Bachelor thesis - Reinforcement learning for something.

Mélanie Fournier

3/15/23

Table of contents

Abstract

Differential equations go brrrrrr

Acknowledgement

Chapter 1

Introduction

Introduction

Numerical methods for differential equations are among the most important and numerous methods in numerical analysis, all with specifics strength and weaknesses. They all have, however, some parameters that need to be chosen, if only for the step size.

These parameters have to be chosen to maximize performances, and depends on the problem. In some cases, the optimal parameters can be determined analytically, in other cases they are taken using some heuristic techniques and in the last case they can be searched for numerically.

In this thesis, we do the latter, on a specific, restricted set of problems. We motivate the use of numerical ODE solvers to solve linear systems. We take as a case study a specific type of linear systems, which appears when discretizing the steady state, one dimensional convection diffusion equation. To solve these systems, This numerical solver has two parameters, a pseudo time step Δt and another parameter α, which need to be chosen.

We then explore how to optimize these two parameters by taking into account two other parameters that define the linear system we have to solve. In particular, we use reinforcement learning to do so.

The method of reinforcement learning can claim its origin from both the field of animal learning (learning by trial and error) and optimal control theory. The general idea is to train an agent to make decisions based on an environment, and rewarding, or punishing this agent when it makes good, or bad decisions. Over time, the agent then learn to make better and better decisions by using its past experiences to maximize the reward it gets. This approach has been used successfully in a number of problems, such as for example playing games at a high level (), or discovering matrix multiplication algorithms [1].

In this thesis, we introduce reinforcement learning main concepts, such as Markov decision process, state, action and rewards. We then introduce policy gradient methods and use it to optimize the solver parameters for the studied linear systems. The results, while positive, are hampered mainly by the fact that the implemented reinforcement learning was “forced” into the problem.

Chapter 2

Motivation

Let A be non singular square matrix of dimension $n \geq 1$ and let $b \in \mathbb{R}^n$. We consider the linear system $Ay = b$, where $y \in \mathbb{R}^n$. The system has for unique solution $y^* = A^{-1}b$. This is a fundamental problem to solve in numerical analysis, and there are numerous numerical methods to solve this, whether they are direct methods or iterative methods. In this thesis, we consider an iterative method. We consider the initial value problem

$$y'(t) = Ay(t) - b, \;\; y(0) = y_0$$

where $y_0 \in \mathbb{R}^n$ and $t \in \mathbb{R}$. Multiplying the equation by e^{-At}, where e^{-At} is the usual matrix exponential, and rearranging the terms yields

$$e^{-At}y'(t) - Ae^{-At}y(t) = e^{-At}b$$

We recognise on the left hand side the derivative of the product $e^{-At}y(t)$, and thus, by the fundamental theorem of calculus,

$$\left[e^{-Au}y(u)\right]_0^t = \int_0^t -e^{-Au}b \; du.$$

Multiplying by $A^{-1}A$ inside the integral in the LHS, we get

$$e^{-At}y(t) - y_0 = A^{-1}\left[e^{-Au}\right]_0^t b = A^{-1}e^{-At}b - A^{-1}b.$$

Multiplying each side by e^{At} and rearranging the terms we get an expression for $y(t)$,

$$y(t) = e^{At}(y_0 - A^{-1}b) + A^{-1}b.$$

Note that each of those step can be taken backward , which means that the solution we have is unique. We have thus proved

Theorem 2.1. *Let A be a non singular, square matrix of dimension $n \geq 1$, $b \in \mathbb{R}^n$ a vector, and consider the initial value problem*

$$y'(t) = Ay(t) - b,\ y(0) = y_0 \tag{2.1}$$

where $t \to y(t)$ is a function from $\mathbb{R}$ to $\mathbb{R}^n$. Then the problem has a unique solution in the form of

$$y(t) = e^{At}(y_0 - y^*) + y^*,$$

where $y^* = A^{-1}b$, and e^{At} is defined using the usual matrix exponential.

Let $\lambda_1, \lambda_2, \dots, \lambda_n$ be the (not necessarily distinct) eigenvalues of A, write $\lambda_i = a_i + iy_i$, where $a_i, b_i \in \mathbb{R}$ are respectively the real part and the imaginary parts of the i^{th} eigenvalue. The following holds

Theorem 2.2. *$y(t) \to y^*$ as $t \to +\infty$ for any initial value y_0 if and only if, for all $i = 1, \dots, n$, $a_i < 0$, that is, all the eigenvalues of A have a strictly negative real part.*

Proof. (In the diagonalisable case)

We assume that A is diagonalisable. Write $A = P\Delta P^{-1}$ where Δ is diagonal.

$$\Delta = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}$$

Then $e^{At} = Pe^{\Delta t}P^{-1}$, where

$$e^{\Delta t} = \begin{pmatrix} e^{\lambda_1 t} & & & \\ & e^{\lambda_2 t} & & \\ & & \ddots & \\ & & & e^{\lambda_n t} \end{pmatrix}$$

Let $z(t) = P^{-1}(y(t) - y^*)$, where $y(t)$ is the unique solution to Equation 2.1 for some arbitrary initial value y_0.

Since P is non singular, $y(t) \to y^*$ if and only if $z(t) \to 0$. We have

$$z(t) = P^{-1}e^{At}(y_0 - y^*)$$

We note that $P^{-1}e^{At} = e^{\Delta t}P^{-1}$, thus

$$z(t) = e^{\Delta t}P^{-1}(y_0 - y^*).$$

Looking at the i^{th} element $z(t)_i$, we have

$$|z(t)_i| = e^{a_i t}\left(P^{-1}(y_0 - y^*)\right)_i$$

where $a_i = \Re[\lambda_i]$. Clearly, if $a_i < 0$, $z(t)_i \to 0$ as $t \to +\infty$. If this holds for any $i = 1, \dots, n$, then $z(t) \to 0$ as $t \to +\infty$. This proves ($\Leftarrow$).

This is also a necessary condition. Indeed, since y_0 is arbitrary, we can chose it so that $P^{-1}(y_0 - y^*) = (1, \dots, 1)^T$. Then $z(t) = (e^{\lambda_1 t}, e^{\lambda_2 t}, \dots, e^{\lambda_n t})^T$ which converges to 0 only if all the eigenvalues have a strictly negative real part.

□

Remark. A general proof is available on [2, Ch. 1]

We now go back to the original problem of solving the linear system $Ay = b$. If all the eigenvalues of A have a strictly negative real part, then, any numerical solver for the initial value problem $y'(t) = Ay(t) - b$ with $y(0) = y_0$ where t is some pseudo-time variable also becomes an iterative solver for the linear system $Ay = b$, as $y(t) \to y^*$.

Remark. If all the eigenvalues of A have a strictly positive real part, then we can simply solve $y' = (-A)y - (-b) = -Ay + b$ instead.

Chapter 3

A test problem - Convection diffusion equation

As a test case for the solver, we consider the one dimensional, steady state convection-diffusion equation with fixed boundary condition

$$u_x = bu_{xx} + 1, \quad u(0) = u(1) = 0 \tag{3.1}$$

where b is some physical parameter, proportional to the diffusion coefficient of the medium. Moreover, $u(x)$ is defined for x on the interval $[0, 1]$. This equation has an analytical solution that is given by

$$u(x) = x - \frac{e^{-(1-x)/b} - e^{-1/b}}{1 - e^{-1/b}}. \tag{3.2}$$

We are however interested in solving this numerically, with a finite difference approach. We partition the interval $[0, 1]$ into equidistant points $x_i, i = 0, ... n + 1$. We note the distance between each points as $\Delta x = \frac{1}{n+1}$, and we have $u(x_0) = u(0) = 0$ and $u(x_{n+1}) = u(1) = 0$. We use the notation $u^i = u(x_i)$. We approximate, for $i \geq 1$ the derivative

$$u_x^i = \frac{u^i - u^{i-1}}{\Delta x}$$

and the second order derivative is approximated by

$$u_{xx}^i = \frac{u^{i+1} - 2u^i + u^{i-1}}{\Delta x^2}$$

Note that the first derivative is approximated backward in time. For $i = 1, ... , n$, we thus have the approximation

$$u_x^i = \frac{u^i - u^{i-1}}{\Delta x} = b\frac{u^{i+1} - 2u^i + u^{i-1}}{\Delta x^2} + 1$$

This can be given in matrix format by letting $u = (u^1, ... , u^n)^T$

$$Au = Bu + d$$

where $d = (1, 1, \dots, 1)^T$,

$$A = \frac{1}{\Delta x} \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \end{bmatrix}$$

and

$$B = \frac{b}{\Delta x^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{bmatrix}.$$

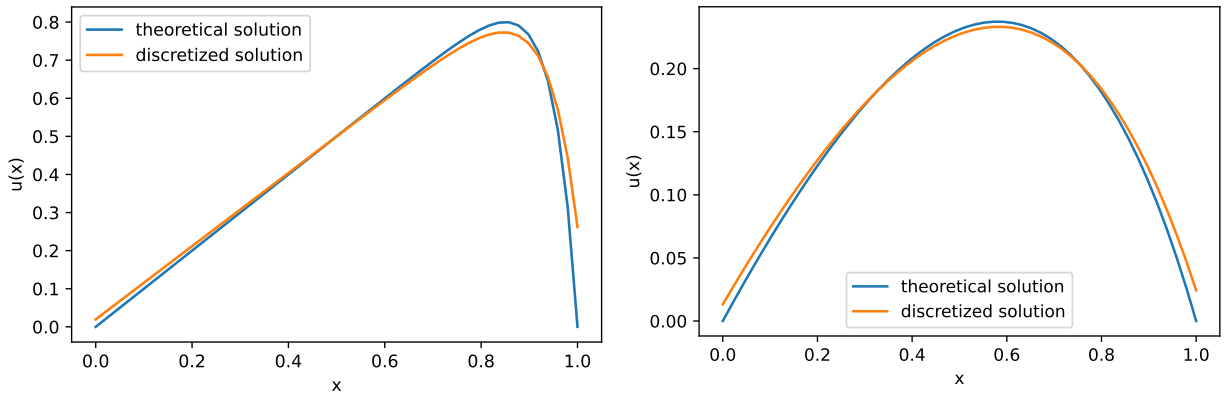
With $N = A - B$, we have to solve the linear system

$$Nu = d \tag{3.3}$$

where N is a square matrix of dimension $n \times n$ and d is the one vector of dimension n.

