Reinforcement Learning for the Optimization of Explicit Runge Kutta Method Parameters

Mélanie Fournier

3/5/23

Table of contents

Abstract

Under some stability condition on a matrix A, solving the linear system $Ax = b$ can be done by solving the initial value problem $x' = Ax - b$. In this bachelor thesis, we study the use of this method to solve linear systems which arise from the discretization via finite differences of the one dimensional, steady state convection diffusion equation. These systems depend on two problem parameters. To solve these initial value problems, we use a two stages explicit Runge-Kutta method with two parameters to chose, which we aim to optimize using reinforcement learning.

Using policy gradient methods, and in particular the REINFORCE algorithm, we find that with some care, reinforcement learning can be used to learn the solver parameters as a function of the two problem parameters. These results are however tempered by some limitations, as the solver can diverge in certain cases, and convergence speed remains low in general. Some different approach to using reinforcement learning are thus suggested.

Acknowledgements

I would like to express my deepest thanks to my supervisor Philipp Birken at the university of Lund for the regular discussion sessions, and without which I could not have written this thesis.

I would also like to thank my partner Sarah (who also fixed an issue I struggled with for weeks in a matter of minutes), for being an amazing partner who is always there to help and encourage me in all aspects of my life. My gratitude also goes to our cat Alyx for her emotional support and the deep talks we have.

Finally, I would like to thank Paulina Ibek, who worked on a similar thesis at the same time, for the discussions which led to exchanging ideas.

Introduction

Numerical methods for differential equations are amongst the most important methods in numerical analysis. All of these methods have specific strengths 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 depend on the problem. In some cases they are taken using some heuristics, but they can also be searched for computationally.

In this thesis, we do the latter, on a specific, restricted set of problems. We first motivate the use of numerical ODE solvers to solve linear systems. As a case study, we have a specific type of linear systems, which appears when discretizing the steady state, one dimensional convection diffusion equation $u_x = bu_{xx} + 1$. Doing so, we end up with two problem parameters, b, which is a physical constant, and n, stemming from the discretization. The studied numerical solver is an explicit Runge-Kutta method, and 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 as a function of the problem parameters b and n. In particular, we use reinforcement learning (RL) to do so.

The method of reinforcement learning can claim its origin from both the fields of animal learning theory and optimal control theory [1]. The general idea is to train an agent to make decisions based on an environment that the agent can interact with, 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 rewards it gets. This approach has been used successfully in a number of problems, such as for example playing games at a high level [2], or discovering matrix multiplication algorithms [3].

In this thesis, we introduce the main concepts of reinforcement learning, such as Markov decision processes, states, actions 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 method used in this thesis makes poor use of what make reinforcement learning powerful in some cases. A discussion on how to redefine the problem to make better use of reinforcement learning follows.

Chapter 1

Motivation : Pseudo time iterations

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 the unique solution $y^* = A^{-1}b$. As directly inverting the matrix is a terrible idea, a fundamental problem in numerical analysis is to find numerical methods to solve this. This can be done with the use of direct methods or iterative methods. In this thesis, we consider an iterative method. Consider now the initial value problem(IVP):

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

where $y_0 \in \mathbb{R}^n$ and $t \in \mathbb{R}$. We adapt the result below from [4, Ch. 9.5].

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 recognize 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 $I = A^{-1}A$ inside the integral above, we get

$$e^{-At}y(t) - y(0) = A^{-1}\int_0^t -Ae^{-Au}b \; du,$$

which can be integrated to get

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

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

$$y(t) = e^{At}(y_0 - A^{-1}b) + A^{-1}b. \tag{1.1}$$

Here, we also used the fact that $e^{At}A^{-1} = A^{-1}e^{At}$. This gives an expression for the solution of the IVP.Since each of those step can be taken backward , the solution we get is unique. We have thus proved:

Theorem 1.1. *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*

$$y'(t) = Ay(t) - b,\ y(0) = y_0, \tag{1.2}$$

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. We write $\lambda_i = x_i + iy_i$, where $x_i, y_i \in \mathbb{R}$ and are respectively the real part and the imaginary parts of the i^{th} eigenvalue. Then, the following results holds[5, Ch. 1]:

Theorem 1.2. *$y(t) \to y^*$ 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.*

We call such matrices *stable* in the rest of this thesis.

Proof. We restrict ourselves to the diagonalizable case. Assume that $A \in \mathbb{R}^{n\times n}$ is diagonalizable and let $\lambda_1, \dots, \lambda_n$ be the eigenvalues of A. Then we can write $A = PDP^{-1}$ where D is the diagonal matrix with the eigenvalues of A, and P is the associated eigenvectors matrix:

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

Then $e^{At} = \sum_{i=0}^{\infty} \frac{(PDP^{-1}t)^i}{i!} = \sum_{i=0}^{\infty} P\frac{(Dt)^i}{i!}P^{-1}$. The P can be moved outside of the sum to get

$$e^{At} = Pe^{Dt}P^{-1}.$$

Since the matrix exponential of a diagonal matrix is simply the matrix of the exponentiated elements, we have

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

Let $z(t) = P^{-1}(y(t) - y^*)$, where $y(t)$ is the unique solution to Equation 1.2 for some arbitrary initial value y_0.

Since P is non singular, we can use a continuity argument to state that $y(t) \to y^*$ 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

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

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

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

The only time dependent term is $|e^{\lambda_i t}| = e^{a_i t}$, with a_i being the real part of λ_i, and $z(t)_i \to 0$ as $t \to +\infty$ if and only if $a_i < 0$.

If this holds for any $i = 1, \dots, n$, then $z(t) \to 0$ as $t \to +\infty$. This proves the sufficient condition.

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.

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 a pseudo-time variable also becomes an iterative solver for the linear system $Ay = b$, as $y(t) \to y^*$.

Remark. The eigenvalues of A are $\lambda_1, \dots, \lambda_n$. If all these eigenvalues have a strictly positive real part, then the eigenvalues of $-A$, which are $-\lambda_1, \dots, -\lambda_n$, have a strictly negative real part. Therefore, $-A$ is stable and to solve the linear problem $Ay = b$, we can simply consider the IVP $y' = (-A)y - (-b) = -Ay + b$ instead, with our best guess of y^* as the initial value.

Chapter 2

A Test Problem, the Convection Diffusion Equation

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

$$u_x = bu_{xx} + 1, \quad u(0) = u(1) = 0. \tag{2.1}$$

Here b is some physical parameter. Moreover, $u(x)$ is defined on the interval $[0, 1]$. This equation has an exact solution that is given by

$$u(x) = x - \frac{e^{-(1-x)/b} - e^{-1/b}}{1 - e^{-1/b}}. \tag{2.2}$$

We are however interested in solving this numerically, with a finite difference approach. We partition the interval $[0, 1]$ using $n + 2$ equidistant points $x_i, i = 0, ... n + 1$. We denote the distance between each points as $\Delta x = \frac{1}{n+1}$. The approximated value of u at the point x_i is denoted by u^i and we have $u^0 = u(0) = 0$ and $u^{n+1} = u(1) = 0$. 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 space, which aligns with the convection being in the right direction. For $i = 1, ..., n$, we thus have the approximation

