ThesisBook

Mélanie

3/15/23

Table of contents

Preface

This is a Quarto book. tests

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

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 $\lambda_1, \lambda_2, \ldots, \lambda_n$ 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 $i = 1, \ldots, n$, $a_i < 0$, 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 $a_i = \mathfrak{R}[\lambda_i]$. 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 (Bellman 1953, chap. 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.

3 A test problem - Convection diffusion equation

As a test case for the solver, we consider the steady state convection-diffusion equation.

$$u_x = bu_{xx} + 1$$

where b is some physical parameter and $u(x)$ is defined on the interval $[0, 1]$. The boundary condition are given by $u(0) = u(1) = 0$. This equation has an analytical solution that is given by

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

We are however interested in solving this numerically, with a finite difference approach. We partition the interval $[0, 1]$ into equidistant points $x_i, i = 0, ...n$. We note the distance between each points as Δx, and we have $u(x_0) = u(0) = 0$ and $u(x_n) = u(1) = 0$. 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 $i = 1, ..., n-1$, 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 $u = (u^1, ..., u^{n-1})^T$

$$Au = Bu + e$$

where $e = (1, 1, ..., 1)^T$,

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

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

$$Mu = e \tag{3.1}$$

where M is a square matrix of dimension $(n-1) \times (n-1)$ and e is the one vector of dimension $n-1$.

Remark. (To make better but I think it works) It is apparent that M is diagonally dominant. Since all elements of the diagonal are positive, then so are the eigenvalues real part. Assuming M is non singular, we have that $-M$ is stable.

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 parametrisation. Setting $\gamma = \frac{1}{\Delta x} + \frac{2b}{\Delta x^2}$, solving Equation 3.1 is equivalent to solving the system

$$Nu = b$$

where with $N = M/\gamma$, $b = e/\gamma$. The eigenvalues of M are also scaled by $1/\gamma$ so $-N$ 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.

$$u'(t) = b - Nu(t)$$

Assuming N is non singular, we can use Theorem 2.2 to guarantee that the ODE will converge to a steady state independently of the initial value we chose. In the next chapter, we will apply the Runge-Kutta scheme we saw earlier to the problem and will see how parameters changes results.

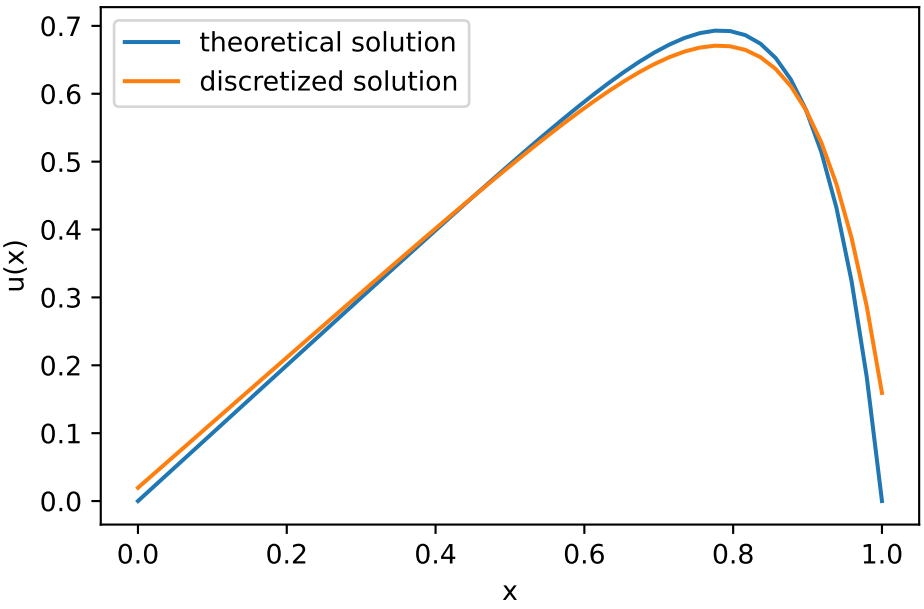

Figure 3.1: Theoretical and discretized solution of the problem with $b = 0.09$, $n = 50$.