Remark. It is apparent that N is diagonally dominant. Since all elements of the diagonal are positive, we can use Gershgorin circle theorem to prove that all the eigenvalues of N have a positive real part. Assuming N is non singular, we then have that $-N$ is stable.

We plot two examples of what the theoretical solution (Equation 3.2) and the discretized solution (Equation 3.3) look like for different b parameters in Figure 3.1



(a) $b = 0.05$, $n = 50$.

(b) $b = 0.5$, $n = 50$.

Figure 3.1: Theoretical and discretized solution of the convection diffusion equation, for different parameters.

To solve this linear system, we use the method highlighted before. To make it easier for later, we chose to scale M so that its diagonal elements are 1. This allows us to have all eigenvalues in the circle centered

around 1 with radius 1 independently of the parametrization. Setting $\eta = \frac{1}{\Delta x} + \frac{2b}{\Delta x^2}$, solving Equation 3.3 is equivalent to solving the system

$$Mu = e \tag{3.4}$$

where with $M = M/\eta$, $e_s = e/\eta$. The eigenvalues of M are also scaled by $1/\eta$ so $-M$ is stable, assuming it is non singular. We are now ready to solve the system iteratively using ODE solver.

We thus introduce a pseudo time variable t and we consider the ODE.

$$u'(t) = e - Mu(t) \tag{3.5}$$

We can use Theorem 2.2 with the non singularity assumption to guarantee that the ODE will converge to a steady state independently of the initial value we chose. In the next chapter, we will then solve this differential equation with a numerical solver.

Chapter 4

Runge-Kutta method and application to the test problem

4.1 Explicit Runge-Kutta method

This section aim to introduce explicit Runge-Kutta methods[3, Ch. 3], and the particular Runge-Kutta method used in this thesis. We consider solving a generic initial value problem of the form

$$y'(t) = f(t, y(t)), \quad y(0) = y_0.$$

If we know, for an instant t_n the value for $y(t_n)$, we can compute the value of y at instant $t_{n+1} = t_n + \Delta t$ by integrating

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(u, y(u))\, du$$

and with the change of variable $u = t_n + \Delta t\tau$, we have

$$y(t_{n+1}) = y(t_n) + \Delta t \int_0^1 f(t_n + h\tau, y(t_n + h\tau))\, d\tau$$

The problem is finding a suitable way to compute the integral above. An elementary approach is to use the current value of $f(t_n, y(t_n))$ and to treat f as constant, thus defining the sequence

$$y_{n+1} = y_n + \Delta t f(t_n, y_n)$$

where $y_n \approx y(t_n)$, $y_0 = y(0)$. This is the explicit Euler's method. We now want to exploit quadrature formulas for integration. Let $c_j \in [0, 1], j = 1, 2, \nu$ be the nodes in the quadrature formula, with their associated weight $b_j, j = 1, 2, ...$ A quadrature formula for the integral is then of the form

$$\int_0^1 f(t_n + h\tau, y(t_n + \Delta t\tau))\, d\tau = \sum_{j=1}^{\nu} b_j f(t_n + \Delta t c_j, y(t_n + \Delta t c_j)).$$

This is all well and good, except that we have to know the values $y(t_n + hc_j)$, which we do not know. We can however, play pretend and compute an approximation of these values $\xi_j \approx y(t_n + \Delta t c_j), j = 1, \dots, \nu$. The ξ_j are called *stage values*. [4], and we compute them using a linear combination of the values $f(t_n + \Delta t c_j, \xi_j)$. That is

$$\xi_i = y_n + \Delta t \sum_{j=1}^{\nu} a_{ij} f(t_n + \Delta t c_j, \xi_j),$$

for $i = 1, \dots, \nu$, where the a_{ij} are some well chosen values, which is not in scope of this thesis. To simplify notation, we note A as the square array containing the a_{ij} parameters, that is $A_{ij} = a_{ij}$, $c = (c_1, \dots, c_\nu)^\top$ the vector of nodes, and $b = (b_1, \dots, b_\nu)^\top$ the vector of weights. A RK methods is then written in the form of the following array, also called Butcher tableau.

$$\begin{array}{c|c} c & A \\ \hline & b^\top \end{array}$$

We remark that if, for for any $j \geq i$, $a_{ij} \neq 0$, then we will need to know ξ_j to compute ξ_i, which involves solving an equation, making the method *implicit*. We consider here *explicit* methods, where we can compute ξ_{i+1} if we know $\xi_j, j = 1, \dots, i-1$. Let $a_{11} = 0$ and $c_1 = 0$. An explicit RK method is then of the form

$$y_{n+1} = y_n + h \sum_{j=1}^{\nu} b_j f(t_n + \Delta t c_j, \xi_j),$$

where the stage derivative ξ_j are computed sequentially as follow

$$\begin{aligned}
\xi_1 &= y_n \\
\xi_2 &= y_n + \Delta t a_{2,1} f(t_n, \xi_1) \\
\xi_3 &= y_n + \Delta t a_{3,1} f(t_n, \xi_1) + \Delta t a_{3,2} f(t_n + \Delta t c_2, \xi_2) \\
&\vdots \\
\xi_\nu &= y_n + \Delta t \sum_{j=1}^{\nu-1} a_{\nu,j} f(t_n + \Delta t c_j, \xi_j)
\end{aligned}$$

4.2 Implementation

We now have to solve the ODE $u'(t) = e - Mu(t)$ where M depends on the problem parameters b and $\Delta x = 1/(n+1)$, and n is the chosen number of subdivisions of $[0, 1]$. Since we are only interested on the asymptotic behavior of u, we only need to care about the stability of the numerical solver we wish to use. We consider in this thesis the following RK method with two stages.

$$\begin{array}{c|cc} 0 & & \\ \alpha & \alpha & \\ \hline & 0 & 1 \end{array}$$

This solver has two parameters, namely the (pseudo) time step Δt and α. The goal is for the solver to converge to a steady state solution as fast as possible. Set $u_0 = u(0) = e$ as an initial value. We define the relative residual after k steps as

$$r_k = ||M_s u_k - e_s||/||e_s||.$$

where $||.||$ is the 2-norm.

If the solver we chose is stable, then $||r_k|| \to 0$ as $k \to \infty$. We define now the convergence at step n to be the ratio of residual at step k and $k-1$. That is

$$\rho_k = \frac{||r_k||}{||r_{k-1}||} = \frac{||Mu_k - e||}{||Mu_{k-1} - e||}$$

where $||.||$ is the 2-norm.