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

This can be given in a matrix-vector format by letting $u = (u^1, \dots, u^n)^\top$:

$$Au = Bu + d,$$

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

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

and

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

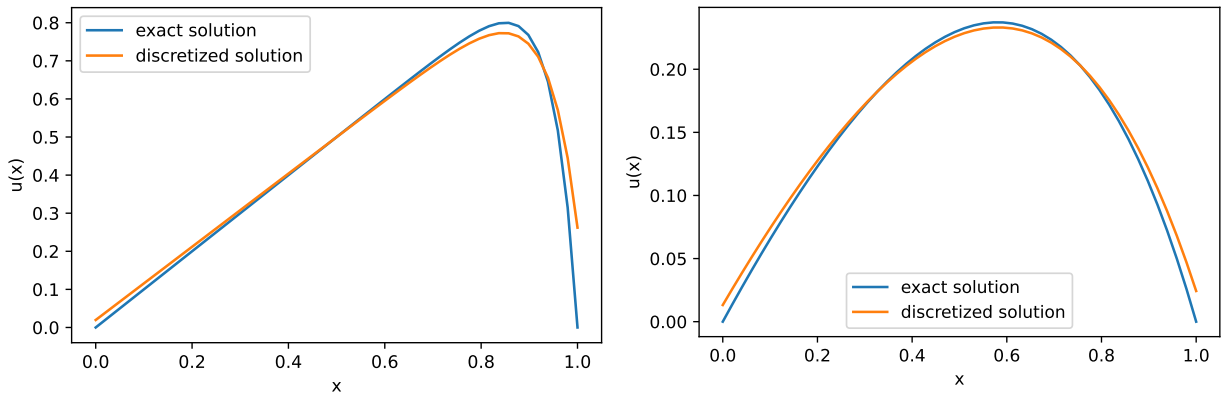
With $N = A - B$, the approximate solution of Equation 2.1 is then the solution of the linear system

$$Nu = d, \tag{2.3}$$

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. We thus only need to assume N is non singular to prove that $-N$ is stable.

We plot two examples of what the exact solution (Equation 2.2) and the discretized solution (Equation 2.3) look like for different values of b in Figure 2.1



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

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

Figure 2.1: Exact 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 choose to scale N 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 2.3 is equivalent to solving the system

$$Mu = e, \tag{2.4}$$

where with $M = \frac{N}{\eta}$, $e = \frac{d}{\eta}$. The eigenvalues of N are also scaled by $\frac{1}{\eta}$, and therefore $-M$ is stable, assuming it is non singular. We are now ready to solve the system iteratively using an ODE solver. To do that, we introduce a (pseudo) time variable t and we consider the ODE

$$u'(t) = e - Mu(t). \tag{2.5}$$

where M and e depends on both n and b, which we will call the problem parameters. We can use Theorem 1.2 with the non singularity assumption to guarantee that $u(t)$ will converge to a steady state independently of its chosen initial value. In the next chapter, we will then introduce the numerical method to solve this differential equation.

Remark. The convection diffusion equation is derived in [6] , chap. 3 from the continuity equation for a scalar quantity u and is

$$\frac{\partial u}{\partial t} + \Delta.(\vec{v}u - \nabla(Du)) = R.$$

We will assume that the quantity u, is the temperature in Kelvin, and has the S.I unit K. The physical quantities are:

- $\vec{v}$, which is the velocity of the medium the quantity is in, in ms^{-1}. (the advection/convection).
- D is the diffusion coefficient, in $\mathrm{m}^2\mathrm{s}^{-1}$.
- R is governing whether the quantity is created when $R > 0$, or destructed when $R < 0$. The unit is $K\mathrm{s}^{-1}$.

We can now simplify the equation by considering it in a single dimension x

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}.(vu - \frac{\partial}{\partial x}(Du)) = R.$$

This further simplifies to $u_t + vu_x - Du_{xx} = R$.

Then, in the steady state, $u_t = 0$ so we get

$$u_x = \frac{D}{v}u_{xx} + \frac{R}{v},$$

and we recognize Equation 2.1, with $b = \frac{D}{v}$, and $1 = \frac{R}{v}$. This also means that we "lose" two parameters in the studied test problem for simplification purposes. Nevertheless, this can be used to give some degree of intuition behind Figure 2.1. When the diffusion is high compared to the convection, the quantity is more centered, but on the other hand, when the convection speed is high compared to the diffusion, the quantity u is "flushed" to the right (assuming $v > 0$).

Chapter 3

Explicit Runge-Kutta Method

3.1 A small introduction to explicit Runge-Kutta methods

This section aims to introduce explicit Runge-Kutta methods, [7, Ch. 3], which we use in this paper. 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

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

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 numerical integration. Let $c_j \in [0, 1], j = 1, 2, \dots, \nu$, where ν is an integer, be the nodes in the quadrature formula, with their associated weight $b_j, j = 1, 2, \dots, \nu$. A quadrature formula for the integral is then of the form

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

This is all well and good, except that we have to know the values $y(t_n + \Delta c_j)$, which we do not possess. 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*. [8]. The main idea to use the ξ_i's to compute ξ_j, using a linear combinations of the terms $f(t_n + \Delta t c_j, \xi_i)$. 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. An RK method is then written in the form of the following array, also called a Butcher tableau:

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

We remark that if, 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 $\xi_j, j = 1, \dots, i-1$. Since we know $f(t_n, y_n)$, we choose $a_{11} = 0$ 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 values ξ_j are computed sequentially as follow

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

3.2 Stability theory for numerical solver

This is where we do the stability theory.

3.3 Application to the test problem

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 $[0, 1]$. We consider in this thesis the following RK method with two stages [8]:

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

Remark. This RK method can be extended to more stages. We only need the last stage value to compute the time step update, and we only need to compute the stage values sequentially using only the last stage value calculated. This makes it possible, when programming the method, to simply to do the update of the variable ξ in place inside the computer memory. Such methods are thus memory efficient.

This solver has two parameters, namely the (pseudo) time step Δt and α, where $\alpha \in [0, 1]$.

The goal is for the solver to converge to a steady state solution as fast as possible. We set $u_0 = u(0) = e$ as an initial value. We define the relative residual after k steps as

$$r_k = \frac{||Mu_k - e||}{||e||}, \tag{3.1}$$

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 residual ratio at step k to be the ratio of the residuals at step k and $k-1$. That is

$$\rho_k = \frac{||r_k||}{||r_{k-1}||} = \frac{||Mu_k - e||}{||Mu_{k-1} - e||}. \tag{3.2}$$

Figure 3.1 shows the evolution of the relative residual, as well as the residual ratio for specific parameters. After a certain number of iterations, the residual ratio stabilizes. This can be however be after a large amount of iterations, so the rate of convergence can be costly to compute.

