LECTURE NOTES | 02465

# Sequential Decision-Making

**Tue Herlau**
tuhe@dtu.dk

June 18, 2025
Version 1.3.3

# Foreword for 02465

The purpose of 02465, *Introduction to reinforcement learning and control*, is to present a unified treatment of control and reinforcement learning. Both subjects address the same underlying problem, namely how to make decisions in sequence to bring about a desired outcome, but with a considerable differences in emphasis, methodology and notation.

For this reason it has not been possible to find a unified treatment of both subjects suitable for a bachelor-level course. It was early on apparent that the excellent resource [SB18] should be used for the second half of the course in reinforcement learning, however, I had considerable difficulties finding suitable reading material for the first part of the course on control.

To address this, I have written a self-contained introduction to sequential decision making, with an emphasis on the DP algorithm and it's application to optimal decision making under certainty and computer-game AI. The second part will consider its applications to control theory, a section which borrows from the lecture notes from *AA 203: Optimal and Learning-Based Control* offered at Stanford, generously made available online by the author James Harrison: `https://github.com/StanfordASL/AA203-Notes`

AA203 has a broader scope than this course, and therefore I have reduced the scope of my version considerable, and added additional chapters to provide a gentler introduction to dynamical systems. A student who is interested in a more in depth and challenging treatment of the topic is encouraged to read the original material.

Finally, the notes have been extended with three chapters to cover the functionality of python we will use in this course. The inclusion of this material was based on the observation that programming skills in the KID study line are not practiced enough during the bachelor study line, which means some are unfamiliar with important parts of the python language. Although there are many excellent resources to learn python online, I hope that a self-contained treatment will provide some much needed overview.

Throughout the notes we use the following symbols:

☞ An algorithm you will implement in the exercises

✸ A section which is slightly technical, and where only the general gist of the result is exam relevant. Exercises with one star should be completed after the exercises with no stars.

✸✸ A section with extra material included for completeness but not relevant for the exam. Exercises with two stars are only for those who are extra interested.

Compared to earlier versions, I have trimmed down the code examples in the notes significantly, but created an online documentation site for the course which is much better suited for this purpose. You can find it here: course page.

**Changes**

**Version 1.3.3 (May 2025)** Fixed an off-by-one issue in the definition of $t_k$ in the beginning of section 15.3.2 + other minor edits in the section. I will take the changes and the potential confusion this may have caused into consideration for the exam.

# Contents

# Chapter X

# Preliminaries

This chapter will collect useful mathematical results we will make use of throughout the remainder of the course.

## X.1 Monte-Carlo sampling

Monte Carlo (method) (MC) is an umbrella name for an arsenal of numerical techniques for computing approximate estimates via random sampling. The estimates are very often given as the result of an intractable integral and the technique could have been named "numerical integration in high dimensions via random sampling". The term Monte Carlo was initially coined during 1940's by von Neumann, Ulam and Metropolis [MU49, MRR$^+$53] to refer to the famous casino of 1940's.[BGJM11] However, due to the central importance of the subject and wide use of the concept, it soon became an established technical term. Monte Carlo techniques have been further popularized in applied sciences with the wider availability of computing power, starting from the 90's, most notably in statistics, computer science, operational research and signal processing.

### X.1.1 The Monte Carlo principle

Monte Carlo

In an abstract setting, the main principle of a **Monte Carlo** technique is to generate a set of samples $x^{(1)}, \dots, x^{(N)}$ from a target distribution $\pi(x)$ to estimate some features of this target density $\pi$. Features are simply expectations of *well behaving* functions that can be approximated as averages:

$$\mathbb{E}_\pi[\varphi] \approx \frac{\varphi\left(x^{(1)}\right) + \cdots + \varphi\left(x^{(N)}\right)}{N} \equiv \bar{E}_{\varphi,N}. \tag{X.1}$$

Provided that $N$ is large enough, we hope that our estimate $\bar{E}_{\varphi,N}$ converges to the true value of the expectation $\mathbb{E}_\pi(\varphi(x))$. More concrete examples of test functions $\varphi(x)$ will be provided in the next section. For independent and identically distributed samples, this is guaranteed by two key mathematical results: the strong Law of Large

Numbers (LLN) and the Central Limit Theorem (CLT)[Ros05]. Assuming that the true mean and variance

$$\mathbb{E}_\pi[\varphi] = \mu, \quad \mathrm{Var}_\pi[\varphi] = \sigma^2 \tag{X.2}$$

LAW OF LARGE NUMBERS

are both finite **Law of Large Numbers** states that

$$\bar{E}_{\varphi,N} \to \mu \quad \text{a.s.}$$

Here, a.s. denotes convergence almost surely, which is the technical statement the limit converge with probability one [Ros05]

$$\Pr\left\{\lim_{N\to\infty} |\mu - E_{\varphi,N}| = 0\right\} = 1 \tag{X.3}$$

CENTRAL LIMIT THEOREM

Since our approximation will be based on a random, finite sample set $x^{(1)}, \ldots, x^{(N)}$, fluctuations are inevitable, but we wish that they are small. This is guaranteed by the **central limit theorem**: for sufficiently large $N$, the fluctuations are approximately Gaussian distributed

$$\bar{E}_{\varphi,N} \sim \mathcal{N}\left(\bar{E}_{\varphi,N}|\mu, \frac{\sigma^2}{N}\right) \tag{X.4}$$

with mean equal to the true mean and standard deviation decreasing at a rate proportional to $\frac{1}{\sqrt{N}}$. This result has important practical consequences. If we can generate i.i.d. samples from the distribution $\pi$ of $x$, we can estimate any expectations $\mathbb{E}[\varphi(x)]$ using Monte Carlo sampling and be guaranteed the following properties

1. Monte Carlo provides a noisy but unbiased estimate of the true value $\mu = \mathbb{E}_\pi(\varphi(x))$
2. The error decreases at a rate proportional to $\frac{1}{\sqrt{N}}$
3. The decrease in error is independent of the dimensionality of $x$

The third point is particularly important. It suggests that, at least in principle, with a quite small number of samples one can compute approximate solutions for arbitrary large parameter estimation problems.

## X.2 Analysis

Control theory will make use of concepts from analysis which should be familiar, such as derivatives and ordinary differential equations (ODEs).

### X.2.1 Vector-valued functions

Highschool math is focused on functions such as $f(x) = \sin(2x)$, $g(x) = e^{-ax}$, and so on. These functions map a single real number to a single real number which is commonly written as $f : \mathbb{R} \to \mathbb{R}$. We can generalize this definition in two ways: A function can either take multiple arguments, or map to multiple outputs (or both). Suppose we adopt the convention of writing vectors as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{X.5}$$

MULTIVARIATE FUNCTION

We could define a **multivariate function** as

$$f(\boldsymbol{x}) = \sin(x_1)e^{x_2} + x_3. \tag{X.6}$$

Which we would write as $f : \mathbb{R}^3 \to \mathbb{R}$. This function takes three input arguments and return a single number. We could also let a function have multiple output arguments. We write this as:

$$\boldsymbol{f}(x) = \begin{bmatrix} \sin(x) \\ \cos(x) \end{bmatrix} \tag{X.7}$$

Which we write as $\boldsymbol{f} : \mathbb{R} \to \mathbb{R}^2$. Obviously the two things can be combined to give, for instance,

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} x_2 \sin x_1 \\ x_2 \cos(x_1 + x_3) \end{bmatrix} \tag{X.8}$$

which we write as $\boldsymbol{f} : \mathbb{R}^3 \to \mathbb{R}^2$. Note the use of boldface to indicate which quantities are vectors.

### X.2.2 Derivatives

The derivative is defined in the ordinary way as $\frac{df}{dx}(x) = \lim_{\Delta \to 0} \frac{f(x+\Delta) - f(x)}{\Delta}$. For instance if $f(x) = \sin(ax)$ then

$$\frac{df(x)}{dx} = f'(x) = a\cos(ax). \tag{X.9}$$

For vector-valued functions there are two generalizations. The first is the partial derivative, which just means we perform the ordinary derivative with respect to a single input variable $x_k$ while keeping the others constant:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} = \lim_{\Delta \to 0} \frac{f(x_1, \ldots, x_{k-1},\ x_k + \Delta,\ x_{k+1}, \ldots, x_n) - f(\boldsymbol{x})}{\Delta}. \tag{X.10}$$

### Gradient and Hessian

GRADIENT

If $f : \mathbb{R}^n \to \mathbb{R}$ the **gradient** is defined as

$$\nabla f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\boldsymbol{x}) \end{bmatrix}$$

HESSIAN

Note the gradient is a function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$. We can also define the equivalent to the second-order derivative namely the **Hessian** defined as:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{X.11}$$

The hessian is a function $H : \mathbb{R}^n \to \mathbb{R}^{n^2}$.

## X.2.3 Jacobian

JACOBIAN MATRIX

Consider now the case where $f$ has multiple outputs, i.e. $f : \mathbb{R}^n \to \mathbb{R}^m$. The **Jacobian matrix** is defined as:

$$\mathbf{J}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

The Jacobian is technically a function $\boldsymbol{J} : \mathbb{R}^n \to \mathbb{R}^{mn}$.

## X.2.4 Approximations

The reason gradients, Jacobians and Hessians are used in machine-learning is because they can be used to approximate functions. First, suppose $f : \mathbb{R}^n \to \mathbb{R}$ has a single output argument. We can then approximate $f$ around $\boldsymbol{x}$ as:

$$f(\mathbf{x} + \boldsymbol{\Delta}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^{\mathrm{T}} \boldsymbol{\Delta} + \frac{1}{2} \boldsymbol{\Delta}^{\mathrm{T}} \mathbf{H}(\mathbf{x}) \boldsymbol{\Delta} \tag{X.12}$$

A similar expression can be obtained for a multi-variate $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$:

$$\mathbf{f}(\mathbf{x} + \boldsymbol{\Delta}) \approx \mathbf{f}(\boldsymbol{x}) + \mathbf{J}_{\mathbf{f}}(\boldsymbol{x}) \boldsymbol{\Delta} \tag{X.13}$$

These expressions are fairly abstract, but we can make sense of them in two ways: First, lets consider the case where $n = 1$. The first equation becomes:

$$f(x + \Delta) = f(x) + \Delta f'(x) + \frac{1}{2} \Delta^2 f''(x) \tag{X.14}$$

Which is just the second-order Taylor expansion. The second equation, if $n = m = 1$, is simply $f(x + \Delta) = \Delta f'(x)$.

The reason why these equations are important is because they allow us to reason about how a function behaves at a point $\boldsymbol{x} + \boldsymbol{\Delta}$ given only information computed at $\boldsymbol{x}$. As an example, given $\boldsymbol{x}$, suppose we consider a new point selected as $\boldsymbol{x}' = \boldsymbol{x} - \alpha \nabla f(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{\Delta}$, $\boldsymbol{\Delta} = -\alpha \nabla f(\boldsymbol{x})$. Plugging this into eq. (X.12) gives

$$f(\boldsymbol{x}') = f(\boldsymbol{x} + \boldsymbol{\Delta}) \approx f(\boldsymbol{x}) - \alpha \|\nabla \boldsymbol{f}(x)\|^2 + \alpha^2 \frac{1}{2} (\nabla \boldsymbol{f}(x))^\top \boldsymbol{H}(\boldsymbol{x}) \nabla \boldsymbol{f}(x) \tag{X.15}$$

Assuming $\alpha$ is small the last term will be much smaller than the middle-term. This tells us that $f(\boldsymbol{x}') \approx f(\boldsymbol{x}) - \alpha \|\nabla \boldsymbol{f}(x)\|^2$ and so $f(\boldsymbol{x}') < f(\boldsymbol{x})$. What this tells us is that adding a term proportional to minus the gradient times $\alpha$ to $\boldsymbol{x}$ will likely decrease the objective function, i.e. gradient descent works.

### X.2.5 Dot-notation

Finally, time will play an important role throughout. Suppose $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^n$ is a function of time, for instance $u(t) = e^{-at}$ would be an exponentially decaying function. The derivative is then written using a dot, or the double-derivative using two dots:

$$\dot{u}(t) \equiv u'(t) = -ae^{-at}, \quad \ddot{u}(t) \equiv u''(t) = a^2 e^{-at} \tag{X.16}$$

Often the time-dependence is suppressed and this is written as $\dot{u}$ and $\ddot{u}$.

### X.2.6 Differential equations

The reader is no doubt fmliar with differential equations. For instance, solve for $f$ if

$$f'(t) = \frac{1}{f(x)} \tag{X.17}$$

(typically the problem should also contain additional information such that $f(t)$ must be non-negative and satisfy an initial condition, but we will omit these details here). Notice we can write this equivalently as:

$$\dot{f} = F(f), \quad F(z) = \frac{1}{z}. \tag{X.18}$$

Since we will be interested in abstract differential equations we will prefer this notation where $F$ is assumed to be some well-behaved function.

Solving a differential equation is often easier done by guessing a solution. In this particular case you can check that $f(t) = \ln(t + k)$ works for any constant $k$.

We can generalize the above way of writing differential equations to the multivariate case:

$$\dot{\boldsymbol{f}} = \boldsymbol{F}(\boldsymbol{f}) \tag{X.19}$$

In this case $\boldsymbol{f} : \mathbb{R} \to \mathbb{R}^n$ is a vector-valued function and $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n$ tells us how $\boldsymbol{f}$ relates to it's rate of change. The notation can alternatively be unpacked into the $n$ coupled differential equations:

$$\begin{aligned} f_1'(t) &= F_1(\boldsymbol{f}(t)) \\ f_2'(t) &= F_2(\boldsymbol{f}(t)) \\ &\vdots \\ f_n'(t) &= F_n(\boldsymbol{f}(t)). \end{aligned} \tag{X.20}$$

Solving such systems of differetial equations is in nearly all cases impossible and we will not be interested in it here. However, note given $\boldsymbol{F}$, we can easily check if a particular function $\boldsymbol{f}$ is a solution by simply differentiating it and checking all $n$ equations agree.

## X.3 Linear algebra

POSITIVE SEMI-DEFINITE

A symmetric matrix $Q$ is **positive semi-definite** if for all $\boldsymbol{x}$ it holds:

$$\boldsymbol{x}^\top Q \boldsymbol{x} \geq 0 \tag{X.21}$$

POSITIVE DEFINITE

It is **positive definite** if the inequality is sharp: $\boldsymbol{x}^\top Q \boldsymbol{x} > 0$. This property is important in optimization since it ensures that $\boldsymbol{x}^T Q \boldsymbol{x}$ resembles a quadratic expression (such as $x^2$) in that it is at most quadratic in $\boldsymbol{x}$ and can never be negative.

# Part I

# Programming

# Chapter 1

# Python basics

Python is a powerful language with a simple syntax, great build-in data-structures, and the best libraries for machine learning and AI. In the last 15 years, python has emerged as *the* standard programming language in machine learning.

## 1.1 Why this course emphasize python

Machine-learning is a new field and so what one ought be able to know or do is less well-established than in other academic disciplines. However, I think it is fair to compare the relationship between being a machine-learning practitioner and programming to

- The relationship between being a surgeon and using surgical tools to operate
- The relationship between being an architect and making technical drawings
- The relationship between an electrician and his tools

In other words, *programming is what allows us to transform theoretical knowledge into something of value*. If you don't have the ability to operate on a person, you cannot perform surgery. If you cannot read and make technical drawings, you cannot be an architect, and if you cannot program, you cannot be a data scientist.

So what are good programming skills? I am going to define it as roughly this:

- You can start with a blank editor and write code
- You know what a dictionary and list is, and how and when to use them
- You know that e.g. a pytorch tensor is a class, and why pytorch made this choice
- Programming does not scare you or feel like trial-and-error

In the first two semesters I ran this course, it was my experience about 40% of the students had deficiencies which made the course unpleasant, and perhaps 20% were struggling to even get started with the exercises. At the same time, I found that poor programming skills did not correlate with study activity or general technical abilities.

I think the reason for these deficiencies should be attributed to there being no python-specific introduction to programming course, as well as a general lack of focus in the study line on programming as a day-to-day activity which directly affects ones grades. In contrast, when I have spoken with good programmers in the study line, they have all attribute their skills to self-study.

Unfortunately, I think this can easily create the impression that if you struggle, you simply lack an innate ability of some sort. *This is wrong.* Programming is a learned skill, similar to blind typing, and our mistake was that you could not reasonably be expected to become good in a single 5 ECTS course.

We are doing what we can to improve on that situation. In the spring 2022 semester, the algorithms and data structure course was moved to an earlier place in the study line, and the introduction to programming course and mathematics course will eventually be re-modeled. However, you need a fix here and now. I have therefore decided to include three chapters to refresh programming concepts we will use, so there is at least a single, self-contained reference for those who prefer to learn in that way.

### 1.1.1 Scope

I think everyone knows what an `if` statement is, how to add numbers in python, and that `s = "Hello World"` defines a variable called `s` which store a bit of text. Despite this, this guide will start from the ground up, since if something relatively simple is missing, this will create a great confusion later on.

### 1.1.2 Starting python

When you learn a new programming language, the most important thing is you can experiment with the examples as you read them. I recommend that you install a good IDE and get yourself set up, you can find more details on DTU Learn, including videos on how to do that.

Once you are done, you can start python by running the command `python` in the console. On my computer it results in the following output:

```
1 Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)] on win32
2 Type "help", "copyright", "credits" or "license" for more information.
3 >>>
```

INTERACTIVE MODE

When python is used this way it is called the **interactive mode**, because commands are typed in directly after the three greater-than signs (`>>>`) and executed when you press enter. The interactive mode is convenient to showcase basic features, but not usable for longer programs. The console mode is also available from Pycharm if you press the `Python console` tab in the bottom of the screen.

The alternative way to run python is to type your commands into `.py`-files and then run them using a command such as `python my_script.py`. Putting your commands in `.py`-files is the recommended way to use python, however, you should not actually type the command `python my_script.py` yourself, but rather run the scripts from Pycharm

or VScode, since this is far more convenient. Please see the installation video for more details.

### 1.1.3 Python version

Most of what the `python` command output is not important, but check your version number (in this case `3.8`). Higher versions are okay, but lower versions will not work for this course[1].

### 1.1.4 What is a program?

When you program, you should have this view in mind: Initially, you have your python program code, usually stored in `.py` files. As an example:

```
# chapter0pythonA/chapter1_python_example.py
x = []                              # Create an empty list
for i in range(5):                  # A for-loop
    if i == 0 or i == 2 or i == 4:  # Check if i is even
        x.append(i)                 # If even, add i to the list x.
print(x)                            # Print the list x
```

INTERPRETER

When you then run your program, a special program called an **interpreter** reads your code and begin to execute the lines in your program starting from the top. During execution, variables are assigned and changed as dictated by each line of code. Eventually, the above program will reach the `print`-function and print out:

```
[0, 2, 4]
```

If you are not entirely sure about everything that goes on in this program don't panic; you will be after the next two chapters. For now, let's just try to name the different elements:

BUILD-IN DATA TYPE

- The number `5`, and the variable `i`, are integers (or in python lingo, an `int`). This is an example of a **build-in data type**. Other examples are decimal numbers and strings.

VARIABLE

DATA STRUCTURE

- The symbol `x` is a **variable**. It is initially initialized to an empty list, which is written as `[]`, which we then append integers to using the `append()` function. Things like lists are called the **data structure** because they allow us to structure the data the program manipulates.

CONTROL FLOW

- Statements like `if` and `for` are called called **control flow** statements, because they control the order (flow) in which the program lines are executed.

COMMENTS

- **Comments** are written as `## This is a comment` and are ignored by the interpreter.

[1] This is because a build-in library in python was changed between version 3.6 and 3.8. The code itself should be compatible with older versions of python.

## 1.2 The primitive data types

The primitive data types are the atoms of a programming language, and probably the most important thing to be familiar with. There are a few extra niche types[2], but you only need to know the following five:

### 1.2.1 Integers (`int`)

INTEGERS

**Integers** are whole numbers such as $-2$, $-1$, $0$, $1$, $2$, $117$. Naturally they support standard algebraic operations like addition and subtraction:

```
>>> 2 + 4                   # Add two integers
6
>>> 50 - 5 * 6
20
>>> (2 + 2) * (3 - 1)       # You can group operations using parenthesis
8
>>> width = 20              # Assign the variable 'width' to a value of 20
>>> height = 5*9            # And also height is assigned the value of 5 * 9 = 45
>>> area = width * height   # Compute the area of a rectangle and asign it to 'area'
>>> area                    # This line shows us the value of 'area'
900
```

### 1.2.2 Decimal (floating point) numbers (`float`)

FLOATING-POINT NUMBERS

**Floating-point numbers** are *decimal* numbers such as $3.14$, $-10.3$ and so on, and support operations like division, multiplication, addition and subtraction. They are defined using the period sign like `7.01`. For instance:

```
>>> 2.5 + 1.5
4.0
>>> 5 / 2       # Division of two integers automatically creates a floating-point number
2.5
>>> 3.5 * 1e2   # Create a floating-point number using scientific notation, i.e. 3.5 * 10^2 = 350.
350.0
>>> 1e-3        # Scientific notation for 10^-3 = 0.001
0.001
>>> 1.25 * 4
5.0
>>> 12 / 3      # Do you find anything surprising about this result?
4.0
>>> cost_per_apple = 1.5
>>> apples = 5
>>> total_cost = apples * cost_per_apple
>>> total_cost
7.5
```

#### Why are there two number-types?

The distinction between an integer-type and a more general floating-point type has to do with how computers store things internally. Integers are easy to store exactly in a

[2]Such as complex numbers, hexadecimal numbers and the binary data type.

binary format, whereas floating point numbers, such as `0.1`, are actually quite tricky to represent efficiently when you only have a finite amount of memory per number.

What the computer does is that it stores a number which is *very close* to `0.1`, and various rounding-tricks are then applied to ensure this distinction is rarely noticed. Consider this example: It should be the case that $1-(1-x) = 1-1+x = x$ for all $x$, but if we actually try the example in python using $x = 0.1$:

```
>>> 1 - (1-0.1)
0.09999999999999998
```

Generally speaking, when you do calculations involving floating-point numbers, the result will tend to accrue minuscule errors, which only in rare cases become a problem[3]. TL;DR: If you can define your program using only integers you should since the result will be exact in that case.

### Number-operations

MODULUS OPERATOR

As we saw, the normal division operator `/` convert integers to floating-point numbers. Python has a special whole-integer division operation `//` which does division and rounding, as well as the **modulus operator** % which computes the remainder after division. Exponentials can be computed using the double multiplication operator `a**b`:

```
>>> 16 / 3 # classic division returns a float
5.333333333333333
>>> 16 // 3   # floor division discards the fractional part
5
>>> 16 % 3    # the % operator returns the remainder of the division. 16 = 5*3 + 1.
1
>>> 17 % 3    # Same as above. In this case 17 = 5*3 + 2
2
>>> 18 % 3    # Equal to 0 because 3 divides 18 so 18 = 6*3 + 0
0
>>> 5 * 3 + 1 # result * divisor + remainder
16
>>> 2 ** 3    # Computes 2^3 (as an integer)
8
>>> 5.0 ** 2  # Computes 5^2 (as a float)
25.0
```

### Conversion

The name of a datatype can be used for conversions. When you convert a floating point number to an integer, the number is rounded down. For instance:

```
>>> int(3.2)      # Integer to float
3
>>> float(3)*2.5  # integer to float (very rarely useful)
```

[3]The magnitude of the errors will depend on the magnitude of the numbers involved, try for instance `(1e18 +1)-1e18`. Rounding problems that arise from floating-point inaccuracies are called overflow or underflow. My personal rule of thumb is to trust floating-point numbers to a precision of about $10^{-10}$.

```
7.5
>>> 3 * 2.5       # Same as above; python does automatic conversion
7.5
```

In other words, you should think about `int` as actually being a function: It takes an argument which is not an integer and converts it to an integer.

### 1.2.3 Booleans (`bool`)

BOOLEAN

A **Boolean** is a special data type which can only take two values, `True` or `False`. Booleans support operations such as `or`, `not` and `and`. They most commonly arise as the result of comparisons, like `x==3`. This expression is either `True` or `False`, depending on `x`, and therefore we can assign it to a variable such as `y = x == 3`. In this case, `y` will be a Boolean variable which will be `True` if `x` is `3` and otherwise `False`. Some examples

```
>>> x = 8
>>> y = x == 3           # y is 'False'
>>> y
False
>>> z = x != 7           # This will be true because 'x' is not equal to 7
>>> z
True
>>> y or x               # This will be True
8
>>> y or not z or 3 == 8 # This will be False
False
>>> i = 3
>>> i == 0 or i == 2 or i == 4  # This will be True if i is an even number less than or equal to 4.
False
```

### 1.2.4 Strings (`str`)

STRING

A **string** is used to store text. Strings are an incredible versatile data type in python and, compared to other languages, very easy to use. Some examples of using strings:

```
>>> "Hamberder"          # A string
'Hamberder'
>>> x = "Green eggs"     # A string-variable
>>> len(x)               # The number of letters in x
10
>>> y = " and ham"       # Another string variable
>>> x + y                # Concatenating two strings to gether
'Green eggs and ham'
>>> x + (y * 2)          # Multiplication can repeat strings
'Green eggs and ham and ham'
>>> x == "Green eggs"    # test if 'x' is "Green eggs"
True
```

Strings can be defined using either single or double-quotes, or using triple-quotes which allow us to write multi-line strings. Regardless, they are all just strings:

```
>>> 'Donald Trump is a "clever" buisnessman'    # Single-quotes allow double-quotes in string
'Donald Trump is a "clever" buisnessman'
>>> "Donald Trump is a 'honest' politician"     # Double-quotes allow single-quotes in string
"Donald Trump is a 'honest' politician"
>>> h = """triple-quotes allow both " and ' (and may span several lines) """
>>> len(h)
62
```

Some, myself included, use the triple-quote strings as a way to write multi-line comments; other people consider this a bad habit, but they are wrong.

### String conversion

A really nice thing about python is that all common-sense things usually works. For instance, this is how you can convert between strings and other data-types such as `int`, `float` or `bool`:

```
>>> x = 107
>>> str(x)              # Convert x to a string
'107'
>>> int("107")          # Convert the string '107' to an integer
107
>>> float("12" + "77")  # Rarely the right way to add numbers
1277.0
```

This may seem like a very small thing to make a fuzz about, but these things usually requires a trip around stackexchange in Java or C++.

### String formatting

If this was a normal programming course I would spend more time on all the cool things you can do with strings, however, we are mostly going to use strings as a way to output things. The **f-string syntax** provides a super convenient way to format output. To provide a few examples which combine the `print`-command and string formatting:

F-STRING SYNTAX

```
>>> apples = 10
>>> print("There are " + str(apples) + " apples")  # Nasty old way
There are 10 apples
>>> print("There are", apples, "apples")           # Using multi-input print (better)
There are 10 apples
>>> print(f"There are {apples} apples")            # Using the automatic f-string formatting feature
There are 10 apples
>>> price = 2/3
>>> f"Total cost of all {apples} apples is {price*apples}"      # Not nice
'Total cost of all 10 apples is 6.666666666666666'
>>> f"Total cost of all {apples} apples is {price*apples:.2f}"  # Format with two decimal after the period.
'Total cost of all 10 apples is 6.67'
```

## 1.2.5 The None-type (`None`)

NONE

The **none** type can just take a single value, `None`. Surprisingly, this turns out to be quite useful since they convention in python is to let `None` stand for *not currently assigned*.

We can test if something is `None` or not using the comparison-operators `==` and `!=` or the special, and sometimes more readable, syntax `is None` and `is not None`.

```
>>> name = "Bob's bakery"
>>> name is None
False
>>> name == None
False
>>> None is None
True
>>> name is not None
True
```

## 1.3 If/else, functions and exceptions

IF

An **if**-statement has the form `if <boolean-variable>:` and execute the indented code if the statement is true. A simple example:

```
>>> x = 42
>>> if x > 10:
...     print("x is larger than 10!")
...
x is larger than 10!
```

ELSE

The `if` can be followed by an **else**-clause which is executed if the `if`-statement is `False`:

```
>>> x = 42
>>> if x > 117:
...     print("x is larger than 117")
... else:
...     print("x is not larger than 117")
...
x is not larger than 117
```

ELSE-IF

The optional `elif` (**else-if**) statements can be used to check multiple conditions in a row:

```
>>> x = 10
>>> if x > 10:
...     print("x is greater than 10!")
... elif x == 10:
...     print("x is equal to 10")
... else:
...     print("x is less than 10")
...
x is equal to 10
```

### 1.3.1 Functions (`def`)

FUNCTION

A **function** takes one (or more) input-arguments, executes the content (body) of the

function, and return a value. Functions are the soul of programming, as they allows the same pieces of code to be re-used in different contexts:

```
>>> def test_my_pet(animal):
...     if animal == "dog" or animal == "cat":
...         print("That is a pretty normal pet!")
...     elif animal == "parrot" or animal == "lizard":
...         print("This is an unusual pet")
...     elif animal == "dragon":
...         print("No way!")
...     else:
...         print("I am not familiar with this kind of animal")
...
>>> test_my_pet("lizard")
This is an unusual pet
>>> test_my_pet("bunny")
I am not familiar with this kind of animal
```

By default, a function will return (i.e., compute) the value `None`, but you can specify another value using the return-keyword. Lets write a function which solves the equation $ax + b = 0$ for $x$:

```
>>> def solve_linear(a, b):  # Solve the equation ax + b = 0
...     return -b / a
...
>>> solve_linear(3, 2)
-0.6666666666666666
>>> solution = solve_linear(10, 3)   # Assign the output of the function to a variable
>>> print(f"The solution of 10x + 3 = 0 is x = {solution}")
The solution of 10x + 3 = 0 is x = -0.3
```

EXCEPTION

This code is buggy in the case where $a = 0$. In this case it might be beneficial to tell the user clearly what is wrong, which can be done by raising an **exception**. The exception will cause the program to fail with an error message we can specify. We can do this as follows:

```
>>> def new_solve_linear(a, b):  # Solve the equation ax + b = 0
...     if a == 0:
...         raise Exception(f"I don't know what to do about the equation: {a}x + {b} = 0")
...     else:
...         return -b / a
...
>>> new_solve_linear(3, 2)
-0.6666666666666666
>>> new_solve_linear(0,2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 3, in new_solve_linear
Exception: I don't know what to do about the equation: 0x + 2 = 0
```

There is more to exceptions than this, for instance the `try`/`except` keywords can be used to anticipate exceptions and handle them without failing, but you will not be using this functionality in this course. Why should you, the code will be perfect!.

### 1.3.2 Inlined functions (`lambda`)

INLINED FUNCTION

If a function just take up a single line, you can define it as an **inlined function** using

the `lambda` keyword. As an example, consider these equivalent ways to define a one-line function

```
>>> def misterfy(x):  # A basic function
...     return "Mr " + x
...
>>> misterfy("Bunny")
'Mr Bunny'
>>> misterfy2 = lambda x: "Mr " + x  # Same thing, one line shorter
>>> misterfy2("Cat")
'Mr Cat'
```

The `lambda`-keyword is a bit obscure, and we will only use it a very few times. The recommended way to define a function is to use the familiar `def function_name():` syntax.

### 1.3.3 What is a good function?

When you write your code, try to break it up into different functions. This both tends to make your code easier to write, read, and will allow you to easier re-structure and re-use your program later. Novice programmers tend to use few functions, and better programmers tend to write more functions.

- If you feel you have a choice between using one or two functions for a task, use two.
- Functions should be short. Try to make them fit on a single screen.
- A function should do a single thing. If the behavior of your function depends on input arguments (for instance, your function may find the maximum of a second degree polynomial or the roots based on an input argument), you should probably use two functions.
- The exception to the above is a public-facing API, such as matplotlibs `plot()`-function. This function does a ton of stuff, but that is because it is intended for a third-party user
- Break these rules if it is more convenient in your concrete setting.

# Chapter 2

# Compound data types and iteration

COMPOUND DATA TYPE

A **compound data type** is a build-in data structure which allows us to collect many instances of the simple data types for later processing. In this section, we will focus on the four most important compound data types, starting with lists. The later half of the chapter will consider iteration techniques for the compound data types.

## 2.1 Lists (`list`)

LIST

The most important compound data type in python is the **list** for two reasons: (i) Lists are just really useful and (ii) strings and most other compound data-types behaves like lists.

A list is simply a sequence of items which can be of any type, including lists themselves. Lists can be manually defined using squared brackets and commas, and the `len` function can be used to compute the length. A list can also be empty:

```
>>> squares = [1, 4, 9, 16, 25] # A list of 5 elements, each element is an integer.
>>> squares
[1, 4, 9, 16, 25]
>>> len(squares)                # Compute the number of elements
5
>>> empy_list = []       # An empty list
>>> empty_list2 = list() # This also creates an empty list
```

The element at the $i$'th place in the list (from the left) can be accessed using the notation `my_list[i]`. Conveniently, the index can be negative, which will access the list from the right. Therefore, `my_list[-1]` will always be the last element, `my_list[-2]` the element before that, and so on:

```
>>> squares[0]   # First element
1
>>> squares[1]   # Second-element
4
>>> squares[-1]  # Last element
25
>>> squares[-2]  # Element before the last
16
```

SLICING

If you want to get a sub-sequences of a list, you can do that using the **slicing**-operator `:`. This can be used in conjunction with negative indexing to get a new list which corresponds to either the first or last part of the list:

```
>>> squares[2:]   # Slice from 2 to end
[9, 16, 25]
>>> squares[:-3]  # Negative indexing works with slicing
[1, 4]
>>> squares[:]    # Creates a copy of the list
[1, 4, 9, 16, 25]
```

Strings and lists are actually quite similar if you think about a string as a list of characters, and strings can also be indexed and sliced. Furthermore, like strings, lists can be added using addition `+` or repeated using multiplication `*`:

```
>>> more_squares = squares + [36, 36, 49, 64]  # Concatenate two lists
>>> more_squares
[1, 4, 9, 16, 25, 36, 36, 49, 64]
>>> even_more_squares = squares * 3            # Repeat list three times
>>> even_more_squares
[1, 4, 9, 16, 25, 1, 4, 9, 16, 25, 1, 4, 9, 16, 25]
```

MUTABLE

It is possible to update the elements in a list. The technical term for a datatype which can be changed like this is that it is **mutable**

```
>>> primes = [2, 3, 4, 7, 9]  # This seems wrong
>>> primes[2] = 5             # Set element 2 equal to '5'.
>>> primes
[2, 3, 5, 7, 9]
```

IN

A cool thing about python how readable the syntax is. If you want to know if an element is **in** a list, just use the `in`-keyword

```
>>> primes = [2, 3, 5, 7]
>>> 5 in primes      # True
True
>>> 4 in primes      # False
False
>>> 7 not in primes  # False
False
```

APPEND

A few options are available to increase/decrease the size of a list. The `append` command is used to **append** elements to the list:

```
>>> primes.append(11) # Add 11 to the list
>>> primes.append(13)
>>> primes
[2, 3, 5, 7, 11, 13]
```

Furthermore, `pop` can be used to remove (pop) elements and return them for processing, `del` can remove elements by index, and the more rarely used `clear()` will remove all the elements in the list:

```
>>> last_element = primes.pop() # Remove the last (right-most) element
>>> last_element                # This will be the value we just removed (popped)
13
>>> primes                      # This contains the remaining elements.
[2, 3, 5, 7, 11]
>>> del primes[1]               # Remove the 2nd element of the list
>>> primes
[2, 5, 7, 11]
>>> primes.clear()              # remove all elements in the list
>>> primes
[]
```

### 2.1.1 Intermezzo: The `for`-loop and lists

FOR LOOP
ITERATE

To provide a bit of context we will take a peak at iteration and lists. The **for loop** allows us to **iterate** over lists. As an example

```
>>> for animal in ["cat", "dog", "dragon"]:
...     print("Look! A large", animal)
...
Look! A large cat
Look! A large dog
Look! A large dragon
```

This allows us to structure similar computations. As an example, let's compute the squares of a sequence of numbers:

```
>>> numbers = [1, 2, 4, 5]
>>> cubes = []
>>> for n in numbers:
...     cubes.append(n ** 3)  # computes n^3
...
>>> cubes
[1, 8, 64, 125]
```

As an additional example, the following functions uses a for-loop and list-addition to compute the reverse of a list (as we will see later, there is a much better way)

```
>>> def reverso(numbers): # A function which reverse a list
...     reversed = []
...     for n in numbers:
...         reversed = [n] + reversed  # reversed.insert(n,0) is faster
...     return reversed
...
>>> print(reverso([1, 1, 2, 3]))
[3, 2, 1, 1]
```

### 2.1.2 The `range`-function

RANGE FUNCTION

Often, we want to iterate over a list of numbers in arithmetic progression. This can be accomplished using the **range function** which, given $n$, counts the integers from 0 to $n-1$:

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

The end-point $n$ is not part of the sequence, and so the sequence contains exactly $n$ values and will be useful to index a list of length $n$.

You can let `range` start from another number, and even specify different increments using a function call such as `range(start, stop, increments)`. This allows `range` to count backwards (note 2 is not included):

```
>>> for i in range(10, 2,-2):
...     print(i)
...
10
8
6
4
```

As an example, we can use this to enumerate the elements in a list:

```
>>> animals = ["dog", "cat", "tiger"]
>>> for k in range(len(animals)):
...     print("Animal", k, "is a", animals[k])
...
Animal 0 is a dog
Animal 1 is a cat
Animal 2 is a tiger
```

If you just print `range`, something strange occurs:

```
>>> print(range(5))
range(0, 5)
```

ITERATOR

So why does this not just output the list `[0, 1, 2, 3, 4]`? What happens is that `range` actually returns an **iterator**. This is a special object which, when used in a for-loop, returns the successive elements in the desired sequence exactly like a list, but computes the items on-the-fly, thus saving memory. Iterators are common in python because they save memory. If you want, you can convert an iterator to a list using the `list()` function:

```
>>> print(list(range(5)))
[0, 1, 2, 3, 4]
```

Using the range function we can begin to build our first useful program. The following program tests if a number $n$ is a prime:

```
>>> def is_prime(n):           # Test if n is prime assuming n >= 2.
...     for k in range(2, n):  # k = 2, 3, 4, .., n-1
...         if n % k == 0:     # Check if the remainder is zero. In that case n =  k * x + 0 and so k divides n.
...             return False   # Divisor found, n is not a prime
...     return True            # No divisors found, so n must be a prime
...
>>> print("Is 11 prime?", is_prime(11))
Is 11 prime? True
>>> print("Is 9 prime?", is_prime(9))
Is 9 prime? False
>>> print("Is 2 prime?", is_prime(2))
Is 2 prime? True
```

To understand this program, what it does is to check, for each $k = 2, \ldots, n-1$, if $k$ divides $n$. This is here done by checking if the remainder is zero. If this is the case, the function returns `False`, and otherwise it will return `True` since a number without divisors is prime.

### 2.1.3 The `break` and `continue`-statements, and `else` in a loop

The `break` statement can be used to stop a `for`-loop before time. Here is an example where we break the inner loop whenever we find a divisor:

```
>>> for n in range(2, 8):
...     for k in range(2, n):
...         if n % k == 0:
...             print(f"{n} = {k} * {n//k}")
...             break
...
4 = 2 * 2
6 = 2 * 3
```

Often you want to do something special depending on whether the loop was terminated early or not. This can be done by using an `else` statement, which is executed *only* if the loop is *not* stopped with a `break`. The effect is easier seen than explained:

```
>>> for n in range(2, 8):
...     for k in range(2, n):
...         if n % k == 0:
...             print(f"{n} = {k} * {n//k}")
...             break
...     else:
...         print(n, "is a prime number")
...
2 is a prime number
3 is a prime number
4 = 2 * 2
5 is a prime number
6 = 2 * 3
7 is a prime number
```

When a `continue`-statement is encountered in a loop the interpreter skips the code following the continue-statement and immediately jump to the next iteration of the the loop. A slightly artificial example:

```
>>> for animal in ['dog', 'dragon', 'cat', 'pig', 'bunny']:
...     if animal in ['pig', 'dragon']:  # Not a nice animal!
...         continue                     # Skip the rest of the for-loop
...     print("What a nice", animal)     # Skipped by 'continue'
...     # If there was more code here, it would also be skipped
...
What a nice dog
What a nice cat
What a nice bunny
```

This is useful when the loop contains a lot of code which you sometimes don't want to execute, but you can often obtain the same result using an `if`-statement (as is certainly the case here). Generally speaking, you should try to avoid `break` and `continue`-statements if you can.

