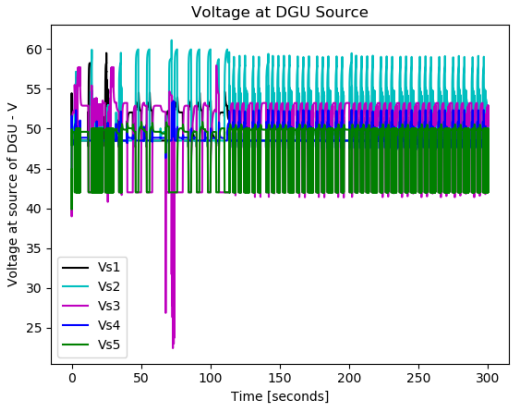
Figure 22 – ***Test results with SARSA v20 - 300 seconds - 1h of training, 0.2s action delay***



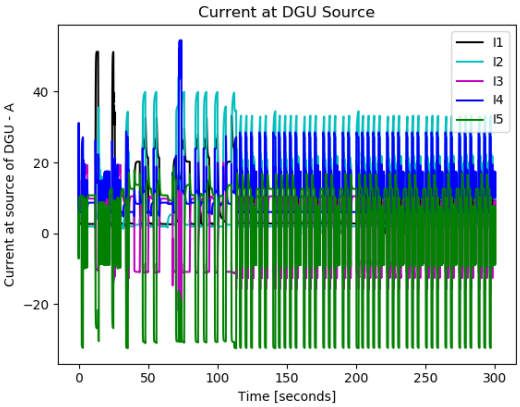
(a) States of Charge



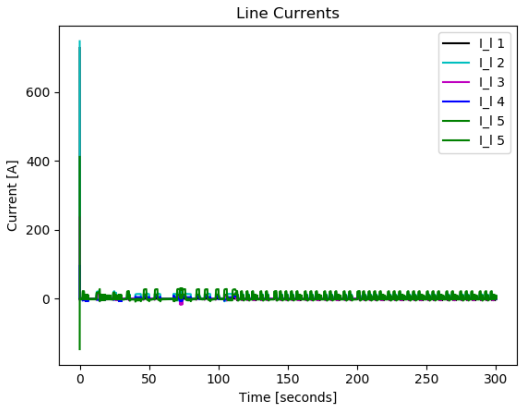
(b) Voltages at PCC



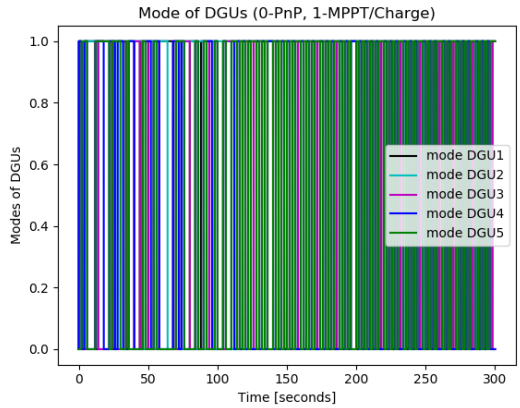
(c) Source Voltages



(d) Source Currents

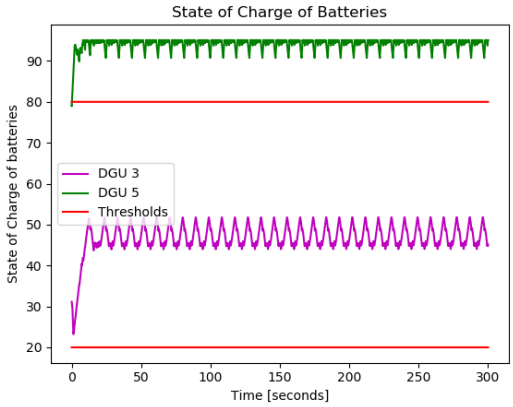


(e) Line Currents

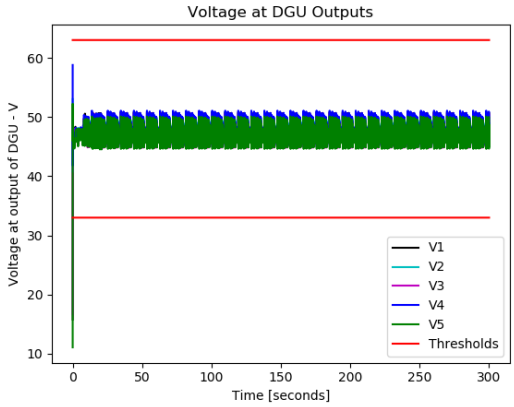


(f) Value Function Weights

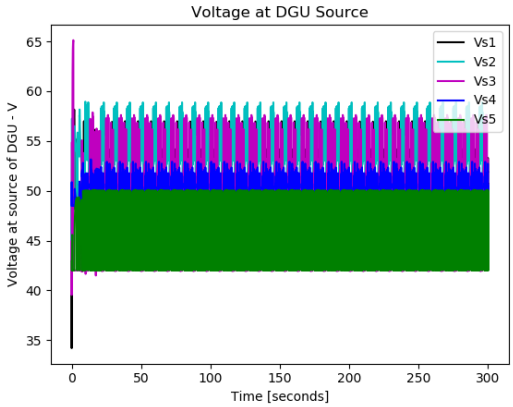
Figure 23 – ***Test results with SARSA v21 - 300 seconds - 2h of training, 2s action delay***