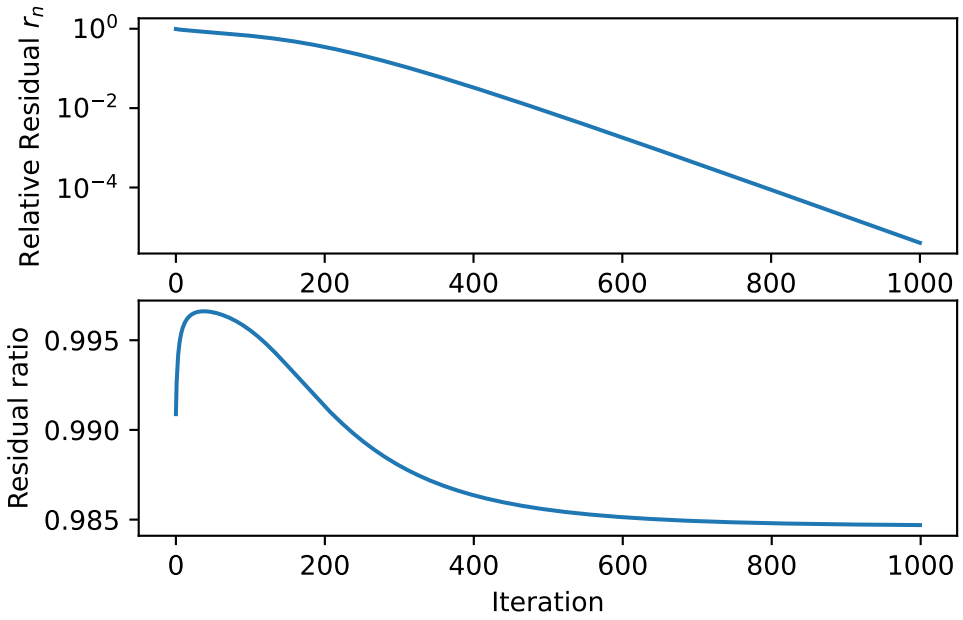


Figure 3.1: Evolution of the residual norm over iteration, with problem parameters $n = 50$ and $b = 0.05$, and RK parameters $\Delta t = 1$ and $\alpha = 0.2$.

Remark. Using the same notation as before for the stage values and the studied RK method, for the equation $u'(t) = f(t, u(t))$, we have $\xi_1 = u_n$,

$$\xi_2 = u_n + \Delta t \alpha f(t_n, \xi_1) = u_n + \Delta t \alpha f(t_n, u_n).$$

The update is thus;

$$u_{n+1} = u_n + \Delta t f(t_n + \alpha \Delta t, \xi_2).$$

In the test problem case, $f(t_n, u_n) = e - Mu_n$, and we get the update

$$u_{n+1} = u_n + \Delta t \left[e - M(u_n + \alpha \Delta t (e - Mu_n)) \right].$$

After a few lines of computation, we get the following iteration:

$$u_{n+1} = \left[I - \Delta t (I - \alpha \Delta t M) M \right] u_n + \left[\Delta t (I - \alpha \Delta t M) \right] e. \tag{3.3}$$

This iteration is of the form $u_{n+1} = Ku_n + Le$, where

$$L = \Delta t \left[I - \alpha \Delta t M \right] \tag{3.4}$$

and

$$K = I - LM = \left[I - \Delta t (I - \alpha \Delta t M) M \right]. \tag{3.5}$$

We recognize this iteration as a linear stationary iteration, which converges to $u^* = M^{-1}e$ if and only if $\rho(K) < 1$, [9, Ch. 2.2], where $\rho(K)$ is the spectral radius of K.

3.4 A small experiment

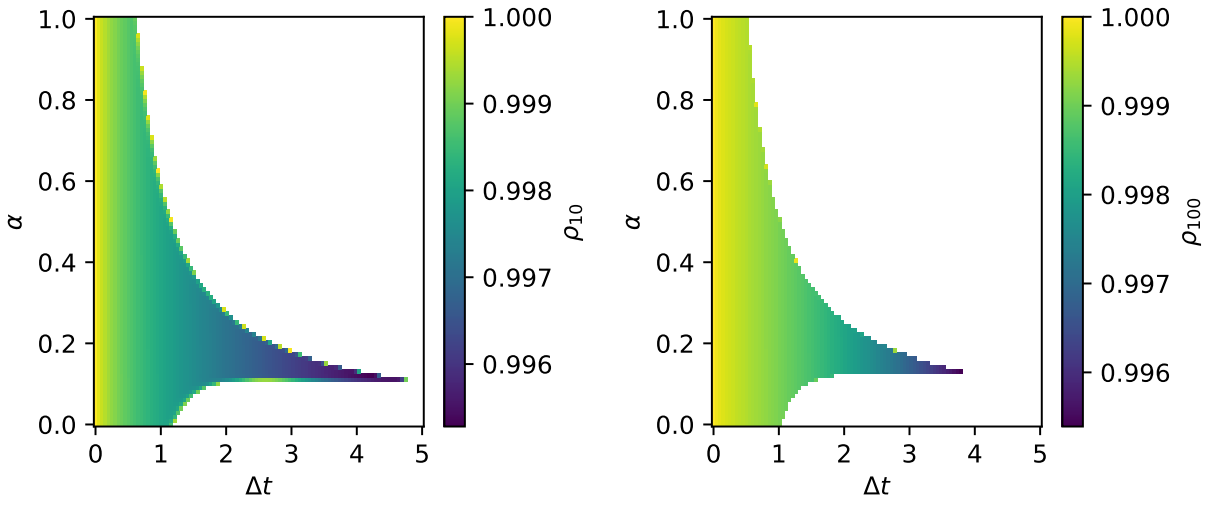
We are interested in finding the best parameters $(\Delta t, \alpha)$ to use for some specific problem parameters (b, n). Ideally, we should minimize the asymptotic residual ratio ρ_∞, but this is computationally intensive, so we restrict ourselves to minimizing the residual ratio ρ_i after a fixed amount of iterations.

As we've seen in Figure 3.1, ρ_i can vary quite a bit depending on i, so we decide to investigate the residual ratio after 10 iterations and 100 iterations. We set the problem parameters $b = 0.05$, and $n = 100$, and we plot $\rho_k = f_k(\Delta t, \alpha)$ for different values of k. This is achieved by making a linear grid for parameters Δt and α of size 100×100, where α varies between 0 and 1, and Δt varies between 0 and 5, then computing the residual ratios on that grid.

We wish to find the optimal parameters for this specific problem, that is, the ones that minimize ρ_k, for different values of k. We are also interested in seeing how much the optimal parameters depend on k.

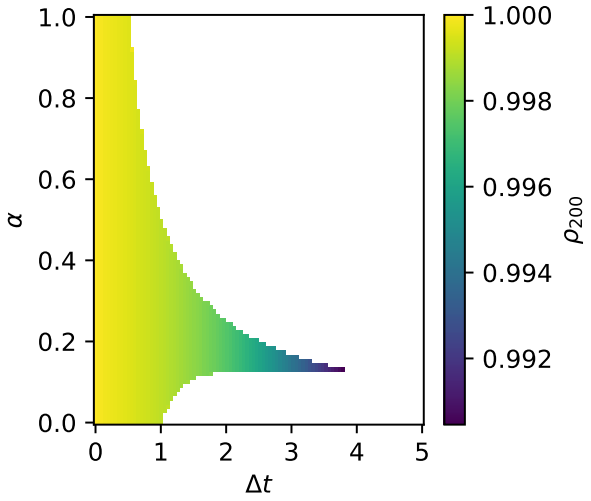
After 100 iterations, we see that we need to choose the parameters in more narrow region than after 10 iterations to get $\rho_{100} < 1$, suggesting that convergence may not hold even if it seems to hold for the first few iterations. However, this doesn't seem to be the case when we consider higher values of k. Nevertheless, we can see how the solver parameters interact with the residual ratio.