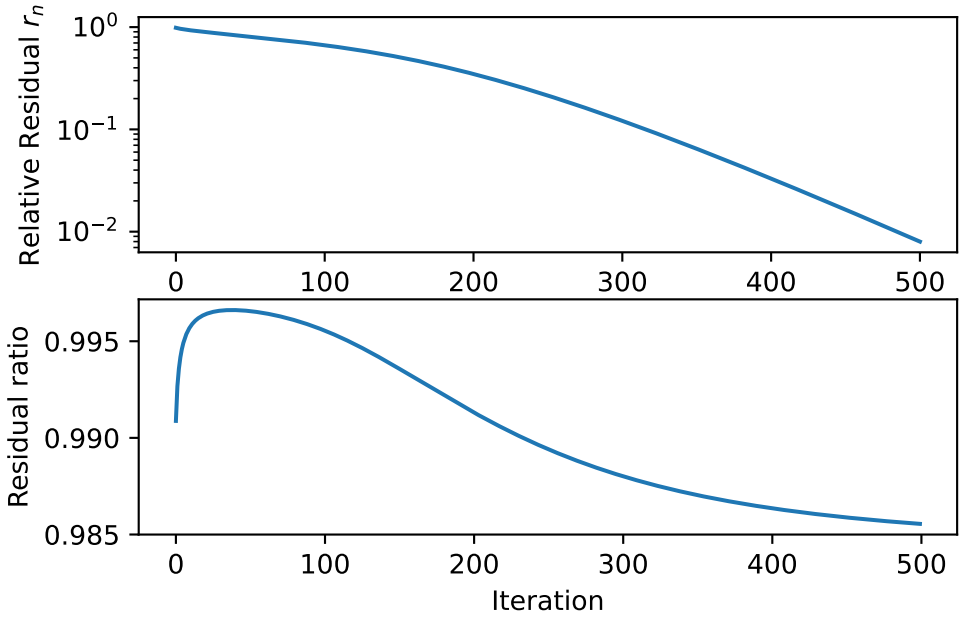


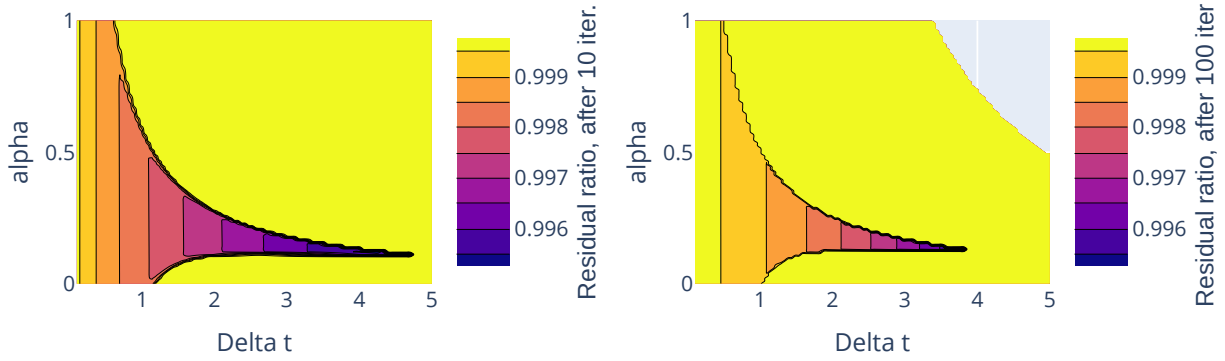
Figure 4.1: Evolution of the residual norm over iteration, with problem parameters $n = 100$ and $b = 0.05$, and RK parameters $\Delta t = 1.5$ and $\alpha = 0.3$.

4.3 A small experiment

We are interested in finding the best parameters $(\Delta t, \alpha)$ to use for some specific problem parameters (b, n). Since the residual ratio vary quite a bit depending on the number of iteration, we decide to investigate the residual ratio after 10 iterations and 100 iterations. So, for the problem parameters $b = 0.05$, and $n = 100$, we plot $\rho_{10} = f(\Delta t, \alpha)$ and $\rho_{100} = g(\Delta t, \alpha)$. We wish to answer the following questions

- Where are the optimal parameters for this specific problem, that is, the ones that minimize ρ_{10} and ρ_{100}, and do they also depend on the iteration number or not.
- What do these functions look like. In particular, we may be interested in the function convexity.

In both cases, we use a contour plot. In **?@fig-resRatio10** and **?@fig-resRatio100**, the residual ratio is clipped when it is ≥ 1 so as to maximize contrast.



(a) Surus

(b) Hanno

Figure 4.2: Famous Elephants

The stability region after 100 iterations is more narrow, suggesting that convergence may not hold even if it seems to hold for the first few iterations. Nevertheless, we can see how the parameters act on the function.

This is of course an exploration of particular problem parameters, and it is not advisable in practice to compute the optimal parameters with a grid search. We thus explore a possible solution to this problem by using a reinforcement learning algorithm to "learn" these optimal parameters.

Chapter 5

Basics of Reinforcement Learning(RL)

In this section, we outline the main ideas behind reinforcement learning and how they can be applied in the context of this thesis. The reader familiar with the material may skip this section.

5.1 A non mathematical, but delicious example

In Reinforcement Learning tasks, we have are training *agent* that interact with its *environment* by taking decisions. In this example, we are the agent, and the environment is the kitchen. Suppose we want to cook a delicious meal. At any point in time, we are making decisions such as

- which ingredients we use. Do we use tofu or seitan? Do we add spice more chili pepper? When do we incorporate the sauce?
- which cookware we use? Cast iron, or non-stick pan?
- whether to put the oven to 200° C or 220° C
- Or simply do nothing!

All of these decisions, which we will call *actions* from now on, are taken based on the current *state* of the cooking process, following a certain *policy*, which is shaped by our previous cooking experience.

After each action, the cooking process get to a new *state* and we get a *reward* that depend on how we did. Maybe the food started to burn in which case we get a negative reward, or maybe we made the food better, in which case we get a positive reward. In this example, there is also a terminal state, in which we finished cooking and get to eat the meal.

But how do we learn how to cook, that is, how do we learn the *policy*? We learn it by trying to make the food as good as possible, which is defined by the *reward* we get after each action. Some of those rewards are immediate. For example, if we add some spices to our food and it tastes better, we may be inclined to do it again the next time we cook a meal. We want to have a *policy* that maximize the total *rewards* we get, which also mean that we have to balance our decision between the immediate reward and the future rewards. Adding a spice may make the meal taste better in the short term, but it may clash later when we add other ingredients, leading to a worse meal and bad *rewards* down the line.

Each time we cook, we learn what works and what doesn't, and remember that for any future time we cook. But, if we want to get better at cooking, we must not just repeat the *actions* that worked! We also have to take some risks, and *explore* the potential actions we can take at each state! On the other hand, we still need to rely and *exploit* what we know, so there is a balance between *exploitation* and *exploration* to find so we can learn as fast as possible.

5.2 Finite Markov decision process

We formalize the above example by defining a Markov decision process (MDP) [5] , [6].

Definition 5.1. (Markov decision process). A finite Markov decision process(MDP) is defined as a discrete time process, where we have

- a state set $\boldsymbol{\mathcal{S}}$,
- an action set $\mathcal{A}$, containing all possible actions,
- for each state and each action, we have a reward set $\mathcal{R}(s,a)$, which contain the potential rewards received after taking action $a \in \mathcal{A}$ from the state $s \in \mathcal{S}$.

A Markov decision process has a model, which consist of

- the probability of getting from state s to the state s' by taking action a, which we call the state transition probability $p(s'|s,a) = \Pr(S_{t+1} = s'|s_t = s, a_t = a)$.
- the probability of getting reward r by taking the action a at a state s $p(r|s,a) = \Pr(R_{t+1} = r|S_t = s, A_t = a)$.

Furthermore, a MDP has a policy function that governs, for any state $s \in \mathcal{S}$, the probability of taking action $a \in \mathcal{A}$, that probability is $\pi(a|s) = \Pr(A_{t+1} = a|S_t = s)$.

Remark. We have implicitly defined the random variables designing the state, action, reward at a time t, those are respectively S_t, A_t, R_t. A diagram of the process is as follow

(Here is a shiny diagram, I should learn tikz..)

Remark. The state space $\mathcal{S}$ and the action space $\mathcal{A}$ can be finite or not. We only consider the case of finite Markov decision process to make matter easier, with generalization only if necessary. This also mean that the model is finite.

The model in a MDP is often impossible to define in advance. This problem is remedied by using *model free* algorithms.

Remark. A fundamental property of the MDP is Markov property, or lack of memory, that is to say that the action taken at instant t A_t is only dependent on the state S_t and not the state before. Mathematically $p(A_t|S_t, S_{t-1}, ..., S_0) = p(A_t|S_t)$.

Example 5.1. (A more mathematical example, adorable)

(More or less a gridworld example to write about)

5.3 State Value and Bellman Equation

We first define a trajectory. We note as S_t the state of an agent at instant t. Then, according to the policy, this agent takes the action A_t. After taking this action, the agent is now at the state S_{t+1}, and it gets the rewards R_{t+1}. Then the agent takes action A_{t+1}, and gets to a new state S_{t+2} with reward R_{t+2}. This can continues indefinitely. We define the trajectory of an agent with starting state $S_t = s$ as the states-rewards pairs $(S_{t+1}, A_{t+1}), (S_{t+2}, A_{t+2}), ...$

Remark. In some environments, it is natural for the agent to have a task that has a starting state and a finishing states (for example, beginning a cooking session and finishing it, or starting a game and winning/losing at it.) We call these tasks *episodic tasks* and in these cases, a finite trajectory $S_0, A_0 \to ... \to S_T$ is also called an *episode*.

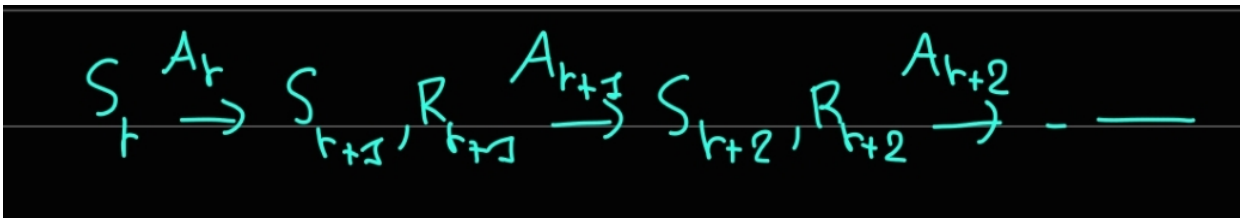


Figure 5.1: A helpful diagram showing trajectory, to be remade with tikz.

In the cases where the task is such that no such state can be defined, a trajectory is not finite and we call these tasks *continuing taks*, which will be the case in this thesis.

Ideally, we would like to chose a policy that aim to maximize rewards along any trajectory, given any starting state. This is the goal of any reinforcement learning algorithm. We now define the discounted return along a trajectory.

Definition 5.2. Let $t = 0, 1, ...$ The (discounted) return along the trajectory $S_t, A_t \to S_{t+1}, A_{t+1} \to S_{t+2}, A_{t+2} \to ...$ is the random variable given by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{t=0}^{+\infty} \gamma^t R_{t+1}$$

where $\gamma \in (0, 1)$ is called the discount rate.

