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

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

Proof. (In the diagonalisable case)

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

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

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

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

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

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

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

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

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

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

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

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

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

$\square$

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 $x_i, i = 0, \dots n$. 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 $i = 1, \dots, n-1$, 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 $u = (u^1, \dots, u^{n-1})^T$

$$Au = Bu + e$$

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

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

and

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

With $M = A - B$, 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.

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

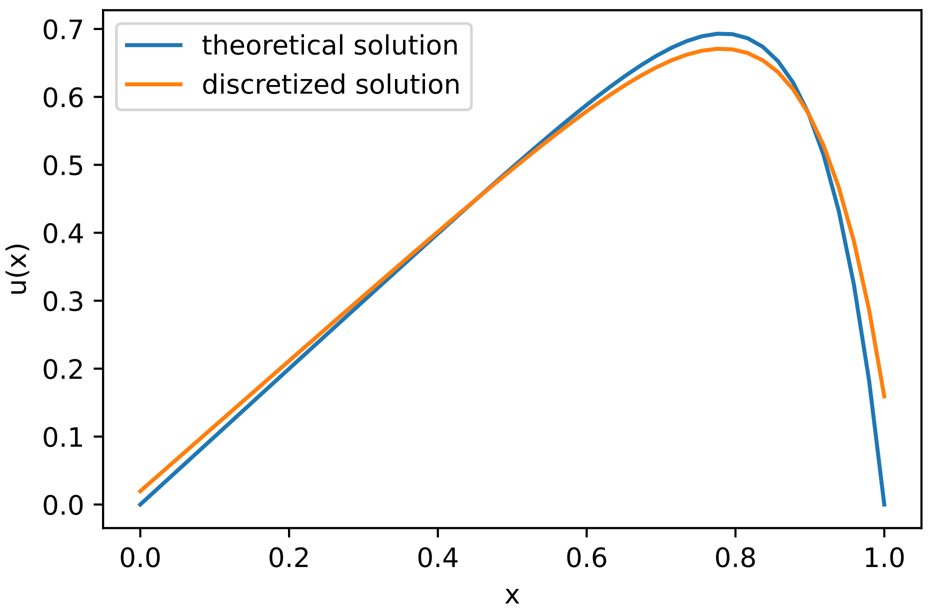

Figure 3.1: Theoretical and discretized solution of the problem with $b = 0.09$, $n = 50$.

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.

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.

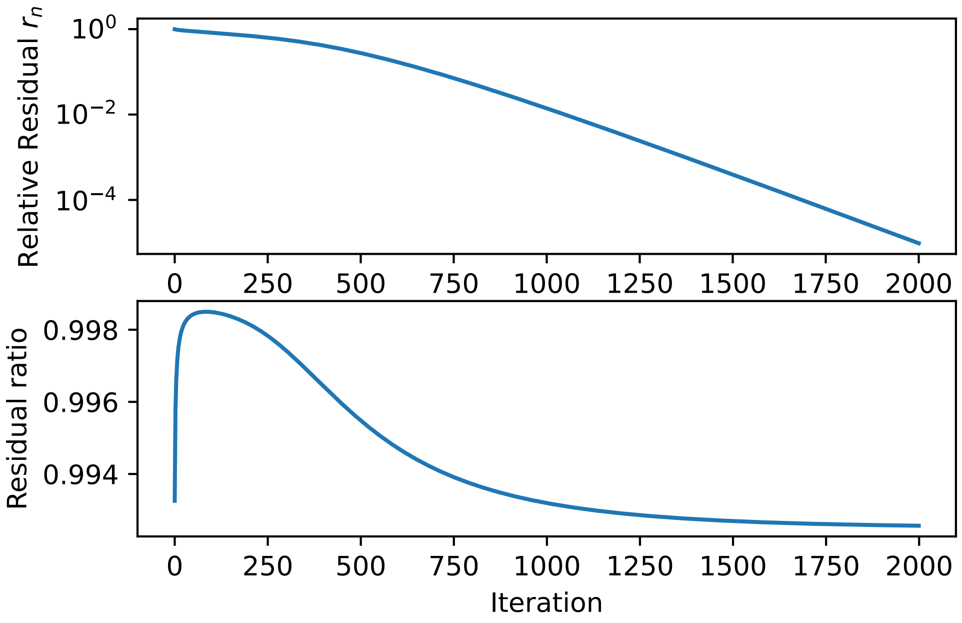

Figure 4.1: Evolution of the residual norm over iteration, with problem parameters $n = 100$ and $b = 0.05$, and RK parameters $\Delta t = 1.6$ and $\alpha = 0.3$.