By doing this experiment, we motivate the following method: using a grid search, look for the solver parameters that minimize ρ_k, where k has to be chosen as low as possible to minimize computing time, but also high enough to ensure that the solver won't diverge after more iterations. This method however need to be repeated for each individual problem parameters. We therefore explore a possible solution to this problem by using a reinforcement learning algorithm to "learn" the optimal solver parameters.

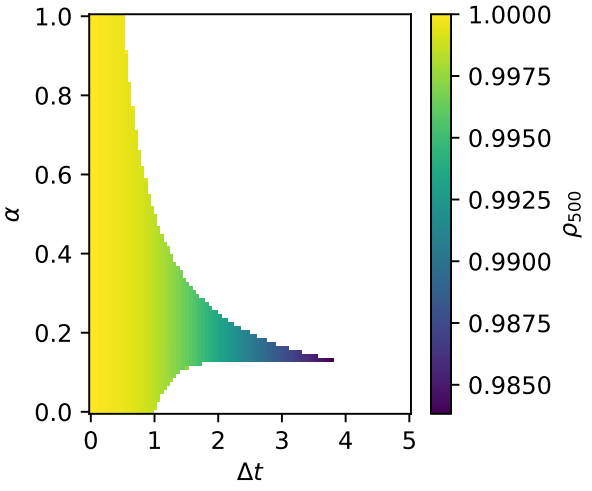


(a) ρ_{10}

(b) ρ_{100}



(c) ρ_{200}



(d) ρ_{500}

Figure 3.2: Contour plot of some residual ratios ρ_k, for different k after different number of iterations, for the specific problem parameters $n = 100$ and $b = 0.05$. Note that the area in white is where $\rho_k > 1$.

Chapter 4

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.

4.1 A non mathematical, yet delicious example!

In Reinforcement Learning tasks, we are training an *agent* that interacts 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 or 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 well 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 enjoy the meal.

But how do we learn how to cook, how do we know what *action* to take at a specific *state*? That is, how do we learn the *policy*? We learn it by trying to make the food as flavorful 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 want to have a *policy* that maximizes the total *rewards* we get over a entire cooking session. This also mean that we have to balance how we prefer the immediate rewards against the future rewards. For example, adding a spice may make the meal taste better in the short term, for which we get a reward, but it may clash later when we add other ingredients, leading to a worse meal and worse *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. There is a balance to find between *exploitation* and *exploration* so as to learn as fast as possible.

4.2 Finite Markov decision process

We formalize the above example by defining a Markov decision process (MDP) [10].

Definition 4.1. (Markov decision process). A finite Markov decision process (MDP) is defined as a discrete time process, where we have:

- A finite state set $\mathcal{S}$.
- An finite action set $\mathcal{A}$, containing all possible actions.
- A reward set $\mathcal{R}(s, a)$, which contains the potential rewards received after taking any action $a \in \mathcal{A}$ from any state $s \in \mathcal{S}$.

We use the notation S_t, A_t as the state and action of the process at time t. The reward R_t is the reward received at time t. S_t, A_t and R_t are random variables.

A Markov decision process also has a model, which consists 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 Markov decision process has a policy that governs, for any state $s \in \mathcal{S}$, the probability of taking action $a \in \mathcal{A}$, that probability is $\pi(a|s) = \Pr(A_t = a|S_t = s)$.

Finally, a Markov decision process has the Markov property, or lack of memory. The action taken at instant t A_t is only dependent on the current state S_t and not the state before. Mathematically, $p(A_t|S_t, S_{t-1}, \dots, S_0) = p(A_t|S_t)$.

An example of Markov decision process with two states can be seen in Figure 4.1.

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 matters easier. This also mean that the model is finite.

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

Example 4.1. A Markov decision process is the mathematical formalization of an agent interacting with its environment [11]. Consider the following grid, in which Leonardo the rabbit is in. Leonardo wants to get to the carrot.

4.3 State Value and Bellman Equation

The Markov decision process are the mathematical formalization of an agent (for example a chess player) interacting with its environment (the chess board)[11].

We first define a trajectory. We denote by 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}.

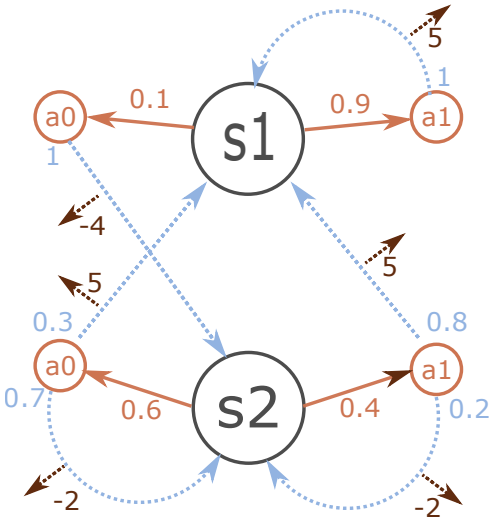


Figure 4.1: An example of Markov decision process with two states and two possible actions for each state. The dashed lines represent the model. After each action, a reward is given, here in dark red.

This can continues indefinitely. We define the trajectory of an agent with starting state $S_t = s$ as the chain of state, actions and rewards from time t to time $t + \tau$:

$$S_t, A_t \to R_{t+1}, S_{t+1}, A_{t+1} \to R_{t+2}, S_{t+2}, A_{t+2} \to \cdots \to R_{t+\tau}, S_{t+\tau}, A_{t+\tau},$$

where τ can either be finite or infinite. Note that due to the Markov property, the value of t is not important.

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*. 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 tasks*, which will be the case in this thesis.

The goal of any reinforcement learning algorithm is to maximize the rewards the agent get, which we elaborate on in the next sections. We now define the discounted return along a trajectory,

Definition 4.2. 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

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

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

Remark. By setting a discount rate that is less than 1 in continuing tasks, we make sure that the discounted return is well defined in the case of bounded rewards. Indeed, if, for any t, $|R_t| \leq M$, then $\sum_{k=0}^{+\infty} |\gamma^k R_{t+1+k}| \leq \sum_{t=0}^{+\infty} \gamma^t M = \frac{M}{1-\gamma}$, so the series converges absolutely.

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 return 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 4.3. State value@Sutton1998 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.

We have

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + ... \right) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{4.1}$$

This expression of the return can be used in conjunction with the definition of the state value above to get

$$v_\pi(s) = E[G_t|S_t = s] = E[R_{t+1}|S_t = s] + \gamma E[G_{t+1}|S_t = s]. \tag{4.2}$$

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 expectations on the first part of the RHS to get

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

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

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

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

$$E[G_{t+1}|S_t = s] = \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} v_\pi(s')\pi(a|s)p(s'|s, a). \tag{4.3}$$

Putting Equation 4.2 and Equation 4.3 together, we get Bellman's equation:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a, s) \left[\sum_{r \in \mathcal{R}} rp(r|s, a) + \gamma \sum_{s' \in \mathcal{S}} v_\pi(s') p(s'|s, a) \right]. \tag{4.4}$$

Remark. The Bellman equation is dependent on the given policy and gives a recursive relation for the state values. Solving this equation is called policy evaluation which involves fixed point iterations(see example below).