4 Runge-Kutta solver applied to the test problem

We now have to solve the ODE $u' = b - Nu$ where M depends on the problem parameters b and $\Delta x = 1/(n+1)$, where 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 the following RK scheme with two stages.

blablbal

This solver has two parameters Δt and α. The objective 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

$$r_k = ||Nu_k - b||/||b||.$$

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

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

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

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 $c_{10} = f(\Delta t, \alpha)$ and $c_{100} = g(\Delta t, \alpha)$. We wish to answer the following questions

- Where are the optimal parameters for this specific problem, that is, the ones that minimize c_{10} and c_{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.

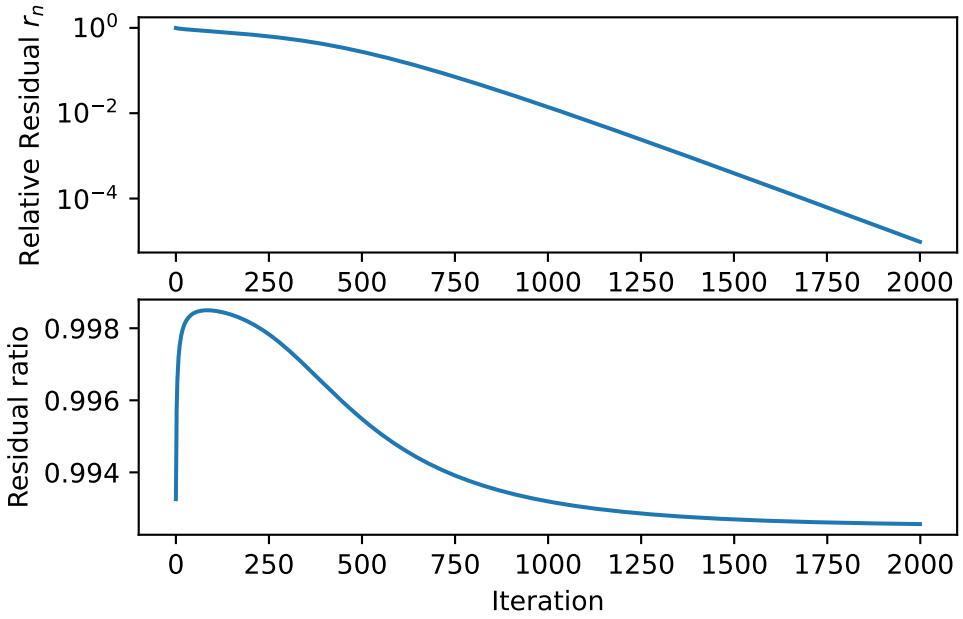

Figure 4.1: Evolution of the residual norm over iteration, with problem parameters $n = 100$ and $b = 0.05$, and RK parameters $\Delta t = 1.6$ and $\alpha = 0.3$.

```
from IPython.display import Image
Image('images/c10.png')
```