Remark. It is apparent that the discount rate should be < 1 in continuing tasks to make sure that the discounted return is well defined in the case of bounded rewards, as otherwise the sum can diverge.

The discounted return is thus the sum of rewards along a trajectory, with a penalty for rewards far in the future. The discount rate is chosen depending on whether we want the agent to favor short term rewards, in which case a discount rate closer to 0 can be chosen, or long term rewards, with a discount rate closer to 1.

Since the discount rate is a random variable, we can look at its expectation, in particular, we are interested in its conditional expectation, given a starting state $S_t = s$. This expectation is called the state value.

Definition 5.3. (State value) The state value of a state s is the function, defined for any $s \in \mathcal{S}$ as

$$v_\pi(s) = E[G_t | S_t = s] = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t]$$

where π is a given policy.

Remark. The Markov property of the MDP means that the state value does not depend on time.

The objective is thus to find a policy π that maximizes the state values. We next derive the Bellman equation.

It is first apparent that

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + ... \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{5.1}$$

Inputting this into the state value yields

$$v_\pi(s) = E[G_t | S_t = s] = E[R_{t+1} | S_t = s] + \gamma E[G_{t+1} | S_t = s]$$

The first term is the expectation of immediate reward, following a certain policy π, the second is the expectation of future rewards. Let us expand on that formula a bit more. We use the law of total expectation on the first part of the RHS to get

$$E[R_{t+1} | S_t = s] = E[E[R_{t+1} | S_t, A_t]] = \sum_{a \in \mathcal{A}} \pi(a, s) \sum_{r \in \mathcal{R}} r p(r | s, a)$$

where $\mathcal{R} = \mathcal{R}(s, a)$ is the set of possible rewards one can get by taking action a at state s.

We now develop the second part of the RHS of the equation to get,

$$E[G_{t+1} | S_t = s] = E[E[G_{t+1} | S_t = s, S_{t+1}]] = \sum_{s' \in \mathcal{S}} E[G_{t+1} | S_t = s, S_{t+1} = s'] p(s' | s)$$

where $p(s'|s) = \sum_{a \in \mathcal{A}} p(s'|s, a)\pi(a, s)$ is the probability of the next state being s' if the current state is s. Because of the Markov property of the MDP, we can remove the conditioning $S_t = s$ and thus, $E[G_{t+1} | S_t = s, S_{t+1} = s'] = E[G_{t+1} | S_{t+1} = s] = v_\pi(s')$. Then

$$E[G_{t+1} | S_t = s] = \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} v_\pi(s') \pi(a|s) p(s'|s, a).$$

Putting everything together, we get a first form of Bellman equation.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a, s) \left[\sum_{r \in \mathcal{R}} r p(r|s, a) + \gamma \sum_{s' \in \mathcal{S}} v_\pi(s') p(s'|s, a) \right] \tag{5.2}$$

Remark. The Bellman equation gives a recursive relation for the state values. Solving this equation is called policy evaluation and involves fixed point iterations, which we will not get into details here.

This equation is dependent on the given policy.

5.4 Action Value

The state value gives information about a specific state, however, we are also interested in knowing how much we stand to gain by taking a particular action at a particular state. This lead to the definition of the action value.

Definition 5.4. The action value is defined as

$$q_\pi(a, s) = E\left[G_t | A_t = a, S_t = s\right]$$

We also have, from Definition 5.3, and the law of total expectation

$$v_\pi(s) = E[G_t | S_t = s] = E\left[E[G_t | S_t = s, A_t = a]\right]$$

then

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a, s) E\left[G_t | S_t = s, A_t = a\right]$$

and we get the relation between state value and action value

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a, s) q_\pi(a, s). \tag{5.3}$$

Chapter 6

Policy gradient methods

Now that we have access to the main definitions used in RL, we can now translate the problem we had at the end of chapter 4.

6.1 Translating the Test Problem

As a reminder, we have a test problem with the following problem parameters:

- a parameter $b \in [0, 1]$ in the steady-state convection diffusion equation, and
- a discretization parameter $n \in \mathbb{N}$ defining the number of points in the linear grid used to solve numerically the equation.

We end up with a linear equation to solve for, which can be solver using the method highlighted before. We wish to find the solver parameters Δt and α that will minimize the residual of REF as fast as possible. To simplify future computation, we will be interested in minimizing the residual ratio after 10 iteration of the Runge-Kutta solver c_{10}. We define this ratio as $c_{b,n}(\Delta t, \alpha)$, a function parametrized by b and n, with arguments Δt and α. We are faced with the following optimization problem:

For any b, n, find

$$(\Delta t^*, \alpha^*) = \arg\min_{\Delta t, \alpha} c_{b,n}(\Delta t, \alpha).$$

We are interested in using reinforcement learning to solve this problem. The last section provided an overview of the elements of reinforcement learning, and we can now translate our problem in a RL setting.

- A individual state can be defined as a pair $s = (b, n)$.
- An individual action can be defined as a pair $a = (\Delta t, \alpha)$.
- Given a state s, the action chosen depend on the policy $\pi(a = (\Delta t, \alpha)|s = (b, n))$. This policy can be deterministic or stochastic.
- Once a state-action pair is chosen, the residual ratio is computed. The reward can then be defined as a function of calculated residual ratio, which is defined in the next section.

The state transition model is more difficult to find a direct translation for. For the purpose of this thesis, the next state is chosen at random after computing an action and a reward. This is not ideal.

There are still several challenges that need to be addressed.

- State transition being random makes for a poor model to apply reinforcement learning to. In a typical RL scenario, the discount rate is usually set close to 1 as the agent need to take into account the

future states it will be in. Here, the next state is independent on the action taken, so it makes no sense to set the discount rate high. As a consequence, we set it low.
- In our problem, the State-Action space is continuous. We previously assumed finite spaces.
- In the definition of a MDP, the reward is a modeled random variable. This is not the case here, as we do not know in advance how the solver will behave.

The first challenge is inherent to the way we translated the problem. We answer the last two challenges in the next sections.

6.2 Model based, model free

One problem we are faced with is the problem of the model. In the last section, we assume that both $p(s'|s,a)$ and $p(r|s,a)$ are known. Depending on the problem, this is not straightforward to define. Thankfully, the model can be empirically estimated via Monte Carlo methods.

In particular, we often have to compute expectation of random variables. The most basic method is simply to sample the desired random variable and to use the empirical mean as an estimator of the desired expectation. Stochastic estimation is also used in numerous reinforcement learning algorithm.

6.3 Dealing with a large state-action space.

In the last chapter, we made the assumption that the every space, be it state, action, or reward is finite. However, this is in practice not always the case, as some state may be continuously defined for example. Even if those spaces are discrete, the *curse of dimensionality* (TODO, should something be cited) may not allow us to efficiently represent every state or action.

We take our problem as formulated before. The state is defined as the problem parameters, that is $b \in [0,1]$ and $n = 1, 2, ...$ Without any adjustment, the state space is of the form $[0,1] \times \mathbb{N}$, and is not finite.

Similarly, the policy is defined by choosing the values $(\alpha, \Delta t) \in [0,1] \times \mathbb{R}^+$, depending on the state. Once again, the action space is continuous.

One approach would be to discretize the entire state $\times$ action space, and then to apply classical dynamic programming algorithm to get some results. Then, after an optimal policy is found, do some form of interpolation for problem parameters outside of the discretized space. This approach has its own merit, as there is 3 dimensions that need to be discretized, and n can be chosen within a finite range. The main issue is that since there are no relationship between the states, solving the resulting Bellman optimal equation is effectively close to brute forcing the problem. (//TODO, this need a stronger argument instead of "My intuition said so".)

Another approach is to use approximation function. A common approach is to approximate the value function $v(s)$ by some parametrization $v(s) \approx \widehat{v}(s,\omega)$ where $\omega \in \mathbb{R}^d$ are d parameters. Such methods are called *value based*. The method we use in this thesis, on the other hand, use an approximation of the policy function defined as $\pi(a|s,\theta)$, where $\theta \in \mathbb{R}^d$ is a parameter vector is dimension d. Such method are called *policy based*. The reason to chose from this class of algorithm is two-fold.

- When thinking about the test problem, one approach which appears natural is to chose the solver parameters as a linear function of the problem parameters. A policy based approach allow us to do exactly this.
- A challenge that we are faced with is the poor model of state transition. Choosing such a linear policy allow us to find some relations between the states.

Remark. Approximation is usually done using neural networks, building on the universal approximation theorem([7]). In our case, a linear approximation is used.

6.4 Policy gradient methods.

6.4.1 Objective function.

We apply a policy gradient method to our problem. Let $\theta \in \mathbb{R}^d$ be a parameter vector and $\pi(a|s,\theta) = p(A_t = a|S_t = s, \theta)$ an approximate policy that is derivable w.r.t θ. We want to define an objective function $J(\theta)$ that we want to maximize in order to find the best value of θ.

To this end, we make the following assumptions, which are specific to our problem. For simplicity, we restrict ourselves to the discrete case.

- The states are uniformly distributed. That is, for any $s \in \mathcal{S}, p(S_t = s) = 1/|\mathcal{S}|$, where $|\mathcal{S}|$ is the number of element of S. This correspond to the idea of taking a new state at random in our problem.

We define the objective function

$$J(\theta) = \overline{v_\pi(S)} = \frac{1}{|\mathcal{S}|}\sum_{s\in S} v_\pi(s)$$

that is, $J(\theta)$ is the average, (non weighted, as per assumption) state value.