## 2.2 Tuples (`tuple`)

TUPLE

On the surface, a **tuple** is exactly like a list except it is defined using parenthesis rather than square bracket:

```
>>> t = (135, "a dog", [5, 7]) # Tuples are defined using parenthesis, and may contain any number of values
>>> print(t)
(135, 'a dog', [5, 7])
>>> s = 1, 4, 9, 16            # You can in fact drop the parenthesis, although it makes your code look worse.
>>> print(s)
(1, 4, 9, 16)
>>> singleton_tuple = (1,)     # A singleton contains just a single element. Note the comma!
>>> print(singleton_tuple)
(1,)
>>> empty_tuple = ()           # Define an empty tuple. Just writing `empty_tuple = ()` will also work.
>>> print(empty_tuple)
()
>>> print("Length of the tuples are:", len(t), len(singleton_tuple), len(empty_tuple))
Length of the tuples are: 3 1 0
>>> x = list(t)      # Convert to a list---
>>> print(x)
[135, 'a dog', [5, 7]]
>>> print(tuple(t))  # ---and back to a tuple
(135, 'a dog', [5, 7])
```

Tuples also behave like lists in the way they are indexed, sliced, added together, and so on. Just a few examples:

```
>>> toys = ("ball", "boat", "doll", "waterpistol")
>>> "boat" in toys             # test for membership
True
>>> toys[2:]                   # Slicing works just like lists
('doll', 'waterpistol')
>>> toys[2]                    # Membership to.
'doll'
>>> toys + ("jack in a box",)  # Addition of tuples
('ball', 'boat', 'doll', 'waterpistol', 'jack in a box')
```

IMMUTABLE

Since tuples appear to be identical to lists, you might wonder why we have two different types. The difference is that tuples are **immutable**, meaning that they cannot

change after you have defined them. For instance, we cannot update an element of a tuple like we can with a list:

```
>>> toys[1] = "lego"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

This guarantees the tuple cannot be changed by other code. Asides this, tuples and lists can in most cases be used interchangeably. By convention, tuples are most commonly used in situations where you have a short sequence of things of different types, and lists are commonly used when you have a (longer) sequence of things of the same type, as in the prime-number example.

### 2.2.1 Sequence unpacking and functions

TUPLE PACKING
TUPLE UNPACKING

Declaring a tuple using just commas and not parenthesis, for instance `2, "Hamberder!"`, is called **tuple packing** because you pack things into a single tuple. **Tuple unpacking** is when you go the other way and define multiple variables in one go from a single tuple. The following example illustrates both:

```
>>> t = 1, "dog", "five beers"
>>> print(t)
(1, 'dog', 'five beers')
>>> children, pet, breakfast = t  # The tuple is 'unpacked' into the three variables.
>>> print(f"Miranda has {children} children, a {pet}, and her last meal was {breakfast}")
Miranda has 1 children, a dog, and her last meal was five beers
```

Tuple unpacking can be used to define multiple variables in one line, and also to solve the completely useless programming pop-quiz question of swapping two variables `x` and `y` in a single line without using any intermediate variables:

```
>>> x, y = 10, 3 # Define two variables in a single line
>>> x, y = y, x  # Swap the order of x and y
>>> print(x, y)
3 10
```

Defining multiple variables is usually looked down upon because it is considered a less readable, but I think it has its uses.

By far the biggest application of packing/unpacking are functions with multiple return values. Consider a function to solve a second-degree polynomial:

```
>>> import math  # Ignore this for now.
>>> def solve_polyal(a, b, c): # Solve ax^2 + bx + c = 0
...     d = b**2 - 4*a*c
...     if d < 0:
...         raise Exception("Oy veh, no solutions")
...     sol1 = (-b - math.sqrt(d))/(2*a)
...     sol2 = (-b + math.sqrt(d))/(2*a)
...     return sol1, sol2  # Note this function returns a tuple with two elements
...
>>> x1, x2 = solve_polyal(1, -5, 4)  # solve x^2 - 5x + 4 = 0
```

```
>>> print(f"Solution x1 =", x1, "Solution x2 =", x2)
Solution x1 = 1.0 Solution x2 = 4.0
```

Although we might *say* that this function *return two values*, in fact all it does is return a tuple, defined using tuple packing as the two-element tuple `(sol1, sol2)`, and then when we use the function, we use tuple unpacking to define `x1` and `x2` at the same time. The following examples should make this obvious by capturing all the output in a single tuple:

```
>>> solutions = solve_polyal(1, -5, 4)
>>> print("The solutions as a tuple", solutions)
The solutions as a tuple (1.0, 4.0)
```

If we are only interested in one of the outputs, assign the other to the (dummy) underscore variable `_`. This tells other programmers that we are going to ignore the corresponding output value:

```
>>> _, sol = solve_polyal(1, -5, 4)
>>> print("A solution is", sol)
A solution is 4.0
```

## 2.3 Sets (`set`)

SET

A **set** is an unordered collection of values which does not contain duplicate elements. Sets are created either using the `set`-function (which can convert lists or tuples to sets) or using curly parenthesis. Sets has some convenient, but less often used, syntax for quickly finding union, intersection and difference:

```
>>> s = {1, 5, 1, 2, 2, 3, 4, 5, 7, 8}  # Define a set. Notice the list has duplicates
>>> print(s)
{1, 2, 3, 4, 5, 7, 8}
>>> len(s)
7
>>> all_numbers = set(range(9)) # Create all numbers 0 to 9 by converting the range(9) sequence to a set.
>>> primes = {2, 3, 5, 7}       # All primes less than 11.
>>> 4 in primes                 # Test for membership
False
>>> all_numbers - primes        # Set difference; non-prime numbers
{0, 1, 4, 6, 8}
>>> s & primes                  # the set of numbers in s which are prime
{2, 3, 5, 7}
```

In the example above, notice how the order of the elements in the first example changes when duplicate elements are automatically removed.

Sets are the least used of the composite datatypes we will consider in this chapter, but have their use. For instance, a common beginner-problem is how to find the unique elements in a list, and as shown below we can either solve this using a clunky (and slow) function, or by using two conversions to weed out the non-unique elements:

```
>>> def unique(s): # Find unique elements in s the hard way
...     u = []
...     for x in s:
...         if x not in u:
...             u.append(x)
...     return u
...
>>> toys = ["car", "ball", "ball", "doll", "car"]
>>> unique(toys)     # The hard way
['car', 'ball', 'doll']
>>> list(set(toys))  # The easy way
['car', 'doll', 'ball']
```

## 2.4 Dictionaries (`dict`)

DICTIONARY

A **dictionary** stores a map from one type of object (the key) to another (the value). You should compare this to a list, which creates a mapping from the index `i` (the key) to `my_list[i]` (the value), but dictionaries greatly extends on this idea by allowing the key to be something more general.

Dictionaries can be defined either using curly parenthesis and colons, or using the `dict()` function. The following examples illustrates various operations on dictionaries; pay attention to how the syntax is similar to that of lists:

```
>>> prices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}  # Define a dictionary
>>> print(prices)
{'apples': 1.0, 'oranges': 1.5, 'pears': 1.75}
>>> dict(apples=1.0, oranges=1.50, pears=1.75)  # Definition using dict. Note the lack of quotes.
{'apples': 1.0, 'oranges': 1.5, 'pears': 1.75}
>>> dict()                                      # As always, this creates an empty dictionary
{}
>>> print("Oranges cost", prices['oranges'])    # Look up value associated with 'oranges'
Oranges cost 1.5
>>> prices['plums'] = 0.75                      # Add a new (key,value) pair to the dictionary
>>> print(prices)
{'apples': 1.0, 'oranges': 1.5, 'pears': 1.75, 'plums': 0.75}
>>> print(list(prices))                         # Print the keys in the dictionary
['apples', 'oranges', 'pears', 'plums']
>>> for fruit in prices:                        # Consistent with the above, this loop over the keys
...     print(fruit, "cost", prices[fruit])
...
apples cost 1.0
oranges cost 1.5
pears cost 1.75
plums cost 0.75
>>>
>>> 'pears' in prices     # Return True because key is in dictionary
True
>>> 'coconuts' in prices  # Returns False
False
>>>
>>> len(prices)           # Number of keys
4
>>> del prices['oranges'] # Remove an element from prices
>>> print(prices)
{'apples': 1.0, 'pears': 1.75, 'plums': 0.75}
```

The keys in a dictionary can be any *immutable* type. That is, they can be a string,

integer, float or tuple of these types, including tuples of tuples and so on. The keys *cannot* be a set, list or a dictionary itself, because they are *mutable*.

Dictionaries give us a lot of freedom in terms of what we consider the keys and values, and therefore they can be a little harder to use when you are just starting out. We are therefore going to look at a couple of examples inspired by the course where dictionaries provide the by far most useful data-types.

### 2.4.1 Example: A probability assignment

We can use a dictionary to represent a probability assignment. Consider the variable $X$ that can take three values:

$$P(X = -1) = 0.2, \quad P(X = 0) = 0.3, \quad P(X = 1) = 0.5 \tag{2.1}$$

We represent this by letting the $X$-value be the key, and the probability the value:

```
>>> Px = {-1: 0.2,
...       0: 0.3,
...       1: 0.5}  # A finite probability distribution
>>>
>>> print("P(X = -1) =", Px[-1])
P(X = -1) = 0.2
```

DEFAULT VALUE

Normally, we access elements in a dictionary using the syntax `dictionary[key]`, but we can also use the `get`-function, which has the benefit of accepting a **default value**

```
>>> Px = {-1: 0.2, 0: 0.3, 1: 0.5}  # A small probability distribution
>>> Px.get(1)    # Same as Px[1]
0.5
>>> Px.get(2, 0) # If the key (`2`) is not found in the dictionary the get-function will return the second argument `0
0
```

### 2.4.2 Example: A tiny grid-world

As another example, we can implement a small gridworld where the player is confined to a $(x, y)$ grid by using a dictionary:

```
>>> gridworld = {(1, 1): "Player",
...              (8, 8): "Goal",
...              (4, 5): "Trap"}
>>>
```

This representation is both very compact, and it is easy to update the players location.

### 2.4.3 Example: The fruit-store

Suppose an online store keep track of prices of fruit, and a customer can submit an order request which specify a fruit he or she wishes to buy as well as the quantity. We

can implement this as follows (note we raise an exception when the customer tries to buy something we don't have):

```
>>> prices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}  # Define a dictionary
>>> def buy_fruit(prices, fruit_type, quantity):
...     if fruit_type not in prices:
...         raise Exception(f"No {fruit_type} for you!")
...     else:
...         return prices[fruit_type]*quantity
...
>>> print("Cost of 3 oranges:", buy_fruit(prices, "oranges", 3))
Cost of 3 oranges: 4.5
```

More realistically, the customers will submit a shopping basket containing the number of each type of fruit the wish to buy. This can naturally be represented as a dictionary, where the keys are fruit, and the values are the quantity they wish to buy. We can then re-use the previous function so we don't have to duplicate the error handling code:

```
>>> def buy_lots_of_fruit(prices, order):
...     total = 0
...     for fruit in order:
...         total += buy_fruit(prices, fruit, order[fruit])
...     return total
...
>>> order = {'apples': 2, 'pears': 10}
>>> print(f"Cost of {order}:", buy_lots_of_fruit(prices, order))
Cost of {'apples': 2, 'pears': 10}: 19.5
```

### 2.4.4 Example: A small database ✸

As a final example, the values can in turn be anything, for instance tuples or dictionaries. In 02450 a dictionary is used to keep track of the students id, their exam (multiple-choice) score, and the score in their reports. We can implement this as a dictionary-of-dictionaries where one of the values in the nested dictionary is a tuple of the report scores:

```
>>> students = {'s210001': {'name': "Jacob Larsen", "exam_score": 74, "report_score": (75,60)},
...             's210002': {'name': "Marie Hansen", "exam_score": 88, "report_score": (90,95)},
...             's210003': {'name': "Line Denielsen", "exam_score": 72, "report_score": (80,50)}}
>>>
>>> students['s210003']['exam_score'] = 80  # Adjust exam score for this student
>>> print("Report score of s210003", students['s210003']['exam_score'])
Report score of s210003 80
```

The point of a representation such as this is we can on-the-fly add new fields. For instance, the following code computes the total score (used for grading) by averaging the exam and report scores:

```
>>> for id in students:
...     students[id]['total'] = 0.75 * students[id]['exam_score'] + 0.25 * sum(students[id]['report_score'])/2
...
```

```
>>> print(students['s210002'])  # All students have a total score which can be discretized to a grade
{'name': 'Marie Hansen', 'exam_score': 88, 'report_score': (90, 95), 'total': 89.125}
```

### 2.4.5 The collections-module (`defaultdict`) ✸

There are other build-in datatypes in python which are more niche, and can be found in the `collections` module[1]. We will only concern ourselves with one of these, the `defaultdict`.

DEFAULTDICT

As the name suggest, a **defaultdict** is just a regular dictionary, but returns a default value when the user look up a key which has not been added to the dictionary yet. The default value is specified by supplying a function to the dictionary, which is invoked to set the default value. Recall that `float` is actually a method, so that `float()` return `0.0`. We can use this to extend the example in section 2.4.1 so that when we ask for the probability of any $x$ where the probability assignment $P(X = x)$ is not defined, it returns a probability of 0:

```
>>> from collections import defaultdict # We return to the import statement later
>>> Px = defaultdict(float, {-1: 0.2, 0: 0.3, 1: 0.5} ) # The first argument set the default argument, second argument
>>> print("P(X=0) =", Px[0])      # Dictionary work as normal
P(X=0) = 0.3
>>> print("P(X=32) =", Px[32])    # Return default value where it is not defined
P(X=32) = 0.0
>>> print("P(X=100) =", Px[100])  # Also return default value
P(X=100) = 0.0
```

The second example extends the students directory example from section 2.4.4 by adding a default record for new students:

```
>>> from collections import defaultdict
>>> students = {'s210001': {'name': "Jacob Larsen", "exam_score": 74, "report_score": (75,60)},
...             's210002': {'name': "Marie Hansen", "exam_score": 88, "report_score": (90,95)},
...             's210003': {'name': "Line Danielsen", "exam_score": 72, "report_score": (80,50)} }
>>>
>>> students2 = defaultdict(lambda: {'name': None, "exam_score": None, "report_score": (None,None) }, students)
>>> students2['s999999']['exam_score'] = 80  # Adjust exam score of this student
>>> students2['s999999']
{'name': None, 'exam_score': 80, 'report_score': (None, None)}
```

## 2.5 Looping techniques

Python has a couple of build-in functions which makes looping over sequences simpler and more readable. The first is the `enumerate` function, which is useful when you also want an index of a sequence:

[1] See `https://docs.python.org/3/library/collections.html`.

```
>>> bands = ["ACDC", "Rolling stones", "Beatles"]
>>> for k, band in enumerate(bands):
...     print(k, band)
...
0 ACDC
1 Rolling stones
2 Beatles
```

Python also has a handy function to loop over a sequence in reverse order

```
>>> for i in reversed(range(4)):
...     print(i)
...
3
2
1
0
```

and sorted in ascending order

```
>>> for f in sorted(["Martin", "Jacob", "Anna", "Charlie"]):
...     print(f)
...
Anna
Charlie
Jacob
Martin
```

Sometimes, you want to loop over two lists at the same time. This can be accomplished by the `zip` function:

```
>>> students = ["s210001", "s210002", "s210003"]
>>> grades = [12, 7, 10]
>>> for id, g in zip(students, grades):
...     print(id, g)
...
s210001 12
s210002 7
s210003 10
```

### 2.5.1 Looping over a dictionary (`items`)

The default behavior when looping over a dictionary is to loop over the keys:

```
>>> prices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
>>> for fruit in prices:
...     print(fruit)
...
apples
oranges
pears
```

More control can be gained by using the `items()` function, which allows you to iterate over both keys and values, and if you want a list of either keys or values use the `keys()` and `values()` methods:

```
>>> prices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
>>> for fruit, price in prices.items():
...     print(fruit, "cost", price)
...
apples cost 1.0
oranges cost 1.5
pears cost 1.75
>>> print(list(prices.keys()))    # All fruit
['apples', 'oranges', 'pears']
>>> print(list(prices.values()))  # All prices
[1.0, 1.5, 1.75]
```

## 2.6 List and dictionary comprehension

LIST COMPREHENSION

Many tasks in python programming consists of transforming one composite data type into another by applying an operation on all elements. **List comprehension** provides an elegant way of doing this using an intuitively appealing syntax.

Consider the following code which computes a sequence of squares

```
>>> squares = []
>>> for x in range(1, 10):
...     squares.append(x**2)
...
>>> print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Using list-comprehension, the exact same code can be written as follows

```
>>> squares = [x ** 2 for x in range(1, 10)]
>>> print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension also allows us to specify a conditional statements so that not all elements are processed. The general form of a list comprehension call is as follows:

```
# chapterOpythonB/pythonOB.py
[f(x) for x in my_list if condition(x)]
```

To illustrate the conditional statement, consider the following code which returns the sub-sequence of positive elements of a list:

```
>>> x = [1, -30, 2, 3, -5 ]
>>> [z for z in x if z > 0]
[1, 2, 3]
```

Or this code, which uses list comprehension to compute a sequence of primes, by only including those numbers where `is_prime` returns `True`:

```
>>> def is_prime(n):
...     for k in range(2,n-1):
...         if n % k == 0:
```

```
...             return False
...     return True
...
>>> primes = [n for n in range(15) if is_prime(n)]
>>> print(primes)
[0, 1, 2, 3, 5, 7, 11, 13]
```

We can in fact also test if a number `n` is a prime by testing if the list of divisors is empty

```
>>> n = 11
>>> len([k for k in range(2,n-1) if n % k == 0]) == 0  # Check if 11 is a prime (compare to is_prime(11))
True
```

Combining these two ideas, the sequence of primes can be computed using a single line as follows:[2]

```
>>> [n for n in range(2,12) if len([k for k in range(2,n-1) if n % k == 0]) == 0]
[2, 3, 5, 7, 11]
```

List comprehension can also be used to define both sets and tuples by simply changing the outer brackets, but since this is rarely used we will not provide any examples.

### 2.6.1 Dictionary comprehension

DICTIONARY COMPREHENSION

**Dictionary comprehension** is like list comprehension, except we are now creating dictionaries. The following example illustrate the idea by creating a new dictionary having names as keys, and the length of the names as the values:

```
>>> names = ["Lancelot", "King Kong", "Odin", "George Washington"]
>>> name_lengths = {name: len(name) for name in names}  # Create a new dicionary with the length of the names
>>> print(name_lengths)
{'Lancelot': 8, 'King Kong': 9, 'Odin': 4, 'George Washington': 17}
```

The syntax should be familiar from list comprehension. As an example, you can use an `if`-statement to filter the dictionary based on the length of the name:

```
>>> short_names = {name: l for name, l in name_lengths.items() if l < 6}
>>> print(short_names)
{'Odin': 4}
```

The following example builds a dictionary where the keys are integers, and the values are a list of divisors of the number:

```
>>> divisors = {n: [k for k in range(1,n) if n % k == 0] for n in range(2,11) }
>>> print(divisors)
{2: [1], 3: [1], 4: [1, 2], 5: [1], 6: [1, 2, 3], 7: [1], 8: [1, 2, 4], 9: [1, 3], 10: [1, 2, 5]}
```

[2] Be warned that packing a ton of functionality into a single line tends to feel better the moment you write it than it does when you read it again 2 months later.

A perfect number is a number which is equal to the sums of its divisors. For instance 6 is a perfect number because $6 = 3 + 2 + 1$. We can find all perfect numbers from the dictionary of divisors by using list comprehension:

```
>>> many_divisors = {n: [k for k in range(1,n) if n % k == 0] for n in range(3000) }
>>> len(many_divisors)  # This is now a long list of numbers
3000
>>> perfect_numbers = [n for n, divisors in many_divisors.items() if sum(divisors) == n]
>>> print(perfect_numbers) # Print the perfect numbers less than 3000. There are only four of them!
[0, 6, 28, 496]
```

## 2.7 More on functions

Next, let's consider a few extra things in our toolbox which won't dramatically change what we can do, but can both make life easier.

### 2.7.1 Named arguments and default values

Input arguments to functions can have default values. Consider this example where the variable `children` will have a default value of `3`:

```
>>> def whois(name, age, children=3):
...     return f"{name} is {age} years old and has {children} children"
...
>>> whois('Jane', 32, 2) # Works as a regular function
'Jane is 32 years old and has 2 children'
>>> whois('Jane', 32)    # Use the default of 3 children
'Jane is 32 years old and has 3 children'
>>> whois('Jane', 32, children=4)  # Works as well
'Jane is 32 years old and has 4 children'
```

Using named arguments, as we did in the last example when we wrote `whois('Jane', 32, children=4)`, has the two-fold benefit of both telling us which argument we are assigning which value, and also making the order of the inputs unimportant. For instance, this works:

```
>>> whois(age=65, children=10, name='Bob')  # Named argument allows us to change the order of inputs (but don't!)
'Bob is 65 years old and has 10 children'
```

Functions with a ton of optional input arguments, such as matplotlibs `plot`-function, makes use of this technique a lot.

### 2.7.2 Variable input arguments (`*` and `**`)

Function can have variable input arguments by using the `*`-symbol. When this symbol is put in front of a tuple, it can be thought of as stripping away the parenthesis (sometimes called exploding the tuple). A slightly contrived example makes use of this to generate a list from a tuple:

```
>>> a = (1, 2, 3)  # A small tuple
>>> b = [*a]       # The star * strips away to the parenthesis; output can be captured to create a list
```

The thing is the symbol also works in reverse, allowing us to capture multiple inputs in a tuple. The one place this is useful is function definitions. Read the following example carefully:

```
>>> def capturing(*args):
...     print("args is:", args) # Args is a tuple of all inputs
...
>>> capturing("icecream", 'hotdogs', 7) # Capture three inputs
args is: ('icecream', 'hotdogs', 7)
```

This mean the function `def capturing` can have any number of input arguments. We can use this to deal with one argument as we normally do, and then use the `*` symbol to capture the rest. In this example, `args` will be a tuple with three elements:

```
>>> def treats(dog, *args):
...     for a in args:
...         print(dog, "eats a", a)
...
>>> treats("Fido", "Steak", "Sofa", "Cat")
Fido eats a Steak
Fido eats a Sofa
Fido eats a Cat
```

### Capturing named arguments

The double-star (`**`) command works the same way except for dictionaries. This can be used to capture named arguments (as dictionaries) as shown below:

```
>>> def multiple_capturing(*args, **kwargs):
...     print("Normal arguments", args)
...     print("Keyword arguments", kwargs)
...
>>> multiple_capturing(1, 2, name='Bob')
Normal arguments (1, 2)
Keyword arguments {'name': 'Bob'}
>>> multiple_capturing(banana='good', pear='bad')
Normal arguments ()
Keyword arguments {'banana': 'good', 'pear': 'bad'}
```

The one place this is most useful is when you have one function which delegates most of it's functionality to another function. In that case, you often simply want to send most (or all) input arguments to the other function. This can be accomplished as follows:

```
>>> def redirector(*args, banana='good', **kwargs):
...     print("The banana is", banana)
...     multiple_capturing(*args, **kwargs) # 'explode' the args, kwargs to call function with these as inputs.
...
>>> redirector(1, 2, 3, banana='bad', pear='fine')
The banana is bad
```

```
Normal arguments (1, 2, 3)
Keyword arguments {'pear': 'fine'}
```

Notice the function treats the named input-argument `banana` in a special way and will not send that argument on to the `multiple_capturing` function. The `banana` input is therefore handled as usual, using the default value, as the following illustrates:

```
>>> redirector("what will happen next?")
The banana is good
Normal arguments ('what will happen next?',)
Keyword arguments {}
```

## 2.8 Example: Conways game of life ✸✸

CONWAYS GAME OF LIFE

Let's consider a complete example of how the data-structure we have seen can be used to create a small animation of **Conways game of life**. Conways game of life play out on an (infinite) grid. Each $(x, y)$ tile in the grid has 8 neighbors (corner-neighbors are included) and can be either occupied (alive) or not (dead). The game proceed in rounds by the following rules

- For each tile $(x, y)$, count the number of neighbors $n$
- If the tile is alive, and $n = 2$ or $n = 3$, the tile stay alive to next round
- If the tile is empty, and $n = 3$, the tile become alive
- In all other cases, the tile is dead in the next round

Many would naturally use a numpy matrix to represent the game world, but since the game world does not have a limit, a more natural choice is a dictionary where the keys are the $(x, y)$ coordinates and a value of 1 means the tile is occupied. This allows us to only track the living tiles.

While we maintain a potentially infinite game world, we will only plot a section of it of width 200. The initialization of the game world is as follows:

```
# chapterOpythonB/conway.py
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation
from collections import defaultdict

n = 200  # size of visible game world.
# Conways game is represented as a defaultdict with keys (i,j).
# A value of 0 means a tile is inactive, 1 that it is active.
# Example:
# > map[(20,10)] == 1  if tile (20,10) is active.
mat = defaultdict(int)
# We initialize the game-world by placing a bunch of random active tiles in the middle:
for _ in range(1000):  # place 1000 random tiles
    mat[(np.random.randint(30) - 15 + n//2, np.random.randint(30) - 15 + n//2)] = 1
```

For plotting, we will extract a $200 \times 200$ sub-set of the game world and turn it into a numpy `ndarray`. This is accomplished using a small function and `matplotlib`:

```
# chapterOpythonB/conway.py
def world2mat(mat):
    # Convert 'mat' into a n x n x 3 RGB matrix for plotting by matplotlib
    G = np.ones((n,n,3))
    for (x,y) in mat:
        if min(x,y) >= 0 and max(x,y) < n: # Check we are inside viewing area
            G[x,y,:] = 0                   # Tile is active, paint it black
    return G

fig = plt.figure()
ax = plt.axes(xlim=(0,n), ylim=(0,n))
image = ax.imshow(world2mat(mat), interpolation='nearest')  # Plot an image of the world in RGB format
```

Next, we have to implement the game rules. The following code execute update number `i` by first counting the number of neighbors, then implementing the game rules based on the neighbor counts. When it is done, it updates the image we just defined:

```
# chapterOpythonB/conway.py
def one_conway_step(i):
    mat2 = defaultdict(int)
    # This loop count neighbours for each (x,y) tile, but only track those with non-zero neighbours
    for x, y in mat:
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                if dx == dy == 0:
                    continue
                mat2[x+dx,y+dy] += 1
    # Conway game rules:
    # * Active Tiles with 2 or 3 neighbours keep living
    # * Active tiles with other neighbour counts die from loneliness or overcrowding
    # * Empty Tiles with 3 neighbours become active (birth)
    mat2 = { (x,y): 1 for (x,y), num in mat2.items() if num == 3 or (num == 2 and mat[x,y] == 1 ) }
    # We use two dictionaries so all updates happen instantaneous
    mat.clear()
    mat.update(mat2)
    image.set_data(world2mat(mat))
    return image,
```

Using this code, we can create an animation by telling python to call the `one_conway_step()` function a number of times and record the frames. You don't have to understand this code; I just copied it from stackexchange:

```
# chapterOpythonB/conway.py
anim = animation.FuncAnimation(fig, one_conway_step, frames=100, interval=20, blit=True)
# save the animation as an mp4. This requires ffmpeg to be installed; uncomment to just show video.
anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
```

That's it! You should now get a nice little animation of Conways game of life and a `mp4`-movie.

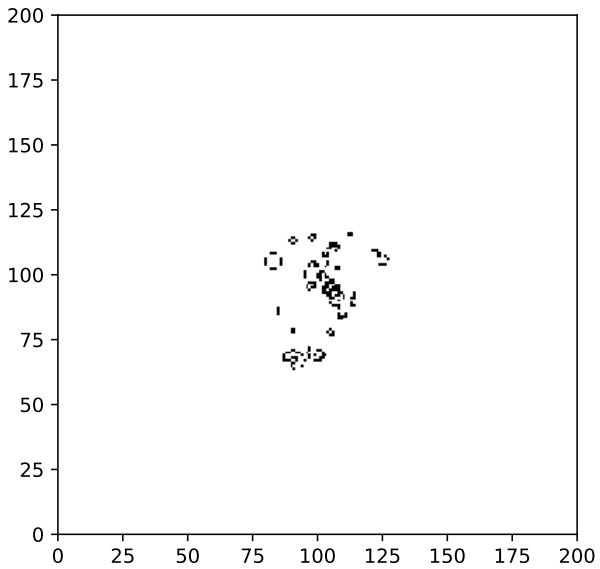

Figure 2.1: Conways game of life

# Chapter 3

# Classes and packages

The previous two chapters have actually shown most of the low-level features of python you need to write normal code, but there is still a big piece missing: Organization! Consider the following example to computes the length of the vector $\begin{bmatrix}3\\4\end{bmatrix}$:

```
>>> import numpy as np
>>> from scipy.linalg import norm
>>> x = np.asarray([3, 4])
>>> x   # What is x?
array([3, 4])
>>> norm(x)
np.float64(5.0)
```

This code makes use of two forms of code organization:

IMPORT

- The first two lines **import** functionality from the two packages `numpy` and `scipy`. What this fundamentally does is that it allows us to split code into multiple files and, more generally, **packages** of files such as `numpy`

PACKAGES

OBJECT

- The line `x = np.asarray([3,4]))` creates an `ndarray` **object** assigned to the variable `x`. This allows us to bundle data (in this case the small vector) and functionality to manipulate the data together.

OBJECT ORIENTED PROGRAMMING

Programming centered on writing and using objects is called **object oriented programming**, and this is pretty much anything interesting in python (tensorflow, numpy, sklearn, pytorch, matplotlib to name just a few). It is my experience some students consider object-oriented programming an esoteric subject they don't need to know, and still use objects without understanding what they are. This is a shame! Firstly, it will prevent one from doing or understanding most things in python (including deep learning), and secondly, it is not hard to learn.

## 3.1 Modules and packages (`import`)

MODULE

When we use the python interpreter interactively, all our code is lost the moment we close it. Therefore, you should write your code in a `.py` file called a script or **module**,

and then edit, run and debug it in your IDE.

As your program gets longer, it becomes useful to split it into several files, simply because a file with many hundreds of line of code is difficult to navigate. The **module name** is simply the name of the `.py` file, and it is possible to **import** definitions of functions, variables and other functionality from one module into another. This is what allows programs to be split into several files.

MODULE NAME
IMPORT

Let's look at an example. Go to any directory in your computer, for instance `c:/Users/tuhe/Documents/sample` and save the following snippet to `c:/Users/tuhe/Documents/sample/fruit.py`:

```
# fruit.py
fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}

def buyFruit(fruitPrices, fruit, numPounds):
    if fruit not in fruitPrices:
        # This is the easiest way to format text. Notice the f in the front.
        print(f"Sorry we don't have {fruit}")
    else:
        cost = fruitPrices[fruit] * numPounds
        print(f"That'll be {cost} please")


if __name__ == '__main__':  # True when script is run
    buyFruit(fruitPrices, 'apples', 2.4)
    buyFruit(fruitPrices, 'coconuts', 2)
```

In the same directory, start the python interpreter by running the `python` command, and you will see you can use the code in the file using the `import` command. As you can see, you can now import the file

```
>>> import fruit
>>> fruit.fruitPrices
{'apples': 2.0, 'oranges': 1.5, 'pears': 1.75}
```

There are a couple of helpful variations of the import command. You can use the `from` keyword to avoid having to write the module name in dot-notation, and also use the `import <module> as <name>` to create a handy alias, which you are no doubt familiar with when you import numpy and matplotlib

```
>>> from fruit import fruitPrices, buyFruit
>>> buyFruit(fruitPrices, 'apples', 10) # Buy some fruit
That'll be 20.0 please
>>> from fruit import *  # Import everything.
>>> from fruit import fruitPrices as prices
>>> print(prices)
{'apples': 2.0, 'oranges': 1.5, 'pears': 1.75}
```

So far we have used the command-line, but we can certainly also import modules from other modules (i.e., scripts). When a module is imported, all lines in the script that contains the module is run. It is often useful to distinguish between when a script is being imported, and when it is being run explicitly using `python <scriptname.py>`. This can be accomplished using the special `if __name__ == "__main__"` check as shown below, since the special `__name__`-variable is only equal to `"__main__"` when the script is run directly using `python`.

```python
# import_sample.py
from fruit import fruitPrices
print("Price of apples", fruitPrices['apples'])

if __name__ == "__main__":
    print("This is only run when the file is run directly using the python-command")
```

In this case, the bottom `print`-statement will execute when we run the script as `python import_sample.py`, but *not* if we import it as `import import_sample`.

## 3.2 packages

PACKAGE

A **package** allows modules to be bundled together in an organized structure (the package) so they can easily be used later. Examples of packages are all the build-in packages like `sys`, `os`, `collections` and so on, as well as third-party packages such as `numpy`, `torch`.

From a practical standpoint, a package is just a directory with some python files (modules) in it. Let's consider the code for this course. All code is distributed as the package `irlc`, and within this package we have **subpackages** (these are simply subdirectories) which allows us to create a more granular organization as shown below:

SUBPACKAGES

```
irlc/                    # Package name
    __init__.py          # Optional init-script which is run when the package is imported
    ex00/                # Each week is a subpackage
        fruit.py         # A module. Your code is in a file such as this.
        (...)
    project0/            # The first test-project
        fruit_project.py # A file relevant for the project
        (...)
    exam/                # Folder for midterms and eventually the exam
        (...)
    utils/               # A subpackage of utility functionality which would clutter each exercise week
        (...)
    (...)
```

That's actually about it. Obviously, in order for python to recognize your package, it must be in a place where python is told to look for packages. System-wide packages such as `numpy` which are installed using the `pip`-command, are placed in a special directory. For custom package, like `irlc`, there are two ways to tell python how to look for them:

- By setting the so-called `PYTHONPATH` environment variable in your operating system
- By specifying it in your IDE

The later is by far the recommended approach and was covered in the installation video.

Once this is done, using the package is super easy using the dot-notation `A.B` which allows you to import from sub-directories in your package structure:

```python
# class_sample.py
# Import example from the fruit-package
from irlc.ex00.fruit import fruitPrices
from irlc import Agent
```

The last line is a bit strange, since the `Agent` is actually a member of the `train` module found in the `ex01` directory. The reason why it works is because when the package `irlc` is imported, it reads (and executes) the `__init__.py` file by default (as you can tell, things surrounded by two underscores are special in python). This file in turn contains the line `from irlc.ex01.train import Agent as Agent` which ensures `Agent` can be imported as in the example.

## 3.3 Classes and objects

Modules and packages allows us to organize code into multiple files. Object-oriented programming (OOP) allows us to organize the logic/functionality of our programs into logical parts called **classes**. That makes OOP a bit hard to illustrate because the simplest examples will have next to nothing to organize (and therefore appear irrelevant) whereas relevant examples are necessarily more complex (and therefore makes object-oriented programming seem difficult).

CLASSES

I have decided to start with the simplest examples, but be aware that these are the real-world concerns that object-oriented problem addresses:

**Organizing programs:** If you have a reinforcement learning system which play a simulated game, it will contain hundreds of functions and small data-structures, containing everything from the game state to the state of the neural network of the agent. Object-oriented programming allows us to bundle data-structures and functions to manipulate the data-structures together, and makes such a system practical.

**Reduce redundant code:** Most things we build are alike in some ways and different in others. Object-oriented programming allows us to save coding effort by sharing functionality where it is useful. In this course, we will write simulation-code once, and all control environments will have the same simulation functionality.

**Working with other people:** Suppose you are building a reinforcement-learning system, and another person is building an interesting test-environment. The only way your system, and the other persons environment, will work together is if you each have a clear, abstract specification in mind of how a system and environment should work. Object oriented programming allows you to formalize such a specification.

### 3.3.1 Defining a class (`class`)

Let's begin with the simplest possible class. This can be defined using the `class`-keyword in a single line:

```
>>> class BasicClass:  # Classnames are usually upper-case
...     pass           # `pass` is a special keyword which does nothing
...
```

INSTANTIATE

Once we have made a class, we can **instantiate** it by calling it like a function. After that, we can assign variables to the `instance` of the class (and read them again) using the dot-notation. The following creates two instances of the `BasicClass` and assign variables to both of them:

```
>>> a = BasicClass()   # Create an instance of the class
>>> a.name = "My first class"  # You can write data to the class like this
>>> b = BasicClass()   # Another instance. a and b are not related and can store different data:
>>> b.name = "Another class"
>>>
>>> print("Class a:", a.name)
Class a: My first class
>>> print("Class b:", b.name)
Class b: Another class
```

Let's just summarize some things:

- `BasicClass` is a class

INSTANCE
OBJECT

- When we do `a = BasicClass()` then `a` is an **instance** of the `BasicClass`, also known as an **object**
- Objects are not classes. Objects make use of the template laid out in `BasicClass`, but can keep their seperate value of variables (state). In this case `a` and `b` are objects each with different values of `a.name` and `b.name`

PROPERTY
ATTRIBUTE

- The object-specific variable `name` is called a **property** or **attribute**

In other words, the relationship between a class, and objects made by instantiating that class, is the same as the relationship between the (abstract idea of) a Dog, and various specific dogs: Specific instances each have the general properties of an abstract Dog, but they are individual and can have different names, ages, moods, breeds and so on.

### Class methods (`self`)

In these examples the class is nothing more than a container of variables. Let's consider a more interesting example in which the class represents a dog. We have now given the dog a build-in `name`-property (sometimes also called an attribute) with a default value and a function. The `name` attribute works as before:

```
>>> class BasicDog:
...     name = "Unnamed dog"  # Each dog-instance will have the property name
...     def read_nametag(self):
...         # This is a class-function. Note we must pass it `self` as a first argument, which is the
...         # instance of the class itself (i.e. the current object). This is how we can access properties of the clas
...         print("This dog is named", self.name, "please give me treats!")
...
>>> dog = BasicDog()
>>> dog.name
'Unnamed dog'
```

The `def read_nametag(self)` function is more interesting. This is how we call it:

```
>>> dog.name = 'Pluto'
>>> dog.read_nametag()  # Invoke the read_nametag() function. Note we don't pass the object to the function!
This dog is named Pluto please give me treats!
```

Class functions requires at least one input argument `self`. This is what allows it to access the objects own properties. That is, when you write `dog.read_nametag()`, then `self` is automatically assigned to `self = dog` when the method is called, which allows the `read_nametag` function to access class attributes.

### Constructors (`__init__`)

In most cases, we don't want variables to take dummy values such as `"Unnamed dog"`. To ensure variables are properly initialized, one should add the special `def __init__(self, ...)` function, known as the **constructor**, as shown in this improved example:

CONSTRUCTOR

```
>>> class BetterBasicDog:
...     def __init__(self, name):
...         self.name = name
...         self.age = 0
...         print(f"The __init__() function has been called with name='{name}'")
...     def birthday(self):
...         self.age = self.age + 1
...         print("Hurray for", self.name, "you are now", self.age, "years old")
...
```

The `__init__` method is called when the object is created, i.e. when you write `BetterBasicDog("Pluto")`, as shown here:

```
>>> d1 = BetterBasicDog("Pluto")        # the __init__ function is now called
The __init__() function has been called with name='Pluto'
>>> d2 = BetterBasicDog(name="Lassie")  # Also support named arguments
The __init__() function has been called with name='Lassie'
```

This ensures that both the `name` and `age` variables have been set, and we now use the `birthday` function to let the Pluto celebrate a couple of birthdays:

```
>>> d1.birthday()
Hurray for Pluto you are now 1 years old
>>> d1.birthday()
Hurray for Pluto you are now 2 years old
```

## 3.3.2 Class inheritance

Things are often alike in most ways and different in others. For instance, if we build two classes representing two control-problems (a mechanical arm and a little car), they may share a lot of information (such as information about constraints, functionality to simulate the systems, etc.) but be unlike in other regards (i.e. the exact dynamics).

INHERITANCE

Class **inheritance** allows two such classes to share most functionality by building a general class which contains the shared functionality, and then allowing the two actual

control problems to specify different dynamics, visualization code and so on. This can be an immensely powerful tool, and one which is vital to understand in order to build new models in e.g. pytorch, but we will try to introduce it as a toy example.

As a little warm-up, the following code selects a random element from a list:

```
>>> import random
>>> random.choice(['Apples', "Bananas", 'Coconuts'])
'Coconuts'
```

With this in mind, let's build a small parrot class we can train to say random words:

```
>>> class Parrot:
...     def __init__(self):
...         self.words = ["Squack!"]
...     def learn(self, word):
...         self.words.append(word)
...     def speak(self):
...         return random.choice(self.words)  # Return a random word
...     def vocabulary(self):
...         return self.words
...
```

The parrot has methods to learn a new word (by simply adding them to a list), and then when it speaks, it returns a random word from the list. A small interaction could look as follows:

```
>>> parrot = Parrot()
>>> words = ["sugar", "sleep well", "(parrot noises)", "*honk*"]
>>> for word in words:
...     parrot.learn(word)
...
>>> for _ in range(3):  # Say three words
...     parrot.speak()
...
'sleep well'
'Squack!'
'sleep well'
>>> print("Vocabulary", parrot.vocabulary())
Vocabulary ['Squack!', 'sugar', 'sleep well', '(parrot noises)', '*honk*']
```

Let's suppose we want to build an alternative parrot which has a memory of just a single word. Clearly, all we need to change is the `learn`-function, and so it would be a shame to copy-paste all the other methods. This is where class inheritance comes in. We can in fact accomplish this as follows:

```
>>> class ForgetfulParrot(Parrot):
...     # The Parot class is used as a template.
...     # All functions in the Parot-class are therefore 'imported' as default, including 'self.words'
...     def learn(self, word):   # This function overwrite the 'actual' learn function in the Parot class
...         self.words = [word]  # This parrot only know a single word
...
```

The trick is the `class ForgetfulParrot(Parrot):` line. What this tells python is that the `Parrot` class should be used as a template, and the new method we define in the

`ForgetfulParrot` class, `def learn`, should overwrite the existing method in the `Parrot` so that it only saves a single word. Asides that, everything works as before:

```
>>> old_parrot = ForgetfulParrot()
>>> old_parrot.learn("damn remote")
>>> old_parrot.learn("Jeopardy")
>>> print("Vocabulary", old_parrot.vocabulary())
Vocabulary ['Jeopardy']
```

INHERITS
EXTENDS

We say that the `ForgetfulParrot` **inherits** from the `Parrot`-class, and equivalently that it **extends** the `Parrot` class.

### 3.3.3 Calling super-class constructors (`super`)

Suppose we get a new requirement: Make a parrot which says parrot noises before and after each word. Since each parrot presumably have different noises, we want to add this to the constructor. Our attempt at implementing this parrot might look as follows:

```
>>> class BadSqueekyParrot(Parrot):
...     def __init__(self, squeek="Quck!"):
...         self.squeek = squeek
...     def speak(self):
...         return f"{self.squeek} {random.choice(self.words)} {self.squeek}"
...
>>> squeeky = BadSqueekyParrot(squeek="Kvak-Kvak")
>>> squeeky.learn("Good night!")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 5, in learn
AttributeError: 'BadSqueekyParrot' object has no attribute 'words'
```

This gives an error, because the `self.words`-variable is assigned in the `__init__` function *as defined in the* `Parrot` *class*. But we just overwrote the `__init__` function, so now the code is broken.

To fix this, we need to use the special `super()`-keyword which allows us to access methods from the parent class. The following shows a correct implementation which also use the `super()`-keyword to borrow the `speak()`-function from the `Parrot` class:

```
>>> class SqueekyParrot(Parrot):
...     def __init__(self, squeek="Quck!"):
...         super().__init__()   # Call the 'Parot' class __init__ method to set up the words-variable.
...         self.squeek = squeek # save the squeek variable
...     def speak(self):
...         word = super().speak() # Use the speak() function defined in the Parrot class.
...         return f"{self.squeek} {word} {self.squeek}"
...
>>> squeeky = SqueekyParrot(squeek="Kvak-Kvak")
>>> squeeky.learn("Good night!")
>>> squeeky.learn("Tell that damn bird to shut it's beak")
>>> squeeky.learn("Sugar!")
>>> squeeky.speak()
'Kvak-Kvak Good night! Kvak-Kvak'
>>> squeeky.speak()
'Kvak-Kvak Squack! Kvak-Kvak'
```

### 3.3.4 Why inheritance is so awesome ★★

In the parrot-example inheritance just saved us a few lines. In more realistic cases, the class we inherit from (in the previous case, the `Parrot`-class) will contain a bunch more functionality, meaning that when we inherit from it, we can build classes with very few lines which can do some really cool things.

As an example, imagine we want to build our own little web service. This may seem difficult, but in fact all we need it to inherit from the `BaseHttpRequestHandler` class, and overwrite the method which return output to the browser. We can then hook this class into a python `HTTPServer` class which exactly expects a `BaseHttpRequestHandler`. The full example looks as follows:

```
# chapterOpythonC/server.py
class BasicServer(BaseHTTPRequestHandler):
    def compute_output(self, url):  # The function which actually computes the output shown to the user
        return "Requested url: " + url

    def do_GET(self):
        self.send_response(200)                         # Special stuff. Tell the browser we are alive
        self.send_header("Content-type", "text/html")   # Set type of output (in this case a html page)
        self.end_headers()                              # End header. The browser knows what is up
        self.wfile.write(bytes("<html><body>", "utf-8"))      # Write the start of the html file
        output = self.compute_output(self.path)         # The actual output we want to display (a string)
        self.wfile.write(bytes(f"<p>{output}</p>", "utf-8")) # Format it and output it.
        self.wfile.write(bytes("</body></html>", "utf-8"))   # Write the end of the html file

if __name__ == "__main__":
    webServer = HTTPServer(("localhost", 8080), BasicServer)  # Actually start the http server
    print(f"Server started http://localhost:8080")          # The url http://localhost:8080 will now work in a browser
    webServer.serve_forever()                               # Start web server
```

When happens is that when you go to the url `http://localhost:8080/page`, all the classes do their magic and eventually end up invoking the `do_GET(self)` method where the field `self.path` will be the (relative) url, in this case `"/page"`.

You then have to create an output the browser will understand, which means you have to generate a header (including content-type information) and write the output in binary encoded UTF-8 format[1]. But once this is over with, whatever string we return in the `compute_output` method will be send to the user and shown in the browser. You can see the result in fig. 3.1.

If we want to build a second webservice, it is even simpler: Instead of inheriting from `BaseHTTPRequestHandler`, we can inherit from the class we just wrote! We can use this to create a functional number-factoring service which can factor any number given in the url such as `http://localhost:8080/48`. An example result can be found in fig. 3.1.