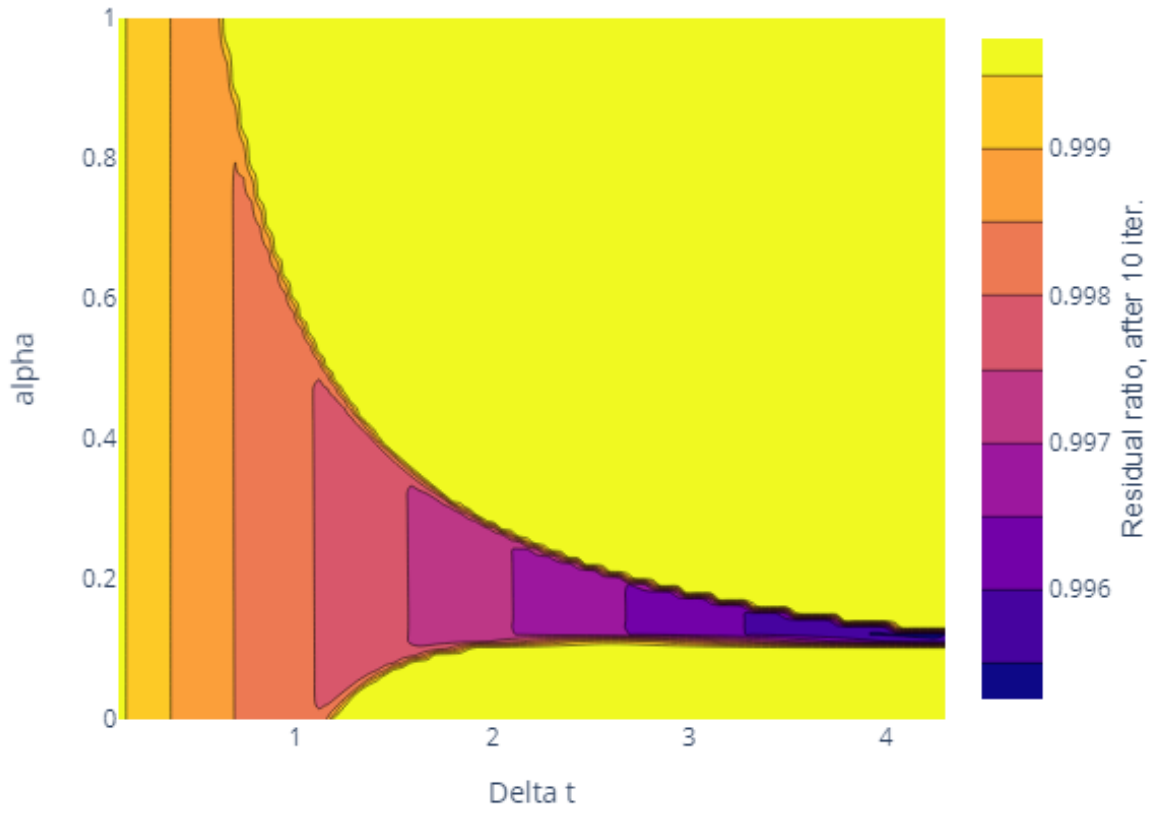

Figure 4.2: Residual ratio at 10 iterations, for problem parameters $b = 0.05$ and $n = 100$, and with varying solver parameters α and Δt

```
Image('images/c100.png')
```

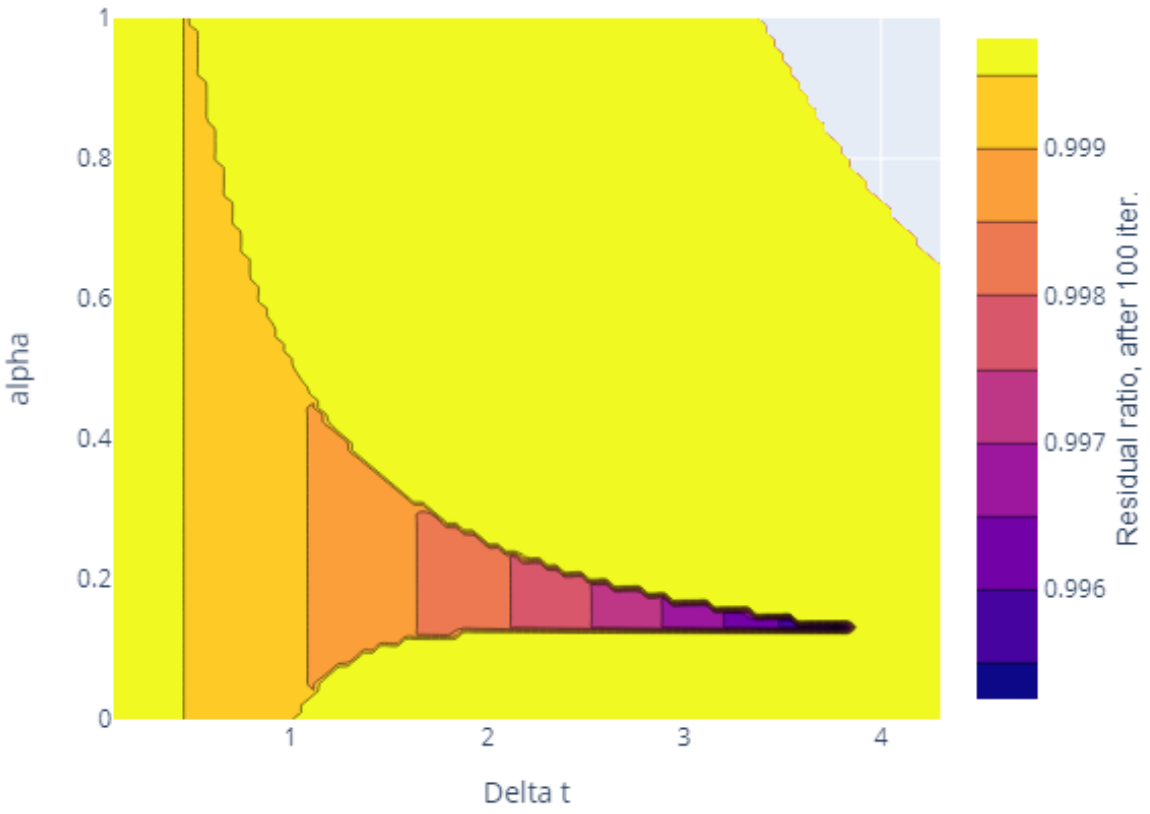

Figure 4.3: Residual ratio at 100 iterations, for problem parameters $b = 0.05$ and $n = 100$, and with varying solver parameters α and Δt