Example 4.2. We can derive directly the state values in the MDP in Figure 4.1. In particular, for the state s_2. There are two possible actions a_0 and a_1 we can take. The policy is to take action a_0 with a probability 0.6, and action a_1 with a probability 0.4. When we take for example action a_0, the probability of the next state being s_1 is 0.3, in which case the reward is 5. Proceeding similarly for all the possible actions and rewards, we get

$$v_\pi(s_2) = 0.6\left[0.7 \times -2 + 0.3 \times 5 + \gamma(0.3 v_\pi(s_1) + 0.7 v_\pi(s_2))\right] + 0.4 \times [\dots].$$

After some computations, we end up with

$$v_\pi(s_2) = 1.5 + \gamma(0.5, 0.5) \begin{pmatrix} v_\pi(s_1) \\ v_\pi(s_2) \end{pmatrix}.$$

Similarly $v_\pi(s_1) = 4.1 + \gamma(0.9, 0.1)(v_\pi(s_1), v_\pi(s_2))^\top$. This leads to the system:

$$\begin{pmatrix} v_\pi(s_1) \\ v_\pi(s_2) \end{pmatrix} = \begin{pmatrix} 4.1 \\ 1.5 \end{pmatrix} + \gamma \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} v_\pi(s_1) \\ v_\pi(s_2) \end{pmatrix}.$$

We stop here to remark that this equation is of the form $v_\pi = r_\pi + \gamma P_\pi v_\pi$. P_π is row stochastic, and is the transition matrix of the Markov chain we get when we consider a given policy. Furthermore, since $\gamma < 1$, we motivate solving the equation by using a fixed point iteration. This is the main idea behind *dynamic programming* [12]. In this case, we can simply solve the system directly. With $\gamma = 0.5$, we have the state values $v_\pi(s_1) = 7.875$, $v_\pi(s_2) = 4.625$. There is, for the given policy, more value in being in the first state than the second.

4.4 Action Value

The state value gives information about a specific state, however, we are also often 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 4.4. Action value 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 4.3, 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 can get the relation between state value and action value:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a, s) q_\pi(a, s). \tag{4.5}$$

Chapter 5

Policy Gradient Method

Now that we have access to the main definitions used in RL, we can study the problem we had at the end of chapter 3 through the lens of RL.

5.1 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 Equation 3.1 as fast as possible. We are interested in minimizing the residual ratio after 10 iteration of the Runge-Kutta solver ρ_{10}. This is not ideal as we have seen that optimal parameters for ρ_{10} can lead to divergence of the solver, but this reduces computation time significantly.

We define this ratio as $\rho_{10}(\Delta t, \alpha)$, 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

$$(\Delta t^*, \alpha^*) = \arg\min_{\Delta t, \alpha} \rho_{10}(\Delta t, \alpha).$$

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 in 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.

5.2 Model-based, model-free

One problem we are faced with is the issue 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 the expectation of functions 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.

5.3 Dealing with a large state-action space.

In the last chapter, we made the assumption that every space, be it state, action, or reward is finite. However, this is in practice not always the case, as some states may be continuously defined for example.

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 algorithms 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.

Another approach is to use an approximation function. One way to do that is approximate the value function $v(s)$ by some parametrization $v(s) \approx \hat{v}(s, \omega)$ 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 methods 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. In the case of this thesis, a linear approximation is used.

5.4 Policy gradient methods.

5.4.1 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 assumption, which are specific to our problem. For simplicity, we also restrict ourselves to the finite 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 by changing the policy parameter θ. To this end, we use a gradient ascend algorithm of the form

$$\theta_{t+1} = \theta_t + \alpha\nabla_\theta J(\theta). \tag{5.1}$$

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

5.4.2 Policy gradient theorem

We prove, using the aforementioned assumptions the policy gradient theorem. This proof is adapted from [11],chap 13.2. To unclutter the notation, we remove θ in $\pi(a|s,\theta)$. The gradient ∇ are also implied to be w.r.t θ.

From the last chapter, we have the expression of the state values

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

We take the gradient of $v_\pi(s)$ w.r.t θ to get

$$\nabla v_\pi(s) = \sum_{a\in\mathcal{A}} \nabla\pi(a|s)q_\pi(a,s) + \pi(a|s)\nabla q_\pi(a,s). \tag{5.2}$$

We now turn our attention to the $\nabla q_\pi(a,s)$ term above. Using the definition of the action value(Definition 4.4) and the definition of return G_t (Definition 4.2), 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 the policy, and therefore θ, and neither is the state transition probability $p(s'|a,s)$. The gradient is thus

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

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 of the metric's gradient $\nabla J(\theta)$ to get $\nabla q_\pi(a,s) = \gamma \nabla J(\theta)$. We insert this in Equation 5.2 and we get

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

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

$$\nabla J(\theta) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left[\gamma J(\theta) + \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s) \right], \tag{5.3}$$

$$= \gamma \nabla J(\theta) + \sum_{s \in \mathcal{S}} \frac{1}{|\mathcal{S}|} \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s). \tag{5.4}$$

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 5.1. ***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 long term probability of being in state s.

Remark. Depending on the model, the objective function that is used 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.

5.4.3 REINFORCE algorithm

Here, we introduce reinforce the classic REINFORCE algorithm [13]. 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:

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

where S_π is the random variable with probability mass function $\mu(s)$. 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 $G_t = E[q_\pi(S_t, A_t)]$. Thus,

$$\nabla J(\theta) \propto E_{S_t,A_t,G_t}\left[G_t\nabla\ln\pi(A_t|S_t)\right]. \tag{5.5}$$

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 5.5 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 5.1, 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 a trajectory

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

Then, we can estimate, via Monte Carlo estimation, the return G_t.Doing this, we also have access to the trajectory

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

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 of length $T + 2$:

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

For any $t = 0, \dots , T$, the estimated return is then 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 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 $\hat{G_t} = \sum_{k=t}^{T} \gamma^{t-k} R_{k+1}$;
Compute the log gradient $\nabla \ln \pi(A_t|S_t, \theta)$;
Update $\theta \leftarrow \theta + \alpha \hat{G_t} \nabla \ln \pi(A_t|S_t, \theta)$;

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 the estimated returns, especially with discount rates $\gamma \approx 1$.

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

Remark. The REINFORCE update can be interpreted as updating the parameters to make it more likely to take an action if the estimated sample return is good, and the opposite otherwise. Furthermore, by looking at the term $\nabla \ln \pi = \frac{\nabla \pi}{\pi}$, we can see that if the probability of taking the action is rare, then the gradient becomes bigger! That way, this gradient act as a balance between exploration and exploitation. Otherwise we would update the parameters as much for a rare action than a common one, and the common action is taken more often which lead to the common action having much more sway in the process.

Chapter 6

Implementation

6.1 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

$$\begin{pmatrix} \mu_\alpha \\ \mu_{\Delta t} \end{pmatrix} = \begin{pmatrix} \theta_0 & \theta_1 \\ \theta_2 & \theta_3 \end{pmatrix} \begin{pmatrix} b \\ n \end{pmatrix} + \begin{pmatrix} \theta_4 \\ \theta_5 \end{pmatrix}. \tag{6.1}$$

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

$$\nabla_\theta \ln(f_1(\alpha)) = \xi_\alpha (b\theta_0, n\theta_1, 0, 0, \theta_4, 0)^\top, \tag{6.2}$$

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,

$$\nabla_\theta \ln(f_2(\Delta t)) = \xi_{\Delta t}(0, 0, b\theta_2, n\theta_3, 0, \theta_5)^\top, \tag{6.3}$$

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 $s = (b, n)$:

$$\nabla_\theta \ln \pi(a|s, \theta) = \xi_\alpha(b\theta_0, n\theta_1, 0, 0, \theta_4, 0)^T + \xi_{\Delta t}(0, 0, b\theta_2, n\theta_3, 0, \theta_5)^\top. \tag{6.4}$$

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([14]). The fact that the probability density function admits values > 1 is not an issue as we can adjust the learning rate accordingly.