```
# chapterOpythonC/server3.py
class PrimeFactorServer(BasicServer):
    def compute_output(self, url):
        n = int(url[1:])                                        # Extract number from url
        s = ', '.join([str(k) for k in range(2, n) if n % k == 0]) # Find and format factors
        return f"Factors of {n} are: {s}"                       # return a formatted string

```

[1] Don't worry too much about what that means; the internet is a mysterious and old thing

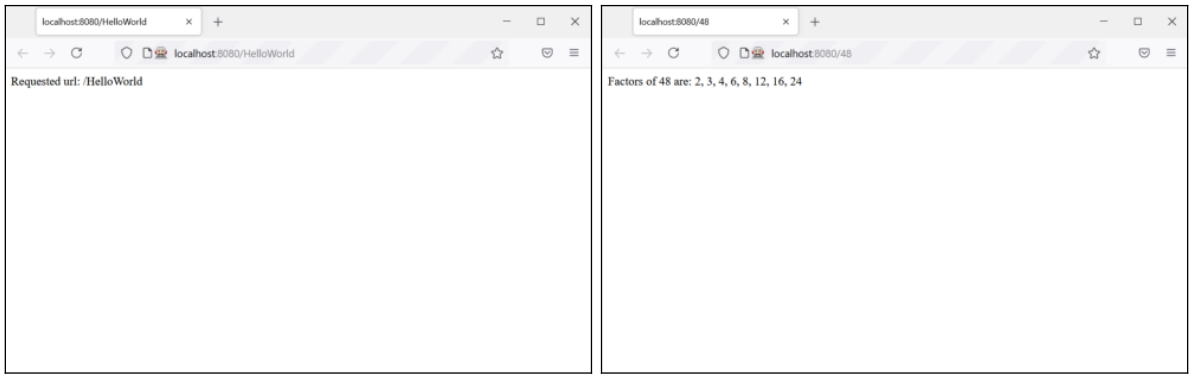


Figure 3.1: Screenshots of our two web-services. Next step, commodification of social relationships!

```
8  if __name__ == "__main__":
9      webServer = HTTPServer(("localhost", 8080), PrimeFactorServer)  # Actually start the http server
10     print(f"Server started http://localhost:8080")                  # The url http://localhost:8080 will now work in c
11     webServer.serve_forever()                                       # Start web server
```

### 3.3.5 Wrappers ✸

WRAPPER-PATTERN

A pattern is a fancy word for an often-used computer-programming trick, and the **wrapper-pattern** is one such trick we will use a couple of times in this course.

Consider this problem: Different people create python classes which represent computer games which a reinforcement learning agent can learn to solve. You are building such an agent and want to test it on several games. The output of the games are images (screenshots), however, the output of different games will be in different resolutions. Our reinforcement learning agent require all images to be in the same resolution of $140 \times 210$ pixels. How do we handle that?

- One approach is to do the re-sizing within your reinforcement learning code. This has the problem of polluting your algorithm with irrelevant functionality, and also ensures it will only work for a single game. Don't do that, it is nearly always a bad idea.
- Another slightly smarter approach is to create new environments which subclass the reinforcement learning game environments, and overwrite whatever methods that creates the images to make sure they have the right resolution. This method is better, but we still need to do this for every single environment, leading to duplicated code.
- Instead, the wrapper-pattern creates a single re-sizing class which can be used with any environment to do the resizing. This is the method openai-gym uses.

This is a bit abstract, so let's illustrate it with the different parrot-classes. Suppose we want to create a new type of parrot, a `TalkativeParrot`. This parrot should work like the regular parrots but just repeat the words it speaks four times.

We want this new parrot-type to work with any old parrot (after all, a forgetful and a regular parrot can both be talkaktive!). The example therefore resemble the image resizing example: The parrots are the game environments, and the repeated-words functionality is the resizing.

When we use the wrapper-pattern, it is accomplished as follows:

- Create a `TalkativeParrot` class.
- The `TalkativeParrot` class is given a concrete `Parrot` instance, `parrot`, in the constructor. This is the object which is wrapped.
- All methods of the `TalkativeParrot` class are delegated to the `parrot` object except the `speak`-function which is extended in order to say four words instead of one.

Practically, this is extremely simple to do:

```
>>> class TalkativeParrot:
...     def __init__(self, parrot):
...         self.parrot = parrot  # The Parrot-object we want to make talkative
...     def learn(self, word):
...         self.parrot.learn(word)  # Learn is delegated to the Parrot-object
...     def speak(self):
...         return " ".join([self.parrot.speak() for _ in range(4)])  # Speak 4 times. It is talkative!
...     def vocabulary(self):
...         return parrot.vocabulary() # Vocabulary is delegated to the Parrot-object
...
```

That's it! Let's put it to work by making both our regular parrot and our old parrot talkative:

```
>>> talkative_parrot = TalkativeParrot(parrot)
>>> talkative_old_parrot = TalkativeParrot(old_parrot)
>>>
>>> talkative_parrot.speak()
'sugar sleep well sugar sleep well'
>>> talkative_old_parrot.speak()
'Jeopardy Jeopardy Jeopardy Jeopardy'
```

This general technique is used in a number of situations, both generally in the reinforcement-learning framework openai-gym (for instance to do observation reshaping, but also framestacking, reward shaping, etc.), and we will also use it in our course software to make different similar-but-distinct models work together. Another benefit is you can stack wrappers, so that one wrapper does re-sizing, and another do something with the reward, and then you just apply both one after another to mix their functionality.

### 3.3.6 Type annotation ★★

Consider the following simple function that takes two arguments and returns a string:

```
# chapterOpythonC/annotation.py
def say_happy_hip_hurray(name, times):
    s = "hip "*times + "hurray, " + name
    return s

print(say_happy_hip_hurray("joe", 3))
```

This code produce the following output:

```
hip hip hip hurray, joe
```

You can probably guess, the first input argument ought to be a `str` and the second an `int`. Some programmers see an advantage in explicitly specifying that this is the case, as it makes it easier to reason about the program. This can be specified using

TYPE ANNOTATION

**type annotation**, in which case the program looks as follows:

```
# chapterOpythonC/annotation.py
def say_happy_hip_hurray2(name : str, times : int) -> str:
    s = "hip "*times + "hurray, " + name
    return s
```

The extra bits is the type annotation: It tells python the first input argument is a `str`, the second an `int`, and the function returns a `str`. There are advantages and disadvantages to type annotation:

- Plus: You can find some bugs quicker (such as swapping input arguments)
- Plus: Modern IDEs can use type annotation to offer better code suggestions
- Minus: Your code is harder to read at a glance
- Minus: Type annotations can be tedious to write

I am going to be using type annotation in a very limited manner in places where I feel it can help you catch certain bugs.

# Part II

# Optimal decision making

# Chapter 4

# Introduction

This course deals with situations where decisions are made one after another. The outcome of each individual decision may not be fully predictable, but can either be anticipated or observed before the next decision is made. The objective is to maximize the reward, which is a numerical measure of what is considered a desirable outcome.

This type of problem has a huge practical significance and arises in a variety of contexts: For instance, to control a robotic arm we must decide which motors to activate at each time step so as to bring about a desired goal, or to play chess against an opponent we must at each turn decide which piece to move and to where. Since the problem is very general, this chapter will introduce basic vocabulary and some key examples used throughout this course.

## 4.1 Introduction

Sequential decision making occurs in many situations. To mention a few:

- Bringing a rocket into orbit
- Driving a car
- Traversing a graph from node $A$ to node $B$
- Playing a computer game
- Maintain a dialog with an user in a question/answer format
- Plan which products to buy and when to maintain a warehouse inventory

A key aspect of these situations is that decisions cannot be viewed in isolation, since one must balance the desire for a high immediate reward with the undesirability of a low future reward. This fundamentally changes the nature of the problem from a typical machine learning task such as classification in two important ways:

- In machine learning, we only have to make a single decision (such as a label in a classification task, or a number in a regression or density estimation task, etc.) based on a fixed quantity of information. In the decision problem we have to take many decisions one at a time based on information which becomes progressively available

- In machine learning, we train the method based on a fixed, known quantity of information (the training set). In the decision problem, the available information is more nebulous, for instance knowledge of the dynamics of the environment, or simply the outcome of our actions as we take them

DECISION PROBLEM

The **decision problem** is an umbrella formulation, and the exact formal statement, as well as which properties of a solution is considered desirable, may differ: For a rocket ship we want a clear plan for how to reach our destination (and ideally, formal guarantees that minor disturbances underway will not lead to a catastrophic outcome), whereas for a computer game an approximate controller which can learn (rather than being programmed) may be desirable. Generally we will focus on two general subject areas:

CONTROL THEORY

**Control theory**, which has its origins in the 1940s with the advent of modern electronics and is by far the more mature field. The emphasis is on stability, robustness and optimality guarantees, which can be achieved by rigorous analysis of simple models. The successful applications of control theory are the Apollo lunar lander, plane autopilots, operation and optimization of industrial plants, and obviously numerous robotics application: Control theory is out there, everywhere, and works.

REINFORCEMENT LEARNING

**Reinforcement learning** which, in its modern form, really came together in the last half of the 1980s. It was partly inspired by neuroscience and psychology, specifically how the human reward systems and planning heuristics might work. Reinforcement learning focuses on the case where the outcome of individual actions are not known, but must be fully learned through experience. This lack of specificity often means there are fewer guarantees on the properties of the algorithms, and the focus is simply on obtaining the maximal reward.

The goal of reinforcement learning is a general learner, who can master any situation simply through experience, but we are not there yet. Perhaps the most impressive applications of reinforcement learning has been the AlphaGo-zero board game learner, and these methods are presently being scaled up to master real-time strategy computer games. Reinforcement learning is therefore primarily in laboratories and supercomputers, but is slowly making it's way into the real world [1].

Asides these two examples, the decision problem is encountered in many other subfields. An overview can be found in fig. 4.1

**Operations research** A mathematical discipline which treats generalizations of the inventory-control example below, which is of huge practical significance

**Economics** Much of economics study the decisions of of rational agents

[1] ChatGPT uses reinforcement learning as part of the training

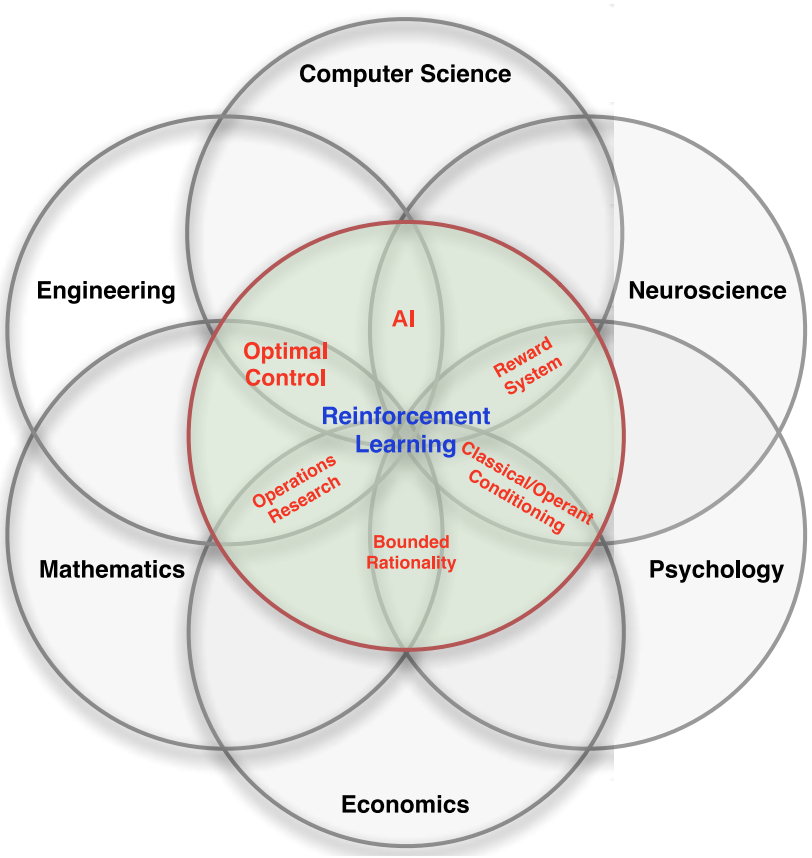


Figure 4.1: The decision problem, when formulated broadly, is encountered across a diverse range of fields, each with their own emphasis

**Neuroscience** Studies how the brain changes (learn) from repeated exposure of reward/punishment

**Psychology** Humans also take rational decisions. What principles guide human decision-making?

**Computer science** AI is to a large extend about building program which can solve sequential decision problems like talking, locomotion, step-wise reasoning/deduction, path-finding, and so on

**Engineering** Control and optimal control theory is centrally concerned with sequential decision making so as to bring about a goal

### 4.1.1 Scope and organization

A distinguishing facet of this course is to focus on an entire class of problems (decision problems) rather than specific methods (Gaussian processes, variational methods, regression trees, kernel methods, and so on).

This would in most circumstances be overly ambitious. However, as it turns out, much of what can fundamentally be said about the decision problem across all these fields rests on a few (*and arguably just one!*) fundamental ideas which are recycled again and again.

In other words, while we will consider a very diverse range of applications (search, multi-agent planning, optimal control, optimal planning, reinforcement learning), these are from one perspective really just special cases of the same algorithm, and the same types of tricks (short-horizon planning, game trees, etc.). These tricks are then re-discovered and re-used, with variations, under different names and notation in different fields.

To mention one example, most treatments of reinforcement learning develops it from the ground-up as a separate subject, however, as we will see the key methods reinforcement learning can just as well be thought of as starting with a minor simplification of the more general decision problem we will encounter in chapter 5, and then replacing a certain expectation with a sample average

$$\int f(x)p(x_i)dx \approx \sum_{i=1}^{n} f(x_i), \quad x_i \sim p.$$

### 4.1.2 Reward, cost, and other annoyances★

Each subfield which studies the decision problem has developed its own terminology and idiosyncrasies. In some cases this is purely cosmetic: Control theory typically calls whatever it is we try to control (for instance, a lunar rocket) the **plant**, whereas reinforcement learning calls it the **environment**. Control theory typically calls the computer program which makes decisions the **controller**, whereas reinforcement learning typically calls it the **agent**. Control theory typically calls the whole (i.e., plant and controller) the **system**, whereas reinforcement learning has no comparable term. I expect to do such a poor job at separating these terms I scarcely expect the reader to notice the difference.

PLANT
ENVIRONMENT
CONTROLLER
AGENT
SYSTEM

Other choices are more annoying. For instance, control theory calls the **state** (of the rocket, etc.) for $x$ and the **control signal** $u$; in reinforcement learning the comparable quantities are $s$ and $a$ (state and **action**). These annoyances will become a bit more pronounced when we implement the methods and have to settle for either $x/u$ or $s/a$.

STATE
CONTROL SIGNAL
ACTION

However, the biggest annoyances is that the objective of reinforcement learning is to maximize a **reward**; whereas the objective of control theory is to minimize a **cost**. This is a completely vacuous distinction; if we define reward as minus the cost both subjects could consistently use either reward or cost, and the distinction is therefore as deep as:

REWARD
COST

$$\max_x [f(x)] = \min_x [-f(x)] .$$

Unfortunately, early on the wrong sort of people got to decide these conventions, and the incompatible notation has now become fixed in the literature.

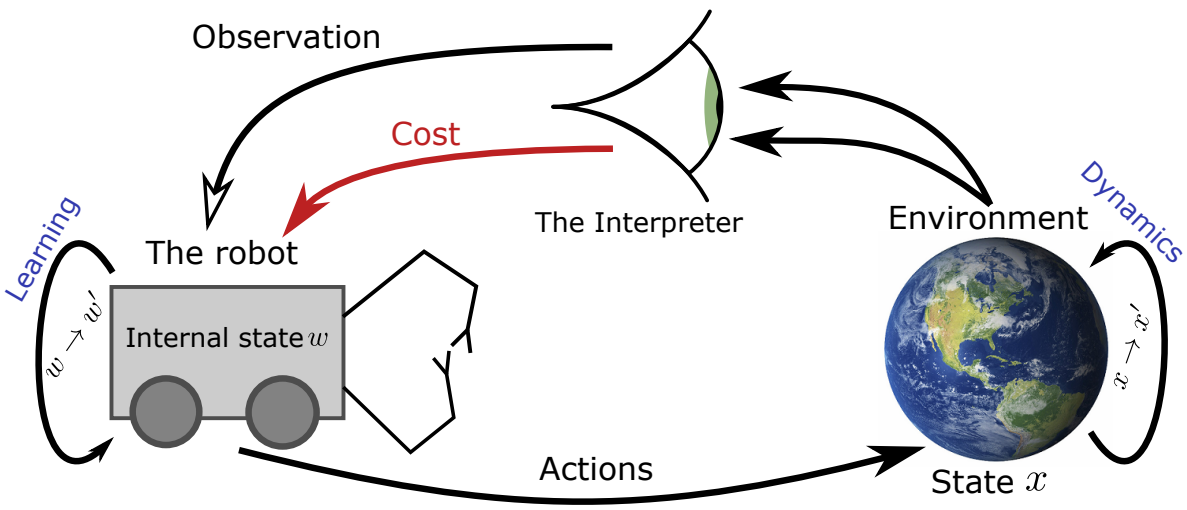


Figure 4.2: The decision problem. Based on observed input the robot selects and execute an action. The environment then change based on the action (and whatever state it was in) according to its dynamics. An interpreter assigns a cost, quantifying what undesirable, and the robot is then informed about the new state of the world (observation) as well as the cost. The robot may maintain an internal state useful for learning.

I have considered fixing these issues by teaching a variant of control theory which used the $s/a$/reward notation, but I think it will create more problems than it is worth, since all external resources will use a different notation. We will therefore stick with the long established conventions: When we discuss a control-theory topic, we typically call the state $x$ and minimize the cost, and when we talk about reinforcement learning, we typically call the state $s$ and maximize the reward.

## 4.2 The decision problem

DECISION PROBLEM

In this course we will use the **decision problem** to refer to the fundamental sequential decision-making problem outlined in the introduction. I have found it useful to have a clear model in mind which is illustrated as the cartoon in fig. 4.2.

ENVIRONMENT

AGENT

The figure has three components: An **environment**, which is the thing we interact with by taking actions (activation of wheels, etc.) and whose behavior is guided by the laws of nature. The second component is the robot (often also called an **agent**), which is what we try to program. What the agent practically does is that at each time step, it obtains measurements of the environment (an observation) and based on this, send out signals to its wheels, arms, or however it can affect the environment.

INTERPRETER

Above all this we have the interpreter; the role of the **interpreter** is to compute the cost function, which is sent into the agent as a signal, and the purpose of the agent (and therefore, our program) is to minimize this cost. Obviously, in a real application

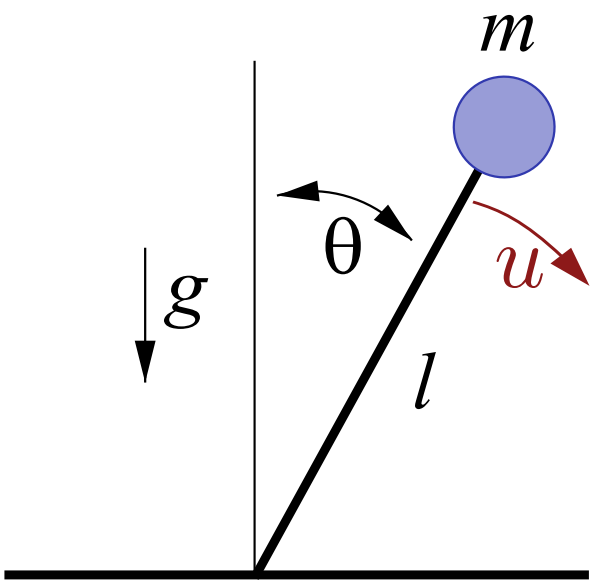


Figure 4.3: The pendulum is an example of a simple simple robotics tasks. The agent must balance the pendulum upright by applying a torque around the joint. Note in most circumstances it is assumed the motor is too weak to simply overcome gravity, and the pendulum must therefore be swung back-and-forth to bring it upright.

the interpreter will be build into the robot, but this does not change the salient point that *we* have to specify (interpret) which states should have a low cost (and therefore be desirable) and which should have a high cost (and therefore undesirable). Without this component, the robot will be useless.

The robot, environment and interpreter can be specified independent of each other: We may choose which control programs to equip the agent with; the same control program should be able to solve different tasks (i.e., different environments); and finally the cost function (interpreter) can encode different goals for the same agent and environment, for instance the difference between landing a rocket ship and bringing it into orbit.

### 4.2.1 Example: The pendulum

The first example is a prototypical continuous-time robot control problem, where the goal is to balance an arm (the pendulum) which is free to rotate around an axis upright (see fig. 4.3). The pendulum is subject to gravity, and we can activate a small motor which apply torque around the joint; it is assumed the motor is not powerful enough to simply balance the pendulum upright from the bottom position. Newtons laws allows us to express how the pendulum behaves. If $\theta(t)$ is the angle of rotation around the axis the pendulum is fixed to, and $u(t)$ is the torque applied at time $t$, then the behavior of

the pendulum is:

$$\frac{d^2\theta(t)}{dt^2} = \frac{g}{l}\sin(\theta(t)) + \frac{u(t)}{ml^2}$$

It is common to suppress the time index and use the dot-notation for time derivatives. In this notation the equations of motion becomes

$$\ddot{\theta} = \frac{g}{l}\sin(\theta) + \frac{u}{ml^2}. \tag{4.1}$$

Due to the way the coordinate system is defined, the pendulum is upright if[2] $\theta = 0$, and we will normally assume the pendulum starts out hanging downwards, which occurs when $\theta = \pi$.

In addition, a robotics system will nearly always be subject to constraints. In our case this may be the force we can apply is bounded

$$-u_{\max} \leq u \leq u_{\max}.$$

but in general constraints can reflect many different things, such as behavior we want the system to avoid (a large angular speed $\dot{\theta}$ will cause wear) or physical limitations (a robots foot cannot go through the floor).

Finally, we have to specify a start and a goal for the robot to complete the problem specification. In our case it might be that the robot starts at $t_0 = 0$ hanging downwards without movement:

$$\theta(t_0) = \pi, \quad \dot{\theta}(t_0) = 0$$

and the goal is to bring the pendulum upright while it is standing standing still at a known future time $t_F$. A convenient way to put this is as finding the control function $u(t)$ for $t \in [t_0, t_F]$ which maximize the cost function[3]:

$$c(\theta, \dot{\theta}) = -\cos(\theta(t_F)) + |\dot{\theta}(t_F)|$$

subject to all known constraints and assuming the system obeys the dynamics given in eq. (4.1).

This example contains all the ingredients for the fundamental problem in control: Find a control function $u$ which tells us what actions $u(t)$ to take at each time point $t$ so that, when we use $u$ to take actions, we obtain a low (and ideally, the lowest possible) cost without violating any constraints.

### 4.2.2 Example: Graph traversal ★

GRAPH TRAVERSAL

**Graph traversal** is likely familiar to the reader: We consider a graph of $n$ vertices, some vertices are connected by edges, and if $i, j$ are two vertices connected by an edge the edge will have an associated cost $a_{ij}$. The problem is then as follows: Starting from

[2] Or more generally, $\theta = 2\pi n$

[3] When is the first term the greatest and when it is the smallest? Why do we use cos and not sin?

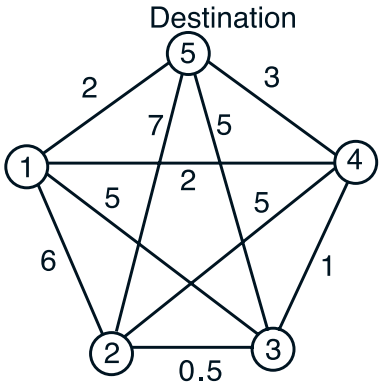


Figure 4.4: A very simple example of a weighted graph with $n = 5$ vertices where each is connected by an edge.

a vertex $s$ and given a goal vertex $t$, find the path traversing the graph from $s$ (start) to $t$ (target) along edges such that the sum of their cost is minimal. For instance, in the small graph in fig. 4.4 the cost of the optimal path from $s = 2$ to $t = 4$ is $a_{2,3} + a_{3,4} = 0.5 + 1$. This problem can be considered a decision problem, in which the robot starts in $s$, and then in each step $k = 0, \ldots, N - 1$ has to take a decision as to which of the available edges it wants to traverse (the action), and the immediate cost is the weight of the edge it traverse.

It may appear a bit odd to think of the problem this way, but as we will see many search algorithms arises as special cases for our more general algorithm.

### 4.2.3 Example: The inventory control problem

INVENTORY CONTROL

The **inventory control** problem will be the most important example, as it features all of the characteristics of the basic decision problem which will be introduced in chapter 5; it is also an important problem in its own right, since a great deal of resources go into managing stocks of various inventory.

A very simple inventory control problem can be defined as follows: We manage a warehouse which only store a single type of items, and we want to manage the stock of this item over $N$ days. Each day customers buy a quantity of the item from the warehouse, and our job is to each day place an order so as to meet the demand. Let us denote

- $x_k$: Number of the item in stock at the start of the $k$'th day (period)
- $u_k$: Stock ordered in the $k$'th period (assumed delivered at the start of the day)
- $w_k$: Demand (number of items bought) during the $k$'th period

The number of items in stock will change as simply $x_{k+1} = x_k - w_k + u_k$, but with the restriction that the quantity in storage cannot be negative and that the warehouse can

hold a maximum of 2 items, i.e. $0 \leq x_k \leq 2$, and any extra stock is simply lost. The update rule is therefore:

$$x_{k+1} = \max\{0, \min\{2, x_k - w_k + u_k\}\} = f_k(x_k, u_k, w_k). \tag{4.2}$$

The customer demand is not known beforehand, and is assumed to follow a probability distribution

$$P_k(w_k = 0) = \frac{1}{10}, \quad P_k(w_k = 1) = \frac{7}{10}, \quad P_k(w_k = 2) = \frac{1}{5}. \tag{4.3}$$

The cost function has to balance the cost of ordering an item, but also what occurs if we have unmet demand or a full inventory (we assume the warehouse has limited space). One (albeit fairly crude) example is:

$$\sum_{k=0}^{N-1} \left(u_k + (x_k + u_k - w_k)^2\right) \tag{4.4}$$

The cost function assumes ordering a single $u_k$ cost 1 unit, and penalizes unmet demand/excess stock quadratically.

In this example, the agent should look at the inventory $x_k$, and decide what quantity of the item to buy $u_k$. One rule for doing so could be to buy one item when the inventory is zero:

$$u_k = \begin{cases} 1 & \text{if } x_k = 0 \\ 0 & \text{otherwise} \end{cases}.$$

More generally, our job is to determine a function which tells us which actions to take based on the observation $x_k$:

$$u_k = \mu_k(x_k) \tag{4.5}$$

POLICY

where $\mu_k$ is what the agent does (we call this the **policy**), so as to minimize the (expected) cost as defined in eq. (4.4). Note that in this problem, it would be very helpful to decide $u_k$ based on $w_k$ (the demand), but this quantity is not known when we decide on $u_k$.

### 4.2.4 Example: Gridworlds★

GRIDWORLD

A **gridworld** (see fig. 4.5) consist of a set of tiles, and an agent which is restricted to move between the tiles. Typically, the agent (here, the blue dot) has four available actions (move `north`, `east`, `south`, `west`) and the result of taking an action is simply to move one step in the desired direction. Certain tiles give the agent a reward, and may signify the end of the episode. In the example given above there are two **goal states**, one with a reward of +1 and the other with a reward of -1. In our gridworld example, the goal states are handled as follows:

GOAL STATES

- The tiles indicated by the numbers 1 and $-1$ function as exit tiles

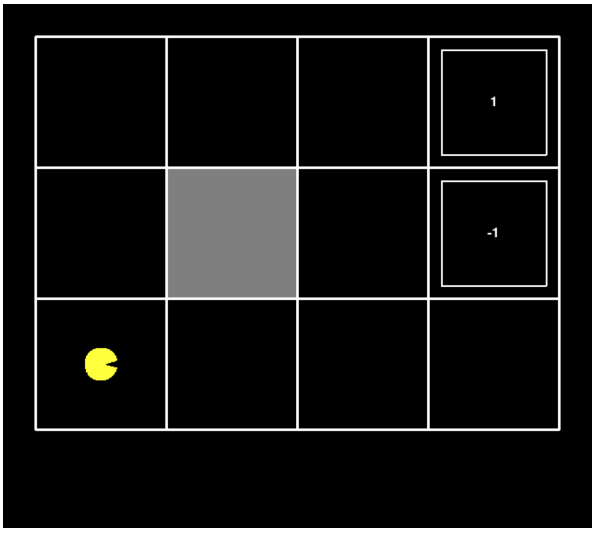


Figure 4.5: Example gridworld environment. The agent may move north, east, south or west, and dependent on which of the two goal tiles it reaches it obtains a reward of +1 or -1 for the episode.

- When the agent lands on an exit tile, it only has access to a single action (`south`)
- When the agent takes this action, the agent obtains the terminal reward (1 or $-1$ in our case), and the environment terminates.

This choice may seem slightly confusing, but it is common in gridworld implementations for visualizations purposes.

When the gridworld is deterministic, i.e. when we can be certain what the outcome of the actions are, the gridworld is equivalent to finding the shortest path in a certain graph. However, gridworlds are often studied in reinforcement learning in which the outcome of the actions are not known, and the dynamics (i.e., result of an action) may be stochastic. Gridworlds themselves have a limited practical applicability, but they are popular because it is visually very easy to visualize what the agent does.

### 4.2.5 Example: Pacman

PACMAN

Various variants of the game **Pacman** (see fig. 4.6) will be a running example throughout these notes. Pacman can move to adjacent square not blocked by a wall, and the goal is to eat the food pellets while avoid being eaten by a ghost. In some levels there are large food pellets which gives Pacman a temporary superpower to chase down and eat the ghosts.

In Pacman, the available actions are `north`, `east`, `south`, `west`, and also `stop`, however an action is only available assuming the square is not blocked by a wall. The state consist of the location of Pacman, the ghosts, the food pellets and the super-pellets. A natural reward function could be the score, although we will often consider a simpler reward based on whether Pacman wins (+1) or not (+0).

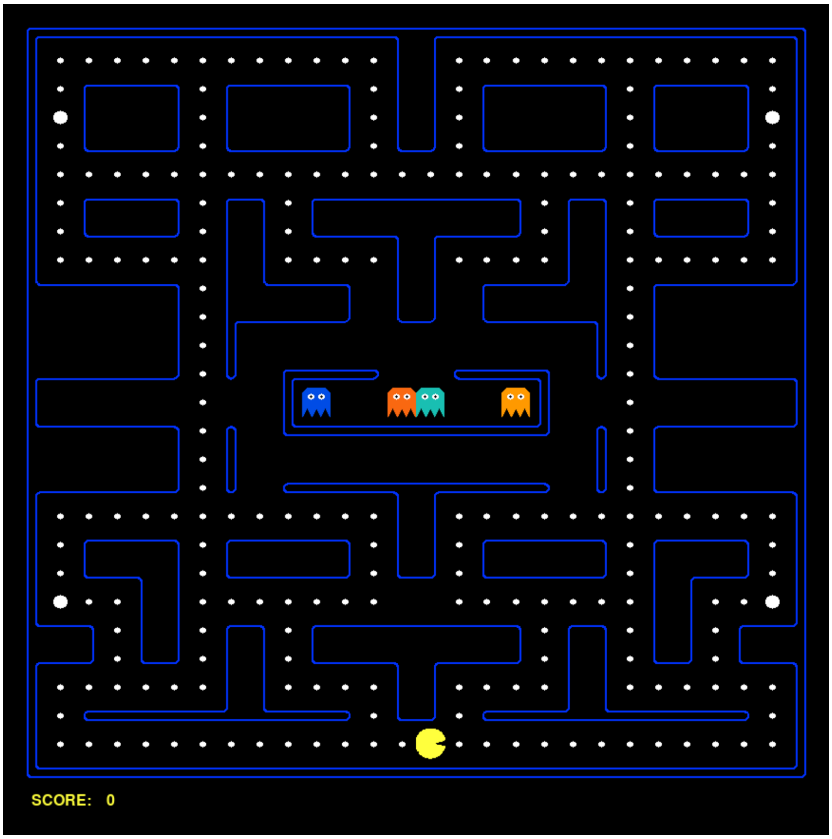


Figure 4.6: Instance of the Pacman game. Note we will often consider greatly simplified game levels.

Pacman shares similarities with all the other problems in that this too is a kind of minimization problem, and from Pacmans perspective it resembles the inventory control problem in that the ghosts movement is not deterministic; however, the game differs from the inventory control problem in that we can view it as a planning problem *from the perspective of the ghost* in which the goal is to eat Pacman; this feature makes it a **multi-agent problem** and therefore actually quite reminiscent of chess and other competitive board games. We will return to this perspective in chapter 9.

MULTI-AGENT PROBLEM

## 4.3 Detailing the decision problem

Based on the preceding examples, we can begin to introduce some common notation for each of the three components of the decision problem.

### 4.3.1 The environment

The environment contains the dynamical rules, i.e. laws of nature, of the problem, and it might also contain constraints (such as in the pendulum example). The environment

has a state which we will commonly denote by $x_k$. In case of the pendulum the state was the position/velocity and the dynamics was an ODE, and in the case of the inventory control the state was an integer and the dynamics was of the form eq. (4.2)

$$x_{k+1} = f_k(x_k, u_k, w_k) \tag{4.6}$$

where $w_k$ is a noise term and $u_k$ is the applied control. The available actions is called the **action space** and the states the environment can be in the **state space**.

ACTION SPACE
STATE SPACE

When there is a noise term the outcome of each action is not known exactly and the environment is called **stochastic**. The alternative case is called a **deterministic** environment, and it is obtained when there is no noise term (or equivalently, where the noise term can only take a single value). In this case the dynamics simplifies to

STOCHASTIC
DETERMINISTIC

$$x_{k+1} = f_k(x_k, u_k) \tag{4.7}$$

In eq. (4.6) the time index is discrete, whereas in the pendulum example it is continuous. The former case is called a **discrete time problem** and the later a **Continuous time problem**. Continuous problems are usually discretized and solved, however the extra step of discretization presents us with various choices and complications. Control theory *usually* treat continuous problems, and reinforcement learning is *usually* focused on discrete.

DISCRETE TIME PROBLEM
CONTINUOUS TIME PROBLEM

The task we consider might terminate or it might not. A problem which eventually terminates is called **episodic**. In this case we imagine the problem is reset after each **episode**. The alternative case is when we the problem is imagined to continue with no time limit and in that case we call the problem **non-episodic**. The first section of this course will focus on the terminating case.

EPISODIC
EPISODE
NON-EPISODIC

Finally, the environment might be **fully observed**. A fully observed environment is one where the agent has access to $x_k$, however in many cases we only observe a function of $x_k$ which can be written as $o_k(x_k)$. The **partially observed** case is obviously more difficult to solve, but as we will see it will in practice either be treated as a special case of the fully observed, or the problem may alternatively simply be ignored.

FULLY OBSERVED
PARTIALLY OBSERVED

### 4.3.2 The agent

The role of the agent is to take actions to minimize the cost. In other words the agent defines a **policy**

POLICY

$$u_k = \mu_k(x_k)$$

In continuous time, we could write this as $u = \pi_t(x)$. As indicated, in the most general case the policy is a function of *both* the state and time. A useful classification for how the agent can obtain a policy is as:

- **Planning** Where the agent know enough about the environment to plan a policy beforehand

PLANNING

- **Learning** Where the agent must arrive at a good policy through interaction with the environment

LEARNING

Control theory has traditionally been concerned with planning, i.e. finding an optimal behavior, whereas reinforcement learning focus on learning since the environment is typically assumed to be unknown.

### 4.3.3 The interpreter

The interpreter computes the cost function, which is a function of the action and state of the environment. As a practical matter, it is computed by the robot, however from the perspective of our control method the computation of the cost is external.

In the discrete case, we assume the reward/cost arrives at each time step, and takes the additive form for a trajectory of length $N$:

$$g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k).$$

COST HYPOTHESIS

Underlying this general formulation is the **cost hypothesis**

**Definition 4.3.1** (Cost hypothesis)**.** *The desired behavior of the agent can be specified as minimizing a cost function.*

For many, this assumption does not feel entirely correct (what about curiosity, intuition, artistry etc.?), but it has so far proved difficult to replace it with anything else.

### 4.3.4 The control loop

CONTROL LOOP
WORLD LOOP

The interaction between the agent, environment and interpreter is called the **control loop** (or **world loop**), and it is worth being specific about what it exactly consist of. In the following it is illustrated in the discrete setting

- Assume at time $k = 0$ the environment is in state $x_0$. Starting at $k = 0$ do
  - The agents policy is queried to compute a control signal
    $$u_k = \mu_k(x_k)$$
  - The control signal is fed into the environment which, unbeknown to the agent, computes a noise disturbance $w_k \sim P_k(W_k|x_k, u_k)$ and a new state:
    $$x_{k+1} = f_k(x_k, u_k, w_k)$$
  - The interpreter computes a cost $c_k = g_k(x_k, u_k, w_k)$
  - The cost and next state is fed back into the agent and the agent may change its policy found in the state $x_k$, the action taken $u_k$, the state it arrived in $x_{k+1}$, and the obtained cost $c_k$

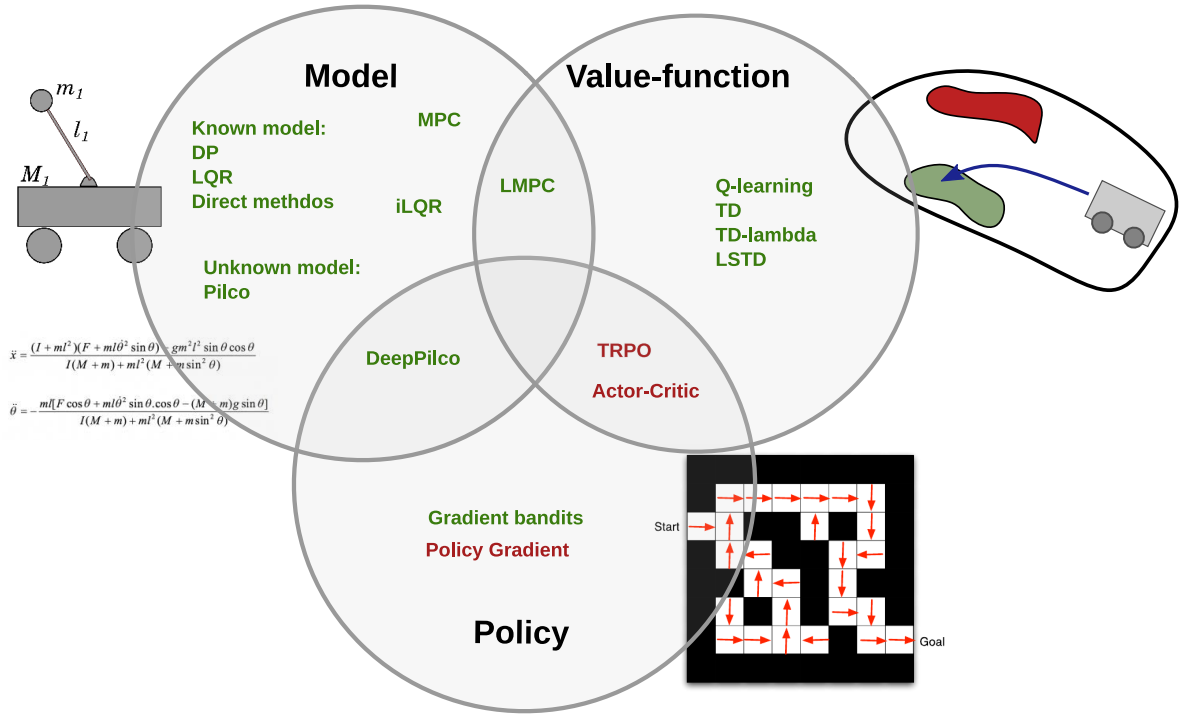


Figure 4.7: A coarse classification of methods used to solve the decision problem based on what the methods primarly choose to store/manipulate in memory

If the environment terminates at time step $N$ we call the states and actions taken

$$(x_0, u_0), (x_1, u_1), (x_2, u_2), \ldots, (x_{N-1}, u_{N-1}), x_N \tag{4.8}$$

TRAJECTORY

the **trajectory**[4]. Note the trajectory does not include a $u_N$ since no action is computed after the environment terminates. For terminating environments, it is common to immediately re-start the world-loop when the environment terminates, thereby letting the agent learn from multiple trajectories/episodes.

### 4.3.5 How to build an agent

The various methods for solving the decision problem can loosely be thought of as falling within three categories. The categories are based on what the agent choose to (primarily) represent in it's internal memory, see fig. 4.7, and the classification comes with the proviso a given method may use ideas from more than one category.

MODEL-BASED

- **Model-based** The agent stores a complete or approximate model of the environment and use it to plan

VALUE-FUNCTION BASED

- **Value-function based** The agent assigns to each state $x_k$ a value (the expected reward) which signifies if it is good or bad, and steers towards those states with a high value

[4]The trajectory may also include the cost

POLICY-BASED

- **Policy-based** The agent directly represents it's policy internally and manipulates it based on whether it did well or less well in terms of the obtained reward

Control theory is traditionally associated with model-based methods, reinforcement learning with value-function based methods. The description of policy-based methods is quite vague and will not be elaborated upon here since we will only tangentially touch upon policy-based methods in this course.

## 4.4 Implementing environments and agents

The promise of reinforcement learning and control is that we should be able to program agents who can control arbitrary environments. A major goal for the software in this course is to make good on this promise, and the way this can be achieved in practice is to ensure the components of the software (the agent, the environment, and so on) have clearly defined roles.

This in turn means the programs we write will all derive from a few, key abstractions; initially this may feel constraining, as it will require us to think in a certain way, however in the long run it will save us a lot tedious code, and ensure that control and reinforcement learning algorithms can be bench-marked on the same problems. This section will itemize these basic components from a user-perspective.

### 4.4.1 Building a robot

If we take an implementation-oriented approach to the control loop, and consider what it involves for a practical robot, there are three major components

**Environment** The environment has an internal state and must implement functionality to compute the result of actions.

**The Agent** The program we implement; must implement functionality for taking actions and learning, and must also maintain an internal state

**The OS** Synthesize sensory data to produce input for the Agent, and ensure the actions from the agent are passed to the environment; should also handle logging for later diagnostics

Obviously, real-world robotics involve a great deal of engineering, such as how to treat a continuous stream of inputs. In this course we will only deal with simulated environments, but assuming the environment was hooked up to a real robot our methods should generalize. The interaction between agent and environment can be illustrated with the following example:

```
1    # chapter1/notes_chapter1.py
2    from irlc.pacman.pacman_environment import PacmanEnvironment
3    from irlc.ex11.q_agent import QAgent
4    env = PacmanEnvironment()
```

```
    agent = QAgent(env)
    train(env, agent, num_episodes=1000)
    env.close()
```

The example illustrates all three components of our learning setup:

- Instantiate the environment, in this case corresponding to a pacman game
- Instantiate a $Q$-learning agent, which can learn to solve the environment
- Train the agent on the environment for 1000 episodes (an episode begins in the start configuration and ends when pacman has either eaten all the dots or been eaten by a ghost), and the environment is re-set after each episode.

The result of this will be a trained agent which can play Pacman reasonably well, i.e. obtain a high reward. In the following we will illustrate how each of these components work with a practical example.

### 4.4.2 The environment

OpenAI Gym

The environments are all build on **OpenAI Gym**. Openai Gym is a reinforcement learning framework `https://gym.openai.com/` and provides the most well-established specification of environments, which means there are many open-source environments in existence build using OpenAIs framework. Asides the availability of many alternative reinforcement-learning environments, OpenAIs framework is well-documented and very simple[5].

As an example, we will implement the inventory control environment we considered earlier. The implementation is as follows:

```
# inventory_environment.py
class InventoryEnvironment(Env):
    def __init__(self, N=2):
        self.N = N                                  # planning horizon
        self.action_space      = Discrete(3)        # Possible actions {0, 1, 2}
        self.observation_space = Discrete(3)        # Possible observations {0, 1, 2}

    def reset(self):
        self.s = 0                                  # reset initial state x0=0
        self.k = 0                                  # reset time step k=0
        return self.s, {}                           # Return the state we reset to (and an empty dict)

    def step(self, a):
        w = np.random.choice(3, p=(.1, .7, .2))    # Generate random disturbance
        s_next = max(0, min(2, self.s-w+a))           # next state; x_{k+1} =  f_k(x_k, u_k, w_k)
        reward = -(a + (self.s + a - w)**2)           # reward = -cost      = -g_k(x_k, u_k, w_k)
        terminated = self.k == self.N-1               # Have we terminated? (i.e. is k==N-1)
        self.s = s_next                               # update environment state
        self.k += 1                                   # update current time step
        return s_next, reward, terminated, False, {}  # return transition information
```

[5]for instance, here: `https://github.com/openai/gym`, but also use shift-click e.g. pycharm to jump to source code definition or similar functionality if you are using another IDE.

### The init-function

The init function defines the `action_space` and `observation_space` attributes. These corresponds to sets $\{0, 1, 2\}$. They are defined using a special class (`Discrete` which may be imported from `gym.spaces.discrete`), which represent a discrete set of integers. The reason we define the action and observation spaces using special gym-provided classes is because it allows our agents to work with *any* discrete space.

### The reset-function

The reset function resets the environment's internal state to the initial value and return it. The reset function must be called before each episode. In this case we set $k = 0$ and $x_0 = 0$ (or rather, `self.s=0` since gym uses the reinforcement-learning convention).

### The step-function

The step function takes an action as input, and computes one step (or update) of the internal state of the environment. The action is in our case an integer (defined by the action-space).

It returns four parameters: The next state, the obtained reward (in this step), whether the environment is done or not (a boolean) and finally a dictionary which may contain additional information. That we have to return a reward is annoying given we earlier talked about cost, but we follow the convention that `reward = -cost` whenever the issue arises.

## 4.4.3 The agent

The second component of our code is the agent. Since we haven't learned how to build an agent, let us just make an agent which generates random actions (i.e. random numbers from the set $\{0, 1, 2\}$) and refuses to learn anything. This agent can be defined as:

```
# inventory_environment.py
class RandomAgent(Agent):
    def pi(self, s, k, info=None):
        """ Return action to take in state s at time step k """
        return np.random.choice(3) # Return a random action
```

### The policy function

The policy is implemented in the function `pi`, which asks the agent what it wants to do in state $s$ at time step $k$. In other words, `pi(s,k)` $= \mu_k(s)$ in our notation from before.

### The training function

The input arguments to the training method consist of the current state, what action the agent took in that state, what reward the agent obtained `r`, the next state[6] the agent transitioned to `sp` and the obtained reward `r`. This corresponds to the following four pieces of information

$$x_k, u_k, g_k(x_k, u_k, w_k), x_{k+1}.$$

The agent may do anything with this information in order to learn a better policy.

## 4.4.4 The training loop

The final component is the training loop, which lets the agent and environment interact with each other. This functionality in collected in a single function `train`. Using this function ensures all training elapse in the same way, and that statistics of the training is collected in a unified manner. To let our agent interact with the environment for one rollout (i.e., $N$ time steps) and print the accumulated reward, we would do:

```
    # inventory_environment.py
    env = InventoryEnvironment()
    agent = RandomAgent(env)
    stats, _ = train(env,agent,num_episodes=1,verbose=False)  # Perform one rollout.
    print("Accumulated reward of first episode", stats[0]['Accumulated Reward'])
```

The reader is invited to look at the code for the practicalities. The main part of the code is the generation of a single rollout, which implements the world loop as described in section 4.3.4, and which can be sketched as:[7]

```
# inventory_environment.py
def simplified_train(env: Env, agent: Agent) -> float:
    s, _ = env.reset()
    J = 0  # Accumulated reward for this rollout
    for k in range(1000):
        a = agent.pi(s, k)
        sp, r, terminated, truncated, metadata = env.step(a)
        agent.train(s, a, sp, r, terminated)
        s = sp
        J += r
        if terminated or truncated:
            break
    return J
```

In other words, the training loop first resets the environment, then feed the first state `s` into the agents policy to produce the first action `a`. This is fed into the step function to obtain the next state `sp` and reward `r`. The agent is trained on this information and we iterate until the trajectory terminates when `done = True`.

[6] It is standard notation to denote the state which follows a given state $s$ as $s^+$; hence the shorthand `sp` for $s$-plus

[7] Naturally, we don't hardcode a maximum number of steps of 1000 in the code, so this is just for illustrations sake

This code computes a single trajectory of length $N$. To estimate the average cost of a given policy we must do this $T$ times and average:

$$\sum_{t=1}^{T} \frac{\{\text{Cost obtained in trajectory } t\}}{T}$$

In practice this is easily accomplished using the train function

```
    # inventory_environment.py
    stats, _ = train(env, agent, num_episodes=1000,verbose=False)  # do 1000 rollouts
    avg_reward = np.mean([stat['Accumulated Reward'] for stat in stats])
    print("[RandomAgent class] Average cost of random policy J_pi_random(0)=", -avg_reward)
```

which will give us an estimate of the cost of a random policy:

```
<<<<<<< HEAD
[RandomAgent class] Average cost of random policy J_pi_random(0)= 4.286
=======
[RandomAgent class] Average cost of random policy J_pi_random(0)= 4.331
>>>>>>> e99c41cdf0c4fe17e48e63e8a332a5aefe6ae9a3
```

The advantage of using the environment, agent and train-functionality is that we get a high degree of reuseability. For instance, we could have saved ourselves a little typing by using that the `Agent` class by default use random actions:

```
    # inventory_environment.py
    stats, _ = train(env, Agent(env), num_episodes=1000,verbose=False)  # Perform 1000 rollouts using Agent class
    avg_reward = np.mean([stat['Accumulated Reward'] for stat in stats])
    print("[Agent class] Average cost of random policy J_pi_random(0)=", -avg_reward)
```

It also allows us to use other learners on the same environments. For instance, we can train a $Q$-learning agent and estimate it's cost:

```
    # chapter3dp/inventory_environment.py
    from irlc.ex11.q_agent import QAgent
    stats, _ = train(env, QAgent(env), num_episodes=1000, verbose=False)  # Perform 1000 rollouts using Agent class
    avg_reward = np.mean([stat['Accumulated Reward'] for stat in stats])
    print("Average cost obtained by Q-learning agent J_Q(0)=", -avg_reward)
```

As expected, the estimated cost of the $Q$-learner is lower than the random agent:

```
Average cost obtained by Q-learning agent J_Q(0)= 2.846
```

### 4.4.5 Advanced features, plotting★

If this was *all* the training loop did there would be little point to it, however it is also use it to collect statistics (and save them for later plotting) as well as managing repeated experiments. Let us consider one such example. The following code trains a $Q$-learning algorithm for 100 episodes and plot the reward obtain in each episode, and the output can be seen in fig. 4.8 and is called a **trace plot**

TRACE PLOT

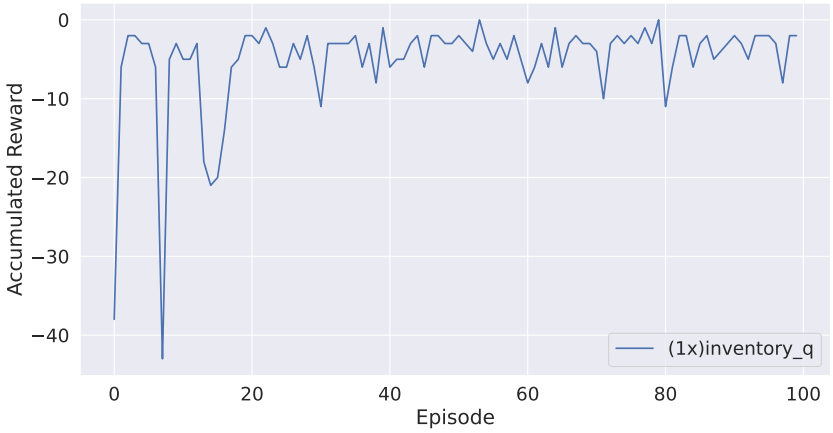


Figure 4.8: Trace plot of the reward obtained by a $Q$-learning agent in the inventory control environment trained over 100 episodes. We see the learner generally improves on the task, but with quite a lot of variability between runs due to the stochasticity of the environment

```
# chapter1/notes_chapter1.py
from irlc import main_plot
env = InventoryEnvironment()
train(env, QAgent(env), num_episodes=100, experiment_name="inventory_q", max_runs=1)
main_plot("inventory_q")
```

As we see, there is quite a bit of variability between each episode, and so in most cases it is useful to re-train the method many times and plot the average. This can very easily be done by simply calling the training method multiple times:

```
# chapter1/notes_chapter1.py
from irlc import main_plot
for i in range(20):
    env = InventoryEnvironment()
    train(env, QAgent(env), num_episodes=100, experiment_name="inventory_q_multiple", max_runs=20)
main_plot("inventory_q_multiple")
```

and the result can be seen in fig. 4.9.

### 4.4.6 Visualizing the environment★

Often, it is informative to see what the environment does as the agent interacts with it. Many environments come with a build-in rendering functionality which can be enabled by passing `render_mode='human'` to the environment, and the environment will then be animated. As an example:

```
# chapter1/notes_chapter1.py
env = PacmanEnvironment(render_mode='human')
agent = Agent(env)
train(env, agent, num_episodes=1)
```

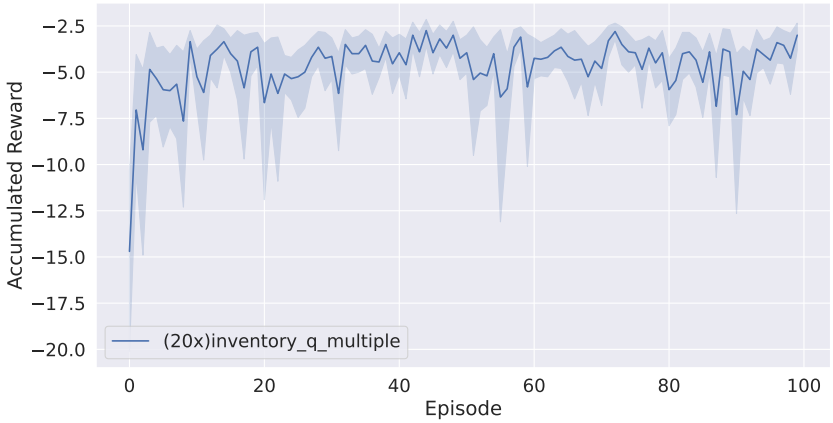


Figure 4.9: Similar to the trace plot of the $Q$-learner in the inventory control environment from fig. 4.8, however the experiment is repeated 20 times and the average reward is shown. We see a more smooth learning curve. The shaded area represents a naive 95% confidence interval of the standard deviation of the mean.

The only documentation contains examples of how you can manually input actions using the keyboard, however, since that does not translate well to written documentation we will simply take a snapshot using the `savepdf` function:

```
# chapter1/notes_chapter1.py
from irlc import savepdf
env = PacmanEnvironment()
env.reset()
savepdf("pacman_chapter1.pdf", env=env)
```

The result can be found in fig. 4.10

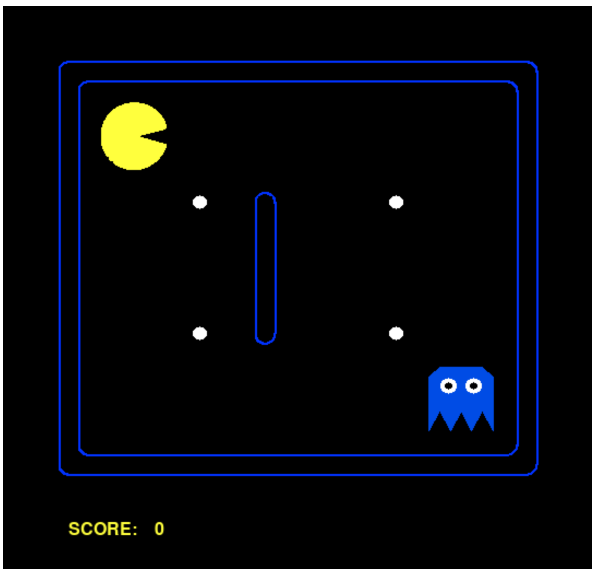


Figure 4.10: Example snapshot of the Pacman environment captured using the toolbox code.

# Chapter 5

# The basic problem

This chapter will provide a more concrete introduction to the basic decision problem. We focus on an idealized situation where the problems are made in discrete time steps. This setup allows us to avoid the technical annoyances of continuous time, and allows a very general problem formulation and optimal solutions. Our second assumption will be that we solve the problems over a **finite horizon**, which means that the problem terminates after a specific number of planning stages. The benefit of this assumption is that it avoids infinities and greatly simplifies the presentation.

FINITE HORIZON

## 5.1 The discrete, finite-horizon decision problem

Our problem setup has three features:

1. An underlying discrete-time dynamic system,
2. A cost function that is additive over time and
3. A finite, known **planning horizon** $N$.

PLANNING HORIZON

Asides these assumptions we will keep everything as general as possible: The dynamics can change over time and it may be stochastic, and the states and actions may be vectors, integers, or something else.

The **system dynamics** describes how the systems state is updated as a consequences of decisions made at discrete instances of time, also called **stages**. The dynamics of the system has the form

SYSTEM DYNAMICS
STAGES

$$x_{k+1} = f_k\left(x_k, u_k, w_k\right), \quad k = 0, 1, \ldots, N-1 \tag{5.1}$$

Fundamentally, the interaction occurs over $N$ stages, and the system terminates in stage $N$. It is assumed $N$ has a fixed value given beforehand. From this we can make two observations

- We must compute $N$ actions $u_0, \ldots, u_{N-1}$

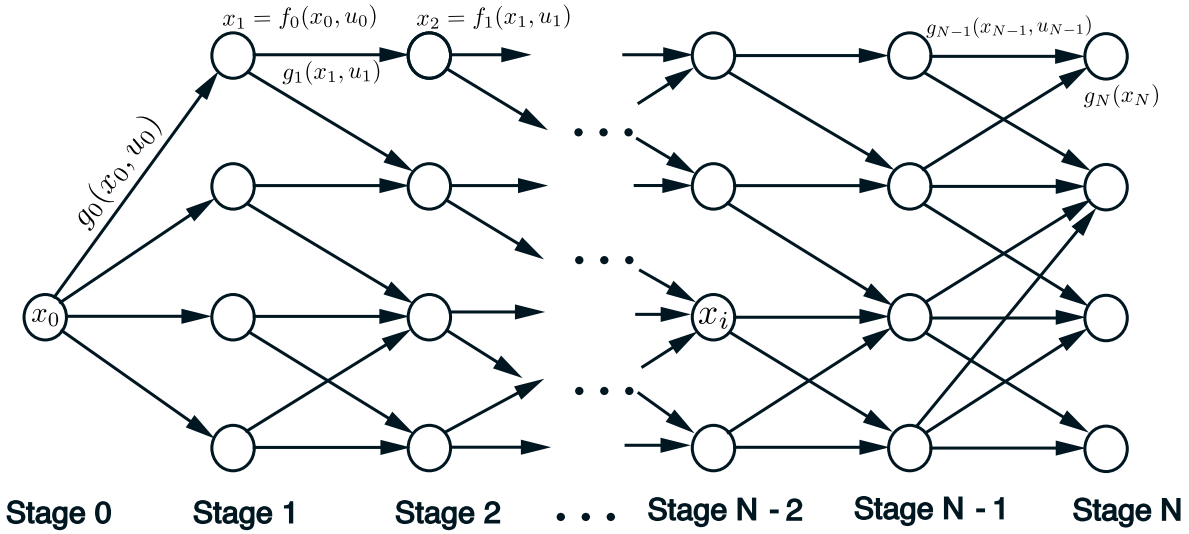


Figure 5.1: A graphical illustration of the paths the planning problem can take starting at stage 0 and ending at $N$. The decision problem can be viewed as finding the path which gives the shortest cost, with the proviso that when we take an action, we may not be guaranteed which path to follow.

- There is a total of $N + 1$ states of the system $x_0, \ldots, x_N$

TRAJECTORY ROLLOUT

The $x_k$'s and $u_k$'s resulting from following the system dynamics is called a **trajectory** or **rollout**. To explain the rest of the terms one at a time:

- $k$ stage number
- $x_k$ The state of the system
- $u_k$ The control selected at stage $k$ and applied to the environment

RANDOM DISTURBANCE

- $w_k$ A random parameter (sometimes also called noise or **random disturbance**). It allows us to describe that the new state of the environment $x_{k+1}$ may not be fully determined by $x_k$ and $u_k$

STATE SPACE

The states the system can be in at step $k$ is called the **state space** written as $x_k \in \mathcal{S}_k$. Similarly, the available actions, at planning stage $k$ and given the agent is in state $x_k$,

ACTION SPACE

is called the **action space** and is denoted $\mathcal{A}_k(x_k)$. Both of these are sets.

The noise terms will be assumed to follow a probability distribution

$$w_k \sim P_k\left(W_k | x_k, u_k\right).$$

This means that the noise can depend on the stage $k$, as well as the current state $x_k$ and action taken $u_k$. This means the noise term is generated after the agent has carried out it's action $u_k$ in state $x_k$.

**The policy**

ADMISSIBLE POLICY

At each stage $k$, the policy (or an **admissible policy**) is a function $\mu_k$ which maps each possible state $x_k \in \mathcal{S}_k$ into a possible control $u_k = \mu_k(s_k)$ such that $\mu_k(x_k) \in \mathcal{A}_k(x_k)$. The full policy is a sequence of $N$ such functions

$$\pi = (\mu_0, \mu_1, \ldots, \mu_{N-1}) \tag{5.2}$$

**The cost**

The cost is assumed to be additive. At each time step $k$ we incur a cost of $g_k(x_k, u_k, w_k)$. The total cost of a particular trajectory becomes

$$g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \tag{5.3}$$

where $g_N(x_N)$ is a special cost function which only depends on the final state. However, because of the presence of the noise terms $w_k$, the cost is generally a random variable.

EXPECTED COST

We therefore formulate the problem as an optimization of the **expected cost**

$$\mathbb{E}\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k) \right\} \tag{5.4}$$

where the expectation is over the noise terms $w_0, \ldots, w_{N-1}$.

This formulation requires that we specify the action sequence $u_0, \ldots, u_{N-1}$ beforehand. But in the situation we are concerned with the actions will be computed using the policy. We therefore define our first fundamental quantity namely the **cost of a policy** $\pi$, by assuming each action is taken using the rule $u_k = \mu_k(x_k)$

COST OF A POLICY

$$J_\pi(x_0) = \mathbb{E}\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right\}. \tag{5.5}$$

This quantity depends on the initial state $x_0$ and policy $\pi$. This quantity is fundamental in control and reinforcement learning, and in the later case it is typically called the **value function**.

VALUE FUNCTION

### 5.1.1 Small graph traversal

Continuing the graph-traversal example section 4.2.2, suppose the graph $G$ has $n$ nodes and the goal is to traverse from node $s$ to node $t$ along the edges $(i, j)$ in exactly $N$ steps along the path with the least weight $a_{ij}$.

In this case the states are the set of nodes $\mathcal{S} = \{1, 2, \ldots, n\}$, and the actions are the nodes connected to a state. I.e. if edge $(i, j) \in G$ then $j$ is one of the available action

from state $i$. The dynamics and cost is then:

$$f_k(x_k = i, u_k = j, w_k) = j \tag{5.6}$$

$$g_k(x_k = i, u_k = j, w_k) = a_{ij} \tag{5.7}$$

$$g_N(x_N) = \begin{cases} 0 & \text{if } x_N = t \\ \infty & \text{otherwise.} \end{cases} \tag{5.8}$$

In this case the disturbances are irrelevant and can be ignored. Note the problem does not specify the shortest path, but rather the shortest $N$-edge path. If we want the shortest path, we have to modify the problem to make sure the terminal node is absorbing, i.e. there is an edge of the form $(t, t) \in G$ with cost $a_{tt} = 0$, see section 5.2.1. We will return to the problem of search in graphs in much greater details in chapter 7 and chapter 8.

### 5.1.2 Inventory control example

Let us return to the inventory control problem from the introduction. Let's assume the number of periods for planning is $N = 2$. The observation and action spaces are:

$$S_k = \{0, 1, 2\}, \quad \mathcal{A}_k = \{0, 1, 2\}. \tag{5.9}$$

And the dynamics and cost-functions as:

$$f_k(x_k, u_k, w_k) = \max\{0, \min\{2, x_k - w_k + u_k\}\} \tag{5.10}$$

We already defined the probability distribution in the problem, however in our new notation:

$$P_k(w_k = 0|x_k, u_k) = \frac{1}{10}, \quad P_k(w_k = 1|x_k, u_k) = \frac{7}{10}, \quad P_k(w_k = 2|x_k, u_k) = \frac{1}{5}. \tag{5.11}$$

and the cost-function is:

$$g_k(x_k, u_k, w_k) = u_k + (x_k + u_k - w_k)^2, \quad q_N(x_N) = 0 \tag{5.12}$$

In our case we will consider the case where $N = 2$. In this case we can write out the cost function eq. (5.5) as:

$$J_\pi(x_0) = \mathbb{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\right] \tag{5.13}$$

$$= \mathbb{E}_{w_0, w_1}\left[\mu_0(x_0) + (x_0 + \mu_0(x_0) - w_0)^2 + \mu_1(x_1) + (x_1 + \mu_1(x_1) - w_1)^2\right] \tag{5.14}$$

If we substitute $x_1 = f_0(x_0, \mu_0(x_0), w_0)$ into this expression it becomes a simple expectation over $w_0$ and $w_1$. We could therefore evaluate the cost of the policy by computing the expression in the parenthesis for all $9 = 3^{N-1}$ combinations of $w_0$ and $w_1$, and if we do this we obtain the true cost of a policy. Note this approach is not very practical, since the number of terms we have to evaluate grows exponentially in $N - 1$.

### Optimal policy

The equation eq. (5.5) denotes the average cost of a policy, i.e. if you take a fixed policy $\pi$, initialize it in $x_0$, and allow the system to evolve to step $N$ this will be the expected reward. If we consider the initial state $x_0$ as fixed, we can consider it as a function of $\pi$.

OPTIMAL COST FUNCTION

This allows us to define the **optimal cost function**, denoted by $J^*(x_0)$, by minimizing eq. (5.5) over all admissible policies,

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0) \tag{5.15}$$

This function map each initial state $x_0$ to an optimal cost $J^*(x_0)$. This allows us to define the goal of control and reinforcement learning, namely to find a policy which achieves this minimum

$$\pi^* = \arg\min_{\pi \in \Pi} J_\pi(x_0) \tag{5.16}$$

In this optimization, the set $\Pi$ denote the set of all admissible policies (i.e., where $\mu_k(x_k) \in \mathcal{A}_k(x_k)$). We can at this point make a few observations, although especially the last may be more obvious when we consider methods for solving the minimization problem in eq. (5.16)

- The minimization task in eq. (5.16) considers a fixed $x_0$, and therefore the policy at the first stage $\mu_0$ is arbitrary for other states in $\mathcal{S}_0$.
- The policy which minimize the expression is not unique; this is both a consequence of the above point, and the simple fact several points may minimize a function. For instance, $x \in \{2\pi n\}_n$ minimize $\sin(x)$
- We should worry about the following: the minimizing policy $\pi^*$ starting in $x_0$, and the minimizing policy $\pi^{*'}$ starting in another state $x_0'$, they might disagree about which action to take later. I.e., if we follow $\pi^*$ to arrive at $x_k$ for some $k$, then it could be the case that following $\pi^{*'}$ from $x_k$ to $x_N$ would be suboptimal. If this was the case it would rule out any notion of a single optimal policy as what a policy did substantially depends on the initial state $x_0$. As it turns out, this is not a problem, however the proof is the optimality of the dynamical programming algorithm itself

## 5.1.3 Example: The chessmatch

Suppose a person (player $A$) is about to play chess match with an opponent (player $B$). Each game in the match can have three outcomes

- Player $A$ wins (and get one point)

- Player $B$ wins (and get one point)
- They draw (and both get $\frac{1}{2}$ points)

The players first play two rounds, and if the match is tied they enter sudden death mode, where they keep playing until one wins a game and thereby the match.

We can turn this into a decision problem by allowing player $A$ to choose between two styles of play in each game:

**Timid play,** $u = 0$ In which player $A$ draws with probability $p_d$ and looses with probability $1 - p_d$

**Bold play,** $u = 1$ In which player $A$ wins with probability $p_w$ and looses with probability $1 - p_w$

Player $A$ can never win using timid play, so if the match enters sudden-death mode the optimal strategy is to always play bold, $u = 1$. The problem can be formulated as a basic decision problem as follows:

- $N = 2$ is the number of games in the ordinary match.
- $x_k = (a_k, b_k)$ is the score of the two players
- $u_k \in \{0, 1\} = \mathcal{A}_k$ denotes timid/bold play respectively
- The dynamics can be selected as

$$f_k(x_k, u_k, w_k) = w_k$$

- $w_k$ contains the rules of the match, such that if for instance we play timid $u_k = 0$ then

$$P_k(w_k = (a_k + 0.5, b_k + 0.5) \mid x_k = (a_k, b_k), u_k = 0) = p_d \tag{5.17}$$
$$P_k(w_k = (a_k, b_k + 1) \mid x_k = (a_k, b_k), u_k = 0) = 1 - p_d \tag{5.18}$$

If the match is drawn after two games, $x_2 = (1, 1)$, it enters sudden-death mode, and as noted player $A$ will always play bold, and therefore the chance of winning sudden-death mode is $p_w$.

Since the objective is to win, the cost can only be computed at the end of the match using $x_2$, defined as:

$$g_k(x_k, u_k, w_k) = 0 \tag{5.19}$$

$$g_N(x_k = (a_k, b_k)) = \begin{cases} -1 & \text{if } a_k > b_k \text{ and played } A \text{ wins} \\ -p_w & \text{if } a_k = b_k \text{ and the match is in sudden-death mode} \\ 0 & \text{otherwise, i.e. if player } A \text{ loose} \end{cases} \tag{5.20}$$

In this definition, $-J_\pi(x_0) = -g_N(x_N)$ will correspond to the probability player $A$ wins the match, hence when we minimize this quantity we are trying to find the bold/timid policy with the highest chance of winning the overall match.

### 5.1.4 Open and closed loop

While eq. (5.16) provides us with a well-defined problem, it is an unusual one. Ignoring the exponential cost of computing the cost function, the minimization is over *functions* $\pi$. A first question to ask is if we can ignore this, and simply minimize eq. (5.16) over action sequences $u_0, \ldots, u_{N-1}$, selected all at once at stage $k = 0$ when we only know $x_0$. This is called an **open-loop** policy since the decisions only depend on the stage index $k$, and can be defined as:

OPEN-LOOP

$$J_{(u_0,\ldots,u_{N-1})}(x_0) = \mathbb{E}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)\right] \tag{5.21}$$

$$(u_0^*, \ldots, u_{N-1}^*) = \underset{(u_0,\ldots,u_{N-1})}{\arg\min}\ J_{(u_0,\ldots,u_{N-1})}(x_0) \tag{5.22}$$

This formulation should be contrasted to our original formulation in eq. (5.16) where we minimize over *functions* $\pi$ and the result is called a **closed-loop** policy. If a problem is deterministic, any closed-loop controller can be turned into an open-loop controller since at each stage $k$, we know which state $x_k$ we will be in, and we can therefore pre-compute $u_k = \mu_k(x_k)$. However, for stochastic problems open-loop controllers will as a rule *not* be optimal, as the following example shows:

CLOSED-LOOP

**Example: Chess match, continued**

Continuing the chess-match example from section 5.1.3, and recall player $A$ has the option to play timid (which draws with probability $p_d$, and otherwise looses) or bold (which wins with probability $p_w$, and otherwise looses). We will first consider all of the open-loop policies; since there are two choices (timid/bold) and 2 rounds there are four such strategies. The chance of winning are

1. Play timid in both games: We have to draw the two games and then win the match in overtime. This occurs with probability $p_d^2 p_w$
2. Play bold in both games: We can either win both games ($p_w^2$) or win one, loose the other ($p_w(1-p_w)$) and win the final game. The chance of winning is therefore $p_w^2 + 2p_w^2(1-p_w) = p_w^2(3-2p_w)$
3. Bold first game, timid in second: $p_w p_d + p_w^2(1-p_d)$
4. Timid in first game, bold in second: Same as above.

The probability of winning under the first policy is always lower than under the two last, so therefore the maximum chance of winning under an open-loop policy is

$$\max\left(p_w^2\left(3-2p_w\right), p_w p_d + p_w^2\left(1-p_d\right)\right) = p_w^2 + p_w\left(1-p_w\right)\max\left(2p_w, p_d\right). \tag{5.23}$$

Let's compare this to the following closed-loop policy: Suppose a player choose to play timid if and only if he is ahead in score, and in all other cases play bold. We can write this as:

$$\mu_k(x_k = (a, b)) = \begin{cases} u = 0 & \text{if } a > b \\ u = 1 & \text{otherwise.} \end{cases} \tag{5.24}$$

Under this policy the player will first play bold and be ahead with probability $p_w$ and behind with probability $p_w$. In the former case he will then play timid, and in the later bold. This gives a chance of winning:

$$p_w p_d + p_w \left(p_w \left(1 - p_d\right) + \left(1 - p_w\right) p_w\right) = p_w^2 \left(2 - p_w\right) + p_w \left(1 - p_w\right) p_d \tag{5.25}$$

This probability may be larger than the chance of winning any single game. For instance, if $p_w = 0.45$ and $p_d = 0.9$ it will be approximately 0.53. On the other hand, the best open-loop policy eq. (5.23) only has a 0.425 chance of winning the match. In other words, not only is a closed-loop policy superior to the best open-loop policy, more remarkable even though the chance of winning any single game using either strategy is less than 50%, combining the different strategies gives us a chance of winning the match above 50%!.

## 5.2 State augmentation

We now turn out attention to the situation where the assumptions of the basic problem appears to be violated, but with a little care only superficially so.

### 5.2.1 Absorbing states

The first situation we consider is one in which the problem may terminate earlier than $N$. For instance, Pacman may win the game or alternatively die before some maximum time $N$ is reached. This situation is easily treated by introducing two special states

$$x_w = \texttt{win}, \quad x_d = \texttt{dead}$$

ABSORBING

then we re-write the dynamics and cost such these states are **absorbing** and have cost 0. For e.g. the winning state:

$$x_w = f_k'(x_w, u_k, w_k), \quad g_k'(x_w, u_k, w_k) = 0.$$

and finally ensure the system must transition into $x_w$ correctly:

$$x_{k+1} = \begin{cases} x_w & f_k(x_k, u_k, w_k) \text{ is a terminal (won) state} \\ f_k(x_k, u_k, w_k) & \text{otherwise} \end{cases} \tag{5.26}$$

If the cost of winning (or loosing) is associated with a large negative (or positive) cost this can be handled by giving the agent the extra cost in $g_N$. Thus, as long as we know an upper-bound on the number of stages, we can easily re-formulate the problem to be amendable to DP.

### 5.2.2 An observation about time

We explicitly model individual transition functions $f_k$ for each stage $k$, i.e. as:

$$x_{k+1} = f_k(x_k, u_k, w_k) \tag{5.27}$$

however this is strictly speaking not necessary. We could just as well have defined the states to contain $k$, i.e. a state $x_k \in \mathcal{S}_k$ could be represented as:

$$\tilde{x} = (x_k, k), \quad x \in \mathcal{S}_k$$

In which case we could use a stationary dynamics $\tilde{f}$ as:

$$\begin{aligned} \tilde{x}_{k+1} &= \tilde{f}(\tilde{x}_k, u_k, w_k) && (5.28)\\ &= (f_k(x_k, u_k w_k), k+1) && (5.29) \end{aligned}$$

Thus, using dynamics which refers to time as in $f_k$, rather than $\tilde{f}$, is a matter of how we define the state. This observation will be relevant when later consider search methods in chapter 8.

### 5.2.3 Time lags ✸

Suppose the natural formulation of the DP problem also depends on previous time steps as in

$$x_{k+1} = f'_k(x_{k-1}, x_k, u_k, w_k) \tag{5.30}$$

Or more generally

$$x_{k+1} = f_k(x_{k-d}, x_{k-d+1}, \ldots, x_{k-1}, x_k, u_k, w_k), \quad d = 1 \tag{5.31}$$

This can easily be re-cast as a DP problem by introducing an augmented state defined as

$$\tilde{x}_k = \begin{bmatrix} x_k \\ x_{k-1} \\ \vdots \\ x_{k-d+1} \\ x_{k-d} \end{bmatrix} \tag{5.32}$$

The augmented dynamics can now be formulated entirely in terms of $\tilde{x}_k$, in which it is an instance of the basic decision problem

$$\tilde{x}_{k+1} = f_k\left(\tilde{x}_k, u_k, w_k\right) = \begin{bmatrix} f'_k(x_k, u_k, w_k) \\ x_k \\ \vdots \\ x_{k-d+1} \end{bmatrix}. \tag{5.33}$$

This re-writing is very common in time-forecast models.

### 5.2.4 Partially observed environments ✸

Suppose that rather than observing the state $x_k$ directly, we only have access to indirect observations $y_k = o(x_k)$. In this case a transition function of the form $y_{k+1} = f_k(y_k, u_k, w_k)$ would not be correct, as it may be past observations $y_{k-d}, y_{k-d+1}, \ldots, y_k$ can provide additional information about $x_k$. We can still cast this as a decision problem by defining the augmented state:

$$\tilde{x}_k = \begin{bmatrix} y_k \\ y_{k-1} \\ \vdots \\ y_0 \end{bmatrix}. \tag{5.34}$$

In this case a model of the form $\tilde{x}_{k+1} = f_k(\tilde{x}_k, u_k, w_k)$ would still be correct. Obviously, in many partially observed situations we do not have access to the model, and at any rate having an ever-expanding state is often computationally prohibited. However, it is a very common trick in (deep) reinforcement learning, when applied to partially observed problems like video games, to use an approximation of this by using eq. (5.33) with for instance $d = 4$.

## 5.3 Implementation details

MODEL

The basic decision problem is our first example of a **model** of the environment. It is worth emphasizing the model is not the environment, but rather captures what the agent *knows* (or *believes*) about how the environment works:

**Environments** The *actual* (simulation) of the environment

- The environment has an internal state which changes when actions are taken
- The environment has a step-function which takes an action and changes the state
- Environments can often be derived from models (we can build an environment which simulates the model)

**Models** How the agent *thinks* the environment behaves

- Contains functions which describes how the agent thinks the environment behaves
- Do not have an internal state
- Can be accurate and complete, or inaccurate and partial

Thus, a model is part of the agent, and a variety of different models can be made to reflect aspects of the same environment. We will capture this by using separate model and environment classes; this may appear more complex than simply cramming model

information into the environment class, however, it is better software design, and it will allow us to equip the same agent with different models.

The model you write will implement different functions corresponding to each part of the basic decision problem. They will therefore all inherit from the following class:

```python
# dp_model.py
class DPModel:
    def __init__(self, N):
        self.N = N  # Store the planning horizon.

    def f(self, x, u, w, k: int):
        raise NotImplementedError("Return f_k(x,u,w)")

    def g(self, x, u, w, k: int) -> float:
        raise NotImplementedError("Return g_k(x,u,w)")

    def gN(self, x) -> float:
        raise NotImplementedError("Return g_N(x)")

    def S(self, k: int):
        raise NotImplementedError("Return state space as set S_k = {x_1, x_2, ...}")

    def A(self, x, k: int):
        raise NotImplementedError("Return action space as set A(x_k) = {u_1, u_2, ...}")

    def Pw(self, x, u, k: int):
        # Compute and return the random noise disturbances here.
        # As an example:
        return {'w_dummy': 1/3, 42: 2/3}  # P(w_k="w_dummy") = 1/3, P(w_k =42)=2/3.
```

Thus, when we implement a model of the basic decision problem, we do so by writing the above functions according to the specification of the problem. Don't worry, in most cases they will be very simple.

# Chapter 6

# Dynamical Programming

In this chapter, we will provide a method, the dynamical programming (DP) algorithm, for solving the basic decision problem optimally. Not only is this algorithm interesting because it allows us to solve a difficult decision problem, but as highlighted in the introduction it is *the* central method in this course: Indeed, the methods we will encounter for search, multi-agent games, control and reinforcement learning will be seen as simplifications and modifications to this algorithm.

## 6.1 The principle of optimality

DYNAMIC PROGRAMMING ALGORITHM
PRINCIPLE OF OPTIMALITY

The **dynamic programming algorithm** rests on a very simple idea, the **principle of optimality**. The name is due to Bellman, one of the great pioneers in artificial intelligence, who popularized DP and transformed it into a systematic tool.

The principle of optimality imbue the intuition that for a policy to be globally optimal, each part of the policy must be optimal as well. More concretely, suppose the shortest path from Copenhagen to Berlin by car takes us through Odense:

$$\text{Copenhagen} \to \text{Odense} \to \text{Berlin}$$

then the principle of optimality says that if we find the shortest path starting in Odense and going to Berlin, that path cannot be shorter than the shortest path we first had in mind. The reason should be obvious: If it was the case, then we could switch to the new path in Odense, and arrive at an even shorter path from Copenhagen to Berlin, in contradiction with the assumption we had the shortest path to begin with. The principle of optimality is the workhorse in the DP algorithm which we will use to solve the basic decision problem momentarily. We will therefore state the PO carefully since the notation will prove useful.

TAIL POLICY

The first component we need is that of a **tail policy**, which is a rigorous way of referring to the trip from Odense to Berlin above. Consider as usual a policy $\pi = (\mu_0, \dots, \mu_{N-1})$. For any for any $k = 0, \dots, N-1$, we let

$$\pi^k = \{\mu_k, \mu_{k+1}, \dots, \mu_{N-1}\} \tag{6.1}$$

TRUNCATED DECISION PROBLEM
TAIL COST

denote the tail policy. The tail policy can be thought of as acting on a **truncated decision problem** of length $N-k$, which starts in state $x_k$ and ends in stage $N$. For this tail policy we can associate a **tail cost** cost of:

$$J_{k,\pi}(x_k) = \mathbb{E}\left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} \tag{6.2}$$

And as before we can define the optimal cost of any policy on the tail subproblem as:

$$J_k^*(x_k) = \min_{\pi^k} J_{k,\pi^k}(x_k) \tag{6.3}$$

Note that $J_0^*(x_0) = J^*(x_0)$. The intuitive statement of the PO is that if we consider the (globally) optimal policy $\pi^*$, then the tail of this policy will also be optimal for the subproblem. Formally stated:

**Definition 6.1.1** (Principle of optimality)**.** *Let $\pi^* = \{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\}$ be an optimal policy for the basic decision problem, and assume that when following $\pi^*$ there is a positive probability we will find ourselves in state $x_i$ at time $i$. Consider the subproblem where we start in $x_i$ at time $i$ and which to minimize the tail subproblem from $i$ to $N$:*

$$\mathbb{E}\left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} \tag{6.4}$$

*Then the truncated policy $\pi^{*,k} = (\mu_i^*, \mu_{i+1}^*, \ldots, \mu_{N-1}^*)$ is optimal for this subproblem:*

$$J_k^*(x_k) = J_{k,\pi^k}(x_k) \tag{6.5}$$

Exactly as before, if $\pi^*$ is the optimal route from Copenhagen to Berlin, then $J_{k,\pi^{*,k}}(x_k)$ is distance that route travels from Odense to Berlin, and this distance will be the shortest between Odense and Berlin $J_k^*(x_k)$. The proof of the PO is somewhat technical, and an interested reader can find it in [Ber05].

## 6.2 The DP algorithm

The principle of optimality suggests the following idea: Suppose we already knew the optimal tail policies at stage $k$ for all states $x_k \in \mathcal{S}_k$. We can then extend this tail policy to the tail policy starting at stage $k-1$ in any given state $x_{k-1} \in \mathcal{S}_{k-1}$, by using that according to the PO, the tail of the $k-1$-tail policy must be one of the original $k$-stage optimal tail policies, each which has the known cost $J_k(x_k)$. The DP algorithm formalizes this idea, and is summarized in theorem 6.2.1.

**Theorem 6.2.1** (DP algorithm)**.** *For every initial state $x_0$, the optimal cost $J^*(x_0)$ is equal to $J_0(x_0)$, and optimal policy $\pi^*$ is $\pi^* = \{\mu_0, \ldots, \mu_{N-1}\}$, computed by the following algorithm, which begins by defining*

$$J_N(x_N) = g_N(x_N), \quad x_N \in \mathcal{S}_N \tag{6.6}$$

☞ **Algorithm 1** The dynamical programming (DP) algorithm, see theorem 6.2.1

$J \leftarrow$ empty list cost functions $N+1$
$\pi \leftarrow$ empty list of policies
**for all** $x \in \mathcal{S}_N$ **do**
    $J_N(x) \leftarrow g_N(x)$ ▷ Implements eq. (6.6)
**end for**
**for** $k = N-1, \ldots, 0$ **do** ▷ Note loop starts at $k = N-1$ and proceeds backwards
    **for all** $x \in \mathcal{S}_k$ **do**
        **for all** $u \in \mathcal{A}_k(x)$ **do**
            $Q_u \leftarrow 0$
            **for all** Noise values $w$ s.t. $p(w) = P_k(w|x,u) > 0$ **do**
                $Q_u \leftarrow Q_u + p(w)\left[g_k(x,u,w) + J_{k+1}(f_k(x,u,w))\right]$
            **end for**
        **end for**
        $u^* = \arg\min_{u \in \mathcal{A}_k(x)} Q_u$
        $J_k(x) = Q_{u^*}$ ▷ Implements eq. (6.7a)
        $\pi_k(x) = u^*$ ▷ Implements eq. (6.7b); we use $\pi_k = \mu_k$ as a shorthand
    **end for**
**end for**

*and then proceeds backward in time from $k = N-1$ to $k = 0$ and for each $x_k \in S_k$ computes*

$$J_k(x_k) = \min_{u_k \in \mathcal{A}_k(x_k)} \mathbb{E}_{w_k} \left\{ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right\} \tag{6.7a}$$

$$\mu_k(x_k) = u_k^* \quad (u_k^* \text{ is the } u_k \text{ which minimizes the above expression}). \tag{6.7b}$$

*Proof.* We prove that the sequence of function $J_k$ produces using the DP algorithm in eq. (6.7) corresponds to the cost of the optimal tail policy defined in eq. (6.3), that is,

$$\begin{aligned} J_k(x_k) &= J_k^*(x_k) \\ &= \min_{\pi^k} \mathbb{E}\left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} \end{aligned} \tag{6.8}$$

The proof is by induction starting at $k = N$ and using eq. (6.8) as the induction hypothesis

**Start of induction $k = N$:** For $k = N$ both sides of eq. (6.8) are equal to $g_N(x_N)$ by definition

**Induction step $k < N$:** Assume for some $k < N$ we have $= J_{k+1}(x_{k+1}) = J_{k+1}^*(x_{k+1})$ by the induction hypothesis. Writing the $k$-tail policy as

$$\pi^k = (\mu_k, \pi^{k+1}) = (\mu_k, \mu_{k+1}, \ldots, \mu_{N-1})$$

the right-hand side of eq. (6.8) can now be re-arranged to be:

$$J_k^*(x_k) = \min_{(\mu_k, \pi^{k+1})} \mathbb{E}\left\{ g_N\left(x_N\right) + \sum_{i=k}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right) \right\} \tag{6.9}$$

As a first step, we move the expectation inside the parenthesis and use the definition of the cost of a tail policy:

$$\begin{aligned} &= \min_{(\mu_k, \pi^{k+1})} \mathbb{E}_{w_k}\left[ g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) + \mathbb{E}_{w_{k+1:N-1}}\left\{ g_N\left(x_N\right) + \sum_{i=k+1}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right) \right\} \right] \\ &= \min_{(\mu_k, \pi^{k+1})} \mathbb{E}_{w_k}\left[ g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) + J_{k+1,\pi^{k+1}}(f_k(x_k, \mu_k(x_k), w_k)) \right] \end{aligned} \tag{6.10}$$

By this definition, $(\mu_k, \pi^{k+1})$ is the optimal policy for the $k$-tail problem, and therefore by the **PO** this policy will also be optimal for the $k+1$ tail-problem, which has optimal cost $\min_{\pi^{k+1}} J_{k+1,\pi^{k+1}}(x_{k+1})$. Therefore:

$$= \min_{\mu_k} \mathbb{E}_{w_k}\left[ g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) + \min_{\pi^{k+1}} J_{k+1,\pi^{k+1}}(f_k(x_k, \mu_k(x_k), w_k)) \right]$$

Which, by definition equal to the optimal tail policy $J_{k+1}^*$:

$$= \min_{\mu_k} \mathbb{E}_{w_k}\left[ g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) + J_{k+1}^*(f_k(x_k, \mu_k(x_k), w_k)) \right]$$

We can now apply the induction hypothesis eq. (6.8) and some simple algebra to obtain:

$$\begin{aligned} &= \min_{\mu_k} \mathbb{E}_{w_k}\left[ g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) + J_{k+1}(f_k(x_k, \mu_k(x_k), w_k)) \right] \\ &= \min_{u_k \in \mathcal{A}_k(x_k)} \mathbb{E}_{w_k}\left[ g_k\left(x_k, u_k, w_k\right) + J_{k+1}(f_k(x_k, u_k, w_k)) \right] \\ &= J_k(x_k) \end{aligned}$$

establishing the induction step. By induction, the hypothesis is true for $k = 0$, which gives us our result.

□

For clarity, the update rules in the DP algorithm have been re-phrased as an algorithm in algorithm 1, which illustrates how the expectations are practically implemented.

### 6.2.1 Example: The small graph problem

As a first example, we will consider the small graph traversal problem we previously encountered in section 4.2.2. The graph considered is re-produced in fig. 6.1. Recall the small graph example does not use the noise terms, which makes it particularly useful

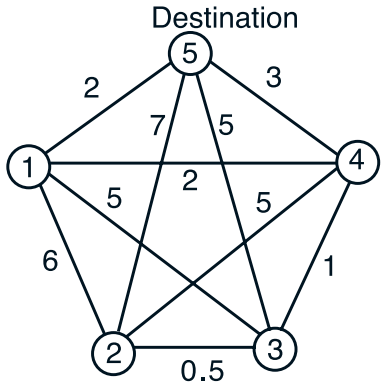


Figure 6.1: Reprint of fig. 4.4. The goal is to find the path with least cost from vertex 2 to 5.

for debugging the DP algorithm. In particular, it is possible to completely omit the loop over $w$ in algorithm 1, line 10 and simply update $Q_u$ as:

$$Q_u \leftarrow g_k(x, u, \texttt{None}) + J_{k+1}(f_k(x, u, \texttt{None})). \tag{6.11}$$

It is recommended you implement this version first (and make sure it works) before adding the noise terms. The version with noise terms should obviously produce equivalent outputs. Carefully review the method to ensure you understand how and why it produces the output below:

**Stage** $k = N$ Because there are 5 vertices in the problem, it can be solved in at most 4 steps. We therefore select $N = 4$. For initialization we have $J_N = g_N$. From the definition of the environment this is:

$$J_4(5) = 0, \quad J_4(s) = \infty \text{ for } s = 1, 2, 3, 4$$

**Stage** $k = 3$ With eq. (6.11) in mind, the update we perform is equivalent to[1]:

$$J_3(x) = \min_u \left[ g_k(x, u) + J_{k+1}(f_k(x, u)) \right]$$

Since the cost term $J_4(x')$ is only finite when $x' = 5$, this is equivalent to the cost to go to node 5 in one step. In other words:

$$J_3(1) = 2, \quad J_3(2) = 7, \quad J_3(3) = 5, \quad J_3(4) = 3$$

Meanwhile, $J_3(5) = 0$ since it is already in the goal state.

[1] we suppress the $w$ terms as they are redundant

**Stage** $k = 2$ Let's consider $J_2(x_2 = 1)$, i.e. the cost-to-go for the first node. It is defined as:

$$J_2(1) = \min_{u=1,2,3,4} \{g_2(1,u) + J_3(u)\} = \min\{2, 13, 10, 5\} = 2$$

Since this corresponds to the first action $u = 1$ we also have $\mu_2(x = 1) = 1$; this means the optimal policy is to stay in vertex 1, then go to vertex 5 in the next move.

One more example: Suppose we are in vertex 2, in this case:

$$J_2(2) = \min_{u=1,2,3,4} \{g_2(2,u) + J_3(u)\} = \min\{8, 7, 5.5, 8\} = 5.5$$

Which corresponds to $\mu_2(x = 2) = 3$, i.e. we go from vertex 2 to vertex 3, and then later from 3 to 5 at a total cost of 5.5. The remaining steps have been omitted for brevity.

### 6.2.2 Example: The chess match

Let us return to the Chessmatch example we encountered in section 5.1.3. Recall the states are player $A$ and $B$'s score $x_k = \begin{bmatrix} a_k \\ b_k \end{bmatrix}$. To apply the DP algorithm we first initialize

$$J_N(x_k) = \begin{cases} -1 & a_k > b_k \\ -p_w & a_k = b_k \\ 0 & a_k < b_k \end{cases}$$

And each update in eq. (6.7) simplifies to:

$$\begin{aligned} J_k(x_k) &= \min_{u_k \in \{0,1\}} \mathbb{E}\left[J_{k+1}(w_k)\right], \quad w_k \sim P_k(\cdot|x_k, u_k) \\ &= \min \left\{ p_w J_{k+1}\left(x_k + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) + (1-p_w) J_{k+1}\left(x_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right), \right. \\ &\qquad \left. p_d J_{k+1}(x_k) + (1-p_d) J_{k+1}\left(x_k + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) \right\} \end{aligned} \tag{6.12}$$

**Stage** $k = 1$ This is a bit daunting, but remembering that $k = 1$ (because $N = 2$) we can first consider the case $A$ is ahead, $x_k = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, in which case eq. (6.12) becomes

$$\begin{aligned} J_1\left(x_k = \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) &= \min\left\{-p_w - (1-p_w)p_w, -p_d - (1-p_d)p_w\right\} \\ &= -p_d - (1-p_d)p_w \end{aligned} \tag{6.13}$$

(and since the second option was selected the optimal policy is to play timid). Next, for a drawn game 1, eq. (6.12) gives us

$$J_1\left(x_1 = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}\right) = \min\left\{-p_w, -p_d p_w\right\} = -p_w \tag{6.14}$$

which corresponds to bold play and finally when $A$ is behind we get from eq. (6.12)

$$J_1\left(x_1 = \begin{bmatrix}0\\1\end{bmatrix}\right) = \min\left\{-p_w^2, 0\right\} = -p_w^2 \tag{6.15}$$

i.e. to play bold, which in hindsight is obvious since timid play always looses.

**Stage** $k = 0$ We can then proceed to find the optimal cost for the start of the match where $x_0 = \begin{bmatrix}0\\0\end{bmatrix}$:

$$\begin{aligned} J_0\left(x_0 = \begin{bmatrix}0\\0\end{bmatrix}\right) &= \min\left\{p_w J_1\left(\begin{bmatrix}1\\0\end{bmatrix}\right) + (1-p_w)J_{k+1}\left(\begin{bmatrix}0\\1\end{bmatrix}\right),\right. \\ &\qquad \left. p_d J_{k+1}\left(\begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix}\right) + (1-p_d)J_{k+1}\left(\begin{bmatrix}0\\1\end{bmatrix}\right)\right\} \\ &= \min\left\{p_w(-p_d - (1-p_d)p_w) + (1-p_w)(-p_w^2),\right. \\ &\qquad \left. p_d(-p_w) + (1-p_d)(-p_w)^2\right\} \\ &= p_w(-p_d - (1-p_d)p_w) + (1-p_w)(-p_w^2) \\ &= -p_w(p_d + (1-p_d)p_w + (1-p_w)p_w) \end{aligned} \tag{6.16}$$

and since the first option was selected, the optimal policy is to play bold. Thus, the DP algorithm both recovered the best policy (timid if and only if ahead in the match), but also provided us with the chance of winning the match using the optimal policy.

### 6.2.3 Example: Inventory control

As a second example, let's apply the DP algorithm to the inventory control problem. Recall that $N = 3$ and there is no terminal cost. We therefore initialize

$$J_3(x_3) = 0 \tag{6.17}$$

and the algorithm will take the general form:

$$J_k(x_k) = \min_{u_k=0,1,2} \mathbb{E}_{w_k}\left\{u_k + (x_k + u_k - w_k)^2 + J_{k+1}(\max(0, \min\{x_k + u_k - w_k\}))\right\} \tag{6.18}$$

**Stage** $k = 2$ In this case the minimization eq. (6.18) becomes

$$J_k(x_k) = \min\left\{\mathbb{E}_{w_k}\left[(x_k - w_k)^2\right], 1 + \mathbb{E}_{w_k}\left[(x_k + 1 - w_k)^2\right], 2 + \mathbb{E}_{w_k}\left[(x_k + 2 - w_k)^2\right]\right\} \tag{6.19}$$

The terms can be computed using the definition of the probability density of $w_k$. The expressions are:

$$\mathbb{E}_{w_k}\left[(x_k - w_k)^2\right] = 0.1x_k^2 + 0.7(x_k - 1)^2 + 0.2(x_k - 2)^2 \tag{6.20}$$

$$1 + \mathbb{E}_{w_k}\left[(x_k + 1 - w_k)^2\right] = 1 + 0.1(x_k + 1)^2 + 0.7x_k^2 + 0.2(x_k - 1)^2 \tag{6.21}$$

$$2 + \mathbb{E}_{w_k}\left[(x_k + 2 - w_k)^2\right] = 2 + 0.1(x_k + 2)^2 + 0.7(x_k + 1)^2 + 0.2x_k^2 \tag{6.22}$$

| Stock | Stage 0 cost-to-go | Stage 0 optimal stock to purchace | Stage 1 cost-to-go | Stage 1 optimal stock to purchace | Stage 2 cost-to-go | Stage 2 optimal stock to purchace |
|---|---|---|---|---|---|---|
| 0 | 3.700 | 1 | 2.50 | 1 | 1.3 | 1 |
| 1 | 2.700 | 0 | 1.50 | 0 | 0.3 | 0 |
| 2 | 2.818 | 0 | 1.68 | 0 | 1.1 | 0 |

Table 6.1: Inventory example results

The possible values of $x_k$ is $\mathcal{S}_k = \{0, 1, 2\}$, and so we obtain:

$$J_2(x_2 = 0) = \min\{1.5, 1.3, 3.1\} \tag{6.23}$$

therefore the optimal action $\mu_2^*(0) = 1$. Proceeding in this manner we get

$$J_2(x_2 = 1) = \min\{0.3, 2.1, 2.1\} \tag{6.24}$$

and so $\mu_2^*(1) = 0$ and $J_2(1) = 0.3$. Similarly we obtain $J_2(2) = 1.1$ and $\mu_2^*(2) = 0$.

**Stage** $k = 1$ For completeness, for $k = 1$ and $x_1 = 0$ we obtain:

$$\begin{aligned} J_1(x_1 = 0) &= \min_{u_k=0,1,2} \mathbb{E}_{w_k}\left\{u_1 + (u_1 - w_1)^2 + J_2\left(\max\left(0, \min\{2, u_1 - w_1\}\right)\right)\right\} \\ &= \min\left\{\mathbb{E}_{w_1}\left[w_1^2 + J_2(0)\right],\right. \\ &\quad 1 + \mathbb{E}_{w_1}\left[(1 - w_1)^2 + J_2(\max\{0, 1 - w_1\})\right], \\ &\quad \left. 2 + \mathbb{E}_{w_1}\left[(2 - w_1)^2 + J_2(\max\{0, 2 - w_1\})\right]\right\} \end{aligned} \tag{6.25}$$

Using what we know about $J_2$ we can evaluate the expressions in the parenthesis to obtain

$$J_1(x_1 = 0) = \min\{2.8, 2.5, 3.68\}$$

so the optimal action if we have an empty inventory in stage 1 is to buy a single item $\mu_1^*(0) = 1$. We can continue in this manner, however since the computations are probably unlikely to rouse much excitement at this point I have summarized them in table 6.1.

Thus, for the inventory control example, the optimal policy is to simply buy a single item if the inventory is empty.

### 6.2.4 Example: Optimal pacman★

As a final case study, let's try to play optimal Pacman. We will consider three Pacman games with zero, one and two ghosts respectively, and a single food pellet as shown in fig. 6.2. The game world is here assumed to be just a $2 \times 4$ grid, and the initial state

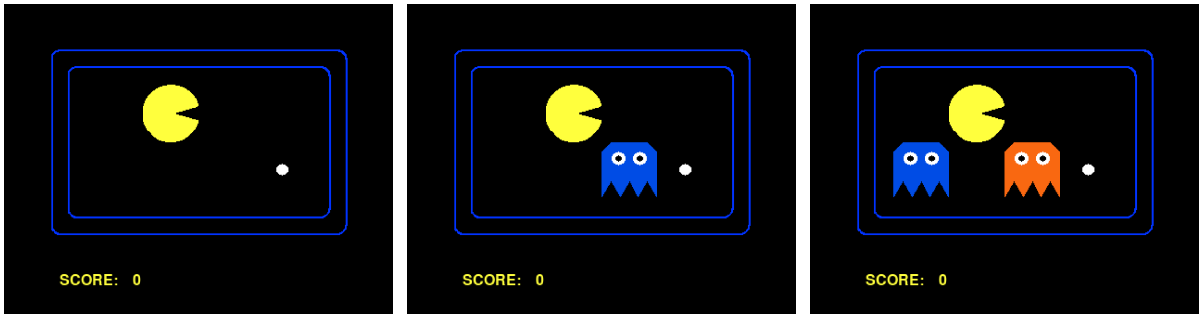


Figure 6.2: Starting position for the three Pacman games we will consider with zero, one and two ghosts respectively. The game terminates in a win if Pacman eats the food pellet before stage $N$ without being eaten himself, otherwise the game is lost.

$x_0$ of pacman (and the ghosts) will always be as indicated in the three figures. The purpose is as usual to eat the food dot and avoid the ghosts. When pacman moves to the dot the game is won, and when pacman and a ghost shares a tile the game is lost.

Let us start with the simplest case with zero ghosts, and suppose we wish to apply DP to solve this problem. The first thing we must do is to select $N$, and in this case we will choose $N = 8$.

### Zero-ghost pacman

We first see that Pacman can actually solve the problem in $k = 3$ steps (`right`, `right`, `down`). This means that when we plan on a horizon of $N = 8$, the game will at that point enter an undefined state. We fix this using the absorbing state trick from section 5.2.1 and define the new states `win`, `lose` to signify if pacman has lost or won. When Pacman is in either of these states, only the action `stop` is available, and pacman always remains in these states. We can now define the cost-function as $g_k(x_k, u_k, w_k) = 0$ and

$$g_N(x_N) = \begin{cases} -1 & \text{if } x_N = \texttt{Win} \\ 0 & \text{otherwise.} \end{cases}$$

Next, the dynamics. In the case of no ghosts the dynamics is deterministic and is defined by the game rules, and the Pacman game comes with a function which can easily allow us to compute the effect of taking an action by Pacman as `x.generateSuccessor(0, u)` where `x` is the game state and `u` is the given action. The set of available actions can similarly be found using `x.getLegalActions(0)`. In both cases, the 0 refers to pacman, and for instance 1 would refer to the available actions (and outcome of taking an action) for ghost 1.

When we apply the DP algorithm, the first step requires us to compute $J_N(x_N)$ for each $x_N \in \mathcal{S}_N$. In the pacman game this creates a problem, since we have no simple mechanism available which will provide us with this set, and although the game world here is so easy it may be possible to explicitly construct the set we should work out a more general solution to obtain the set of states. A little thought reveals that if we

| Game type | $\lvert \mathcal{S}_1 \rvert$ | $\lvert \mathcal{S}_N \rvert$ | DP optimal winrate $-J_0(x_0)$ | Simulated winrate | Average game length |
|---|---|---|---|---|---|
| No Ghosts | 4 | 18 | 1.00 | 1.00 | 3.00 |
| One Ghosts | 11 | 118 | 1.00 | 1.00 | 4.00 |
| Two Ghosts | 16 | 781 | 0.93 | 0.91 | 4.78 |

Table 6.2: Dynamical programming applied to the simple Pacman levels shown in fig. 6.2.

know the dynamics of the game, we can easily work out the set $\mathcal{S}_1$ from $\mathcal{S}_0$ and in general $\mathcal{S}_{k+1}$ from $\mathcal{S}_k$ by simply trying out all available solution. In general, in the deterministic case, this operation is:

$$\mathcal{S}_{k+1} = \{f_k(x_k, u_k) \mid x_k \in \mathcal{S}_k, u_k \in \mathcal{A}(x_k)\} \tag{6.26}$$

If we do this, we see that the size of the state spaces changes as [2].

$$\lvert \mathcal{S}_0 \rvert = 1, \quad \lvert \mathcal{S}_1 \rvert = 4, \quad \lvert \mathcal{S}_2 \rvert = 12, \quad \lvert \mathcal{S}_3 \rvert = \cdots = \lvert \mathcal{S}_N \rvert = 18$$

It may appear surprising that there are not $\lvert \mathcal{S}_N \rvert = 8$ states (7 which can be occupied plus `win`), however Pacman also has an orientation (determined by the previous state), and even though it is irrelevant insofar as the game rules are concerned, it is counted as a unique state. There are 2 available orientation for the two left-most states, one for the upper-right state (pacman faces the wall to the right; after all, he cannot enter this state from below) and three available orientations for the four middle state and including the win state this does indeed agree:

$$2 \times 2 + 3 \times 4 + 1 \times 1 + 1 = 18.$$

With this information we can apply the DP algorithm. Unsurprisingly, we obtain an expected cost of $-1$ (Pacman always wins), however the number of steps may differ from 3 since it makes no difference if Pacman wins in 7 or 3 steps in terms of the cost. This problem may be fixed by adding a miniscule constant cost of e.g. $10^{-8}$ to each step, and now we do see pacman perfectly solves the maze, as shown in table 6.2.

### 6.2.5 Multi-ghost pacman

Multi-ghost pacman requires a slight elaboration on the game rules. For multiple ghosts the rules are:

- Pacman is labelled 0, the ghosts are labelled $1, 2, \ldots Q - 1$

[2] An eagle-eyed student spotted a problem in this counting argument, see `https://piazza.com/class/lcyz0561au01sm/post/21`. TL;DR: The counting-argument works for the project, but the story is a little more complicated for this particular example due to where pacman starts.

- When the game is in state $x_k$, and pacman takes action $u_k$, the following occurs:
  - Pacman transitions to a new state deterministically, and food Pacman land on is immediately eaten. If the new state corresponds to a win/loose, the game terminates
  - In turn, $1, 2, \ldots$ ghost $p$ moves. The ghost selects it's action uniformly at random from the available actions and moves deterministically. If at any point a ghost lands on Pacman, the game terminates
- The resulting position is now $x_{k+1}$

For this type of dynamics, it is convenient to simply let $w_k$ be the new state (corresponding to the effect of the procedure above, including the random ghost movement) and define:

$$f_k(x_k, u_k, w_k) = w_k \tag{6.27}$$

then $w_k$ is generated by carrying out the above actions in sequence, which also makes it easy to track the probabilities. The state spaces also have to be computed. In the stochastic case this is defined as:

$$\mathcal{S}_{k+1} = \{f_k(x_k, u_k, w_k) \mid x_k \in \mathcal{S}_k, u_k \in \mathcal{A}(x_k), w_k \text{ st } P_k(W_k = w_k | x_k, u_k) > 0\} \tag{6.28}$$

i.e. for each $x_k$, we try each available action $u_k$, and let the set of available state be those which can be reached with non-zero probability.

### 6.2.6 One-ghost Pacman

For the one-ghost game we obtain:

$$|\,\mathcal{S}_1\,| = 11, \quad |\,\mathcal{S}_8\,| = 117.$$

There is still a 100% chance of winning the one-ghost game (the expected cost is $J_0(x_0) = -1$). This may appear strange, as it would seem there is no way to get past the ghost without a chance the ghost catches pacman, however, pacman exploits some subtleties in the game rules: In the first move Pacman always chooses the `Stop` action. If the ghost go `left`, Pacman makes a run for the dot and win in three additional moves (the game terminates as soon as pacman eats the food dot regardless of whether the ghost move onto pacman on the ghosts turn).

On the other hand, if the ghost goes `up`, pacman chooses the `down` action, and in the next move pacman chooses `right`; Pacman exploits that if the ghost has gone up in one turn, the ghost cannot choose `down` in the next and so the resulting square is safe. The case where the ghost go `right` pacman will choose `down`, `right`, `right`, once more exploiting that the ghost cannot reverse course in one move.

**Two-ghost pacman**

For two ghosts pacman is not guaranteed to win. It is easy to imagine a case where two ghosts can adopt a strategy where they will catch pacman with probability 1, and in that light it may be surprising that the game is still relatively easy and with optimal strategy we will win about 90% of the games ($J_0(x_0) = -0.898$). Another observation is that the problem has become quite a bit harder and we have:

$$|\,\mathcal{S}_1\,| = 16, \quad |\,\mathcal{S}_8\,| = 768.$$

### 6.2.7 Shortcomings of DP

For each new food dot the number of states will (approximately) double, and for each new ghost the number of states will roughly be multiplied by the number of squares. On top of that, for each state we have a computational burden corresponding to the number of actions *times* the number of possible random disturbances $w_k$. These computations must be carried out $N$ times.

In practice, this means that a *direct* application of DP has a limited applicability due to the computational demands, and would not be suited for the real pacman game. This does *not* mean DP is irrelevant. As we will see in this course, these shortcomings can each be addressed, typically as a tradeoff between exactness versus feasibility. These various ways to address the shortcomings give rise to central methods in control theory, search and reinforcement learning.

## 6.3 Reformulations

In this section, we will discuss two re-formulations of the DP algorithm which have important applications in control theory, AI and reinforcement learning

### 6.3.1 Evaluation

Let's begin with the tail cost defined in eq. (6.2)

$$J_{k,\pi}\left(x_k\right) = \mathbb{E}\left[g_N\left(x_N\right) + \sum_{i=k}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right)\right]. \tag{6.29}$$

Using that the expectation is a linear operation we can pick out the first cost term:

$$\mathbb{E}\left[g_N\left(x_N\right) + \sum_{i=k+1}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right) + g_k(x_k, \mu_k(x_k), w_k)\right] \tag{6.30}$$

$$= \mathbb{E}_{w_k}\left[g_k(x_k, \mu_k(x_k), w_k) + \mathbb{E}\left\{g_N\left(x_N\right) + \sum_{i=k+1}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right)\right\}\right]. \tag{6.31}$$

The inner-most expectation is just equal to $J_{k+1,\pi}(x_{k+1})$, with $x_{k+1} = f_k(x_k, \mu_k(x_k), w_k)$, so therefore this implies:

$$J_{k,\pi}(x_k) = \mathbb{E}_{w_k}\left[g_k(x_k, \mu_k(x_k), w_k) + J_{k+1,\pi}(x_{k+1})\right]. \tag{6.32}$$

If you compare this to the update rule for the DP-algorithm in eq. (6.7a), it means there is a simple, DP-like algorithm for evaluating the cost of a policy $J_{0,\pi}(x_0)$. Later reinforcement learning, this derivation is the equivalent to one of the Bellman-equations, and the resulting method is called policy evaluation.

### 6.3.2 Adversarial setting

Suppose at step $k$ the available random disturbances are $w_k \in W_k(x_k, u_k)$ and recall the usual DP algorithm take the form (see eq. (5.5)):

$$J_\pi(x_0) = \mathbb{E}_{\substack{w_k \in W_k(x_k, \mu_k(x_k)) \\ k=0,\ldots,N-1}}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k\left(x_k, \mu_k(x_k), w_k\right)\right] \tag{6.33}$$

and the goal is to find the optimal policy $\pi^*(x_0) = \arg\min_\pi J_\pi(x_0)$. In the so-called
MINIMAX **minimax** formulation the expectation is simply replaced with a max in the objective:

$$\hat{J}_\pi(x_0) = \max_{\substack{w_k \in W_k(x_k, \mu_k(x_k)) \\ k=0,\ldots,N-1}}\left[g_N(x_N) + \sum_{k=0}^{N-1} g_k\left(x_k, \mu_k(x_k), w_k\right)\right]. \tag{6.34}$$

and the optimal policy is still defined as $\pi^*(x_0) = \arg\min_\pi \hat{J}_\pi(x_0)$. The intuitive meaning of this change is that we assume the noise disturbances $w_k$ represents the *worst case* from our perspective. There are two main applications of this formulation:

**Control** In this case using the minimax formulation provides a guarantee that the system will not undergo a disaster because of a particular bad set of control vectors; this setup is relevant if people may die as a result of failure as in e.g.
ADVERSARIAL CONTROL aerospace applications and is known as **adversarial control**.

**AI** In games, it is referred to as *adversarial search*. Recall in the pacman example section 6.2.5, for a single ghost, the noise disturbance was the ghosts move, and so the adversarial setting implies the opponent play optimally.

Intuitively, the minimax formulation is simply a particular choice of noise distributioni $P(w_k|x_k, u_k)$, namely the *worst* distribution from our perspective. It is therefore not surprising (a reader interested in a formal argument can consult the excellent resource [Ber05]) that the optimal control rule is exactly as the DP algorithm except the average is now a maximum (contrast this with eq. (6.7)):

$$\begin{aligned} J_k(x_k) &= \min_{u_k \in \mathcal{A}_k(x_k)} \max_{w_k} \left\{g_k(x_k, u_k, w_k) + J_{k+1}\left(f_k(x_k, u_k, w_k)\right)\right\} \\ \mu_k(x_k) &= u_k^* \end{aligned} \tag{6.35}$$

### 6.3.3 Finite-horizon formulation

Our starting point is the DP objective eq. (6.33). The DP algorithm determines the optimal expected cost function $J^*$ and policy $\pi^*$ by initializing $J_N(x_N) = g_N(x_N)$ and iterating the DP update eq. (6.7). Three observations:

- At any step $k = d - 1$ of the DP algorithm, it is in fact solving a truncated DP problem starting at step $k = 0$ and of length $N' = d$ with the new terminal cost:

$$g_{N'}(x_{N'}) = J_d(x_d) \tag{6.36}$$

- The truncated policy is optimal so therefore

$$g_{N'}(x_{N'}) = J_d(x_d) = J_d^*(x_d) \tag{6.37}$$

- Since the DP algorithm determines the optimal policy for any problem, it must also determine the optimal policy in the problem truncated at $k$.

Putting these observations together we have shown the optimal value function can be found as

$$J^*(x_0) = \underset{\mu_0,\ldots,\mu_{d-1}}{\arg\min}\, \mathbb{E}\left[ J_d^*\left(x_{k+1}\right) + \sum_{k=0}^{d-1} g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) \right] \tag{6.38}$$

And the $d$-long policy $\mu_0, \ldots, \mu_{d-1}$ will still be optimal, i.e. agree with the first $d$ policy functions of the optimal policy $\pi^*$. In itself this result is not of much use, since in order to find $J_d^*$ we still have to plan in the full $N$-step problem. However, it becomes useful the moment we can make approximations: Assume we have an approximation $\hat{J}_d \approx J_d^*$, then the $N$-horizon DP planning problem is reduced to a short-horizon $d$-long planning problem. Furthermore, even if the approximation is very poor, we can increase $d$ and the importance of $J_d^*$ will typically diminish. This is perhaps the single most useful approximations and is used in an abundance of places:

**AI** Any sort of complex planning (see following chapters)

MODEL-PREDICTIVE CONTROL
RECEDING HORIZON CONTROL

**Control** It is known as **model-predictive control** (or **receding horizon control**) and is the principal way of solving complex control tasks and dealing with model uncertainty

$n$-STEP METHODS

**Reinforcement learning** The re-write is the basis of **$n$-step methods**

# Chapter 7

# Shortest path formulation

This chapters deal with the basic decision problem from chapter 6, but with an emphasis on the deterministic case. In other words, the setting considered here is a special case where the next state $x_{k+1}$ and obtained cost is fully determined by $x_k$ and $u_k$.

As we saw in section 5.1.4, in this case we can convert our closed-loop controller into an open-loop one (recall an open-loop controller is one where the actions are determined beforehand). As a consequence, it suffices to search over open-loop policies $\pi = (u_0, \ldots, u_{N-1})$. As we will see, this greatly simplifies the implementation of DP and hence makes it applicable to larger problems, and gives rise to the popular search methods as we will discuss later. The discussion in this chapter will also be useful when we later discuss multi-agent systems and approximate adversarial search methods familiar from games such as chess or Pacman.

## 7.1 Deterministic decision problem

DETERMINISTIC DECISION PROBLEM

The **deterministic decision problem** corresponds to the case where the noise disturbances can only take a single value and can hence be omitted from the problem specification:

$$x_{k+1} = f_k(x_k, u_k) \tag{7.1}$$

$$J_\pi(x_0) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k) \tag{7.2}$$

As discussed in section 5.1.4, in the deterministic case any closed-loop controller can be converted to an equivalent open-loop one we can therefore simply consider the policy as being comprised of an $N$-element list of controls

$$\pi = (u_0, u_1, \ldots, u_{N-1}), \quad u_k \in \mathcal{A}_k(x_k) \tag{7.3}$$

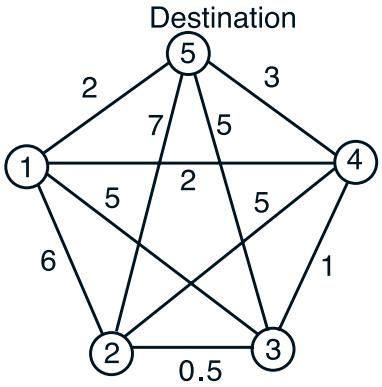


Figure 7.1: A simple graph of $n = 5$ vertices

and the objective remains to minimize the cost over all admissible policies:

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0) \tag{7.4}$$

$$= \min_{u_0, u_1, \ldots, u_{N-1}} \left[ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k) \right]. \tag{7.5}$$

Since this is a special case of the basic problem we can simply apply the DP algorithm to find an optimal solution. However, the problem structure in eq. (7.5) is much simpler in two important respects:

- There is no longer an expectation
- The minimization is over a vector rather than a function.

The implication of these simplifications is that the solution algorithm can be written much more efficiently. We already saw a few instances of the deterministic decision problem, for instance the graph-traversal problem in section 4.2.3 or the zero-ghost Pacman example in section 6.2.4. Indeed most classical computer science problems fall within this description, for instance the traveling salesman problem

### 7.1.1 Traveling Salesman

Consider the graph-traversal problem in fig. 7.1. The traveling salesman problem consists of finding the path which visit all vertices just once, ends up in the same node as it started in, and has the minimum cost; the name derives from an example where the verticess are cities, the edge-cost a travel distance, and the traveling salesman must visit all cities once and end up where she started.

Which edges to traverse can be seen as decisions, however to be able to represent that a path should visit all vertices we must know which have been visited already. We can therefore view it as a decision problem where the states are lists of vertices which have been visited, for instance:

$$x_3 = (1, 2, 4).$$

Formally, suppose the graph has $n$ vertices $\mathcal{S} = \{1, 2, \ldots, n\}$ and an edge-cost $A_{ij}$. For the traveling salesman problem the initial state is $s = ()$. We define the actions as the nodes we can travel to

$$\mathcal{A}_k(x_k = i) = \{j \in \mathcal{S} \mid A_{ij} < \infty\} \tag{7.6}$$

The dynamics is now to simply add the selected node to the state

$$f_k(x_k = (i_0, i_1, \ldots, i_k), u_k = j) = (i_0, i_1, \ldots, i_k, j) \tag{7.7}$$

The cost functions are defined as $g_k(x_k, u_k) = 0$ and the terminal cost at $N = n + 1$ is finite if and only if the path is a solution to the traveling salesman problem, i.e.

$$g_N(x_N = (i_0, \ldots, i_{n+1})) = \begin{cases} \sum_{k=0}^{n} A_{i_k, i_{k+1}} & \text{if } i_0 = i_{n+1} \text{ and all states are visited} \\ \infty & \text{otherwise} \end{cases}. \tag{7.8}$$

### 7.1.2 An issue with the DP algorithm

If we try to solve the traveling salesman problem using the DP algorithm the first step we arrive at is the initialization:

$$J_N(x_N) = g_N(x_N), \quad x_N \in \mathcal{S}_N, \tag{7.9}$$

however, already here we encounter the same problem as in the pacman example from section 6.2.4: In the traveling salesman example the individual states $x_N$ must be $N$-long lists of all vertices's we have traversed[1]; in other words, to solve the problem, we must first generate *all* such lists (i.e., all lists of $n$ vertices's), and compute their cost.

Since there are $n!$ such lists this is not only unfeasible, once we *have* the list (and have computed their terminal costs) we can just select the state with the lowest cost and return it as an optimal solution. In other words, in this problem, simply initializing the DP algorithm corresponds to solving the problem using complete enumeration.

This feature, that the state-spaces $\mathcal{S}_k$ are difficult to compute, is a general problem in discrete application, and arises because the DP algorithm proceeds *backwards* from $k = N$ to $k = 0$. The solution is to find a way to run the DP algorithm in reverse, which is exactly what the **forward DP algorithm** does.

FORWARD DP ALGORITHM

## 7.2 The deterministic decision problem and graphs

The natural way of introducing the forward-DP algorithm is to use a connection between the deterministic control problem and graph traversal, and so we will first begin by reformulating the decision problem.

In a deterministic problem, any state $x_k$ and control $u_k$ will result in a transition from $x_k$ to the state $x_{k+1} = f_k(x_k, u_k)$ with associated cost $g_k(x_k, u_k)$.

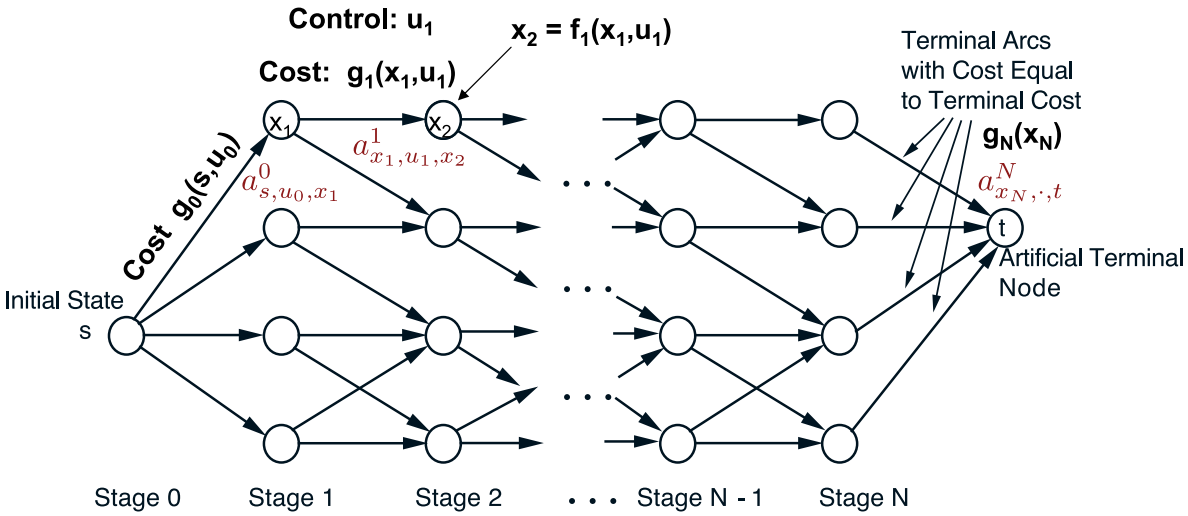


Figure 7.2: Transformation of the basic problem into a graph-traversal form. $s = x_0$ is the initial state (which is considered fixed) and a *new* terminal node $t$ is added. The terminal node is connected to all states in $\mathcal{S}_N$ with a cost of $g_N(x_N)$. The red inserts corresponds to the quantities in the graph view, see fig. 7.3.

GRAPH

We can think of this as traversing a **graph**, where each state $x_k$ is a vertex, and two nodes $x_k$ and $x_{k+1}$ are connected by an edge if there is a control $u_k$ such that $x_{k+1} = f_k(x_k, u_k)$. In this case, the edge has an associated cost $g_k(x_k, u_k)$, see fig. 7.2.

TERMINAL STATE

To complete this identification, we introduce a **terminal state** $t$, distinct from all other nodes, and consider each state $x_N \in \mathcal{S}_N$ to be connected to $t$ with an associated cost $g_N(x_N)$.

A control sequence therefore corresponds to finding a path from the initial state $x_0$ (and we denote this by $s = x_0$) to the terminal state $t$, and the cost of the control sequence is the sum of cost of the edges traversed.

One slightly unusual property of this formulation is that there may be multiple edges connecting two nodes $x_k$ and $x_{k+1}$, specifically this will be the case if there are two distinct actions $u'_k \neq u''_k$ such that

$$x_{k+1} = f_k(x_k, u'_k) = f_k(x_k, u''_k).$$

MULTIGRAPH

What this means is that the graph we have to find the shortest path in will be a **multigraph**, meaning an edge has the form $e = (x, u, x')$ where $x, x'$ are the two end-nodes and $u$ is used to identify this particular edge, see fig. 7.3. These definitions can be summarized as follows:

[1] If the state was just a single vertex, we would have no way to know if $x_N$ corresponded to a traveling salesman path or something else!

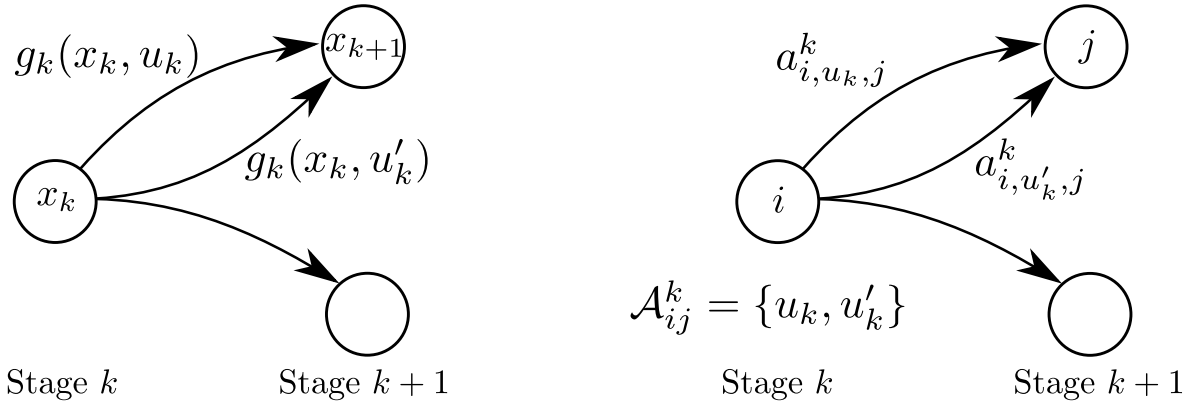


Figure 7.3: Illustration of how a basic problem is transformed into a graph. In the left figure, the state $x_k$ has three available actions, two of which brings it to the node $x_{k+1}$. This is equivalent to a graph, where the vertices are labeled $i = x_k$, $j = x_{k+1}$, there is an edge between $i, j$ if $j = f_k(i, u)$ for some $u$, and the weight in that case is $a^k_{i,u_k,j} = g_k(i, u_k)$. Note this graph can have multiple edges connecting the same vertices.

**Algorithm 2** Backwards DP in the deterministic case

$J_{N+1}(t) \leftarrow 0$
**for** $k = N, \ldots, 0$ **do** ▷ Starting at $N$ instead of $N-1$
    **for all** $i \in \mathcal{S}_k$ **do**

$$J_k(i) = \min_{\substack{j \in S_{k+1}, \\ u \in \mathcal{A}^k_{ij}}} \left[ a^k_{i,u,j} + J_{k+1}(j) \right]$$

    **end for**
**end for**

$$a^k_{i,u,j} = \left\{ \begin{array}{l} \text{Cost of transitioning from } i \in \mathcal{S}_k \text{ to } j \in \mathcal{S}_{k+1} \text{ under action } u. \\ \text{I.e. } g_k(i,u) \text{ assuming } j = f_k(i,u) \text{ and } u \in \mathcal{A}_k(i) \\ \text{(otherwise there is no edge)} \end{array} \right\}, \tag{7.10}$$

$$a^N_{i,\cdot,t} = \left\{ \begin{array}{l} \text{Terminal cost } g_N(i) \text{ of state } i \in \mathcal{S}_N \\ \text{(the dot } \cdot \text{ means the action is irrelevant)} \end{array} \right\}, \tag{7.11}$$

$$\mathcal{A}^k_{ij} = \left\{ \begin{array}{l} \text{Actions available in state } i \text{ which will allow us to transition} \\ \text{to state } j \end{array} \right\}. \tag{7.12}$$

Finally, we introduce the (extra) set of states $\mathcal{S}_{N+1} = \{t\}$ to be comprised of our new terminal node. We can then simply translate the DP algorithm into this new notation and the result can be found in algorithm 2. Upon termination, the cost of a shortest path from $s$ to $t$ will be computed as $J_0(s)$.

**Algorithm 3** Forward DP in tilde-notation

**Ensure:** $J_0(\tilde{s})$ is the cost of the optimal path $s \to t$

$\tilde{J}_{N+1}(\tilde{t}) \leftarrow 0$ ▷ Equivalent to $\tilde{J}_{N+1}(s) \leftarrow 0$ by eq. (7.13)

**for** $k = N, \dots, 0$ **do**

  **for all** $i \in \widetilde{\mathcal{S}}_k$ **do** ▷ Equivalent to $i \in \mathcal{S}_{N+1-k}$ by eq. (7.14)

    $\tilde{J}_k(i) \leftarrow \min_{\substack{j \in \widetilde{S}_{k+1}, \\ u \in \widetilde{\mathcal{A}}_{ij}^k}} \left[ \tilde{a}_{i,u,j}^k + \tilde{J}_{k+1}(j) \right]$ ▷ Equivalent to $\min_{\substack{j \in S_{N-k}, \\ u \in \mathcal{A}_{ji}^{N-k}}} \left[ a_{j,u,i}^{N-k} + J_{k+1}(j) \right]$ by eq. (7.15) and eq. (7.16)

  **end for**

**end for**

### 7.2.1 The forward view of DP

The problem with the DP algorithm is that it proceeds backwards in time, i.e. that it starts at the states at $\mathcal{S}_N$ and ends at $\mathcal{S}_0$. We can fix this by the following simple observation based on fig. 7.2: An optimal path from $s$ to $t$ is equivalent to an optimal path from $t$ to $s$ in a problem where each edge has been reversed but the cost of an edge is unchanged. In other words, simply flip the direction of the edges in the figure, start from $t$, and find the shortest path to $s$.

The reversed problem will be denoted by a tilde symbol. Specifically, for $k = 0, 1, \dots, N$ it is defined as

$$\tilde{t} = s, \quad \tilde{s} = t \tag{7.13}$$

$$\widetilde{\mathcal{S}}_k = \mathcal{S}_{N+1-k} \tag{7.14}$$

$$\tilde{a}_{i,u,j}^k = a_{j,u,i}^{N-k}, \quad j \in \mathcal{S}_{N-k}, \quad i \in \mathcal{S}_{N-k+1}, \tag{7.15}$$

$$\tilde{\mathcal{A}}_{ij}^k = \mathcal{A}_{ji}^{N-k} \tag{7.16}$$

and the basic DP algorithm, applied to this problem, can now be seen in algorithm 3. We can now back-substitute the definition of the tilde symbols in algorithm 3 (shown by red), and since it is more convenient to let $k = 0$ and end at $N$ we can re-index the $J$ symbols. The resulting, equivalent, algorithm is shown in algorithm 4

### 7.2.2 Search problems and forward-DP

The re-indexed DP algorithm proceeds forward in time, however as stated it still requires us to access the sets of states $\mathcal{S}_k$ without providing a clear mechanism for how to compute them, and it is not obvious how to implement the minimization problem. A different, and from an implementation point of view much more useful, view of the forward DP algorithm can be obtained by considering it as a search problem. This leads to the following more general definition:

**Algorithm 4** Forward DP

**Ensure:** $\tilde{J}_{N+1}(t)$ is the cost of the optimal path $s \to t$

1: $\tilde{J}_0(s) \leftarrow 0$
2: **for** $k = 0, \dots, N$ **do**
3: **for all** $i \in \mathcal{S}_{k+1}$ **do**
4:

$$\tilde{J}_{k+1}(i) \leftarrow \min_{\substack{j \in S_k, \\ u \in \mathcal{A}_{ji}^k}} \left[ a_{j,u,i}^k + \tilde{J}_k(j) \right]$$

5: **end for**
6: **end for**

**Definition 7.2.1** (Search problem)**.** *A search problem is defined as*

- *A set of states $\mathcal{S}$ (the search nodes)*
- *An initial state $s \in \mathcal{S}$*
- *A goal test which returns true if we are in the terminal state*
- *A transition function $T$ which returns the set of available edges we can traverse in state $T$ and their cost*

*Specifically, the transition function returns a set of available transitions in the form of triplets:*

$$T(x) = \{\cdots, (x_i', u_i, c_i), \cdots, \} \tag{7.17}$$

*with the interpretation that if $(x', u, c) \in T(x)$ then $x$ is connected to $x'$ by an edge (identified by $u$) and with cost $c$.*

The search-problem does away with the time index $k$ and the maximum number of steps $N$ and is therefore more general than the DP decision problem we have seen. To transform a DP problem into a search problem, we will use the we need to include the state index $k$ into the state, since otherwise this information will be lost (see section 5.2.1). Asides this the identification is trivial:

**Theorem 7.2.2** (The dp problem is a search problem)**.** *We can consider an $N$-step DP problem as a search problem as follows*

- *Define a new terminal state $t$. A valid choice is simply* `t = "terminal_state"`*. Also define a dummy action $u_t = 0$.*
- *Absorb the time index into the states. If $x_k = x$ is the state of the DP problem at stage $k$, this will correspond to a new state $\tilde{x}_k = (x, k)$ in the search problem (see section 5.2.2)*

☞ **Algorithm 5** Forward DP for a search problem

**Ensure:** $\tilde{J}_{N+1}(t)$ is the cost of the optimal path $s \to t$

$\tilde{J}_0(s) \leftarrow 0$
$\mathcal{S}_0 \leftarrow \{s\}$
**for** $k = 0, \dots, N$ **do**
    $\mathcal{S}_{k+1} \leftarrow \emptyset$
    **for all** $i \in \mathcal{S}_k$ **do**
        **for all** $(j, u, c) \in T(i)$ **do**
            **if** $j \notin \mathcal{S}_{k+1}$ **or** $c + J_k(i) < J_{k+1}(j)$ **then** ▷ Check if this transition from $i \to j$ is better than current best estimate
                $\mathcal{S}_{k+1} \leftarrow \mathcal{S}_{k+1} \cup \{j\}$ ▷ Mark state $j$ as visited
                $J_{k+1}(j) \leftarrow c + J_k(i)$ ▷ Update best estimate of shortest path from $s$ to $j$
                $\pi_k(i) \leftarrow u$ ▷ Update best policy
            **end if**
        **end for**
    **end for**
**end for**

- *Assume $\tilde{x}$ is the current state in the search problem. It can either be $\tilde{x} = t$ or $\tilde{x} = (x, k)$ where $x \in \mathcal{S}_k$. The transition function is defined as:*

$$T(\tilde{x}) = \begin{cases} \{(t, u_t, 0)\} & \text{if } \tilde{x} = t \\ \left\{ \left( f_k(x, u_k), u_k, g_k(x, u_k) \right) \mid u_k \in \mathcal{A}_k(x) \right\} & \text{if } k \leq N - 1 \\ \left\{ \left( t, u_t, g_N(x_N) \right) \right\} & \text{if } k = N \end{cases} \tag{7.18}$$

  *The first line in the conditional ensures the terminal state is absorbing. The middle line provides all possible transitions as obtained by trying all possible actions, and the last ensures we connect the states at $k = N$ to the goal state (the last arrows in fig. 7.2)*

- *The goal test is simply if the current state is equal to $t$*

Using the transition function, we can obtain a practical implementation of algorithm 4 as follows: Firstly, at step $k$ we know $\mathcal{S}_k$. We can then try all transitions $T(x_k)$. If one of these transitions takes us from $x_k$ to $x_{k+1}$, at a lower cost than the current best estimate of this transition, we update $J_{k+1}(x_k)$. The resulting method is given in algorithm 5

The re-writing of the DP algorithm to the forward view has important algorithmic implications. Firstly, it makes it much simpler to implement the environments as we now only has to implement the transition function, and the algorithm will compute the intermediate steps along the way.

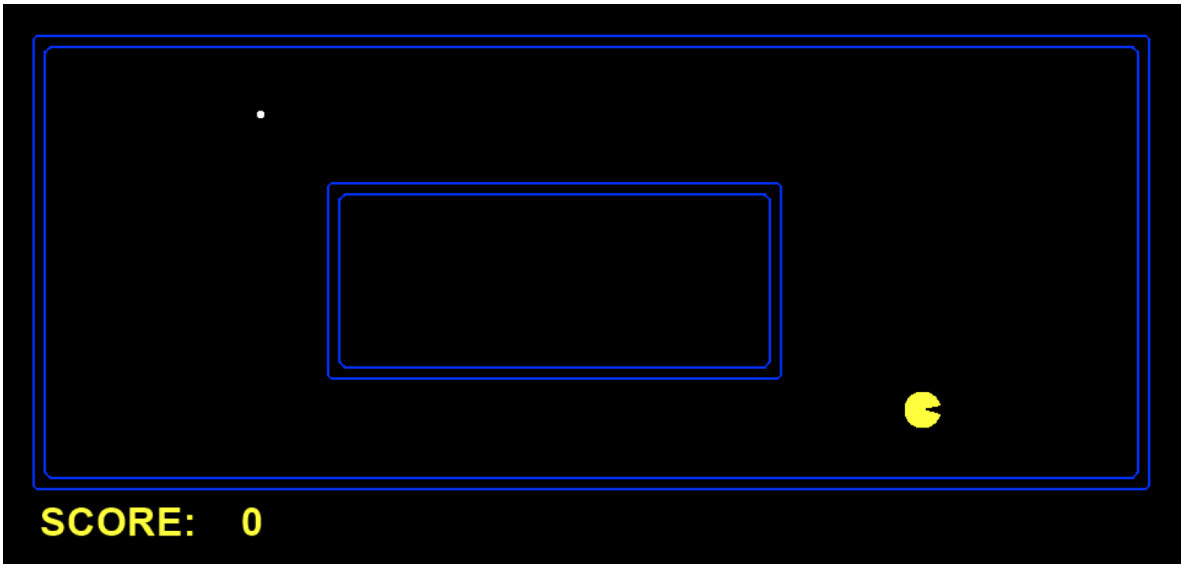


Figure 7.4: Example pacman level where the goal for pacman is to eat the food dot

Secondly, the algorithm is now real-time. We can delay computing the transitions at step $k$ until they are required. This makes the algorithm useful for filtering applications.

There are two main drawbacks of the algorithm. The first is we still need to specify $N$; for problems that terminate earlier than $N$ this can be remedied using absorbing states (section 5.2.1). The second issue is the algorithm requires us to visit all states; this is necessary to obtain the optimal cost; after all, if we fail to visit a certain edge, we cannot know if that edge may have had a large negative cost associated with it. However as we will see in the next chapter, if we make minimal assumptions about the cost function it is possible to derive variants of the forward algorithm which are much more efficient.

### 7.2.3 Example: Shortest-path graph traversal with no restrictions

Consider again the graph traversal problem. In this case the nodes of the graph will be denoted by $\mathcal{S} = \{1, 2, \ldots, N\}$ and $A_{ij}$ is the cost of moving from node $i$ to $j$. We let $A_{ij} = \infty$ when $i, j$ are not connected by an edge.

The goal of the graph-traversal problem is to find a shortest path from a given node $s \in \mathcal{S}$ to $t \in \mathcal{S}$, i.e. a sequence of edges $(i_0, i_1), (i_1, i_2), \ldots, (i_k, t)$. To make the problem well-defined, we allow zero-cost self-transitions $A_{ii} = 0$ and assume there are no cycles (i.e. a path which starts and ends at the same node) with negative cost. This assumptions is required since otherwise we could generate a path of arbitrarily low cost by repeating the negative-cost cycle again and again.

Assuming there is a path from $s$ to $t$, the problem is now solvable in $N$ moves. We

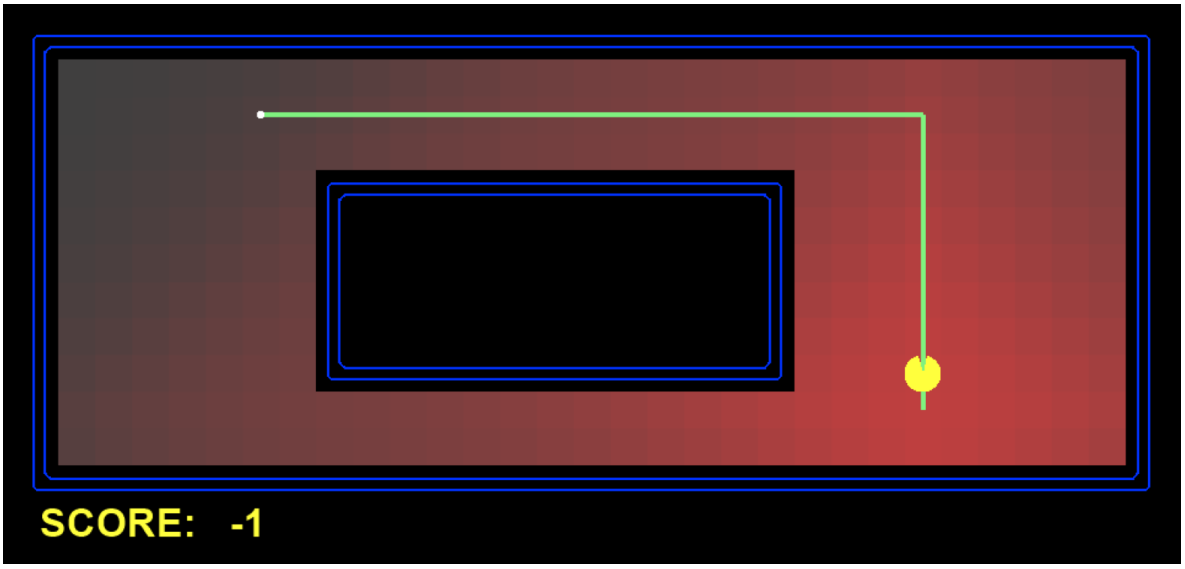


Figure 7.5: Forward DP applied to the food search problem using $N = 250$.

can formulate it as a search problem by defining the transition function as

$$T(i) = \{(j, u_0, A_{ij}) \mid j \in \mathcal{S} \text{ where } A_{ij} < \infty\} \tag{7.19}$$

The goal test is obviously if the current vertex $i$ is equal to $t$. Simply applying the DP algorithm will now find the optimal path with none of the limitations of familiar graph-traversal algorithms such as non-negative cost.

### 7.2.4 Example: Pacman food pellet search

As another example, consider the Pacman level in fig. 7.4. This is a scaled-up version of the zero-ghost example we previously considered in section 6.2.4. Recall Pacman was rather complicated to solve using DP because we had to define all states $\mathcal{S}_k$ iteratively for all $k$ before we could run the algorithm, see eq. (6.26). When we consider Pacman as a search problem this issue goes away: We can easily define the DP transition function $f_k$ as in section 6.2.4, and therefore the transition function $T$ is easy to define. We can now solve any deterministic task using DP search, simply by defining different goal tests. In our example, the goal test is if all food pellets are eaten, but we could define the goal as eating a certain number of pellets or visiting certain squares.

The result can be seen in fig. 7.5. We have inserted the optimal path found by forward-DP search (planning using a horizon of $N = 250$) as the green line, and the squares explored, i.e. a square $x$ is red if $T(x)$ has been invoked by algorithm 5. The shade reflects the order in which Pacman visited them. This visualization is not very helpful for forward-DP search, as it mostly indicate all squares are visited, but when we consider more powerful search methods cutting down on the excess red will be a useful benchmark.

### 7.2.5 Where to go from here?

The DP algorithm provides an optimal solution in the case of noise, and the forward DP algorithm is an implementation of the DP algorithm with minimal requirements on the environment (we only have to specify a transition function) which provides the optimal solution for deterministic environments.

Both methods suffer from two defects: Firstly, that the transition model is often not known but has to be learned – we will consider this in more details in later chapters. Secondly, and from our purpose more importantly, that even if the transition model is known, the two methods are very demanding in terms of memory and computation time. As this problem is more fundamental, we will first study how this problem can overcome by either assuming certain structure to the cost functions (search), or by allowing solutions which are only approximately optimal (multi-agent search and reinforcement learning). These strategies will provide us with methods which are of much greater practical significance than the raw DP algorithm, and the ideas we can study in this setting are re-used under various incarnation in reinforcement learning and control theory.

# Chapter 8

# Search

Search is concerned with finding the path from a start state to a goal state. Potentially, there may be many possible goal states, and the problem is assumed to be deterministic. Efficient search methods are of immense practical utility, and in some cases in ways that are not obvious:

- Path-finding problems (both when using a map or in the real world),
- Robot navigation
- Certain forms of trajectory planning
- Automatic theorem proving (apply axioms in sequence to prove a theorem)
- automatic assembly sequencing
- chip design
- Stocking machines on shop floors
- Computer game AI

The forward DP algorithm is an example of a search algorithm, and it has the benefit of always finding the optimal path for any cost function provided $N$ is large enough. The drawback is that it is computationally inefficient.

In this chapter we will investigate ideas for making the forward DP algorithm more efficient, and our tools will be to either make additional assumptions about the environment, or by allowing sub-optimal paths. Asides introducing us to some of the most important sequential decision methods, the ideas we will see in this section are re-used in both multi-agent systems, control and reinforcement learning.

## 8.1 Search methods

If we take a step back, the forward DP algorithm, algorithm 5, works as follows:

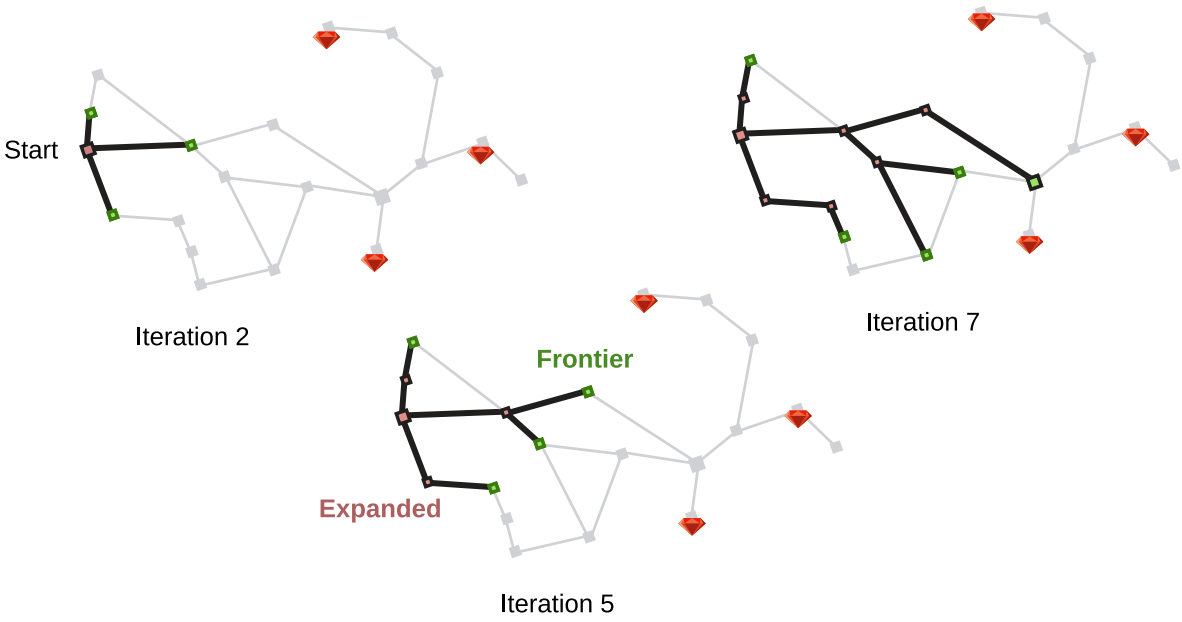


Figure 8.1: Example of how a generic search method solves a problem. Nodes are iteratively expanded (i.e., $T(i)$ is called). Expanded nodes are painted red, and once a node is expanded, the method learns about new potential nodes which can be expanded, here marked by green; the set of such nodes is called the frontier. The job of a search method is to determine which of the nodes in the frontier should be expanded next to find an optimal-cost path to a goal state (indicated by diamonds).

EXPANDED SET

- At step $k$ the algorithm is concerned about the set of states in $\mathcal{S}_k$. These are visited one at a time, and when a state $x_k \in \mathcal{S}_k$ is visited, the transition function $T(x_k)$ is called to obtain the states in $\mathcal{S}_{k+1}$ and update the cost-to-go function $J(x_k)$. We call the set of states which have been expanded, i.e. where $T(x_k)$ has been called, the **expanded set** and it is visualized as the red vertices in fig. 8.2.

FRONTIER

- When a state $x_k$ is expanded using $T(x_k)$, we learn about new states in $\mathcal{S}_{k+1}$). This set of nodes waiting to be expanded is called the **frontier** and are visualized using green in fig. 8.2

- In summary, just at iteration $k$ begins, the frontier consist of $\mathcal{S}_k$, and in iteration $k$ the frontier is moved one step to the nodes $\mathcal{S}_{k+1}$

When we think of a more efficient search algorithm, the foremost choice is the order in which the elements in the frontier is expanded, and when the algorithm terminates. Rather than searching over all the states in $\mathcal{S}_k$, a more efficient search method might begin to expand more promising states first, and which therefore perhaps reach a terminal state earlier. For such a method to be possible we have to make assumptions about the cost function: **The central assumption of this chapter is that the cost function is non-negative**. If this was not the case, it might be that at some high

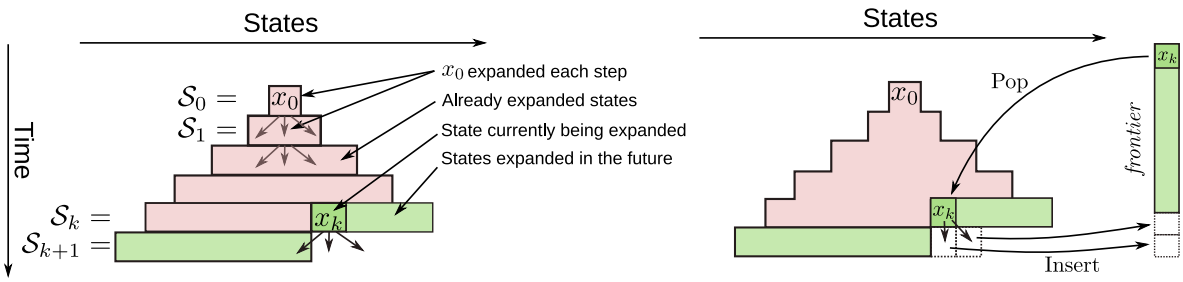


Figure 8.2: Left: Illustration of the forward-DP algorithm algorithm 5, in which the states $\mathcal{S}_{k+1}$ are iteratively defined by exanding states in $\mathcal{S}_k$. Right: This is equivalent to simply defining a queue of states, the frontier queue, where we add states to one end and take states from the other.

☞ **Algorithm 6** FIFO que