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 a particular problem, and it makes no sense in practice to compute the optimal parameters with a grid search. We thus explore a possible solution by using a reinforcement learning algorithm to "learn" these optimal parameters.

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.

5.1 A non mathematical, but delicious example.

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 stir or not. It may stick and burn at the bottom of the pan if we don't, but we are lazy and our laziness has to be weighted in.
- Or simply do nothing!

All of these decisions, which we will call *actions* from now on, are taken in reaction to 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*.

Each time we cook, we learn what works and what doesn't, and remember that for the 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

Before introducing reinforcement learning, we first need to define a Markov decision process (MDP).

Definition 5.1. (Markov decision process). A finite Markov decision process 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 $p(s'|s, a) = P(S_{t+1} = s'|s_t = s, a_t = a)$.
- the probability of getting reward r by taking the action a at a state s $p(r|s, a) = P(R_{t+1} = r|S_t = s, A_t = a)$.

Furthermore, a policy function is also given that governs for a given 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 can be in practice impossible to define in advance. This is remedied by using so called *model free* reinforcement learning algorithms.

Remark. Markov property applies, in particular lack of memory. (TODO)

Example 5.1. (A more mathematical example, adorable)

(More or less a gridworld example to write about)

At this point, we may be able to roughly define how to translate the problem (TODO, should formulate it better so I can reference it).

- The state space is defined by the problem parameters (b, n). A simplification that could be possible would be to discretize the state space.
- Actions would be choosing the solver parameters $(\Delta t, \alpha)$. This can also be discretized.
- Reward can be defined to be proportional to convergence rate after a certain number of iterations, where better convergence rate leads to better rewards.
- The model can be partially described, in the sense that while we can't model rewards, we can define the state transition probabilities by simply choosing a new state at random.

5.2.1 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 $(S_{t+1}, A_{t+1}), (S_{t+2}, A_{t+2}), ...$

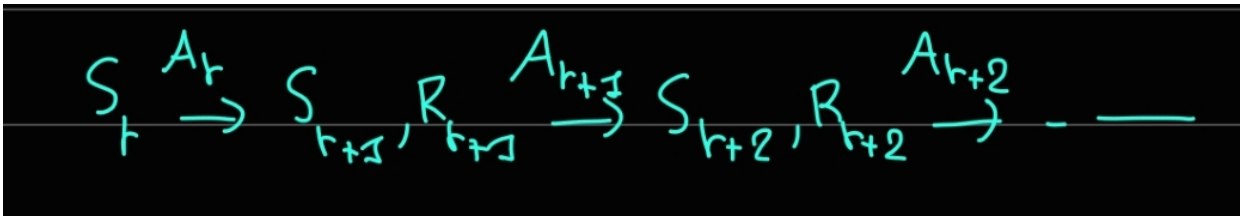

Figure 5.1: A helpful diagram showing trajectory, to be remade with tikz.

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 $(S_{t+1}, A_{t+1}), (S_{t+2}, A_{t+2}), ...$ is the random variable given by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

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

The discounted return is thus the sum of rewards along a trajectory. 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

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

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 for (finite) MDP.

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

Some remarks

- 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 valid for a given policy.

The expression between the brackets is called the action value $q_\pi(s, a)$. Bellman's equation is then simply

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a, s) q_\pi(s, a) = E_\pi[q]$$

(///TODO)

5.3 Bellman optimality equation

Given some state value, one may ask the fundamental question.

(test Zhao (2023) , Sutton and Barto (2018) , Williams (1992), Lillicrap et al. (2019)).

6 Policy gradient methods

We introduce policy based method in this chapter.

6.1 The test problem, in a RL framework

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 defined the problem. We answer the last two challenges in the next sections.