6.2 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 for these 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.

6.3 Computing the reward.

Once a state and action is chosen, the reward need to be computed. 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 then set to be

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

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. 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.

6.4 Implementation of the REINFORCE algorithm.

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

6.4.1 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 $\alpha = 1 \times 10^{-8}$.
- The discount rate is set to $\gamma = 0$, as the state transitions are completely random, there is no reason to prefer long term rewards.
- Because the discount rate is so low, there is no bias added by estimating the returns at the end of the episodes. 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 Figure 3.2a, it seems that a policy of $\alpha = 0.3$ and $\Delta t = 2$ is a reasonable choice. 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 6.1).

Since the discount rate is set to 0, in any state, the return is the instant reward received by the agent over a single episode. So, for an episode of length l, we have the rewards $r_1, r_2, \dots, r_l$. Then, we can plot the average reward $r_{av} = \frac{r_1 + r_2 + \cdots + r_l}{l}$ over each episode. Because the variance of r_{av} is still high, we use the rolling average of r_{av} over the last $k = 50$ episodes as a smoother.

The average reward is the no scaling reward in Figure 6.2 and is trending upward with successive episodes, which is the intended behavior of the algorithm. However, there are certain problems that have been made apparent by the two plots:

- Despite running the algorithm for a long time, some of the θ parameters have barely changed, 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.
- After the full It seems that θ_1 and θ_3 vary 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 leads to divergence issues.

The high variance is typical of reinforcement learning tasks, and in particular 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 [11, Ch. 13.5], or proximal policy optimization(PPO) [15]. Both of these methods are not explored in this thesis.

6.5 Scaling the parameters

6.5.1 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 given by

$$\nabla f(x, y) = (2x, 18y)^\top.$$

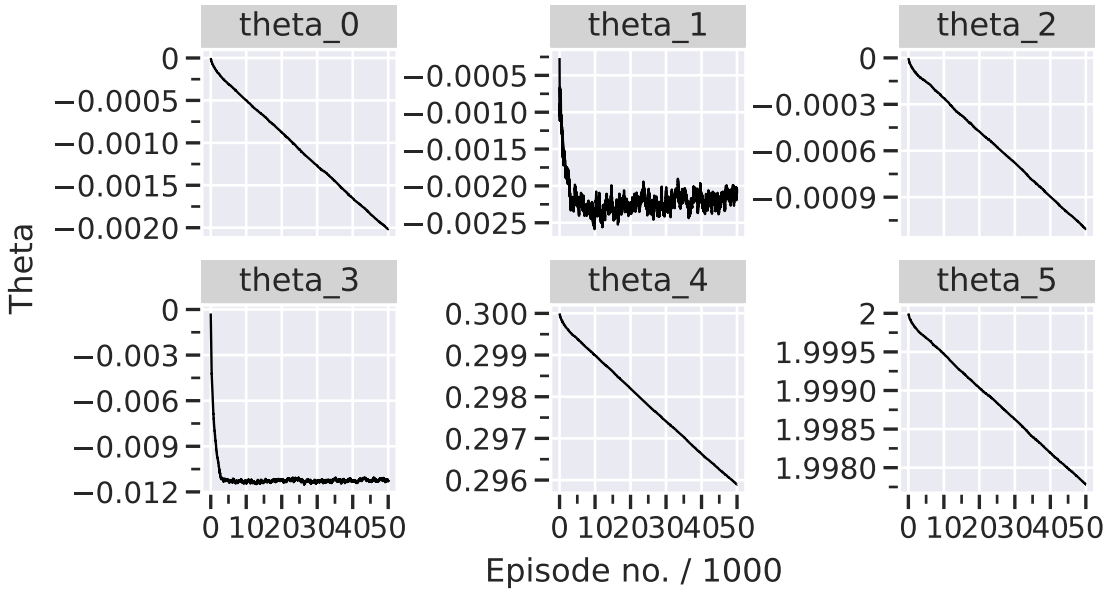


Figure 6.1: Evolution of the θ parameters in a first experiment.

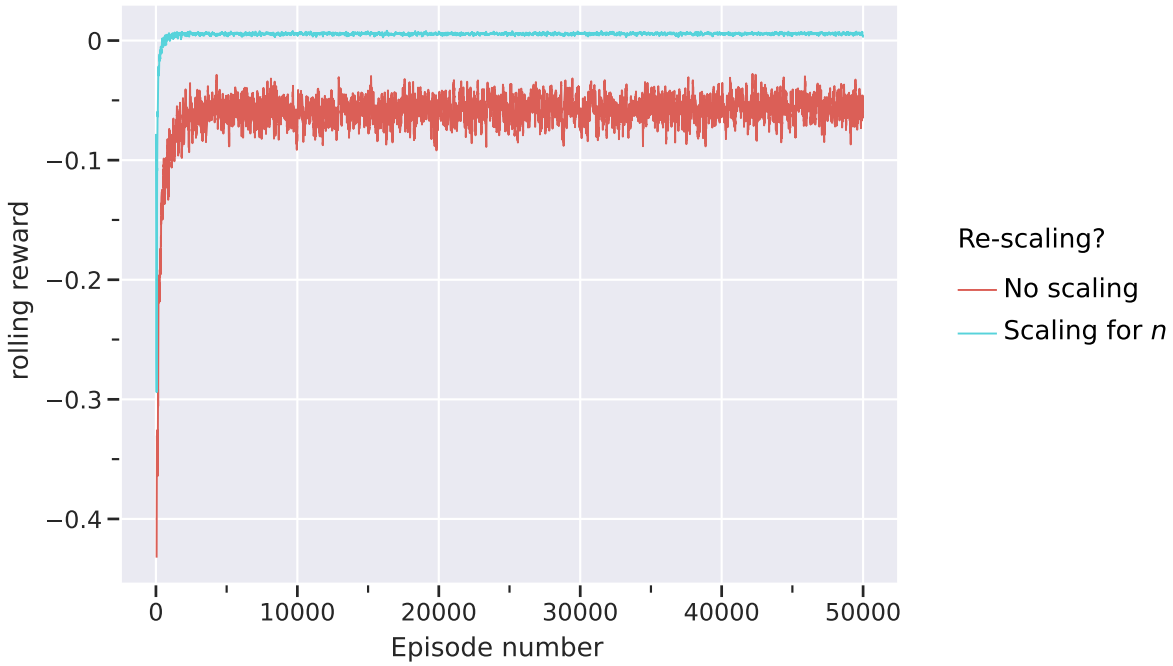


Figure 6.2: Evolution of the rolling average (k=50) of the average episode reward, with or without scaling.

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!

6.5.2 Changing the variable

We have an expression for the policy of the gradient of the policy in Equation 6.4. In particular, in Equation 6.2 and Equation 6.3, 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 ones.