```
Maintain a list of items l = [x_1, x_2, ..., x_k] in memory
function POP( )
    l ← [x_2, ..., x_k]                          ▷ Remove first element
    return x_1
end function
function INSERT(x*)
    l ← [x_1, ..., x_k, x*]                      ▷ Append to list
end function
```

value of $k$, there was a transition with an enormously low cost. The next sections will show different ways to make use of that assumption.

### 8.1.1 Frontier queues

As a first step, we note that each set of states $\mathcal{S}_k$ are used in only a very limited way (see fig. 8.2). The states in these sets are used only once when we loop over $x_k \in \mathcal{S}_k$, and they are defined only when we insert states into $\mathcal{S}_{k+1}$. We can visualize this by imagining all state spaces are concatenated into a large list where we take items from $\mathcal{S}_k$ and insert at the very right-most end:

$$(\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{k-1}, \underbrace{\cdots \mathcal{S}_k \cdots}_{\text{Contains current states } x_k}, \ldots) \tag{8.1}$$

We can simplify this by discarding all states to the left of $x_k$ which are never used. Formally, this is equivalent to using a queue-structure, which is just a list of states where all states are added to the right-most end, and states are taken (and removed) from the left-most end.

$$(\underbrace{x}_{\text{Current element } x_k}, x_1, x_2, \ldots, x_m, \underbrace{n'}_{\text{New elements added here}}). \tag{8.2}$$

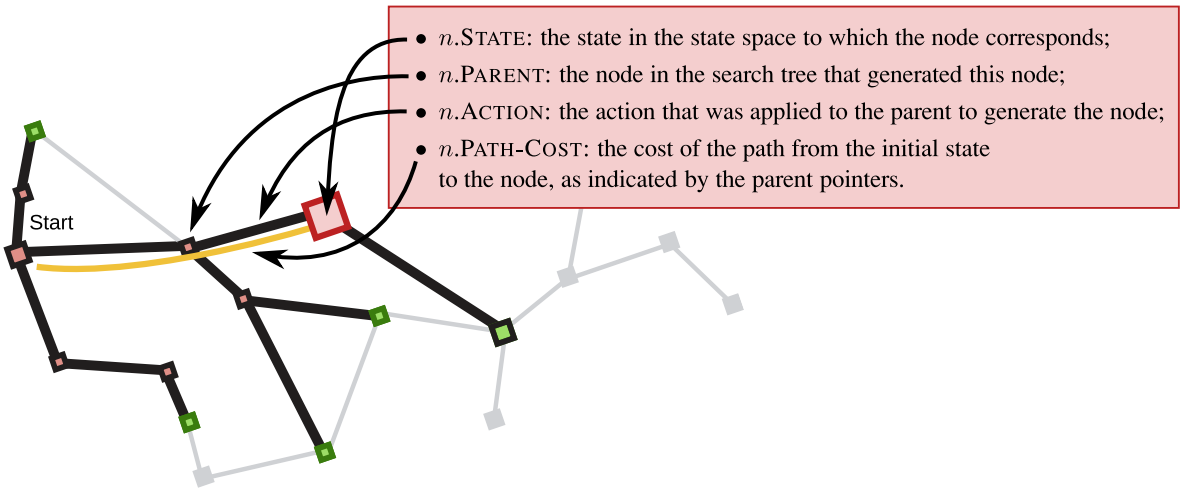


Figure 8.3: A search node is a simple data structure associated with each node in the expanded set (see fig. 8.2). It tracks the current state $x_k$, the parent node, the action required to be taken in the parent to end in the current node, and the current best-cost estimate of the path $J(x_k)$.

QUEUE

In general, a datastructure where we can add and remove elements is called a **queue**, and the particular instance in eq. (8.2), in which the older element is removed, is known as a first-in first-out or **FIFO queue**. Pseudocode can be found in algorithm 6

FIFO QUEUE

### 8.1.2 Search nodes

In the forward-DP algorithm we tracked the path-cost and policy separately. For search algorithms it is common to track these using a single data structure consisting of search nodes. A **search node** $n$ is a data structure associated with a particular state $x$ with the following fields (see fig. 8.3):

SEARCH NODE

- $n$.STATE: The state $x$ to which this node corresponds
- $n$.PARENT: The parent node
- $n$.ACTION: Action to take in $n$.PARENT to get to this node
- $n$.PATH-COST: The path from $s$ to $n$.STATE as indicated by the parent pointers

The parent fields tells us which node immediately proceeded this node on the optimal path from $s$ to $n$, and the action field which action to take in the parent node to arrive at $n$. Together, these allows us to recursively backtrack from a given node towards the starting state, and thereby get the optimal path. Meanwhile, the PATH-COST field maintains the cost of the (optimal) path from $s$ to $n$, corresponding to $J_0(s)$.

As a small warmup, let's try to restructure the algorithm 5 using search nodes and a FIFO queue. The result can be found in algorithm 7. Note that to turn this formulation

**Algorithm 7** Sketch of re-formulation of algorithm 5 using queues and nodes

*frontier* $\leftarrow$ an empty queue of search nodes to explore
$n_0 \leftarrow$ NODE(state$=s$, cost$=0$, action$=\emptyset$, parent$=\emptyset$)
*frontier* $\leftarrow$ INSERT(*frontier*, $n_0$)
**while** True **do**
    $n =$ NODE(state$=x$, cost$=C$, action$=u_p$, parent$=n_p$) $\leftarrow$ POP(*frontier*)
    **for all** $(x', u, c) \in T(x)$ **do**
        **if** there is no node $n' \in$ *frontier* s.t. $n'$.STATE $= x'$ **then**
            $n' \leftarrow$ NODE(state$=x'$, cost$=c+C$, action$=u$, parent$=n$)
            *frontier* $\leftarrow$ INSERT(*frontier*, $n'$)
        **end if**
        **if** there is a node $n' \in$ *frontier* s.t. $n'$.STATE $= x'$ and $c + C < n'$.cost **then**
            $n'$.PATH-COST $\leftarrow c + C$ ▷ Found a better path to $n'$, update
            $n'$.ACTION $\leftarrow u$
            $n'$.PARENT $\leftarrow n$
        **end if**
        (Keep track of nodes $n$ corresponding to terminal states)
    **end for**
**end while**

into a practical method, we would need to keep track of the nodes corresponding to terminal states, and at the end of the method return the terminal node with the smallest cost.

### 8.1.3 Breadth-First search

Let's do a bit of exploration of the new notation and how we can find a more efficient search method. To this end, consider again the Pacman food-pellet problem from section 7.2.4. The forward-DP solution is re-produced in fig. 8.4, and recall the red tiles $(x, y)$ indicate which states have been expanded (similar to fig. 8.2), and the shade corresponds to the order in which the expand-function $T$ is called. In this problem, each step has a fixed cost of $c = 1$, and we will for now assume the cost-per-step is a positive constant. We can speed up the method in two ways:

- Since each step has the same, constant cost $c$, the first time we get to a goal state there is no need to keep search and we can terminate the method.
- There is no need to expand the same state twice: Once we have found a path to a state in $k$ steps, i.e. with cost $ck$, any other path to that state will either have cost $ck$, or a greater cost than $ck$ (in the pacman-example, this can occur if Pacman backtracks).

To implement the later item, we will keep track of a list of visited states, and only add states to the frontier if they have not been visited. With these modifications we obtain algorithm 8:

☞ **Algorithm 8** Generic graph-search algorithm. Algorithm 7 but tracking visited states