6.2 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 c_{10}. With that ratio, we need to define an appropriate reward metrics. We design the reward such that:

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

The reward is

$$r(c_{10}) = \begin{cases} 100 \times (1 - c_{10}) & \text{if } c_{10} < 1 \\ \max(-10, 1 - c_{10}) & \text{if } c_{10} \geq 1 \end{cases}$$

When $c_{10} < 1$, 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. Because the ratio tends to be very close to 1, we multiply everything by 100, adding more contrast to the rewards.

When, on the other hand $c_{10} \geq 1$, 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 -10.

6.3 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.

6.4 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 $v(s) \approx \hat{v}(s, \omega)$ 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 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(Hornik, Stinchcombe, and White (1989)). In our case, a linear approximation is used.

6.5 Policy gradient methods.

6.5.1 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

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

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

6.5.2 Policy gradient theorem

We prove, using the aforementioned assumptions the policy gradient theorem. This proof is adapted from (Sutton and Barto 2018,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

$$\nabla J(\theta) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left[\gamma J(\theta) + \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(a,s) \right] \tag{6.1}$$

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

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.

6.5.3 REINFORCE algorithm

Here is the pseudocode.

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 define the following stochastic policy, define first

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

Then we chose the random policy $\alpha \sim \mathcal{N}(\mu_\alpha, \sigma^2)$,and similarly $\Delta t \sim \mathcal{N}(\mu_{\Delta t}, \sigma^2)$. The term σ^2, which is the variance of the policy is chosen fixed, and will help us balance exploration vs exploitation. 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

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

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,

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

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 $s = (b, n)$,

$$\nabla_\theta \ln \pi(a|s, \theta) = \xi_\alpha(b\theta_0, n\theta_1, 0, 0, \theta_4, 0)^T + \xi_{\Delta t}(0, 0, b\theta_2, n\theta_3, 0, \theta_5)^T. \tag{7.1}$$

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(Lillicrap et al. (2019)). The fact that the pdf admits values > 1 is not an issue as we adjust the learning rate accordingly.

7.2 Application of the REINFORCE algorithm.

The Reinforce algorithm is used in Here are the results, (Experiments will be rerun, those are placeholder graph)

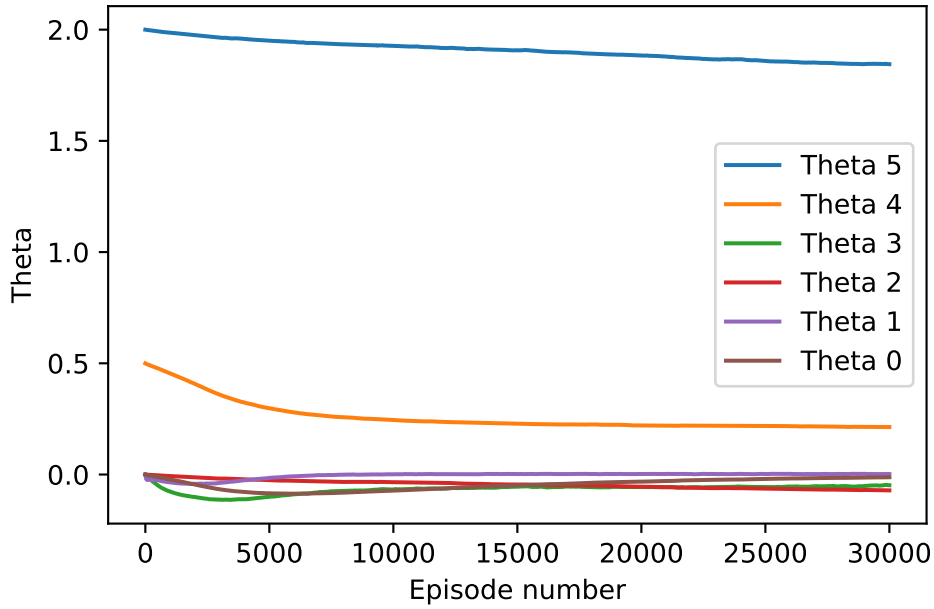

Figure 7.1: Evolution of the theta parameters with episode number. The hyperparameters are sigma=0.1, learning rate 5e-6….