We want to maximize this objective function. To this end, we use a gradient ascend algorithm of the form

$$\theta_{t+1} = \theta_t + \alpha\nabla_\theta J(\theta). \tag{6.1}$$

We are faced with the immediate issue that the algorithm requires knowing the gradient of the objective function.

6.4.2 Policy gradient theorem

We prove, using the aforementioned assumptions the policy gradient theorem. This proof is adapted from [6],chap 13.2.

Using the expression //TODO: REF, the expression for a specific state value can be written as

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(a,s)$$

We take the gradient w.r.t θ to get

$$\nabla v_\pi(s) = \sum_{a\in\mathcal{A}} \nabla\pi(a|s)q_\pi(a,s) + \pi(a|s)\nabla q_\pi(a,s)$$

Using the expression //REF for the action value we get

$$\nabla q_\pi(a,s) = \nabla\left[\sum_r p(r|s,a)r + \gamma\sum_{s'} p(s'|a,s)v_\pi(s')\right]$$

It is apparent that the first part of the RHS is not dependent on θ, therefore, and neither is the state transition probability, the gradient becomes

$$\nabla q_\pi(a,s) = \gamma \sum_{s'} p(s'|a,s) \nabla v_\pi(s').$$

By assumption $p(s'|a,s) = 1/|\mathcal{S}|$, and thus

$$\nabla q_\pi(a,s) = \gamma \sum_{s'} \frac{1}{|\mathcal{S}|} \nabla v_\pi(s').$$

We recognize the expression for the gradient of the metric to get $\nabla q_\pi(a,s) = \gamma \nabla J(\theta)$.

$$\nabla v_\pi(s) = \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s) + \gamma \pi(a|s) \nabla J(\theta)$$

Since the policy $\pi(a|s)$ is a probability over the action space, it sums to 1 and we can get the second part of the RHS out of the sum

$$\nabla v_\pi(s) = \gamma J(\theta) + \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s)$$

Using $\nabla J(\theta) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \nabla v_\pi(s)$, we get

$$\nabla J(\theta) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left[\gamma J(\theta) + \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s) \right] \tag{6.2}$$

$$= \gamma \nabla J(\theta) + \sum_{s \in \mathcal{S}} \frac{1}{|\mathcal{S}|} \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s) \tag{6.3}$$

And after a small rearrangement of the terms

$$\nabla J(\theta) = \frac{1}{1-\gamma} \sum_{s \in \mathcal{S}} \frac{1}{|\mathcal{S}|} \sum_{a \in \mathcal{A}} \nabla \pi(a|S) q_\pi(a,S)$$

This is an expression of the policy gradient theorem. The reason to put the fraction $1/|\mathcal{S}|$ inside the first sum is to get a parallel with the more general expression, where in general, we have a weighted sum with different weight depending on the state. Depending on the metric used and the environment, this can be the stationary distribution of the states for a given policy.

We state the policy gradient theorem in a more general form

Theorem 6.1. *(Policy gradient theorem)*

Given an appropriate objective function $J(\theta)$, the gradient of the metric is proportional to the weighted sum

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(a,s) \nabla \pi(a|s)$$

where $\sum_s \mu(s) = 1$. The weights depends on how the problem is formulated. In the case of continuing problems, that is, problems where there exist no end states, $\mu(s)$ can be, for example the probability of being in state s, following the stationary distribution of the Markov process on policy π.

Remark. Depending on the model, the objective function may change. Nevertheless, the expression of the policy gradient theorem stay similar. In particular, the constant of proportionality may change.

The policy gradient theorem is powerful in the sense that we can derive the gradient of the objective function, something that is tied to the environment, to establishing the gradient of the parametrized policy function, which we have more control over.

6.4.3 REINFORCE algorithm

Here, we introduce reinforce the classic REINFORCE algorithm [8]. Even with the policy gradient theorem, we are still faced with the problem of estimating the action values q_π. But we remark that the formula in the policy gradient is an expectation.

$$\nabla J(\theta) \propto E_{S_\pi}\left[\sum_a q_\pi(a,S)\nabla\pi(a|S)\right]$$

where S_π is the stationary distribution of the state.
By using the identity $\frac{\nabla f}{f} = \nabla \ln f$, we can also rewrite the inner term as

$$\sum_a q_\pi(a,S)\nabla\pi(a|S) = \sum_a \pi(a|S)q_\pi(a,S)\nabla\ln\pi(a|S),$$

which is also an expectation, and thus

$$\nabla J(\theta) \propto E_{S\sim S_\pi, A\sim A_\pi}\left[q_\pi(A,S)\nabla\ln\pi(A|S)\right].$$

We also know from before that the action value is also an expectation of the return $G_t = E[q_\pi(S_t,A_t)]$. Therefore,

$$\nabla J(\theta) \propto E_{S_t,A_t,G_t}\left[G_t\nabla\ln\pi(A_t|S_t)\right]. \tag{6.4}$$

Since this is an expectation, we can estimate it by using samples. The k'th, which we note as e_k have to be chosen as follow.

- Chose a state $S_t = s$ at random, following its stationary distribution.
- Chose an action $A_t = a$ according to the policy $\pi(A_t = a|S_t = s)$.
- Compute the log policy gradient. Then, get the return $G_t = g$ for the state-action pair (a,s). The sample is then $e_k = g\nabla\ln\pi(a,s)$.

Then, the estimator for the RHS in Equation 6.4 is given by

$$\hat{E}_n = \frac{1}{n}\sum_{k=1}^{n} e_k$$

where n is the amount of samples we gathered. Using the gradient ascent algorithm in Equation 6.1, we can update the parameters θ.

$$\theta_{t+1} = \theta_t + \alpha\frac{1}{n}\sum_{k=1}^{n} e_k$$

One of the problem of this approach is the low sample efficiency. Indeed, to compute the return of a specific state action pair, we have to generate a sufficiently long trajectory starting from that state. This mean that any information we get about the state visited along the trajectory is discarded. However, suppose we have an episode

$$S_t, A_t \to S_{t+1}, A_{t+1} \to S_{t+2}, A_{t+2} ...$$

Then, we can estimate the return G_t, but we also have access to the trajectory

$$S_{t+1}, A_{t+1} \to S_{t+2}, A_{t+2} ...$$

and thus we can also estimate the return G_{t+1}! Therefore, we can use a single episode to estimate multiple samples! Using this idea, we can generate an episode

$$S_0, A_0 \to S_1, A_1, R_1 \to S_2, A_2, R_2 \to ... \to S_{T+1}, R_{T+1}$$

then, for any $t = 0, ..., T$, the estimated return can be defined as

$$\hat{G_t} = \sum_{k=t}^{T} \gamma^{t-k} R_{k+1}.$$

We can now state the REINFORCE algorithm, in pseudo code format

---

REINFORCE algorithm pseudocode
INPUT: A parameter vector $\theta \in \mathbb{R}^d$, and a parametrized policy $\pi(a|s, \theta)$ with derivable gradient $\nabla_\theta \pi(a|s, \theta)$.
Hyperpararameters
- Learning rate α
- Discount rate γ
- Episode length $T + 2$;
OUTPUT: The updated θ parameters ;

FOR any number of episode:
Compute an episode, following $\pi(a|s, \theta)$, of length T+2 the form $S_0, A_0 \to S_1, A_1, R_1 \to S_2, A_2, R_2 \to ... \to S_{T+1}, R_{T+1}$;
FOR t=0 ... T
Compute the estimated return $G_t = \sum_{k=t}^{T} \gamma^{t-k} R_{k+1}$;
Compute the log gradient $\nabla \ln \pi(A_t|S_t, \theta)$;
Update $\theta \leftarrow \theta + \alpha G_t \nabla \ln \pi(A_t|S_t, \theta)$;

Remark. Note that we compute the log-policy gradient and update it directly after estimating the return G_t. This mean that except for the first estimated return, we estimate the return using an outdated policy, introducing bias unless we generate new episodes often.

On the other hand, a low episode length also introduces bias in estimated returns, especially with discount rate $\gamma \approx 1$.

A balance must therefore be found for episode length in continuing cases, which is the case in this thesis.

Chapter 7

Implementation

7.1 A linear approximation of the policy

We want to have a policy of the form $(\Delta t, \alpha) = A(b, n)' + \mathrm{c}$, where A is a two by two matrix and c a 2-vector.

We first define the deterministic policy

$$\begin{pmatrix} \mu_\alpha \\ \mu_{\Delta t} \end{pmatrix} = \begin{pmatrix} \theta_0 & \theta_1 \\ \theta_2 & \theta_3 \end{pmatrix} \begin{pmatrix} b \\ n \end{pmatrix} + \begin{pmatrix} \theta_4 \\ \theta_5 \end{pmatrix} \tag{7.1}$$

As we need a stochastic policy in the REINFORCE algorithm, we add some Gaussian noise to the policy $\alpha \sim \mu_\alpha + \mathcal{N}(0, \sigma^2)$,and similarly $\Delta t \sim \mu_{\Delta t} + \mathcal{N}(0, \sigma^2)$. The term σ^2, which is the variance of the policy is chosen fixed in this thesis. Since α and Δt are chosen independently, the joint probability density of both parameters is the product of both marginal pdf, that is

$$f(\alpha, \Delta t) = f_1(\alpha) \cdot f_2(\Delta t)$$

where

$$f_1(\alpha) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\alpha - \theta_0 b - \theta_1 n - \theta_4)^2}{2\sigma^2}\right)$$

and similarly,

$$f_2(\Delta t) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\Delta t - \theta_2 b - \theta_3 n - \theta_5)^2}{2\sigma^2}\right)$$

.