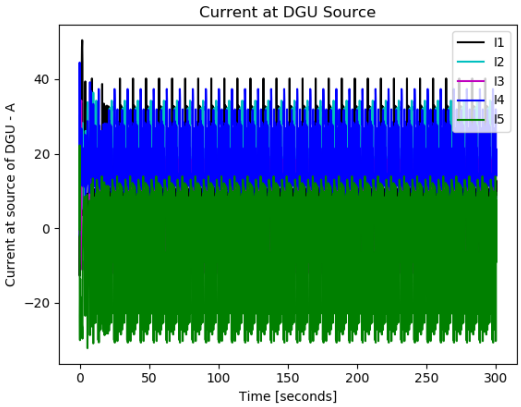
(a) States of Charge



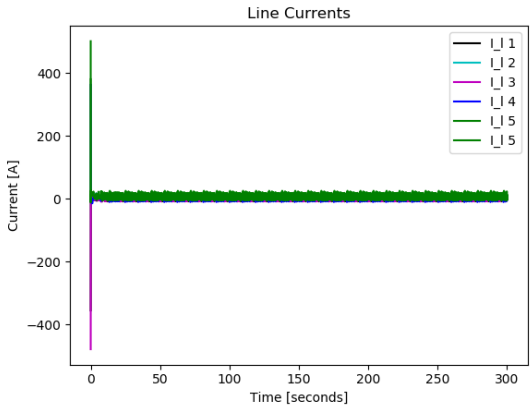
(b) Voltages at PCC



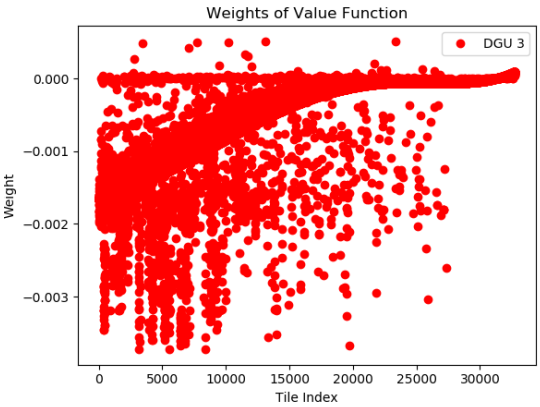
(c) Source Voltages



(d) Source Currents

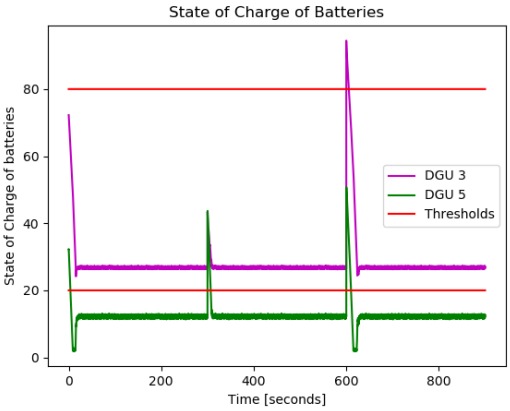


(e) Line Currents

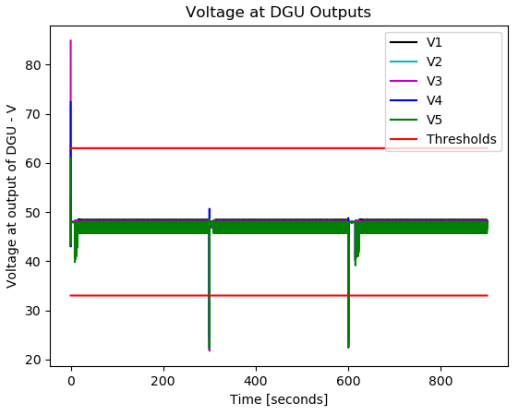


(f) Value Function Weights

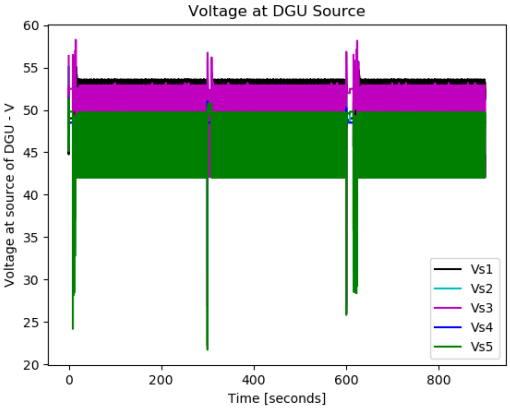
Figure 24 – ***Test results with Qfit v2.4 - 300 seconds - 1h of training, 0.2s action delay***



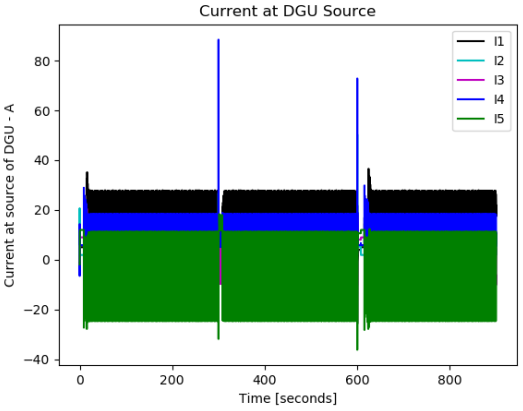
(a) States of Charge



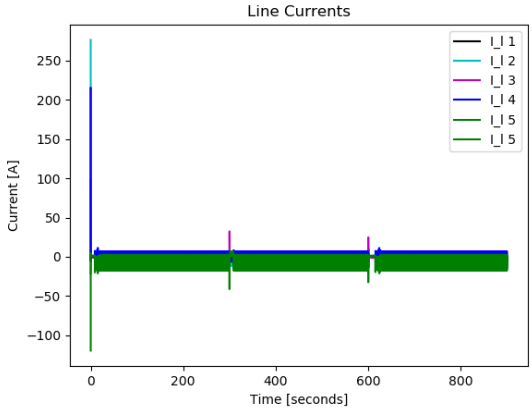
(b) Voltages at PCC



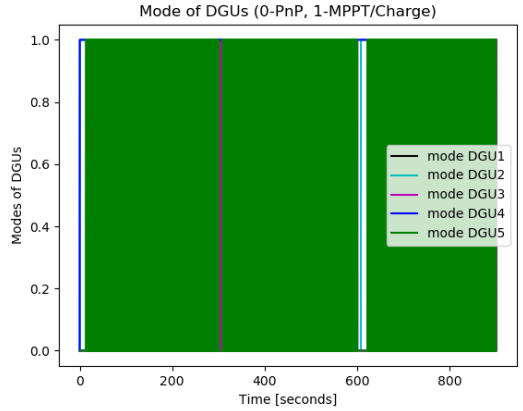
(c) Source Voltages



(d) Source Currents



(e) Line Currents



(f) Value Function Weights

Figure 25 – ***Test results with NAF v4 - 3 episodes of 300 seconds - 1h of training, 0.1s action delay***

References

[1] *Plug-and-Play control of DC microgrids: modeling of power sources and integration of batteries* by Davide Riccardi

[2] *A Closer Look at State of Charge (SOC) and State of Health (SOH) Estimation Techniques for Batteries* by Martin Murnane and Adel Ghazel

[3] *Plug-and-Play control of DC microgrids: analysis of bus-connected topologies and integration of photovoltaic sources* by Giuseppe Tagliaferri

[4] *A Passivity-Based Approach to Voltage Stabilization in DC Microgrids with ZIP loads* by Pulkit Nahata, Raffaele Soloperto, Michele Tucci, Andrea Martinelli and Giancarlo Ferrari-Trecate

[5] *Continuous Deep Q-Learning with Model-based Acceleration* by Shixiang Gu Timothy Lillicrap Ilya Sutskever Sergey Levine

[6] *An Overview of Generic Battery Models* by Ala Al-Haj Hussein and Issa Batarseh

[7] *Modeling Stationary Lithium-Ion Batteries for Optimization and Predictive Control* by Emma Raszmann Kyri Baker, Ying Shi, and Dane Christensen

[8] *Reinforcement Learning - An introduction* by Richard S. Sutton and Andrew G. Barto

[9] *Reinforcement Learning and Dynamic Programming using Function Approximators* by Lucian Busoniu, Robert Babuska, Bart De Schutter and Damien Ernst