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

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

which can now vary between 0 and 1. Everything then follows by simply replacing n by n' in Section 6.1. The new deterministic policy is

$$\begin{pmatrix} \mu_\alpha \\ \mu_{\Delta t} \end{pmatrix} = \begin{pmatrix} \theta_0 & \theta_1 \\ \theta_2 & \theta_3 \end{pmatrix} \begin{pmatrix} b \\ n' \end{pmatrix} + \begin{pmatrix} \theta_4 \\ \theta_5 \end{pmatrix}, \tag{6.5}$$

and the 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 is now set to $\alpha = 2 \times 10^{-4}$ without divergence. Compared to the first experiment, the evolution of the policy parameters is much higher(see Figure 6.3)! The average episode reward is also much better, as seen in Figure 6.2.

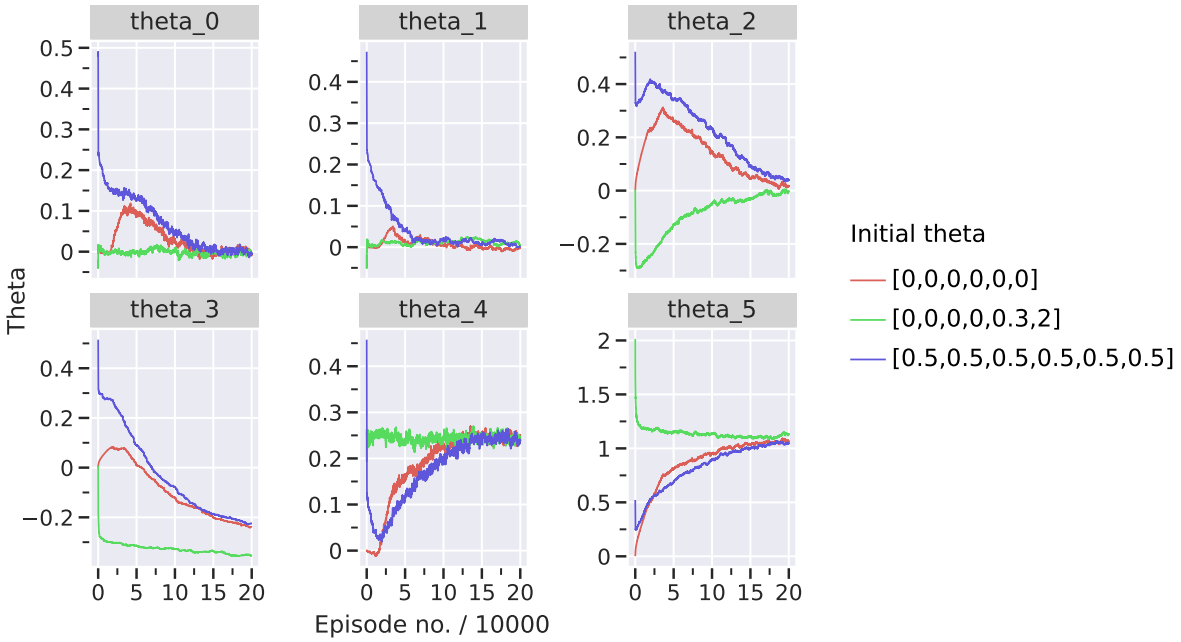


Figure 6.3: Evolution of the theta parameters with different initial parameters.

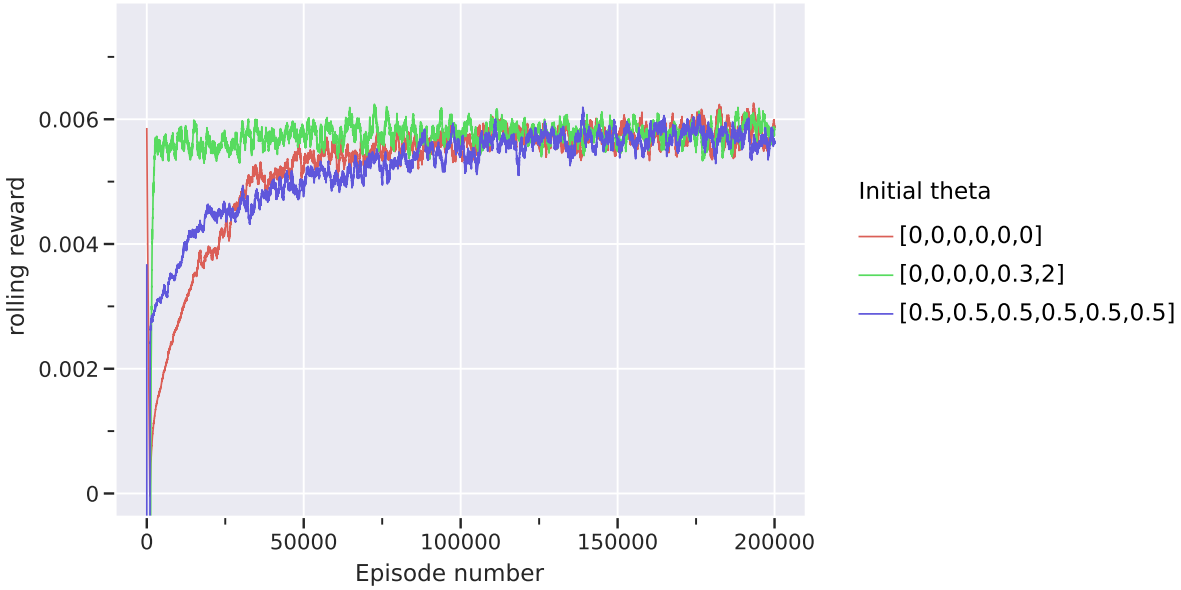


Figure 6.4: Evolution of the rolling average($k = 1000$) of the average episode reward for different initial parameters.

6.6 Impact of initial conditions

Gradient descent algorithms use the local information about an objective function $J(\theta)$ to compute the update $\theta \to \theta - \alpha \nabla J(\theta)$. This local behavior also means that any convergence of gradient descent is to a local minimum, and 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. The third experiment is then to run the algorithm with the same parameters as before, but with varied initial conditions for the θ vector, and to visualize the results, both in the average rewards and the evolution of θ parameters over 200000 episodes.

The evolution of the θ parameters is in Figure 6.3, and the rolling average of the average episode reward is plotted in Figure 6.4 for different initial value for θ. It turns out that while convergence in reward is to the same values, the θ_3 parameter does not seem to converge to the same value. Furthermore, even with such a large amount of episodes, it is not clear if the other parameters converged.