Taking the logarithm, we get $\ln(f(\alpha, \Delta t) = \ln(f_1(\alpha)) + \ln(f_2(\Delta t))$. Thus,

$$\ln(f_1(\alpha)) = \ln(\frac{1}{\sqrt{2\pi}\sigma}) - \frac{(\alpha - \theta_0 b - \theta_1 n - \theta_4)^2}{2\sigma^2}$$

.

We now take the gradient w.r.t θ to get

$$\nabla_\theta \ln(f_1(\alpha)) = \xi_\alpha(b\theta_0, n\theta_1, 0, 0, \theta_4, 0)^T \tag{7.2}$$

where $\xi_\alpha = \frac{(\alpha - \theta_0 b - \theta_1 n - \theta_4)}{\sigma^2}$.

Doing a similar thing with Δt, we get the gradient,

$$\nabla_\theta \ln(f_2(\Delta t)) = \xi_{\Delta t}(0, 0, b\theta_2, n\theta_3, 0, \theta_5)^T. \tag{7.3}$$

where $\xi_{\Delta t} = \frac{(\Delta t - \theta_2 b - \theta_3 n - \theta_5)}{\sigma^2}$ We now add both gradients together to we get the gradient of the policy, for a specific action $a = (\alpha, \Delta t)$ and state $s = (b, n)$,

$$\nabla_\theta \ln \pi(a|s, \theta) = \xi_\alpha(b\theta_0, n\theta_1, 0, 0, \theta_4, 0)^T + \xi_{\Delta t}(0, 0, b\theta_2, n\theta_3, 0, \theta_5)^T. \tag{7.4}$$

We used here the standard notation. That is, $\pi(a|s, \theta) = f(\alpha, \Delta t)$.

Remark. One may remark that the REINFORCE algorithm uses a discrete policy space. This is not an issue. Instead of using the probability mass function of the policy, we will instead use the probability density function as a substitute([9]). The fact that the probability density function admits values > 1 is not an issue as we can adjust the learning rate accordingly.

7.2 State Space

In the test problem, the b parameter is a physical parameter and can take any value in $[0, 1]$, while the value of n depends on the discretization of the differential equation, and can be any integer value. As the value of n get bigger, computation of a single time step in the RK solver can take quite a bit more time, and we will be doing a lot of those computations in the coming section! Therefore, we decide to limit the value of n to an arbitrary maximum of 200. We also cap the minimum value of n to an arbitrary minimum of 5 as those values are simply too low to get a acceptable discretization error, and we do not want to train an agent to solve theses states.

In the last chapter, we wrote that the state transitions were random, more precisely, this now means that a state transition is defined as

- choosing a new value of $b \sim \mathcal{U}(0, 1)$, and
- choosing a new value of n randomly, between 5 and 200.

7.3 Computing the reward.

Once a state and action is chosen, the reward need to be computed. We said before that, for each state and action, we compute the residual ratio after 10 iterations ρ_{10}. With that ratio, we need to define an appropriate reward metrics. We design the reward such that:

- The lower the ratio ρ_{10}, the better the convergence rate and the better the reward should be.
- It appears natural to have a positive reward when $\rho_{10} < 1$, which implies convergence, and a negative reward otherwise.

The reward is

$$r(\rho_{10}) = 1 - \rho_{10}$$

When $\rho_{10} < 1$, the reward is positive as we are currently converging, and the lower the ratio, the better the convergence and thus we want a better reward. The term s is there t.

When, on the other hand $\rho_{10} \geq 1$, the reward is negative as we are diverging. The higher the ratio, the lower the reward. As the ratio can get very big with very bad parameters, we cap the negative reward at -3.

7.4 Implementation of the REINFORCE algorithm.

Now that everything as been defined, the REINFORCE algorithm can be applied to find an optimal policy.

7.4.1 A first experiment

We implement the REINFORCE algorithm to the test problem. There are a few hyperparameters to set.

- The learning rate is set to $\alpha = 2 \times 10^{-8}$.
- The discount rate is set to a low value of $\gamma = 0.1$, as the state transitions are completely random, there is no reason to prefer long term rewards.
- Because the discount rate is so low, the episodes length is set to 20 as we want to use the updated policy as often as possible.
- The standard deviation of the policy parameters is set to $\sigma = 0.1$.

This leaves the choice of the initial value for θ. While it is possible for the parameters to be random, or all set to 0, we use the experiment done in chapter 4 to use. In **?@fig-resratio10**, it seems that a policy of $\alpha = 0.3$ and $\Delta t = 2$ is reasonable. Since this was done only for a single set of problem parameters, we have no idea of the relationship between problem parameters and optimal solver parameters. Therefore, we only set the parameter $\theta_4 = 0.3$, and $\theta_5 = 2$, the other parameters are set to 0.

The algorithm is run for 50000 episodes, and we observe the evolution of the theta parameters(Figure 7.1). The objective function $J(\theta)$ can not be observed. We have access, however to the average reward received by the agent over a single episode in **?@fig-experimenttheta**. Because of the high variance of the method, we visualize the rolling average of the average episode reward, over the last $k = TODO$ episodes.

The reward in Figure 7.1 is trending upward, which is the intended behavior of the algorithm. However, there are certain problems that have been made apparent.

- Despite running the algorithm for a long time(approximately 20 minutes on a typical computer), the θ parameters have not changed very much, and it is clear that we are far from any convergence of the reward function.
- Even with smoothing, it is apparent that the method has a high variance.
- It seems that θ_1 and θ_5 varies quite a bit over time whereas the other parameters have a steady rate of change.

The slow apparent convergence rate can not be mitigated by a higher learning rate, as this empirically lead to divergences issues. The high variance is typical of reinforcement learning tasks, as well as Monte Carlo based methods, which REINFORCE is a part of. That being said, there exists much better methods that can reduce this variance, at the expense of introducing some bias, such as for example actor-critics methods [6, Ch. 13.5], or proximal policy optimization(PPO) [10]. Both of these methods are not explored in this thesis.

7.4.2 Scaling the parameters

7.4.2.1 An example of gradient descent

Consider the objective function $f(x, y) = x^2 + 9y^2$. The function admits a global minimum at $x = y = 0$, and its gradient is

$$\nabla f(x, y) = (2x, 18y)'$$

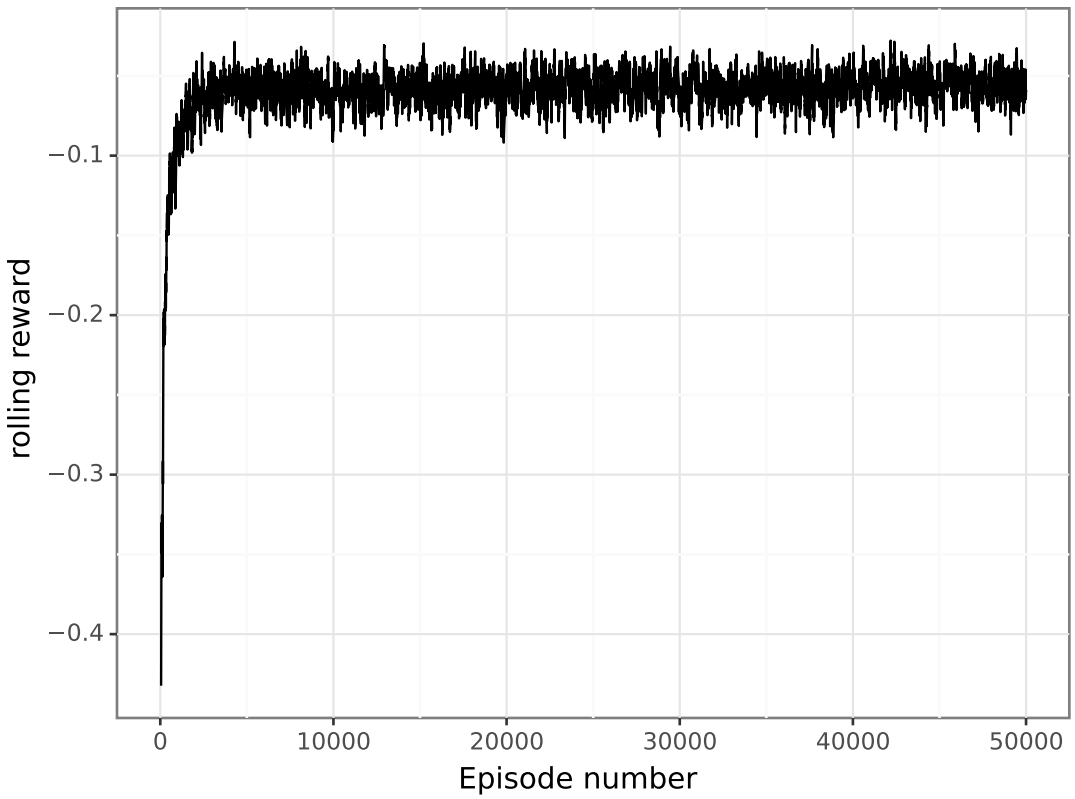


Figure 7.1: Evolution of the rolling average (k=1000) of the average episode reward.

Therefore, the gradient descent iteration, with learning rate $\alpha > 0$ is the iteration

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix} = \begin{pmatrix} x_t \\ y_t \end{pmatrix} - \alpha \begin{pmatrix} 2x_t \\ 18y_t \end{pmatrix}$$

.