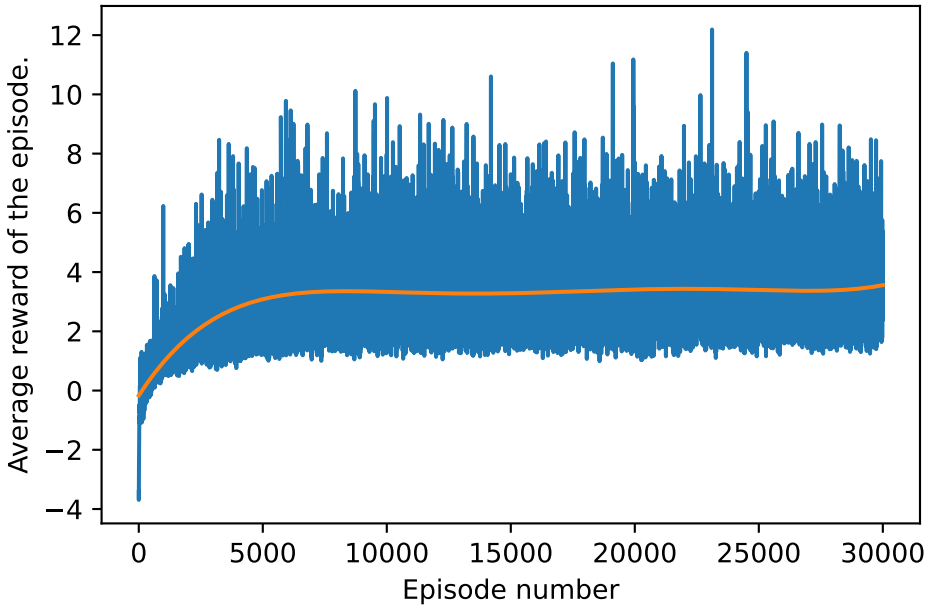

Figure 7.2: Evolution of the average reward inside the episode with episode number. The hyperparameters are sigma=0.1, learning rate 5e-6…. The variance is high! The trendline is a polynomial fit not to be trusted too much.

there are problem with the results, namely

- The learning rate must be kept low, otherwise things explode nastily.
- The convergence is extremely slow, to the point were we may be wondering if it converges at all

(The next sections are things I've done to mitigate these problem, with varying degrees of successes, the log are there, but also I need to clean up my code and rerun it for reproducibility sake as I was tinkering a lot at the time, the goal is to have a program were each hyperparameter can be chosen at the beginning/as environment variables, and making all the experiments easily reproducible).

7.3 Exploration vs exploitation tradeoff.

Changing σ^2 to a higher value makes for a flatter gradient, so not only are we taking more risks, we can actually afford a higher learning rate without exploding!

7.4 Gradient ascent with different learning parameters.

Here is how we can get better convergence by changing the learning parameters.

One problem is that if we look at the gradient directions, how steep they are depend on the problem parameters. And since n can vary between 5 and 300, this make for a muuuuuch steeper gradient than the direction associated with b, which only varies between 0 and 1. As a result, the direction associated with n tend to converge but not the other. This can be remedied by applying varying learning rate in each direction. It actually works quite well! '

7.5 Impact of initial condition.

Gradient based algorithm have a tendency to converge to local minima. (in our case maxima, but same thing really), therefore, it would be interesting to see how initial policy impacts the learned policy now that we have convergence. (Not done yet, but I suspect some nasty surprises!)

7.6 Moving beyond the basics.

(Some discussion on what to do next, etc…) Example include.

- Better algorithm, with less sample variance and more sample efficiency.
- Moving beyond a linear policy. (approximation using NN).
- Meta learning, maybe?
- Applying the concept learned here to a more varied set of problem, instead of just confining ourselves to steady state diffusion-convection equation.

8 Summary

In summary, this book has no content whatsoever.

References

Bellman, Richard Ernest. 1953. *Stability Theory of Differential Equations /.* New York : McGraw-Hill,.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. "Multilayer Feedforward Networks Are Universal Approximators." *Neural Networks* 2 (5): 359–66. https://doi.org/https://doi.org/10.1016/0893-6080(89)90020-8.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2019. "Continuous Control with Deep Reinforcement Learning." https://arxiv.org/abs/1509.02971.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8 (3): 229–56. https://doi.org/10.1007/BF00992696.

Zhao, Shiyu. 2023. "Mathematical Foundations of Reinforcement Learning." 2023. https://github.com/MathFoundationRL/Book-Mathmatical-Foundation-of-Reinforcement-Learning.