6.7 Further results

The average reward of the episode is a nice way to report on the performance of the method. However, it is difficult to interpret how exactly the model is performing once we have some found 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 6.1, 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 are(with some rounding off)

$$\theta^* = (-3.606 \times 10^{-3}, 4.476 \times 10^{-3}, -3.598 \times 10^{-4}, -0.3542, 0.2435, 1.1305)^\top.$$

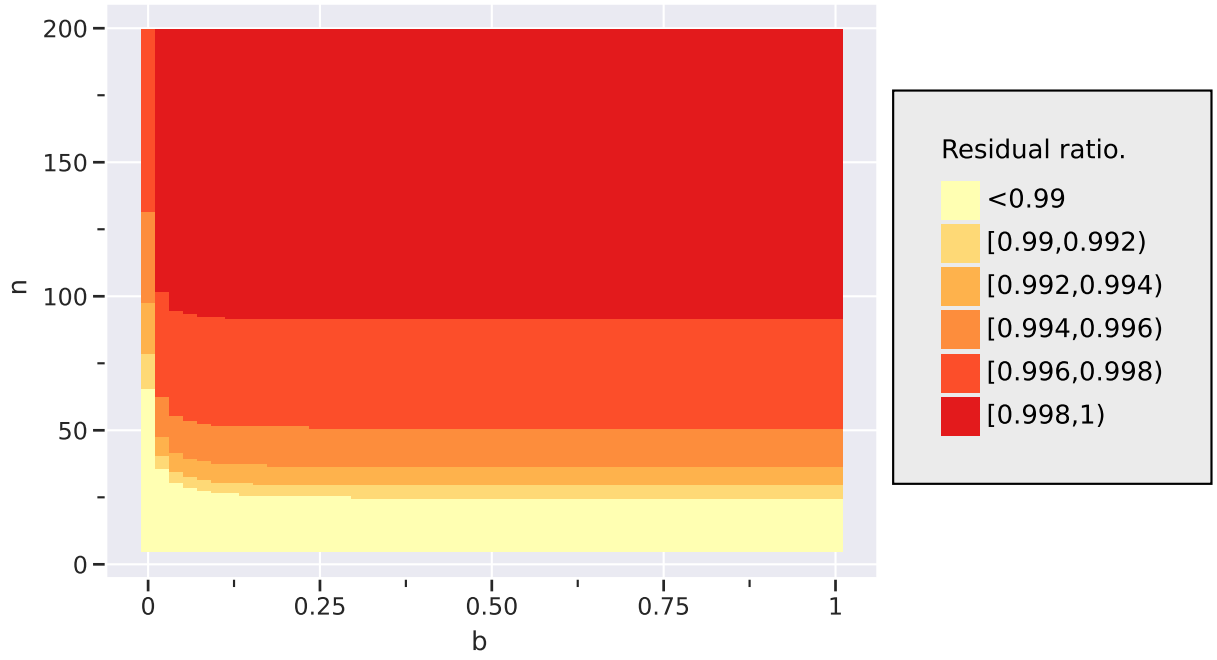
Then, we compute the value of ρ_{10}, for this policy and for different values of n and b, the results are as below in Figure 6.5. While we have convergence at any point, the convergence is slow, and the maximum value for ρ_{10} is 0.99917. Referring back to the grid search experiment(see Figure 3.2), this slow convergence is also partly an issue with the solver itself.

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 6.5. 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.

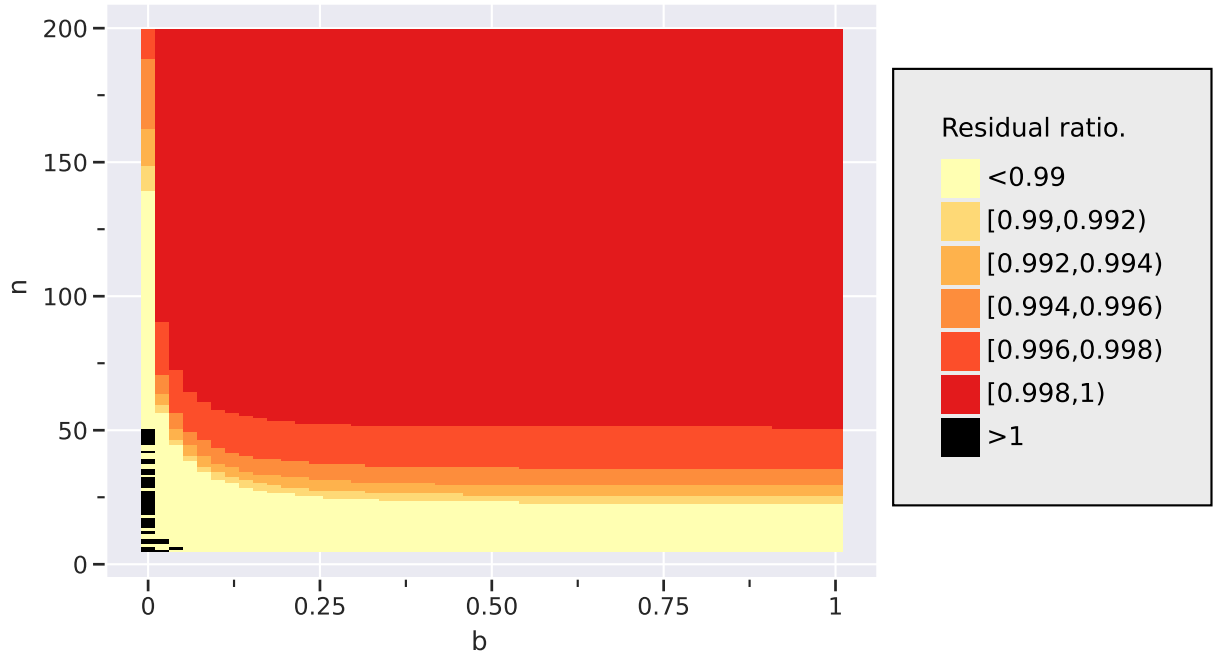
This divergence indicates that it may be a good idea, as a further study to further train the learned policy by computing ρ_{100}, and having a reward of $1 - \rho_{100}$ instead of $1 - \rho_{10}$. This of course mean that the training time will be longer.

6.8 (Possible section with comparison of results).

This possible section is to see if we can compare results.



(a) ρ_{10}. Maximum residual ratio: 0.99922.



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

Figure 6.5: Evolution of the residual ratio ρ_{10} and ρ_{100}, with the learned policy, depending on the problem parameters n and b.

Chapter 7

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, as otherwise we risk "overfitting" the parameters, which could lead to divergence of the solver after more 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.

At last, we need not restrict ourselves to just one type of solver. We could potentially train an intelligent agent to chose which numerical solver to use, depending on the problem.

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 the problem is approached. 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 [16], where we learns the hyperparameters of an optimization algorithm such as gradient descent.

References

[1] C. Mahoney, "Reinforcement learning: A review of the historic, modern, and future applications of this special form of machine learning." https://towardsdatascience.com/reinforcement-learning-fda8ff535bb6, 2021.

[2] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[3] 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.

[4] W. A. Adkins, M. G. Davidson, and S. (Online service), *Ordinary differential equations*. in Undergraduate texts in mathematics,. New York, NY : Springer New York :, 2012. Available: http://dx.doi.org/10.1007/978-1-4614-3618-8

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

[6] A. Atangana, *Fractional operators with constant and variable order with application to geo-hydrology*. Academic Press, 2018. Available: https://ludwig.lub.lu.se/login?url=https://www.sciencedirect.com/science/book/9780128096703

[7] 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.

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

[9] Wolfgang. Hackbusch and S. (Online service), *Iterative solution of large sparse systems of equations /*, 2nd ed. 2016. in Applied mathematical sciences,. Cham : Springer International Publishing :, 2016. Available: http://dx.doi.org/10.1007/978-3-319-28483-5

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

[11] 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

[12] R. Bellman, R. E. Bellman, and R. Corporation, *Dynamic programming*. in Rand corporation research study. Princeton University Press, 1957. Available: https://books.google.se/books?id=rZW4ugAACAAJ

[13] 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.

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

[15] 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

[16] 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