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.

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

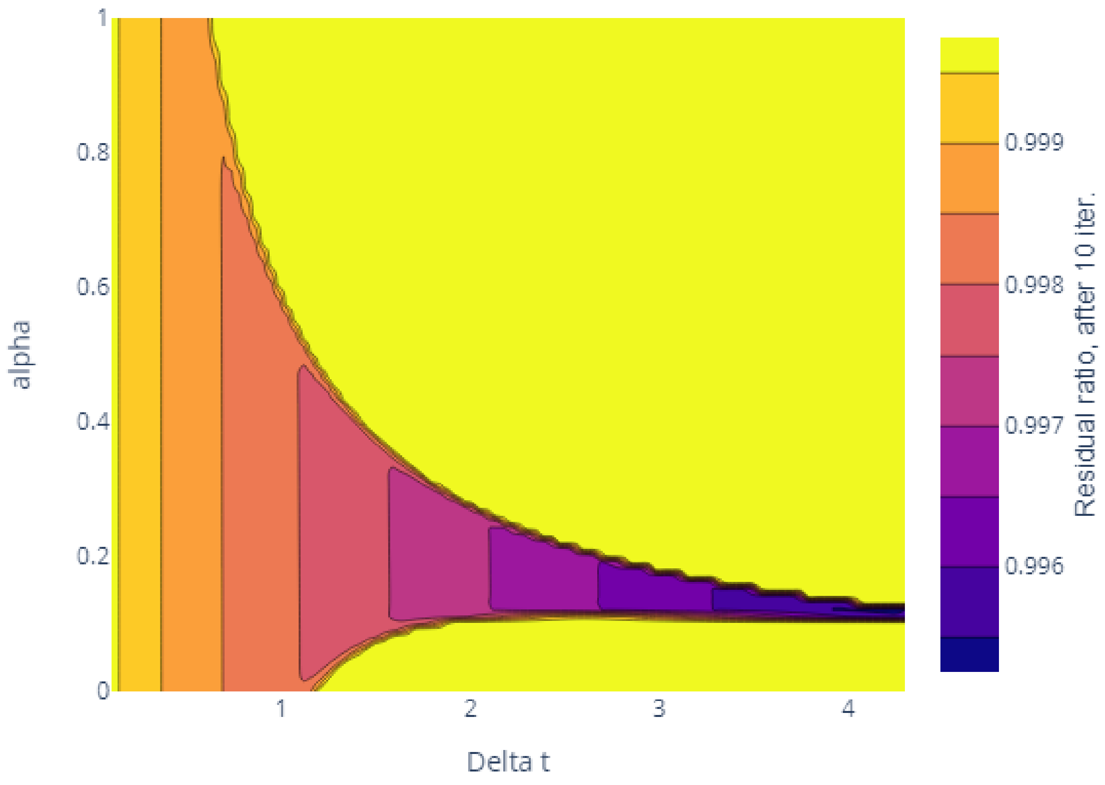

Figure 4.2: Residual ratio at 10 iterations, for problem parameters $b = 0.05$ and $n = 100$, and with varying solver parameters α and Δt