That is $x_{t+1} = (1 - 2\alpha)x_t$ and $y_{t+1} = (1 - 18\alpha)y_t$. The algorithms converge to $x = y = 0$ if and only if $\alpha < 1/9$. If however, $\frac{1}{9} < \alpha < 1$, we will have convergence for x, but not for y.

The reason for this is that the gradient is steeper in the y direction than the x direction, which leads to comparatively bigger change in y than x in the gradient descent iterations.

To remedy this, we can use a change of variable $z = 3y$. Then $f(x, z) = x^2 + z^2$. The gradient descent iteration is then given by

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix} = \begin{pmatrix} x_t \\ y_t \end{pmatrix} - \alpha \begin{pmatrix} 2x_t \\ 2y_t \end{pmatrix}.$$

That is, $x_{t+1} = (1 - 2\alpha_x)x_t$ and $z_{t+1} = (1 - 2\alpha_y)y_t$. This converges to 0 if and only if $0 < \alpha < \frac{1}{2}$, which means we can afford a much bigger learning rate. With $\alpha = \frac{1}{2}$, the gradient descent algorithm can now converge to 0 in a single iteration!

7.4.2.2 Changing the variable.

We have an expression for the policy of the gradient of the policy in Equation 7.4. In particular, in Equation 7.2 and Equation 7.3, we remark that the gradient value in the directions of θ_1 and θ_3 have a similar expression. Namely, it "depends" strongly on n, which can get up to values of 200. Similarly, θ_0 and θ_2 depends on b, and θ_4 and θ_5 depend on neither. However, b can only take values between 0 and 1. This motivates the idea that the gradient may be in practice way steeper in the θ_1 and θ_3 directions than the other one.

Therefore, instead of using n directly, we implement the scaled variable

$$n' = \frac{n - 5}{200}$$

which can vary between 0 and 1. Everything then follows by simply replacing n by n' in Section 7.1. The new deterministic policy is

$$\begin{pmatrix} \mu_\alpha \\ \mu_{\Delta t} \end{pmatrix} = \begin{pmatrix} \theta_0 & \theta_1 \\ \theta_2 & \theta_3 \end{pmatrix} \begin{pmatrix} b \\ n' \end{pmatrix} + \begin{pmatrix} \theta_4 \\ \theta_5 \end{pmatrix} \tag{7.5}$$

an expression of the gradient is unchanged, with the exception of replacing n by n' everywhere.

With this change implemented, we rerun the first experiment. All the parameters are the same, except that the learning rate can now be set to $\alpha = 1 \times 10^{-4}$ without divergence! The results are in

The results are in Figure 7.2.

Compared to the last experiment, we get a much faster reward

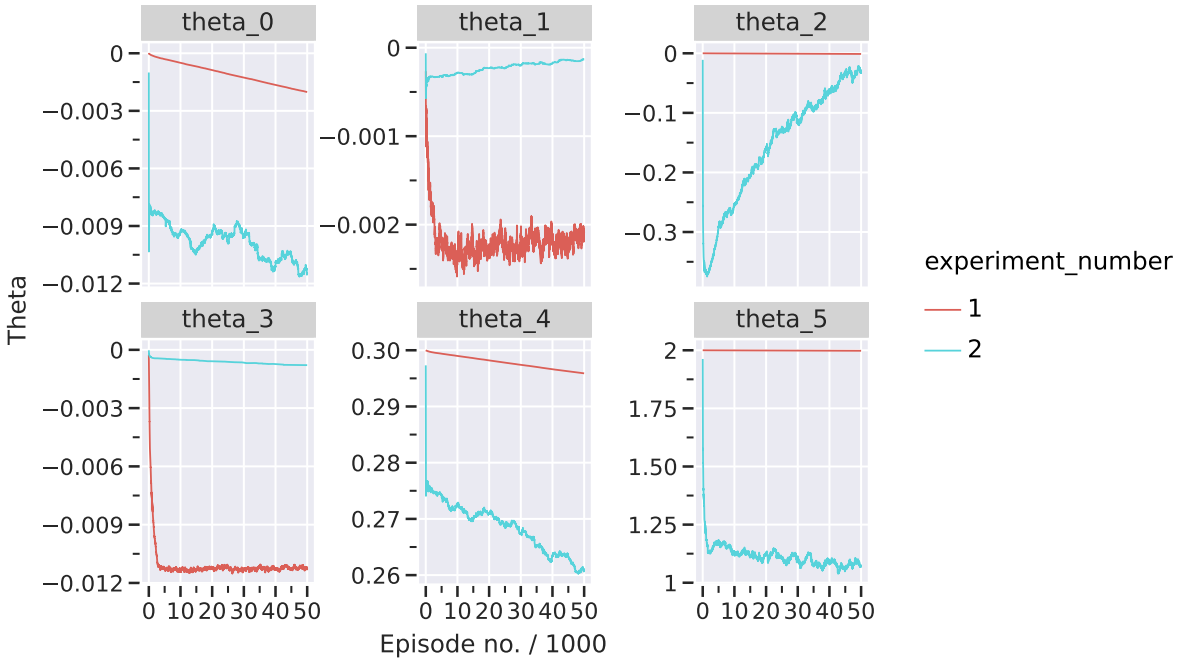


Figure 7.2: Evolution of the theta parameters with episode number, with direction based learning rate.

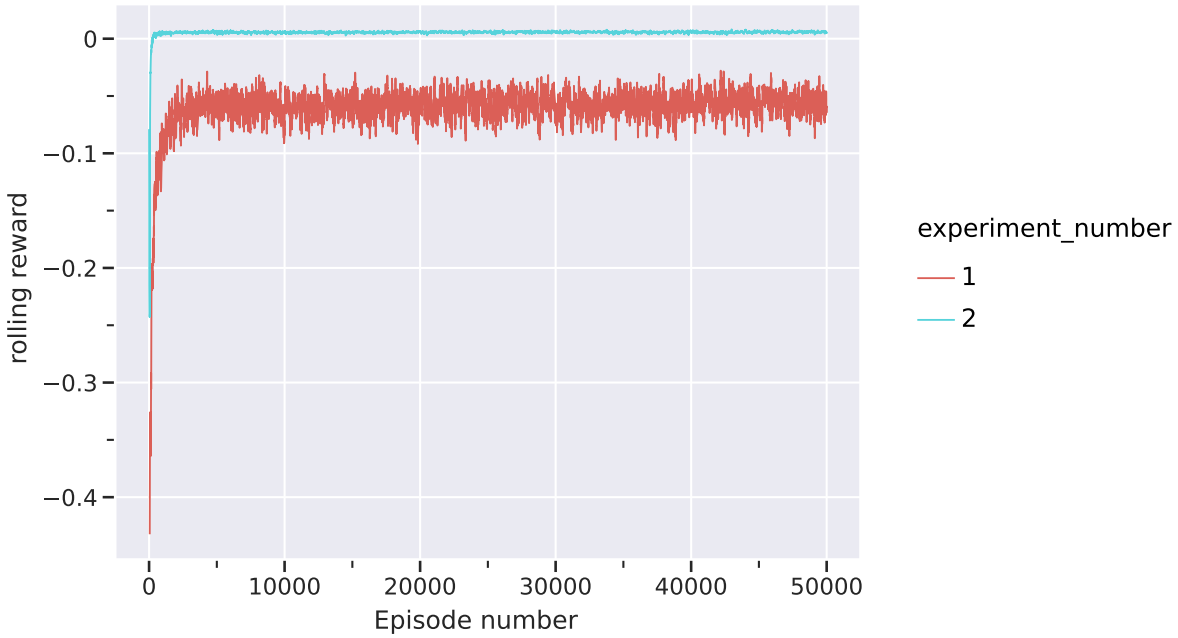


Figure 7.3: Evolution of the rolling average (k=50) of the average episode reward.

7.5 Impact of initial condition.

Gradient descent algorithm use the local information about a loss function $l(\theta)$ to compute the update $\theta \to \theta + \alpha \nabla l(\theta)$. When gradient descent converges, we can only be sure that the convergence is to a local minimum, but we can't be certain that this mimimum is a global mimimum. Therefore, it is interesting to test whether the algorithm converges to the same values regardless of initial conditions. Turns out it sorta does?

7.6 Further results

The average reward of the episode is a nice way to report on the performance of the method as it is closely linked to the return function. However, it is difficult to interpret how exactly the model is performing once we have some optimal parameters θ^*.