```
1: visited ← empty list of visited states
2: frontier ← an empty queue of search nodes to explore
3: n_0 ← NODE(state=s, cost=0, action=∅, parent=∅)
4: frontier ← INSERT(frontier, n_0)
5: while True do
6:     n = NODE(state=x, cost=C, action=u_p, parent=n_p) ← POP(frontier)
7:     if x not in visited then
8:         Append x to visited
9:         if x is the terminal state then
10:            return n
11:        end if
12:        for all (x', u, c) ∈ T(x) do
13:            if x' not in visited then                ▷ Don't re-visit nodes
14:                n' ← NODE(state=x', cost=c + C, action=u, parent=n)
15:                frontier ← INSERT(frontier, n')
16:            end if
17:        end for
18:    end if
19: end while
```

BREADTH-FIRST SEARCH

When the method is used with a FIFO queue, the resulting method is called **breadth-first search**, and it will be optimal in the case where the cost of transition is constant. The same method applied to the Pacman search problem can be found in fig. 8.4 (right).

### 8.1.4 Search performance

We have only loosely talked about search performance, however to compare different search methods we need a slight bit of notation. For a search problem, one usually speaks about

DEPTH

- The **depth** $d$ of a problem we mean the shortest number of steps from the root to a terminal node. For the Pacman food pellet problem this is 26

BRANCHING FACTOR

- The **branching factor** $b$ refers to the to the maximum number of successors any node can have. In the Pacman game it is 5 (number of actions)
- Finally, $m$ will refer to the maximum length of any path; for the Pacman game $m$ is infinite

And for a given search method we say

COMPLETE

- It is **complete** if it is guaranteed to find a solution if there is one

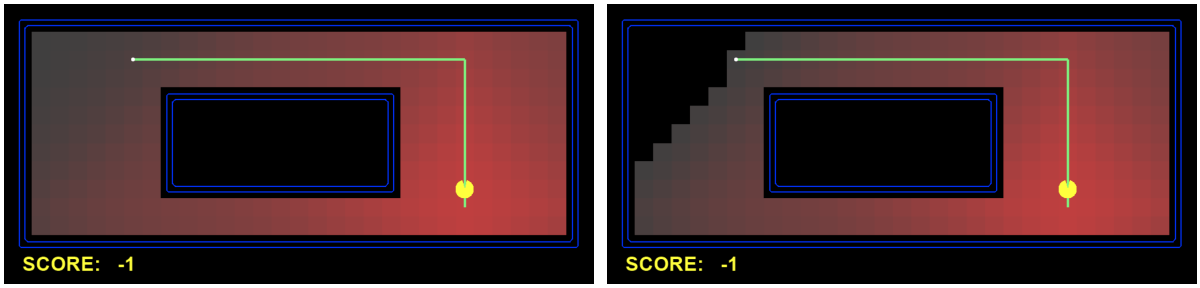


Figure 8.4: Left: Forward DP applied to the food search problem from section 7.2.4 using $N = 250$. The red nodes are the expanded set. Since the cost per step is uniform, pacman could have terminated the first time he arrived at the food pellet. Right: Search-node forward DP from algorithm 5 applied to the food search problem from section 7.2.4 using $N = 250$. The method is still optimal, and terminates as soon as the goal state is found, thereby saving computation time.

| Search method | Cost of path | Search nodes expanded | Unique search nodes expanded |
|---|---|---|---|
| DP (N=250) | 26.0 | 28556.0 | 241.0 |
| BFS | 26.0 | 214.0 | 215.0 |
| DFS | 142.0 | 188.0 | 189.0 |
| A* (Euclid) | 26.0 | 121.0 | 122.0 |
| A* (Manhattan) | 26.0 | 92.0 | 93.0 |

Table 8.1: Summary of search cost in the Pacman food pellet problem

OPTIMAL

- It is **optimal** if the solution is optimal (i.e., has lowest possible cost)
- Efficiency measured as time-complexity, i.e. the number of nodes that are expanded

Breadth-first search is both complete and optimal, assuming all transitions have the same cost. However breadth-first search is not very efficient. Suppose the problem has depth $d$, then an upper-bound on the number of expansions can be found by assuming the first node has $b$ successors, each of these have $b$ successors (hence, $b^2$ nodes), and so on up to $d$. This gives a total cost of

$$b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d) \tag{8.3}$$

This means the cost is exponential in $d$, and usually makes breadth-first search infeasible if $d$ is of the order of 10. Note this is a worst-case analysis, and for the Pacman game it is not quite as bad. What saves us is that the maximum number of expansions are limited by the number of states, and so we only need to perform 214 expansions (the result of the search is summarized in table 8.1. Note this number is very substantially smaller than the DP search method; what saves us is the early termination, and that we maintain a list of visited nodes.

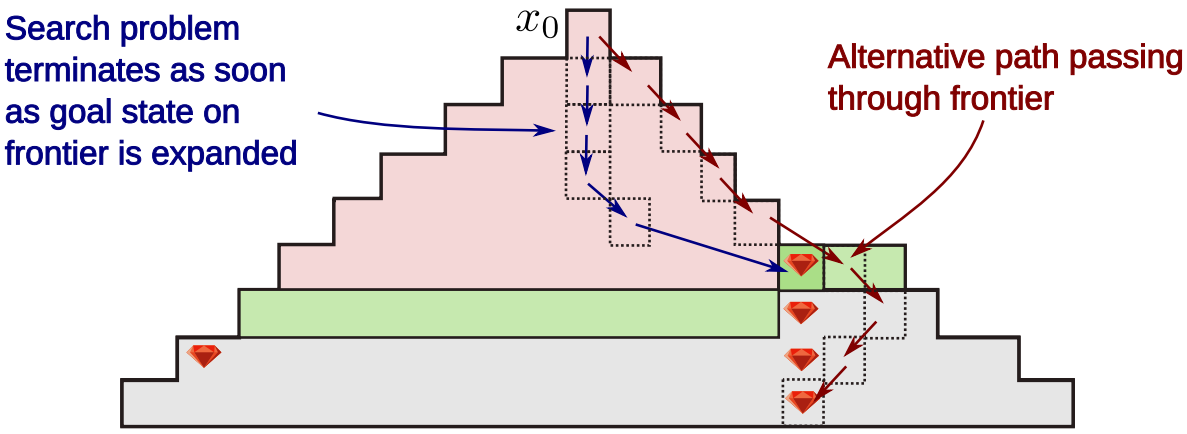


Figure 8.5: Optimality of uniform-cost search. When a node (diamond, green background) is expanded by uniform-cost search, the path (blue) must be optimal. Otherwise, there would be another path to the node (red), which would necessarily have to pass through the frontier. But if so, the place it pass through the frontier would have a higher cost than the current node (because otherwise we would have selected it for expansion) and therefore it cannot have a lower total cost.

## 8.2 Uniform cost search

Breadth-first search is easily generalized to the more interesting case where all transitions have a non-negative cost. Suppose that the queue operates by not taking the first-inserted item, but rather the item with the currently lowest estimated cost. In other words, we always expand the node $n$ on the frontier with the smallest value of $n$.PATH-COST. This can be implemented by only modifying the queue in algorithm 8 to a **priority queue** as indicated in algorithm 9

PRIORITY QUEUE

Uniform cost search is still optimal, and the proof is guaranteed by the following argument:

- If a node $n$ is selected for expansion, the optimal path to that node must have been found: If this was not the case, there would have to be a frontier node on the optimal path from the start to $n$. But in that case this node would need to have a lower cost, and hence it would have to be selected first by the priority queue
- Because the step costs are nonnegative, paths only increase in cost as more nodes are added. Hence if a goal node is selected for expansion it must be optimal

The argument is also illustrated in fig. 8.5. Suppose the blue path to the node with the diamond is not optimal, and suppose the red path is in fact the optimal one. In that case it must pass through the frontier. But since we always select the lowest element of the frontier for expansion, and since the path increase in cost with more edges, it cannot in fact be optimal.

The advantage of uniform-cost search is that it will find the optimal path in the general case, however the time complexity will depend on the cost structure and how

☞ **Algorithm 9** Priority queue

Maintain a list $l = [(x_1, c_1), (x_2, c_2), \ldots, (x_k, c_k)]$ of items and their cost memory
**function** POP(d)
    Find element $x_i$ with lowest cost $c_i$
    $l \leftarrow [(x_1, c_1), \ldots, (x_{i-1}, c_{i-1}), (x_{i+1}, c_{i+1}), \ldots, (x_k, c_k)]$ ▷ Remove element with lowest cost
    **return** $x_i$
**end function**
**function** INSERT($x^*$, $c^*$)
    **if** There is an $(x_i, c_i)$ in the list where $x_k = x^*$ and $c^* < c_i$ **then**
        Overwrite $x_k$ with $x^*$, and set $c_i = c_i^*$
    **else**
        Otherwise just append $(x^*, c^*)$ to the list.
    **end if**
**end function**

☞ **Algorithm 10** Uniform cost frontier

Let $Q$ be a priority queue
**function** POP
    **return** POP($Q$) ▷ Pop element with lowest cost-to-go
**end function**
**function** INSERT($n$)
    INSERT($Q, n, n$.PATH-COST) ▷ Insert node with priority equal to node cost
**end function**

well a greedy strategy will lead to the goal. When all edges have the same cost, the method is equivalent to breadth-first search.

## 8.3 Depth-first search

Depth first search is an approximate search method and makes no claim for optimality. What depth-first search does is it searches blindly, and as quickly as possible, to the deepest possible level; the difference is illustrated in figs. 8.6 and 8.7. If it finds a terminal state it immediately terminates, and otherwise it backtracks, with preference to backtracking as little as possible, and once more searches to the deepest level. This strategy can be implemented by simply letting the frontier be a so-called last-in first-out

LIFO QUEUE

or **LIFO queue**, which is characterized by expanding the most recently added node, see algorithm 11.

DFS is complete assuming the problem has a finite number of states, but will obviously not find the shortest path (see fig. 8.8). The worst-case performance is also quite poor, and it may search all possible nodes before finding the optimal solution, even

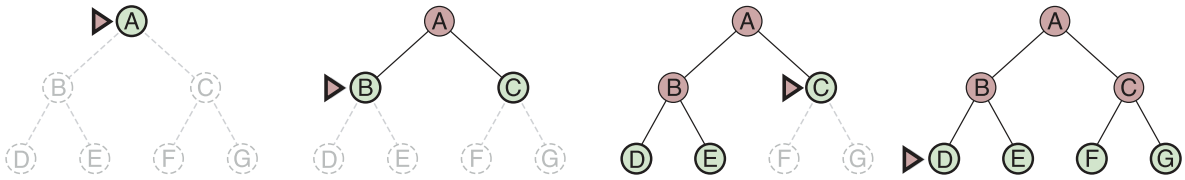


Figure 8.6: Illustration of breadth-first search. The frontier is expanded at each step starting from the initial node $A$

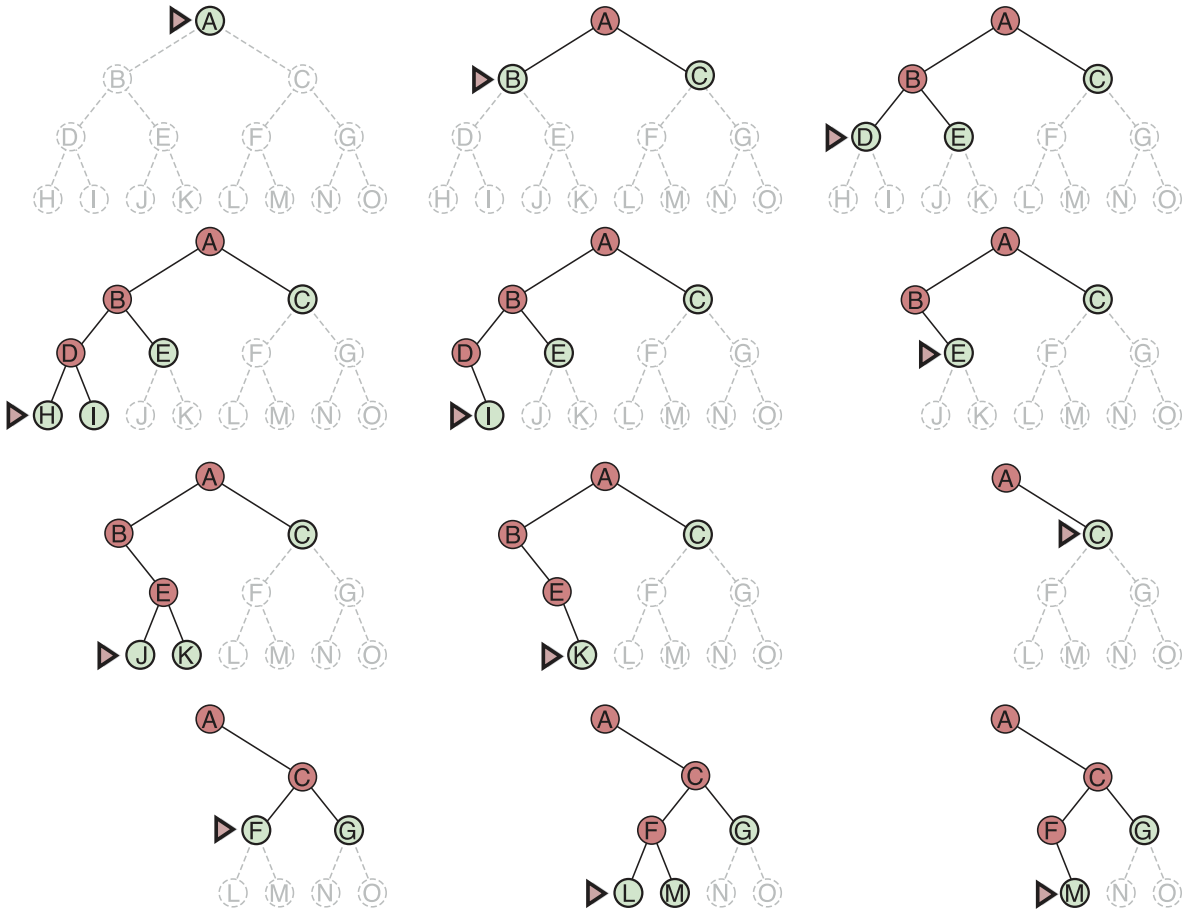


Figure 8.7: Illustration of depth-first search, starting from $A$ and attempting to find $F$. Depth-first search searches towards the deepest part of the problem first, and then tries to backtrack.

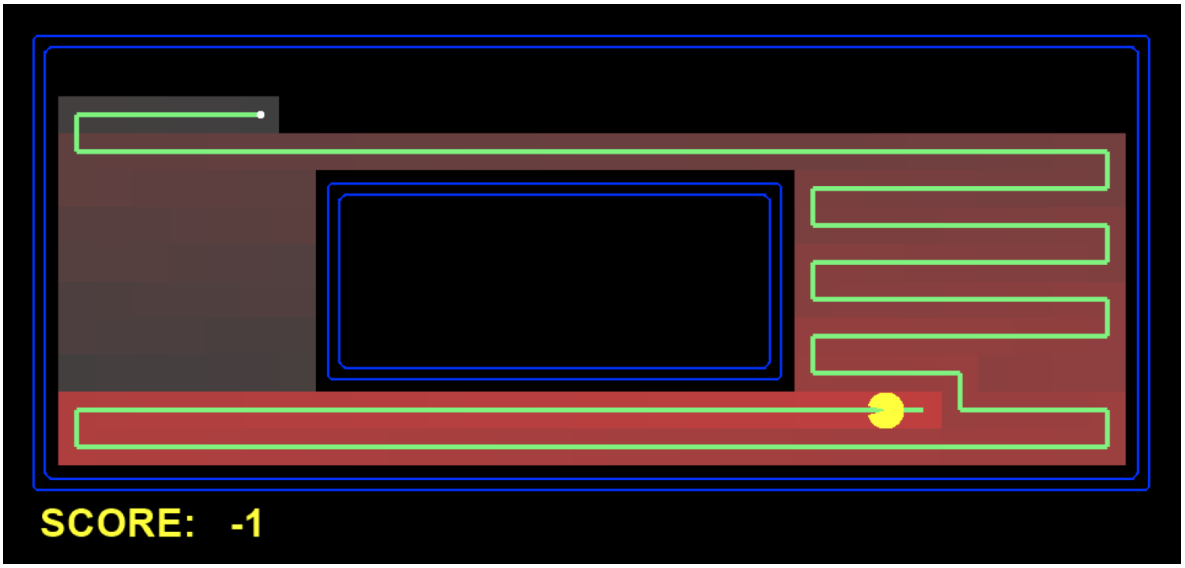


Figure 8.8: Depth-first search applied to the Pacman food pellet problem. A path is found, but it is obviously sub-optimal.

☞ **Algorithm 11** LIFO queue

```
1: Maintain a list of items l = [x_1, x_2, ..., x_k] in memory
2: function POP
3:     l ← [x_1, x_2, ..., x_{k-1}]                ▷ Remove last element
4:     return x_k
5: end function
6: function INSERT(x* )
7:     l ← [x_1, ..., x_k, x*]                     ▷ Append to list
8: end function
```

though the optimal solution is only one step from the initial state. Since it can search to maximum depth before finding the solution, the performance is upper-bounded by the branching factor raised to the power of the maximum depth:

$$b + b^1 + \cdots + \cdots b^m = \mathcal{O}(b^m) \tag{8.4}$$

There are several ways to improve on depth-first search, for instance by first letting it search to a fixed depth $h$ which is progressively improved; an interested reader can consult [RN09] for more information.

These shortcomings – that depth-first search is not complete and may visit all nodes before finding the optimal path – might make it seem irrelevant, however depth-first search is commonly used in practice, since DFS allows us to search deep at a limited memory. This is in particular relevant for situations where any solution can be expected to be optimal or good-enough (a theorem proving system is an example of this).

☞ **Algorithm 12** $A^*$ frontier queue

```
1: Let Q be a priority queue
2: Let h be the heuristic function
3: function POP
4:     return POP(Q)                    ▷ Simply call the pop method of the priority que
5: end function
6: function INSERT(n)
7:     f_n ← n.PATH-COST + h(n.STATE)   ▷ Compute f(n) using the heuristic
8:     INSERT(Q, n, f_n)                ▷ Insert node with given priority
9: end function
```

## 8.4 Structured search and $A^*$ ✸

INFORMED SEARCH

**Informed search** refers to the situation where we have external, problem-specific knowledge to help us select which nodes to expand. Informed search is generally *much* more efficient than uninformed search, and the most widely used variant, $A^*$, has all the nice properties (completeness and optimality), works for any non-negative cost function, and can scale to solve highly complex search tasks such as navigation in e.g. videogames.

The idea is quite obvious. Consider the BFS method applied to Pacman, as shown in fig. 8.4. The BFS solution searches away from the starting location with no preference to any direction. This is obviously a bad idea: Pacman should give preference to states that are closer to the food pellet, and in particular there is no reason to walk towards the lower-right corner. This is what we want to *inform* the search method about.

### 8.4.1 Heuristic functions

EVALUATION FUNCTION

Our approach is very similar to uniform-cost search, but with a small twist. We assume there is an **evaluation function** $f$ which evaluate the search nodes $n$ in the frontier, such that $f(n)$ is an cost estimate of how good it would be to expand this node, and the node in the frontier with the lowest cost estimate is expanded first. In uniform-cost search, $f$ was simply the current path-cost estimate, but by changing $f$ we can tell Pacman to never expand more distant nodes.

The optimal choice of $f$ would be the cost of the shortest path from $s$ to $t$ which passed through $n$. In this case, we would always expand nodes on the optimal path, and reach our destination without ever expanding a suboptimal path. We can immediately get some of the way: We already know that the optimal path from $s$ to $n$ has a cost of $n$.PATH-COST, so all we need is an approximation of the cost from $n$ to $t$; this approximation is written as $h(n)$ where $h$ is called a **heuristic function**. This gives the equation:

HEURISTIC FUNCTION

$$f(n) = n.\text{PATH-COST} + h(n). \tag{8.5}$$

$A^*$ SEARCH

A search method which select nodes according to such a criteria is known as $A^*$ **search**.

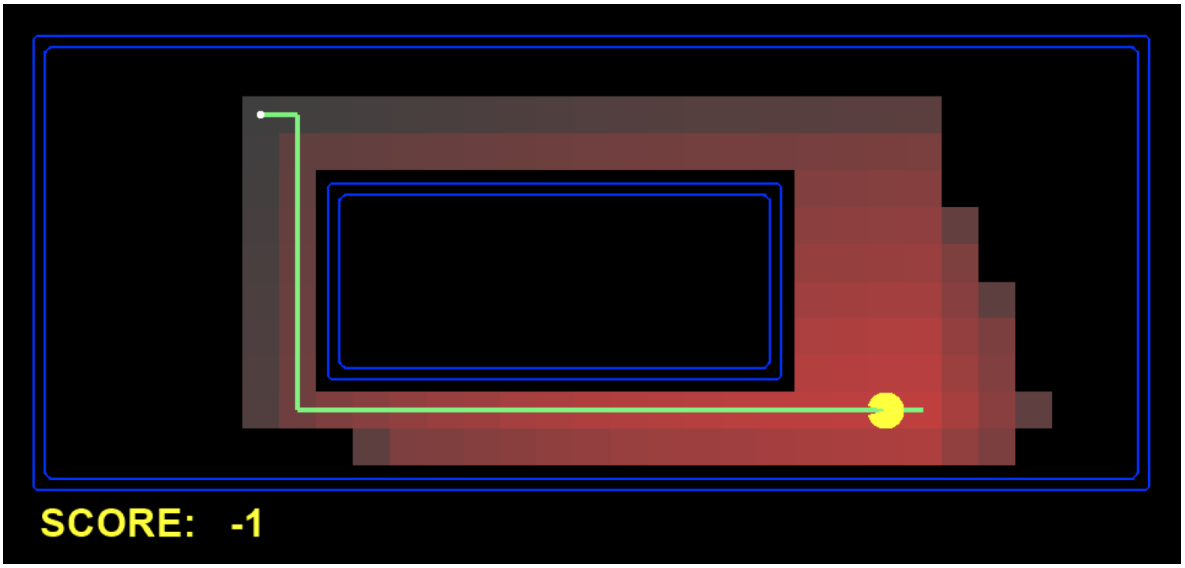


Figure 8.9: $A^*$ search using a Euclidean distance heuristic

Implementing this idea is very easy: We re-use our master search method from algorithm 8, and modify the frontier queue we used in uniform-cost search (section 8.2) to use $f$. The result can be seen in algorithm 12

### 8.4.2 Heuristic functions

As mentioned, the heuristic function $h(n)$ should estimate the distance from $n$ to the goal, and intuitively the better this estimate is the faster $A^*$ will find the correct solution. However, for $A^*$ to be complete and optimal we need a few more conditions on $h$.

The most obvious criteria is that the heuristic $h(n)$ never overestimates the cost the cost from $n$ to $t$, in other words the heuristic should be optimistic. If this is true, the heuristic is said to be an **admissible heuristic**.

ADMISSIBLE HEURISTIC

The reason why this is important can be grasped intuitively: Suppose our heuristic sometimes over-estimated the cost from $n$ to the goal. That would mean it could grossly over-estimate the cost of a node $n$ on the optimal path, and fail to expand it. Suboptimal paths could meanwhile reach the goal, and the method will terminate sub-optimally in line 10 of algorithm 8. In practice this criteria is easy to satisfy; in the Pacman example, we can simply use the Euclidean distance from a state $s = (x, y)$ to the goal pellet at $t = (x_g, y_g)$

$$h(n = (x, y)) = \sqrt{(x - x_g)^2 + (y - y_g)^2} \tag{8.6}$$

which is obviously optimistic since Pacman cannot walk diagonally and at any rate must walk around the hole in the middle. The states expanded can be seen in fig. 8.9 and from table 8.1 we see the path is still optimal, and substantially fewer nodes are expanded.

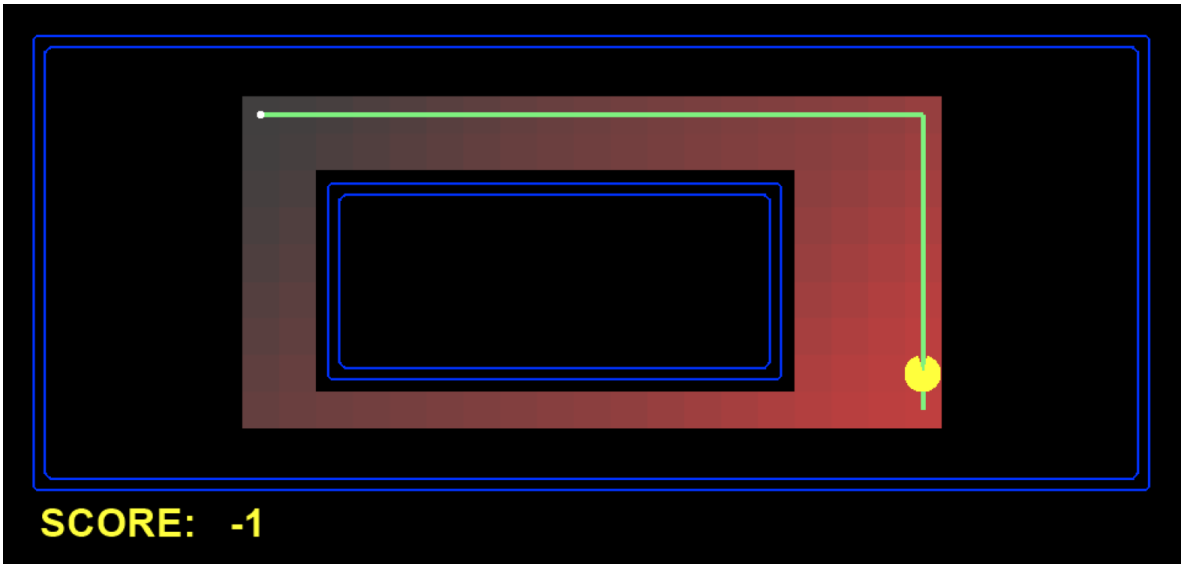


Figure 8.10: $A^*$ search using a Manhattan distance heuristic

Admissibility is not quite enough to guarantee optimality. In order for $A^*$ to be optimal, the heuristic must be consistent. Consistency carries the same intuition as the triangle inequality: For any node $n$, any successor $n'$, we suppose the shortest path between $n$ and $n'$ is $c_{n,n'}$. A Heuristic is then **consistent** if:

CONSISTENT

$$h(n) \leq c_{n,n'} + h(n'). \tag{8.7}$$

This condition is natural if we recall the heuristic is supposed to be optimistic: In that case the true cost of the path to $n'$ from $n$, $C_{n,n'}$, should always disappoint the heuristic. In reality, most reasonable heuristics which are admissible will also be consistent, and it is for instance the case by the Euclidean heuristic which can easily be seen from the triangle inequality.

When selecting heuristics, one should look for heuristics which are admissible, consistent, and which are as close to the true distance as possible. For Pacman, an even better choice is the Manhattan distance

$$h(n = (x, y)) = |x - x_g| + |y - y_g|^2 \tag{8.8}$$

which is the actual distance from the start to the goal state. The result is shown in fig. 8.10, and in this case Pacman expands no unnecessary nodes.

# Chapter 9

# Multi-agent systems

This chapter will consider the dynamical programming problem in the non-deterministic case, i.e. where the dynamics is of the form

$$x_{k+1} = f_k(x_k, u_k, w_k) \tag{9.1a}$$
$$w_k \sim P_k(W_k|x_k, u_k). \tag{9.1b}$$

This problem has already been optimally solved in chapter 6, however the dynamical programming algorithm in the most general form is of limited applicability due to the computational requirements (the optimal dynamical programming agent for Pacman should have convinced us of this much) and the assumption the noise model, eq. (9.1b), is known. This chapter will consider two ways of making the DP algorithm more practical

- Firstly, the computational requirements, specifically planning on finite horizon as in section 6.3.3
- Secondly, we will consider the situation where the noise model is *not* known. Our primary example will be a chess game, where the noise model is the opponents moves, and the overall approach is to assume the opponent always plays the best move.

We will consider multi-agent games, which is particularly welll suited for the methods, which will lead us to understand e.g. a chess playing engine such as Deep Blue. The methods have applications beyond video-game playing. In control theory, the methods are very similar to model-predictive control (MPC), and in reinforcement learning they offer an important pre-cursor to Monte-Carlo decision trees.

## 9.1 Multi-agent games

COMPETITIVE GAMES

Multi-player games encompas a number of problem formulation, but we will restrict our attention to the following situation. Firstly, we will focus on **competitive games**, in which we struggle against (rather than corporate with) one or more opponents. We

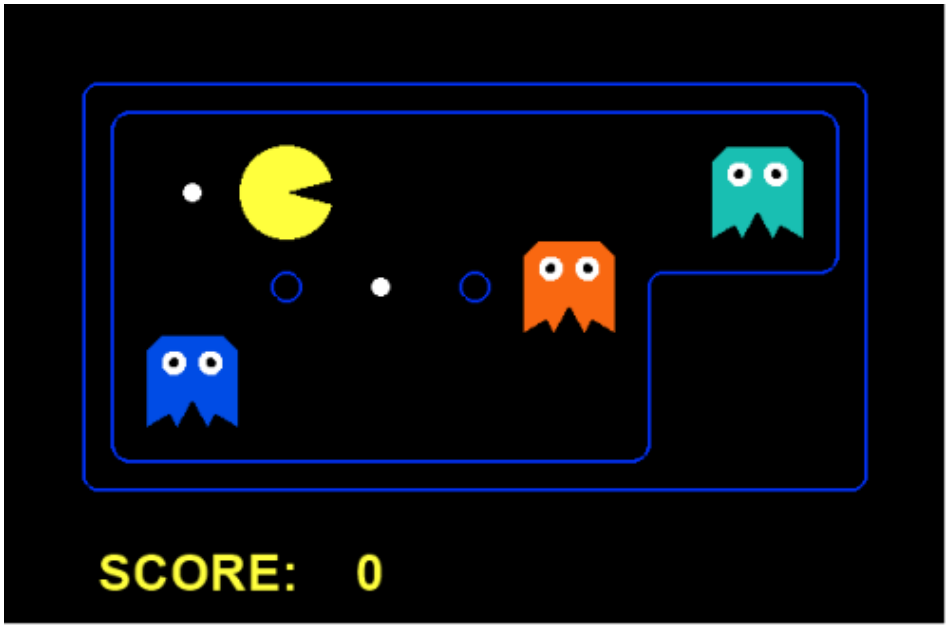


Figure 9.1: Example of a multi-agent game problem. Pacman must capture the two food pellets while avoiding the three ghosts

assume the game world is fully known, and that the game is zero-sum, which means if we do well our opponents do poorly and visa-versa. Pacman and chess are examples of this setting, but not poker since it is not fully observed.

Consider for a moment the game of chess, and suppose we play as white. The state $x_k$ would correspond to the board position at (our) turn, the actions $\mathcal{A}_k(x_k)$ would be all legal moves, $u_k$ would some particular move of a piece, and the noise disturbance, $w_k$, would correspond to our opponents move *conditional* on the board state that arose after we played move $u_k$ in position $x_k$. Hence, the update:

$$x_{k+1} = f_k(x_k, u_k, w_k) \tag{9.2}$$

encapsulates *two* decisions, namely both our and the opponents moves (this is called two **ply** in common terminology), and the second decision, the opponents move, is similar to ours.

PLY

This situation is exacerbated in e.g. Pacman, where $w_k$ consist of the moves of all the other ghosts. Since we want to describe the decision problem from the *ghosts* perspective (or black, in the chess game) we will re-formulate the problem to be more symmetric. The definition of a game, or equivalently a multi-agent search problem, is given in theorem 9.1.1

GAME

**Definition 9.1.1.** *Multiagent search problem Formally, we define a turn-based* ***game*** *with $Q$ players as consisting of:*

- $x_0$*: The initial state of the game*

- *$x$: A game state, belonging to a state of spaces $\mathcal{S}$. It is assumed that $x$ contains information about whose turn it is to play*
- PLAYER($x$)*: Return which players turn it is to move*
- $\mathcal{A}(x)$*: The available actions for the current player*
- $f(x, u)$*: The transition model which computes the result of playing move $u \in \mathcal{A}(x)$ in state $x$*

TERMINAL TEST

- TERMINAL-TEST($x$) *A **terminal test** which returns true if the game is over. If the terminal test is true $x$ is also called a terminal state*

UTILITY FUNCTION

- UTILITY($x, p$)*: The **utility function** which defines the* final *numerical value for a game which ends in state $x$ for player $p$. This may be different than the score-functions*
- *We follow the convention that player $p = 0$ refers to us – i.e. we want to maximize the utility of player 0 – and players $p = 1, \ldots, Q-1$ are the opponents.*

With this terminology, we can define what it means for the game to be zero-sum as saying the final utility for a terminal state $x$ must be constant:

$$\text{Constant } = \sum_{p=0}^{Q-1} \textsc{Utility}(x, p). \tag{9.3}$$

The later is not a strict requirement since it can be ensured by demanding that if one player obtains a positive reward, the other players obtain a small negative reward: $\sum_{p=0}^{Q-1} \textsc{Utility}(x, p) = 0$ or by simply letting the winner get a score of $+1$ at the end of the game and all other obtain a score of 0.

## 9.2 Expectimax

Expectimax is simply another name for the DP algorithm, in the finite-horizon formulation, applied to a multi-player game. It is easier to first discuss the DP formulation and only later introduce the specific notation of a multi-agent problem above.

Our starting point will be the same setup as the multi-ghost pacman example described in section 6.2.5. Recall we consider a game with $Q$ players (pacman and $Q-1$ ghosts), the game state was $x_k$, pacmans move was $u_k$, and all the ghosts moves were $w_k$. As we saw in the finite-horizon formulation, section 6.3.3, optimally planning on a long horizon of $N$ is equivalent to planning on a horizon of $d$ assuming the terminal cost is $J_d^*$ (the optimal cost function for the DP problem starting at step $d$)

$$J^*(x_0) = \underset{\mu_0,\ldots,\mu_{d-1}}{\arg\min}\, \mathbb{E}\left[ J_d^*\left(x_{k+1}\right) + \sum_{k=0}^{d-1} g_k\left(x_k, \mu_k\left(x_k\right), w_k\right) \right] \tag{9.4}$$

**Algorithm 13** Finite-horizon DP