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

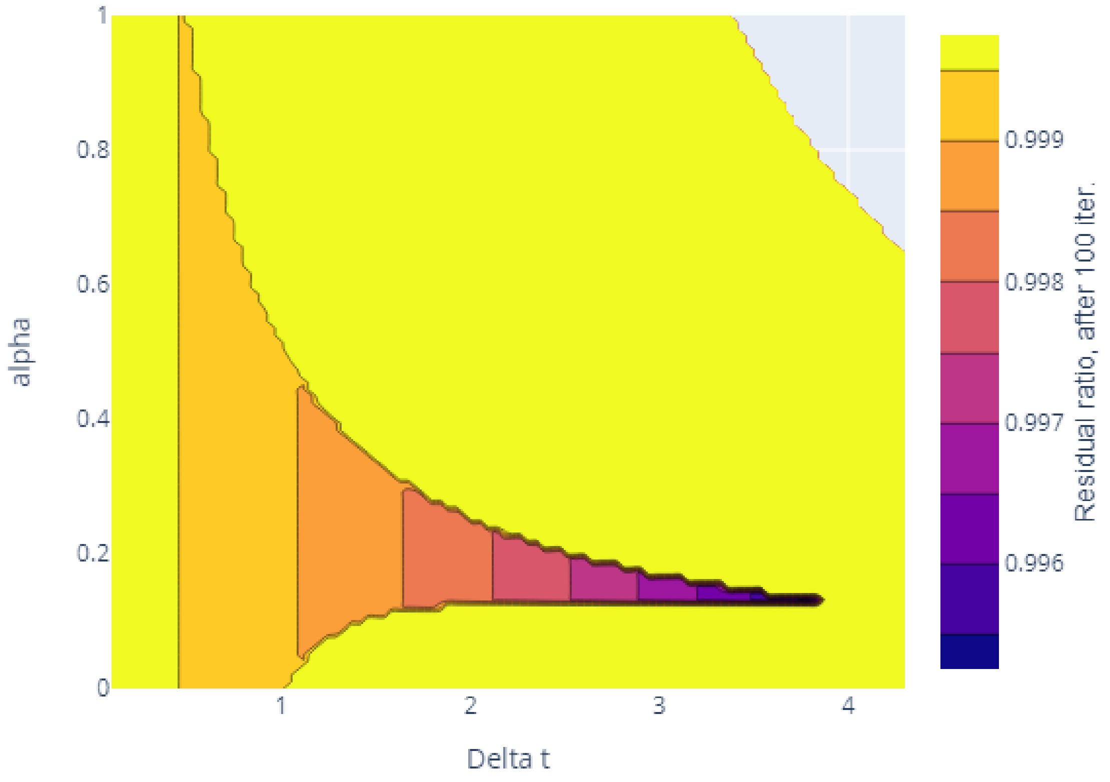

Figure 4.3: Residual ratio at 100 iterations, for problem parameters $b = 0.05$ and $n = 100$, and with varying solver parameters α and Δt

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 Reinforcement learning - Basic

In this section, we outline the main ideas behind reinforcement learning and how they can be applied in the context of this thesis.

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.

5.1 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.

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

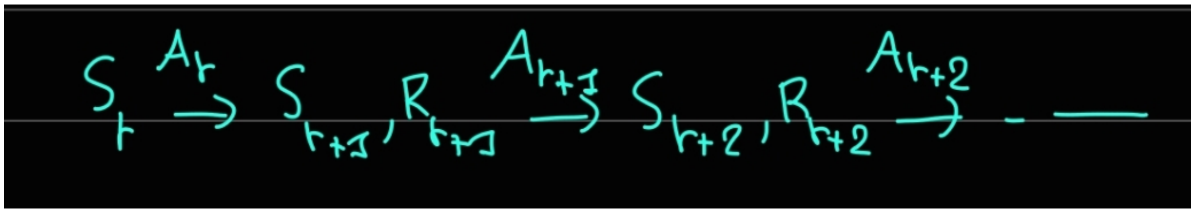
$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \xrightarrow{A_{t+1}} S_{t+2}, R_{t+2} \xrightarrow{A_{t+2}}$ - ——

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.

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

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

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

5.2 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 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.

7 Policy gradient methods

Here's the rundown

Several problem

- State transition being random defeats the point of RL.
- State-Action is continuous
-

7.1 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, these 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.

7.2 From discrete to continous.

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* 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.

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 $\omega \in \mathbb{R}^n$ are n 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 $\theta \in \mathbb{R}^n$ is a parameter vector is dimension n. 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.
- I know I had other reasons, but it's easy to implement may not be a good one.
- Using approximation allow use to extrapolate better.

Approximation is usually done using neural networks, building on the universal approximation theorem(Hornik, Stinchcombe, and White (1989)).