In particular, while the policy function is stochastic during training, the actual policy we choose can be deterministic. So, at the risk of adding some bias, we simply remove the noise σ in the policy and chose to use the deterministic policy $\alpha = \mu_\alpha$, $\Delta t = \mu_{\Delta t}$, as in Equation 7.1, and we note this policy $\pi_d(a|s, \theta^*)$. For the value of the parameters, we use their last values in the second experiment, which is

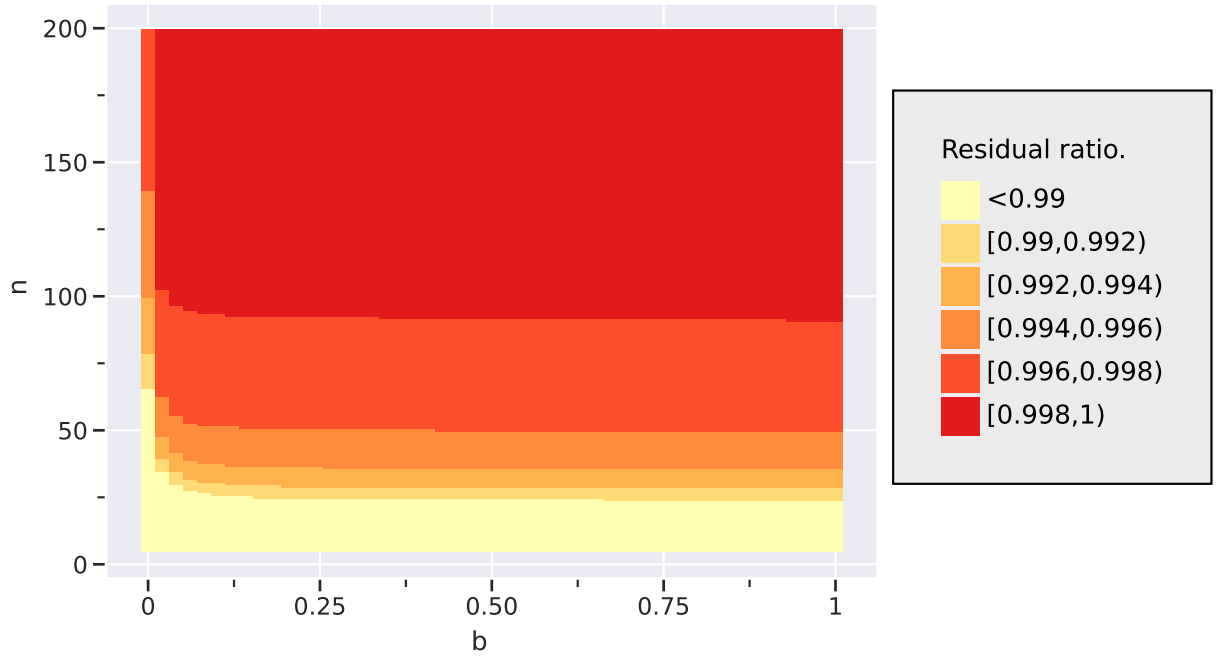
$$\theta^* = (-0.0114, -1.317e-04, -0.0318, -7.88e-04, 0.26, 1.07)^\top$$

Then, we compute the value of ρ_{10}, for this policy and for different values of n and b, the results are as follow Figure 7.4. While we have convergence at any point, the convergence is slow, and the maximum value for ρ_{10} is 0.99917. Referring back to REFFFF, this slow convergence is also partly an issue with the method, like we've seen in the grid search.

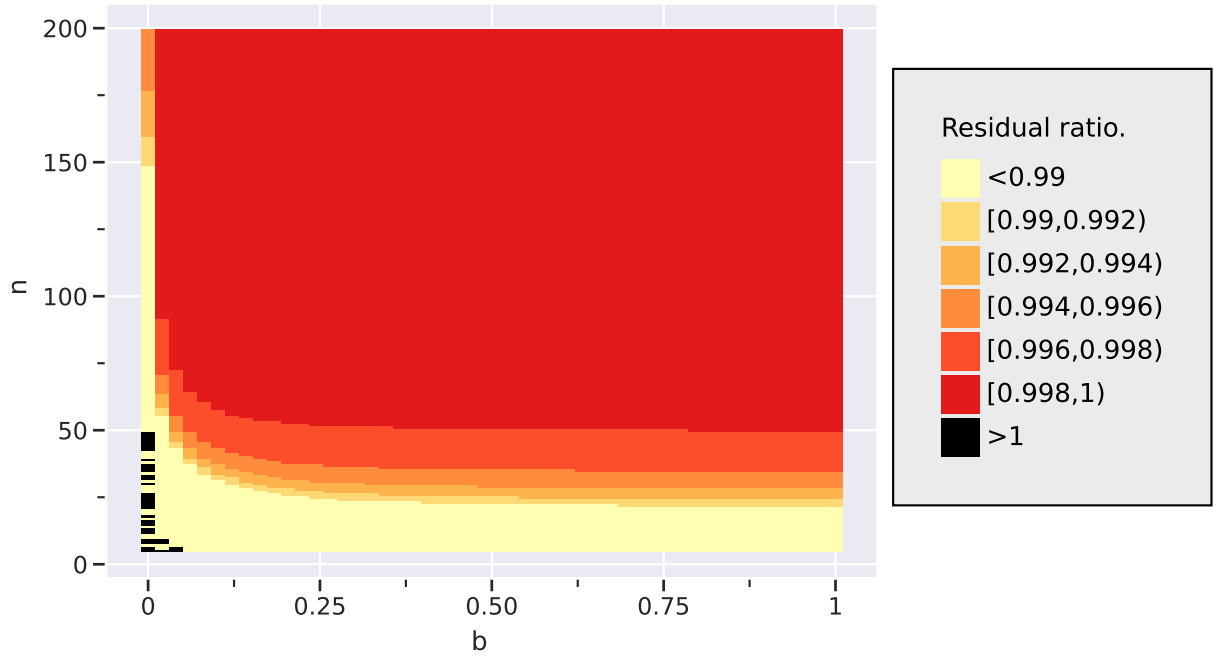
Since we trained the policy on ρ_{10}, it may be a good idea to check if the convergence still holds when we compute more iterations of the solver. The result are in Figure 7.4. There are some points where the solver diverges, which is a problem in particular because the point where it diverges are for small values of b, which is often the case physically.

This divergence indicates that it may be a good idea, as a further study to further train the learned policy by computing ρ_{100}, and having a reward of $1 - \rho_{100}$ instead of $1 - \rho_{10}$, this of course mean that the training time will be longer.

7.7 (Possible section with comparison of results).



(a) ρ_{10}. Maximum residual ratio: 0.99916.



(b) ρ_{100}. Note the divergence in black, for low values of b.

Figure 7.4: Evolution of the residual ratio ρ_{10} and ρ_{100}, with the learned policy, depending on the problem parameters n and b.

Chapter 8

Summary and discussion

In this thesis, we started with the idea of using numerical differential equations solver as an iterative solver to a linear system. More specifically, we turned our attention to a specific RK method, which has two parameters to chose from, which we called the solver parameters. We also chose a specific type of linear equation which arises from the discretization of the steady state, one dimensional convection-diffusion equation. This linear equation depends on two parameters, which we called the problem parameters. The goal was then to see if we could optimize for the solver parameters, as a function of the problem parameters, to maximize the convergence rate of the method. To do that, we used reinforcement learning. In particular, we applied the classical REINFORCE algorithm to our problem. Using the implementation in this thesis, we observed that the implemented solution works, with limited results. In particular, the optimized parameters that we learnt do not always guarantee long term convergence. There are some avenues to improve these results, in particular:

- On the technical front, the implemented algorithm is very elementary, and suffers from the issue of high sample variance, being a Monte Carlo method. This issue can be addressed by more performant algorithms.
- The policy used was a linear function of the problem parameters. We may want to explore if choosing a policy taking into account interactions between the problem parameters, or applying some transforms to them before fitting a linear policy. It is also possible to fit a neural network to the policy. Note that this most likely involves making sure that the convergence holds after the first 10 iterations.
- Possible incremental improvements can also be made. This involves for example experimenting with the reward function design, or setting a decaying learning rate to improve convergence of the RL algorithm.
-

There is on the other hand one glaring issue with the way that reinforcement learning was applied to this problem. A core philosophy of reinforcement learning is that the states, actions and rewards are all interdependent. This interdependence was absent in this thesis, with the state transition being completely random. This severely hampers the utility of using reinforcement learning instead of other methods. While it was possible to adapt this philosophy as presented in this thesis, this severely hampers the utility of using reinforcement learning instead of other methods. In particular, one may wonder if the implementation presented here is essentially "gradient descent, with extra steps".

It is therefore preferable to change how we approach the problem. One approach could be train an agent to dynamically change the solver parameters over successive iterations for some specific set of parameters. In that case, the agent would need information about the evolution of the residual, which complicates the modeling problem. Another approach would be to make use of meta learning [11]. As we are faced with an

optimization problem, we use a generic optimizer for which we can learn its optimal parameters.

References

[1] A. Fawzi *et al.*, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, Oct. 2022, doi: 10.1038/s41586-022-05172-4.

[2] Bellman Richard Ernest, *Stability theory of differential equations /*. New York : McGraw-Hill, 1953.

[3] A. Iserles, *A first course in the numerical analysis of differential equations /*, 2. ed. in Cambridge texts in applied mathematics. Cambridge ; Cambridge University Press, 2009.

[4] P. Birken, "Numerical methods for stiff problems." Lecture Notes, 2022.

[5] S. Zhao, "Mathematical foundations of reinforcement learning." 2023. https://github.com/MathFoundationRL/Book-Mathmatical-Foundation-of-Reinforcement-Learning (accessed Mar. 30, 2023).

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, Second. The MIT Press, 2018. Available: http://incompleteideas.net/book/the-book-2nd.html

[7] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, doi: https://doi.org/10.1016/0893-6080(89)90020-8.

[8] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992, doi: 10.1007/BF00992696.

[9] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning." 2019. Available: https://arxiv.org/abs/1509.02971

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017, Available: http://arxiv.org/abs/1707.06347

[11] M. Andrychowicz *et al.*, "Learning to learn by gradient descent by gradient descent," *CoRR*, vol. abs/1606.04474, 2016, Available: http://arxiv.org/abs/1606.04474