```
function POLICY(x)                          ▷ Return (approximately) optimal action in state x_0
    return arg min_{u∈A(x)} E[g(x, u, w) + J-STAR(f(x, u, w), d − 1)]
end function
function J-STAR(x, i)                       ▷ Evaluate J*_{d−i} in the finite-horizon problem
    if i = 0 then
        return J̃(x)                         ▷ Evaluating J_d: return the terminal cost approximation
    else                                    ▷ Do a single DP step
        return arg min_{u∈A(x)} E[g(x, u, w) + J-STAR(f(x, u, w), i − 1)]
    end if
end function
```

The first idea is that instead of using $J_d^*$, which we cannot compute, we will use an approximation which we can compute $\tilde{J}$:

$$J_k^* \approx \tilde{J} \tag{9.5}$$

This will allow us to quickly obtain the optimal policy at step $k = 0$ by planning in the $d$-long problem. Then, after we take the optimal action (and the ghosts make their move) the game transition to state $x_1$ and we plan on a new $d$-long problem starting in $x_1$ and once more with terminal cost $\tilde{J}$.

The second idea is to implement a recursive version of the DP algorithm. I will assume the various functions are stationary, i.e. the pacman game-rules stay the same throughout a game, and in this case the optimal action in state $x_0$ is computed as:

$$u^* = \arg\min_{u \in \mathcal{A}(x_0)} \mathbb{E}_{w_0} \left[ g(x_0, u, w_0) + J_1(f(x_0, u_0, w_0)) \right] \tag{9.6}$$

In turn, we can imagine the functions $J_k^*$ are implemented recursively by considering the left-hand side of the DP update as being computed by evaluating, by a function call, the $J_{k+1}^*$ function on the right-hand side:

$$J_k^*(x_k) = \min_{u \in \mathcal{A}\, x_k} \mathbb{E}_{w_k} \left[ g(x_k, u_k, w_k) + J_{k+1}^*(f(x, u_k, w_k)) \right]. \tag{9.7}$$

If we put these two ideas together we obtain the method shown in algorithm 13 for approximately optimal planning in a state $x_0$ The recursive formulation of DP is much less inefficient, as we may end up evaluating the same states many times. The advantage it is that it is easier to implement. The rest of the section will consist of adapting this method to multi-agent games.

### 9.2.1 Formulating the opponents choice as a DP update

Consider the game of chess. Suppose the board state is $x_k$ in turn $k$, and we play move $u_k$. The opponent will then see the game state $x'_k = f(x_k, u_k)$. Based on this she will

select her move $u'_k$ (this will be equivalent to $w_k$) and the game state in the next turn will be

$$x_{k+1} = f(x'_k, u'_k) = f(f(x_k, u_k), u'_k)$$

In other words, in this case the DP transition model relates to $f$ as

$$x_{k+1} = f_k(x_k, u_k, w_k) = f(f(x_k, u_k), u'_k) \tag{9.8a}$$
$$u'_k \sim P_k(\cdot|x_k, u_k) = P(\cdot|f(x_k, u_k)) \tag{9.8b}$$
$$P(u|x) = \{\text{Chance player } p = \textsc{Player}(x) \text{ play move } u \text{ in state } x\}\,. \tag{9.8c}$$

This can be generalized to a game of $Q$ players as follows. In a given turn $k$, we let $x'_p, u'_p$ be the state and action of player $p$. In this way a single move is broken up into the $Q$ plys:

$$\underbrace{(x_k, u_k) = (x'_0, u'_0)}_{\text{Our turn/move}}, \underbrace{(x'_1, u'_1), (x'_2, u'_2), \ldots, (x'_p, u'_p), \ldots, (x'_{Q-1}, u'_{Q-1})}_{Q-1 \text{ opponents states/moves}}, \underbrace{(x'_Q, u'_Q) = (x_{k+1}, u_{k+1})}_{\text{Our turn/move}}$$

In this representation, $x'_{p+1} = f(x'_p, u'_p)$. If we assume the cost function can be computed on a per-ply basis

$$g_k(x_k, u_k, w_k) = \sum_{q=0}^{Q-1} g_q(x'_q, u'_q) \tag{9.9}$$

a single step fo the DP algorithm can be re-written as:

$$\begin{aligned} J_k(x_k) &= \min_{u_k} \mathbb{E}_{u'_1,\ldots,u'_{Q-1}} \left[g_k(x_k, u_k, w_k) + J_{k+1}(x_{k+1})\right] \\ &= \min_{u_k} \left[\mathbb{E}_{u'_1}\mathbb{E}_{u'_2}\cdots\mathbb{E}_{u'_{Q-1}} \left[g_0(x'_0, u'_0) + \cdots + g_{Q-1}(x'_{Q-1}, u'_{Q-1}) + J_{k+1}(x_{k+1})\right]\right] \\ &= \min_{u_k} \left[g_0(x'_0, u'_0) + \mathbb{E}_{u'_1}\left[g_1(x'_1, u'_1) + \mathbb{E}_{u'_2}\left[\cdots \underbrace{\mathbb{E}_{u'_{Q-1}}\left[g_{Q-1}(x'_{Q-1}, u'_{Q-1}) + J_{k+1}(x_{k+1})\right]}_{= J_{k,Q-1}(x'_{Q-1})}\right]\right]\right] \end{aligned} \tag{9.10}$$

If we focus on the inner-most expectation, it looks very similar to the standard DP expectation. If we define $J_{k,Q} = J_{k+1}$ we may define $J_{k,q}$, for $q = 0, \ldots, Q-1$, as:

$$J_{k,q}(x'_q) = \mathbb{E}_{u'_q}\left[g_q(x'_q, u'_q) + J_{k,q+1}(f(x'_q, u'_q))|x'_q\right] \tag{9.11}$$
$$= \sum_{u'_q \in \mathcal{A}(x'_q)} P_q(u'_q|x'_q)\left[g_q(x'_q, u'_q) + J_{k,q+1}(f(x'_q, u'_q))\right] \tag{9.12}$$

We can iteratively plug in eq. (9.12) into eq. (9.10) to see that

$$J_k(x_k) = \min_{u \in \mathcal{A}(x)} J_{k,1}(f(x_k, u)) \tag{9.13a}$$

**Algorithm 14** Simplified Expectimax search

**function** J-STAR($x$, $q$, $d$) ▷ Evaluate state $x$ given it is player $q$'s turn
  **if** $d = 0$ **then** ▷ Evaluate the state
    **return** $\tilde{J}(x)$
  **end if**
  $q' \leftarrow (q + 1 \mod Q)$ ▷ Next agent. If $q = Q - 1$ this will be $q' = 0$
  **if** $q = 0$ **then** ▷ Players turn, eq. (9.13a)
    **return** $\min_{u \in \mathcal{A}(x)} [g_q(x, u) + \text{J-STAR}(f(x, u), q', d - 1)]$
  **else**
    **return** $\mathbb{E}_u [g_q(x, u) + \text{J-STAR}(f(x, u), q', d)]$ ▷ Opponents turn, eq. (9.12)
  **end if**
**end function**
**function** POLICY($x$) ▷ Compute best action
  **return** $\arg\max_{u \in \mathcal{A}(x)} [g_0(x, u) + \text{J-STAR}(f(x, u), 1, d)]$
**end function**

| Agent | $d = 1$ | $d = 2$ | $d = 3$ | $d = 4$ |
|---|---|---|---|---|
| Minimax | 1.0 | 0.5 | 0.0 | 1.0 |
| Expectimax | 0.0 | 0.5 | 1.0 | 1.0 |
| Alpha-Beta | 1.0 | 0.5 | 0.5 | 0.5 |

Table 9.1: Win rate for multiagent search applied to the pacman level in fig. 9.1

In other words, what we have done is to define $J_{k,q}$ such that $J_{k,0} = J_k$. We have then shown that these functions are related such that $J_{k,0}$ (our turn) can be computed in terms of $J_{k,1}$ using the min-operation (see eq. (9.13a)), and when it is the opponents turn, $q \geq 1$, they are related by eq. (9.12). This formulation of the DP algorithm is commonly called Expectimax and we will therefore therefore adopt this term from now on; the version with the changes discussed is shown in algorithm 14.

What remains is simply an issue of translating the Expectimax-version of the DP algorithm into the language of a multi-agent game (theorem 9.1.1). To do so, we note there is no intermediate cost-function $g_q$ unless the player is terminated (i.e. TERMINAL-TEST is true) in which case the UTILITY is returned). In our implementation we choose to let $\tilde{J}$ be simply the UTILITYfunction, however better approximations of the true tail cost will lead to better results. A final change of the algorithm is that we compute the optimal action as part of the method rather than having an additional minimization step when computing the policy. The resulting method can be seen in algorithm 15, with the changes highlighted in red. Note the minimums have been changed to maximums because the multi-agent search problem is defined in terms of reward.

How well expectimax works depends on the planning depth. The larger depth, the more it will resemble an (inefficient!) implementation of the DP algorithm, and

☞ **Algorithm 15** Expectimax search

```
1: function EXPECTIMAX(x, q, d)        ▷ Returns (optimal score, optimal action)
2:     if d = 0 or Terminal-Test(x) then
3:         return UTILITY(x), none          ▷ Return leaf utility and dummy action
4:     end if
5:     if q = Q − 1 then      ▷ It is the last players turn; reset q and decrease depth
6:         Next agent q′ ← 0 and next depth d′ ← d − 1                  ▷ End of round
7:     else
8:         Next agent q′ ← q + 1 and next depth d′ ← d                  ▷ Within a round
9:     end if
10:    for all u ∈ A(x) do
11:        V, none ← EXPECTIMAX(f(x, u), q′, d′)          ▷ Score of playing action u
12:    end for
13:    if q = 0 then
14:        u* ← arg max_{u∈A(x,q)} V_u                    ▷ Compute optimal player action
15:        return V_{u*}, u*                          ▷ Return optimal cost, optimal action
16:    else
17:        return Σ_{u∈A(x,q)} p_q(u|x)V_u, none  ▷ Irrelevant dummy action for opponent
18:    end if
19: end function
20: function POLICY(x)
21:    v*, u* ← EXPECTIMAX(x, 0, d)
22:    return u*
23: end function
```

thereby provide near-optimal results. We have illustrated the win-rate (obtained by simulating 10'000 games) of expectimax, when applied to the Pacman game level in fig. 9.1 in table 9.1, and as shown in the table the win-rate increases when Pacman plan on a longer (but still fairly shallow all things considered) depth. The winrate can be compared to the DP algorithm when evaluated using various $N$, see table 9.2. The later table includes the estimates cost of the starting state $J_0$, the (simulated) win percentage, the average length of a game and the size of the state space.

The cost estimate $J_0$ and win-rate agrees as they should, however comparing the win-rate to expectimax we see expectimax, at the same depth, outperforms the DP algorithm, and the DP algorithm needs a depth greater than 8 to compete with the expectimax algorithm using depth 4. The reason for this is that the expectimax algorithm re-plans at each step and so while the policy is shallow, it is more relevant to the current board configuration. We also see that increasing the depth from 12 to 20 does not change the winrate. This is because the randomness in the game makes long-term planning irrelevant as the ghosts position cannot be predicted with enough accuracy. These features are the reason control theory is nearly exclusively concerned with rolling-horizon controllers.

| N | $J_0$ | Win pct | Length | $\lvert\mathcal{S}\rvert$ |
|---|---|---|---|---|
| 1 | 0.00 | 0.00 | 1.00 | 12.0 |
| 2 | 0.00 | 0.00 | 2.00 | 41.0 |
| 3 | 0.00 | 0.00 | 2.50 | 155.0 |
| 4 | 0.75 | 0.72 | 3.72 | 278.0 |
| 6 | 0.81 | 0.81 | 4.30 | 1098.0 |
| 8 | 0.82 | 0.82 | 4.33 | 3565.0 |
| 12 | 0.85 | 0.86 | 4.54 | 18956.0 |
| 16 | 0.85 | 0.84 | 4.51 | 37516.0 |
| 20 | 0.85 | 0.84 | 4.56 | 47811.0 |

Table 9.2: Results of the DP algorithm to the pacman level in fig. 9.1

## 9.3 Minimax search

In most circumstances, we cannot assume to know the noise distribution $P_k(W_k|x_k, u_k)$, however we might know enough of the environment to know which values of the noise disturbances are possible, denoted by the set $W_k(x_k, u_k)$. This situation covers most games, in which $w_k$ is a stand-in for the opponents action. It this case, it is often reasonable to assume the opponent chooses the *worst* action (from our perspective) and plan on that assumption. This is known as **adversarial search**, which we already encountered in section 6.3.2. To recap, it was simply the replacement of the expectation with a maximum in the DP cost function:

ADVERSARIAL SEARCH

$$J_\pi(x_0) = \max_{w_k \in W_k(x_k, \mu_k(x_k))} \left[ g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right] \tag{9.14}$$

The DP algorithm therefore becomes our regular DP algorithm with a max rather than an expectation, hence the name **minimax** search:

MINIMAX

$$J_k(x_k) = \min_{u_k \in U(x_k)} \left[ \max_{w_k \in W_k(x_k, u_k)} \left[ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right] \right]. \tag{9.15}$$

The discussion of expectimax still holds true, and this leads to a simpler algorithm: Our adversaries simply select the actions which are *worst* (from our perspective) and we select the actions that appear *best*. The listing is given in algorithm 16, and the consequential changes from the expectimax algorithm are highlighted in red.

If we consider the performance of minimax search in the multi-ghost example, see table 9.1, we see the win-rate is lower than expectimax. The reason is minimax, by always assuming the ghosts does the worst possible, is very risk-adverse, and for a game such as this it means it fails to take necessary risks. We can see this in another way by looking at the estimated value of the initial state $J_0(x_0)$ (i.e., simply the output of minimax/expectimax when called on this state). The expectimax estimate increases from 9 (+10 bonus from eating the first food pellet), then decreases due to the cost of

☞ **Algorithm 16** Minimax search

**function** MINIMAX($x$, $q$, $d$) ▷ Returns tuple of (optimal score, optimal action)
  **if** $d = 0$ **or** TERMINAL-TEST($x$) **then**
    **return** UTILITY($x$), none ▷ Return leaf utility and dummy action
  **end if**
  **if** $q = Q - 1$ **then** ▷ It is the last players turn; reset $q$ and decrease depth
    Next agent $q' \leftarrow 0$ and next depth $d' \leftarrow d - 1$ ▷ End of round
  **else**
    Next agent $q' \leftarrow q + 1$ and next depth $d' \leftarrow d$ ▷ Within a round
  **end if**
  **for all** $u \in \mathcal{A}(x)$ **do**
    $V_u$, none $\leftarrow$ MINIMAX($f(x, u), q', d'$) ▷ Score of playing action $u$
  **end for**
  $u^* \leftarrow \max_{u \in \mathcal{A}(x)} V_u$ **if** $q = 0$ **else** $\min_{u \in \mathcal{A}(x)} V_u$ ▷ Use min rather than average
  **return** $V_{u^*}, u^*$
**end function**
**function** POLICY($x$)
  $v^*, u^* \leftarrow$ MINIMAX($x, 0, d$) ▷ Optimal action is $u^*$
**end function**

| Agent | $d = 1$ | $d = 2$ | $d = 3$ | $d = 4$ |
|---|---|---|---|---|
| Minimax | 9.0 | 8.0 | 7.0 | -492.000 |
| Expectimax | 9.0 | 8.0 | 7.0 | 326.125 |
| Alpha-Beta | 9.0 | 8.0 | 7.0 | -492.000 |

Table 9.3: Estimated expected cost $J_0(x_0)$ for the state in fig. 9.1

moving ($-1$ per step), and at step 4 it jumps to 326 because expectimax (correctly!) believes there is a large chance of winning ($+500$) the game in 4 moves. Minimax is more pessimistic, and correctly note that if the ghosts do their absolute worst they will catch Pacman. The pessimism of the minimax agent can often lead to sub-optimal behavior, such as refusing to do anything because it believes it is doomed anyway.

### 9.3.1 An issue with expectimax and minimax

If the depth is set to $d = \infty$, and assuming the cost function is only defined on the terminal nodes, both methods returns the optimal policy assuming they terminate and have a complexity of $\mathcal{O}(b^m)$, where $m$ is the maximum number of *plys* of the deepest branch of the game. In most circumstances this is unfeasible and a more shallow depth is used.

The number of search nodes expanded by expectimax/minimax can be found in table 9.4, and we see they require just around 6000 search nodes for a depth of just 4.

| Agent | $d = 1$ | $d = 2$ | $d = 3$ | $d = 4$ |
|---|---|---|---|---|
| Minimax | 33.0 | 211.0 | 1160.0 | 5916.0 |
| Expectimax | 33.0 | 211.0 | 1160.0 | 5916.0 |
| Alpha-Beta | 26.0 | 150.0 | 643.0 | 1315.0 |

Table 9.4: Number of search nodes expanded by multisearch agents

This is much greater than DP at the same depth (1100, see table 9.2; the discrepancy is because the recursive formulation may expand the same node many times if there are recurrent configurations), but much less than the number of states at $N = 20$.

For many games the depth required for reasonable play makes expectimax/minimax unfeasible. For instance, chess games has a branching factor of 20 or more in the early game and so even just a few plys, which is far too little for reasonable play, is too much.

## 9.4 Alpha-Beta search ✸✸

The problem with minimax is the number of nodes in the game tree grows exponentially in the branching factor, and for many applications this is simply unacceptable. This growth is dictated by the nature of the decision problem and so we cannot get rid of it, but as it turns out we can reduce the *exponent* by a factor of two while still returning minimax optimal paths. The technique we will consider here is called alpha-beta pruning, and it rests on the simple observation that some branches, assuming we know a bit about them, can be marked as irrelevant.

How does this work? Consider the minimax algorithm for a two-player game where as usual we attempts to maximize our utility and our opponent tries to maximize it. For $d = 1$ the minimax search tree can be visualized in fig. 9.2 (each player has 3 actions).

Since it is our turn, the minimax evaluation for node $A$ is as follows:

$$\textsc{Minimax}(A) = \max\left[\textsc{Minimax}(B), \textsc{Minimax}(C), \textsc{Minimax}(E)\right] \tag{9.16}$$

These three calls to minimax are evaluated on the opponents turn, who will use the min. In other words, just expanding one of the calls for clarity, we obtain:

$$\max\left[\min\left\{\begin{array}{c}\textsc{Minimax}(E)\\ \textsc{Minimax}(F)\\ \textsc{Minimax}(G)\end{array}\right\}, \min\left\{\begin{array}{c}\textsc{Minimax}(H)\\ \textsc{Minimax}(I)\\ \textsc{Minimax}(J)\end{array}\right\}, \min\left\{\begin{array}{c}\textsc{Minimax}(K)\\ \textsc{Minimax}(L)\\ \textsc{Minimax}(M)\end{array}\right\}\right] \tag{9.17}$$

As the algorithm is called, it will begin to evaluate these branches one at a time. Suppose we evaluate all nodes except two as follows:

$$= \max\left[\min\left\{\begin{array}{c}7\\ 3,\\ 14\end{array}\right\}, \min\left\{\begin{array}{c}2\\ \textsc{Minimax}(I)\\ \textsc{Minimax}(J)\end{array}\right\}, \min\left\{\begin{array}{c}7\\ 5\\ 4\end{array}\right\}\right]$$

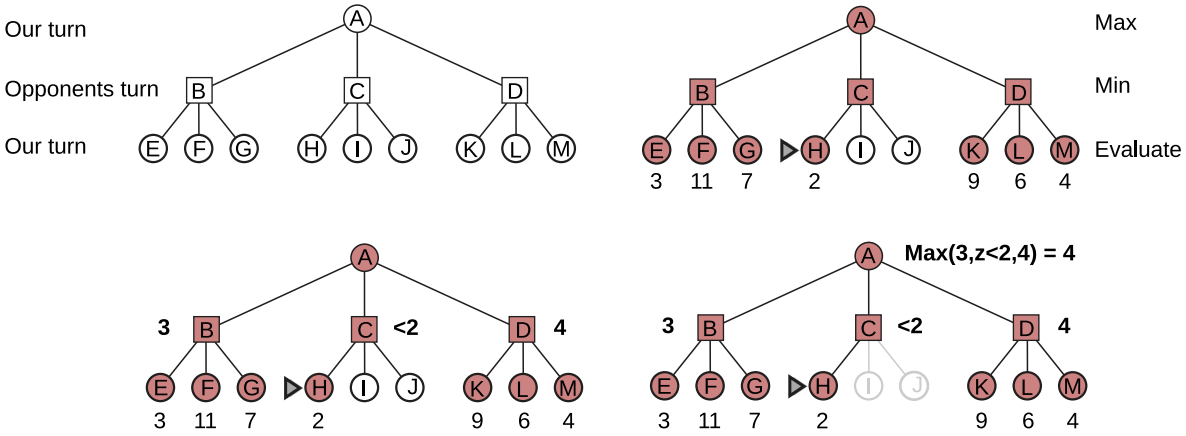


Figure 9.2: Illustration of minimax search for $d = 1$. Top-left: Minimax has to search a small tree (it is assumed we and the opponent have 3 actions), at the root $A$ we perform a max-operation, the opponent three min-operations, and finally the 9 leaf states are evaluated. Top-right: Minimax search the tree using a depth-first order, and we suppose in this case Minimax has just evaluated leaf $G$ to have a utility of 2. Bottom-left: At this point we know the left-branch has value $B = 3$, the right-branch $B = 4$ (due to the min), and the middle branch must have value less than 2 (dependent on the value of $H$ and $I$). Bottom-right: Since we want to compute the Max at the root, there is no reason to further evaluate the second branch: We know the best action is $E$, and has a value of 4.

The left-most branch will be evaluated to $\min\{7, 3, 14\} = 3$. Meanwhile, we know that the middle branch is $\leq 2$ and the right-most branch is $\min\{14, 5, 4\} = 4$. Hence, at the above step the algorithm knows, without evaluating any further nodes, that:

$$= \max\left[3, C \leq 2, 4\right] = 4$$

This is true *regardless* of the value of the nodes $I$ and $J$. In other words, the minimax structure implies we can *sometimes* evaluate the tree *exactly* even without knowing the value of all the branches.

### 9.4.1 Alpha-beta pruning

For each state ($A, B$ and so on) we define two numbers $\alpha$ and $\beta$. The role of $\alpha$ is to track the reward we are guaranteed (assuming we make optimal actions), and $\beta$ tracks the reward the opponent is guaranteed (assuming maximally adversarial actions). These numbers are initialized at the root, and passed down the tree: At a Max-node (for instance, $A$), once we *know* a subtree has at least some value, we can update $\alpha$.

As an example, in fig. 9.2, once the $B$ branch is evaluated this guarantees that $\alpha = 3$ at the $A$ node. Later, when the $D$-branch is evaluated, we are guaranteed a utility of at least $\alpha = 4$. The value of $\alpha = 4$ is then passed down to the $C$-branch, which is a min-node. Once the min-node evaluates the $H$ branch to 2, it can check that

$$\text{Utility}(H) \leq \alpha$$

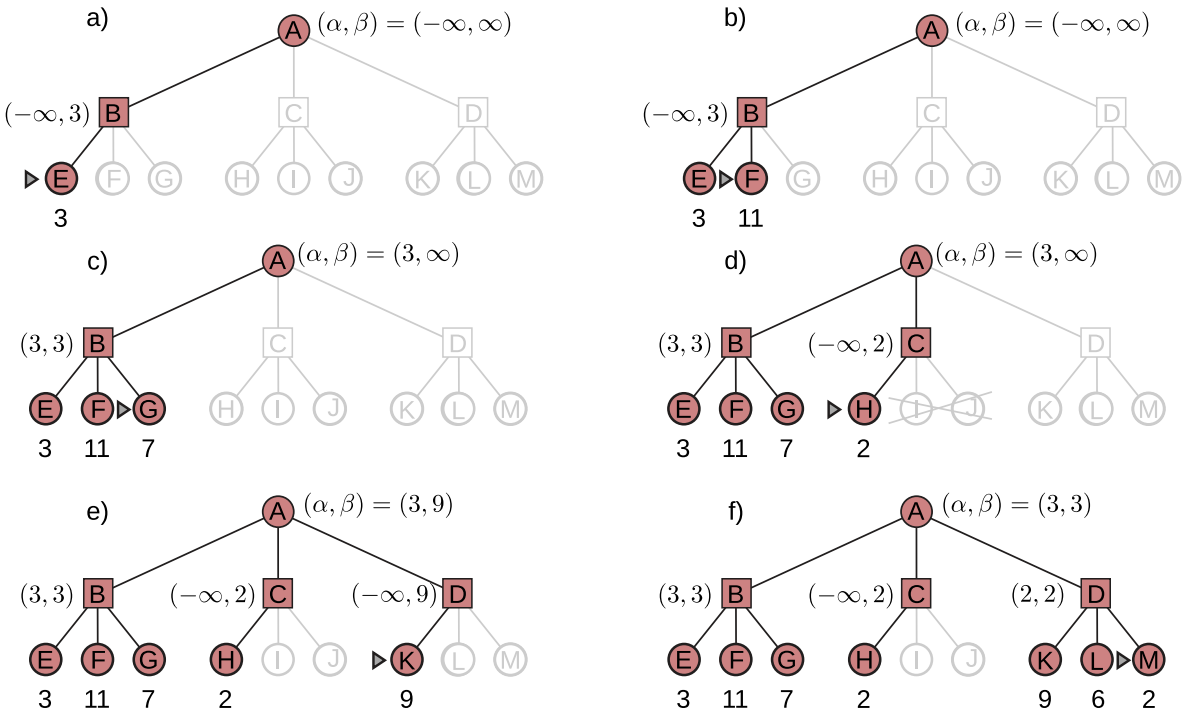


Figure 9.3: Illustration of alpha-beta search (see text). The *alpha*-value is a (guaranteed) highest utility the max-player is guaranteed. Once the max-player begins to see what happens at next layer $\alpha$ can be assigned. If we tell $\alpha$ to the min-player at the middle branch, she can reason certain actions are never going to be used.

the min-node can then conclude that *whatever else happens*, its result will never be used. A similar argument can be used to terminate evaluation at a max-node (our turn), however in that case we need the *minimal* utility we are already guaranteed. To make **alpha-beta pruning** rigorous, we define the values:

ALPHA-BETA PRUNING

$$\alpha = \begin{Bmatrix} \text{The highest value the Max-player is so far guaranteed} \\ \text{by taking optimal actions} \end{Bmatrix} \tag{9.18}$$

$$\beta = \begin{Bmatrix} \text{The lowest value the Min-player is so far guaranteed by} \\ \text{taking optimal actions (from their perspective)} \end{Bmatrix} \tag{9.19}$$

We will provide exact pseudocode for how they are updated momentarily, however their role is best understood using the example in fig. 9.3:

- **Init)**: We start in node $A$ (max-node), and initially $\alpha = -\infty$ and $\beta = \infty$, signifying no players have any guarantees
- **a)**: We proceed using DFS. At node $B$ (a min-node) there is a for-loop over the leafs, and $\beta$, the min-nodes guaranteed best (minimal!) utility, is updated. Since the min-node learns $E = 3$, it updates $\beta = 3$ since this is now guaranteed
- **b)**: The min-node iterates over other choices. Since $11 > 3$, there is no need to update $\beta$ as it would never choose 11
- **c)**: After the $G$ is evaluated node $A$, a max-branch, knows that the value of $B$-branch is 3. It can thus update $\alpha = 3$ since this value is now guaranteed
- **d)**: We then proceed to $C$, a min-node, where $\beta$ is updated. At $C$ the min-node learns a value of $\beta = 2$ is guaranteed, however it also knows that the top-node $A$ has a value of $\alpha = 4$ available; hence, there is no need for further tests
- **e)**: The method proceed to the $D$-branch, where $\beta$ is updated to 9 for the $K$-node
- **f)**: After the min-node learns about $M = 2$ it updates $\beta = 2$ since this is now guaranteed at this node. The evaluation of $A$, $B$ and $C$ has happened as a loop in the max-node at $A$, which tries to find the maximal utility. Since this was obtained at $B = 3$, it has $\alpha = 3$ and returns the action corresponding to $B$.

The method is implemented in algorithm 17; although the listing is rather dense, note the max/min steps are nearly identical. In summary, we maintain the $\alpha$ and $\beta$ values for each node, however at a max-node only the $\alpha$-parameter is updated and the $\beta$-parameters are just passed along to be used at the following min-node. Meanwhile, the rest of the program is similar to minimax, in that we still plan at a maximal depth of $d$. The method can be used to compute the optimal action $u^*$ and utility $v^*$ for the player $q = 0$ as:

$$v^*, u^* = \textsc{Alpha-Beta-Search}(x, 0, d, \alpha = -\infty, \beta = \infty).$$

**Algorithm 17** Alpha-Beta search

```
1: function ALPHA-BETA-SEARCH(x, q, d, α, β)
2:     if d = 0 or TERMINAL-TEST(x) then
3:         return UTILITY(x), none
4:     else if q = 0 then
5:         return MAX-VALUE(x, q, d, α, β)
6:     else
7:         return MIN-VALUE(x, q, d, α, β)
8:     end if
9: end function
10: function MAX-VALUE(x, q, d, α, β)
11:     v* ← −∞                                   ▷ Best (maximal) utility for any action
12:     for all u ∈ A(x, q) do                    ▷ Loop over all actions
13:         v ← MIN-VALUE(f(x, u), 1, d, α, β)    ▷ Utility of this branch
14:         if v > β then return v, u end if
15:                                   ▷ Pruning: If there is a min-node closer to the root
                                      with a guaranteed low cost of β lower than the
                                      current estimate v this branch is never chosen.
16:         if v > v* then
17:             (u*, v*) ← (u, v)     ▷ If the utility of branch u is better, use this branch
18:             α ← max(α, v*)        ▷ Update the α-value to match best available option
19:         end if
20:     end for
21:     return v*, u*
22: end function
23: function MIN-VALUE(x, q, d, α, β)
24:     (q', d') ← (0, d − 1) if q = Q − 1 else (q', d') ← (q + 1, d)   ▷ As in minimax
25:     v* ← ∞                                ▷ Score for best branch for min-agent
26:     for all u ∈ A(x) do
27:         v ← MAX-VALUE(f(x, u), q', d', α, β)
28:         if v < α then return v, u end if                            ▷ Prune
29:         if v < v* then
30:             (u*, v*) ← (u, v)
31:             β ← min(β, v*)
32:         end if
33:     end for
34:     return (v*, u*)
35: end function
```

### 9.4.2 Comments on efficiency

Alpha-beta search is a more efficient implementation of minimax, and will therefore find solutions with the same cost (see table 9.1 and table 9.3). However, it can be much more efficient.

Since whether a node can be pruned depends on the order in which other nodes have been evaluated beforehand alpha-beta search has the same worst-case performance of $b^d$ as minimax. However, if the nodes are accessed in an optimal order, i.e. the nodes which actually contains the optimal paths are expanded first, we get a performance of $\mathcal{O}(b^{\frac{d}{2}})$, meaning we can evaluate a search tree twice as deep. If nodes are expanded in random order the behavior is $\mathcal{O}(b^{\frac{3d}{4}})$, i.e. half of the benefit in terms of extra search depth. Even for the Pacman game this gain is quite substantial (see table 9.4).

### 9.4.3 Tricks and chess

For chess, alpha-beta search is augmented by a few tricks

- Since alpha-beta search achieves the best performance of $\mathcal{O}(b^{\frac{d}{2}})$ by expanding the optimal path first, much can be gained by searching the more promising nodes first. In chess, this is commonly done by considering moves that are a-priori promising first, such as captures, pawn promotions, checks, moves forward, and finally moves backwards in that order.

- It is also common to change the TERMINAL-TEST() to stop searching really bad positions, for instance if the board evaluation fall below some threshold.

- As expectimax and minimax, alpha-beta search can expand the same position many times. By keeping an index of already evaluated positions the amount of work can be reduced

- In chess, a very basic utility function tracks the value of the pieces (a pawn is 1 point, a bishop is 3, a queen is 9 and so on), however a real chess engine uses much more refined features, for instance double pawns, free pawns, king safety, number of available moves, distance between king and pawns, and so on. Especially for end-game play these features are important

Up until very recently, Alpha-Beta search, along with move-order heuristics, a carefully crafted utility functions and pruning methods, was the class of methods which played chess far better than any other method (deep blue was an alpha-beta search method); this was possible because chess has a intermediate branching factor. For games such as go, the branching factor is too high and alpha-beta search, even with modern computer resources, remains inferior to top human play.

As we will see in the second half of this course, it is possible to re-use the idea of a game tree, but with a *learned* utility function and an adaptive approach to exploring the tree to build a more general class of algorithms, and it is this class of algorithms, known as monte-carlo search trees which have recently been used to build expert Go engines and chess engines with performance comparable to the best chess engines[1]

[1]For chess, the MCST approaches have a different hardware requirement than normal chess engines, however it seems at least reasonable to say they have a comparable performance and will likely exceed chess engines, even using similar computational resources, in the near future

# Part III

# Control

# Chapter 10

# The control problem

Control theory deals with a similar problem to that which we have already considered, namely how to make decisions, one after another, so to bring a system into a desirable state.

However, control theory differs both in the details of how this problem is formulated, and perhaps more importantly, in what is considered desirable behavior in a controller and therefore emphasized in control theory:

**Formulation** The problems treated by control theory (such as temperature management of a watertank, a plane autopilot, balancing a Segway, etc.) represents concrete, physical systems which evolve in continuous time. Accordingly, the time variable should be considered as a continuous variable $t \in \mathbb{R}$, and therefore both the state $\boldsymbol{x}(t)$ and actions (which we now call **control**) $\boldsymbol{u}(t)$ are going to be functions of time.

CONTROL

**Emphasis in control theory** Control theory is often concerned with robustness, that is, being able to reason exactly about what the controller does and guarantee it works. This often mean that the methods used in control theory can appear superficially simple from a machine-learning background because simpler methods are more predictable.

**Practical problems** Control theory face many practical obstacles we have so far been spared from. In the inventory control example, it was reasonably to assume that the states $x_k$, actions $u_k$ and noise disturbance distributions are fully known. If we try to apply control theory to a robot, this will most likely no longer be the case:

- The state $\boldsymbol{x}(t)$ is probably not fully known. For instance, if some part of $\boldsymbol{x}(t)$ is the current location of self-driving robot, we need to solve the problem of figuring out the locating of the robot given e.g. camera measurements.
- A robot is likely controlled by a specialized robotics-OS, which integrate different sensory information; these are probably sampled at different speeds, at different times, and with varying precision precision.

- Even if we have a really good CAD drawing of the robot it is unlikely we can accurately predict its behavior exactly due to friction, wear-and-tear, etc.

**The big picture: Models and the real world** Control theory grew out of physics, and the approach to complexity is largely the same: To study simpler, mathematically well-described models in order to obtain insight and concrete methods. These methods are then applied to real world problems with the hope the real world is not *too* different from the model (but understanding that it is at least a *bit* different).

The idealized problems we always start with will be differential equations which arises from classical mechanics. In the following, this will be called the **continuous-time** formulation of the problem.

CONTINUOUS-TIME

There are two operations we will be interested in:

- To be able to *accurately* predict what the continuous-time do when a control signal is applied. This will form the basis of building environments which we can later test the methods in
- To build approximate, and very often discrete, models of the continuous-time formulations. These will form the basis of our control methods we implement in the agents.

Note that the real-world (physical) system may differ from the continuous-time model we use to build our controller. The main takeaway is that we can no longer assume the model is behaves exactly the same as the environment.

**Approach in this course** I have chosen to emphasize practical computational tools for real world optimal control problems. In order to ensure what we are doing is still a recognizable form of control theory I have made three choices:

- To test the control methods independently on an (reasonably exact) simulation of the continuous-time control problem. This is our attempt to address that our control-theory models are likely not an exact representation of the real world.
- We will therefore maintain the clear distinction between agent, environment and model. This will become more important since the models will typically be approximate
- The main methods we will focus on are real-world control methods such as iLQR, direct methods, PID control, etc.

**Prerequisites:** Control theory will make use of concepts from analysis such as derivatives and ordinary differential equations (ODEs). If the notation is at all unclear, consider looking at section X.2.

## 10.1 The continuous-time control problem

In order to solve a problem, we need a precise mathematical formulation. When we discuss control, we will always assume the dynamics is described by an ODEs (ordinary differential equation). We already saw an example from the Pendulum environment (eq. (4.1)), in which the system was characterized by the angle $\theta$, an applied force $u$, and satisfied the second-order differential equation:

$$\ddot{\theta} = \frac{g}{l}\sin(\theta) + \frac{u}{ml^2}. \tag{10.1}$$

This equation use notation common in engineering and physics, where it is understood that both $\theta$ and $u$ are functions of time and that the dot indicate time derivative (see section X.2). An equivalent way of writing the problem is therefore

$$\frac{d^2\theta(t)}{dt^2} = \frac{g}{l}\sin(\theta(t)) + \frac{u(t)}{ml^2}.$$

The variables $g, l, m$ are constants denoting the gravitational constant, pendulum length and the mass attached to the end of the pendulum.

We can re-formulate any second order differential equation to a first order differential equation using a simple trick: For each second-order derivative, $\ddot{\theta}$, we introduce the new coordinates $\boldsymbol{x}$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}. \tag{10.2}$$

Evidently, the derivative of $x_2$ with respect to $t$ is $\ddot{\theta}$, and therefore we can re-write the system as:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} x_2 \\ \frac{g}{l}\sin(x_1) + \frac{u}{ml^2} \end{bmatrix} = \boldsymbol{f}(\boldsymbol{x}, u). \tag{10.3}$$

Where we have introduced the vector-function $\boldsymbol{f}$.

This approach (i.e., replacing a double-derivative such as $\ddot{\theta}$ with two variables $x_1 = \theta$ and $x_2 = \dot{\theta}$) can be used to get rid of all second-order derivatives.

We will therefore study systems whose behavior from a start-time $t_0$ until and end-time $t_F$, and for all $t \in [t_0, t_F]$, can be described using a state $\boldsymbol{x}(t) \in \mathbb{R}^n$ and a control vector $\boldsymbol{u}(t) \in \mathbb{R}^d$, which will evolve as a first-order system of ODEs characterized by a function $\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^n$:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t). \tag{10.4}$$

DYNAMICS

This function $\boldsymbol{f}$ is called the **dynamics** of the control problem.

## 10.2 Constraints

Any physical system will be subject to constraints, which may either reflect what the system simply *cannot* do, what we *don't want it to do*, and *what we want it to do*.

We often distinguish between equality and inequality constraints on a variable $x$:

$$\text{Equality constraint:} \qquad x = c \tag{10.5}$$

$$\text{Inequality constraint:} \quad a \leq x \leq b \tag{10.6}$$

From a formal perspective we can express the equality constraint as an inequality constraint by letting $a = b = c$. At the same time, if we let $a = -\infty$ and $b = \infty$, then the inequality constraint becomes equivalent to no constraint. It is therefore common to write all constraints as inequality-constraints.

### Simple constraints

These are by far the most common and something we invariantly have to face in nearly all applications. They take the form:

$$\begin{aligned} \boldsymbol{x}_{\text{low}} &\leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}} \\ \boldsymbol{u}_{\text{low}} &\leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upp}} \end{aligned} \tag{10.7}$$

and simply signify the given variables lie in certain ranges.

### End-point constraints

The end-points $t_0$ and $t_F$ of a robots path are often of particular importance, and so we might have special constraints which only apply at these. As an example:

$$\begin{aligned} \boldsymbol{x}_{0,\text{ low}} &\leq \boldsymbol{x}\left(t_0\right) \leq \boldsymbol{x}_{0,\text{ upp}} \\ \boldsymbol{x}_{F,\text{ low}} &\leq \boldsymbol{x}\left(t_F\right) \leq \boldsymbol{x}_{F,\text{upp}}. \end{aligned} \tag{10.8}$$

which could be used to specify that the robot starts in a particular configuration ($\boldsymbol{x}(t_0) = \boldsymbol{x}_0$), or that the robot must end in a particular terminal state ($\boldsymbol{x}(t_0) = \boldsymbol{x}_F$) *and absolutely no deviation is allowed*.

### Time-constraints

The end-times $t_0$ and $t_F$ may themselves be considered variable (although often this will not be the case). For instance, if we launch a satelite into orbit, we probably know when the launch starts, $t_0$, however we have no interest (and no practical way) to specify when the sattelite should reach orbit $t_F$. This can be specified as

$$\begin{aligned} t_{0,\text{ low}} &\leq t_0 \leq t_{0,\text{ upp}} \\ t_{F,\text{ low}} &\leq t_F \leq t_{F,\text{upp}}. \end{aligned} \tag{10.9}$$

That the end-time is variable signifies an important departure from the dynamical programming problem considered in the earlier section and is important in minimum-time problems (i.e., complete a control task in minimal time). Note not all methods we will encounter will handle end-time constraints.

### 10.2.1 Non-linear constraints ★

In the exercises we will limit ourselves to linear constraints. However, many constraints are non-linear. For instance non-linear boundary constraints:

$$\boldsymbol{h}^b\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right) \leq \mathbf{0}, \quad \text{boundary constraint} \tag{10.10}$$

This could be useful if we are planning a robots gait over one full step between time $t_0$ to $t_F$ and we want to tell the robot the gait must be periodic $\boldsymbol{x}(t_0) - \boldsymbol{x}(t_F) = 0$.

#### Non-linear path constraints

Some constraints are inherently a function of both the state and action during the control sequence. Examples include that the limbs of a robot cannot intersect each other, that the dynamical load on a rocket ship cannot exceed a certain threshold or that a car must follow a particular path. The general form is:

$$\boldsymbol{h}(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) \leq \mathbf{0} \tag{10.11}$$

With the understanding $\boldsymbol{h}$ may be a function of arbitrary many output dimensions (this allows us to specify both upper and lower-bounds).

## 10.3 Policy and cost

Recall the cost-function for the basic DP problem was a function $J_\pi(x_0)$ of the policy $\pi$ and initial state $x_0$. We will follow this convention and denote the policy as $\boldsymbol{u}(t) = \pi(\boldsymbol{x}(t), t)$. The special case of finding an optimal open-loop policy, $\boldsymbol{u}(t) = \pi(t)$, is known as **trajectory optimization** in control theory, and a closed-loop policy is often referred to as a **feedback policy**.

TRAJECTORY OPTIMIZATION

FEEDBACK POLICY

**Closed-loop or open-loop?** Although it is true an open-loop policy is sufficient for the deterministic dynamics defined by $\boldsymbol{f}$ in eq. (10.8), we will later see that both numerical and practical concerns will in nearly all cases result in the dynamics we plan according to, and the exact dynamics defined by $\boldsymbol{f}$, are different. In those cases there is a great benefit in using closed-loop policies.

**Admissible policies** It is common to think of the differential equation $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)$ not as generating $\boldsymbol{x}$, but rather as a constraint: Given the initial and final times $t_0$ and $t_F$ and arbitrary functions $\boldsymbol{x}(t), \boldsymbol{u}(t)$ for $t \in [t_0, t_F]$ we can check if they satisfy the differential equation as well as whatever other constraints affect the problem.

ADMISSIBLE

If this is the case the trajectory $(\boldsymbol{x}, \boldsymbol{u})$ is said to be **admissible** as it represents a possible solution. If $t_0$, $t_F$ and $\boldsymbol{x}_0$ is considered given, then the differential equation will define $\boldsymbol{x}(t)$ at subsequent time points. In this case we will say the control $\boldsymbol{u}$ is admissible if the resulting trajectory satisfy whatever other constraints that are imposed on the problem.

### 10.3.1 Cost function

Recall the cost-function defined in chapter 5 had the form $J_\pi(x_0)$ since it was dependent on both the policy and initial state $x_0$ (but nothing else). As mentioned above, an open-loop policy will suffice for the (abstract) statement of the control problem, and we will therefore refer to the cost as $J_{\boldsymbol{u}}$. The cost includes $t_0$ and $t_F$ since these can be free variables. If they take a fixed value we will omit them from the problem statement.

The most general form of cost function we will encounter is:

$$J_{\boldsymbol{u}}(\boldsymbol{x}, t_0, t_F) = \underbrace{c_F\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right)}_{\text{Mayer Term}} + \underbrace{\int_{t_0}^{t_F} c(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau)) d\tau}_{\text{Lagrange Term}} \tag{10.12}$$

The cost-function is quite similar to the dynamical programming problem we have previously seen: One term, $c_F$, is equivalent to $g_N$ in that it concerns itself with the end-location of the system (often $t_0$ and $\boldsymbol{x}(t_0)$ will be fixed), and the integral of $c$ play the same role as $g_k$ in that it relates to what the system does during the trajectory.

## 10.4 The continuous-time control problem

The control problem is then stated as minimizing the cost function eq. (10.12) while ensuring that whatever constraints apply to $\boldsymbol{x}$, $\boldsymbol{u}$ and $t_0$ and $t_F$ are satisfied. Formally, we define the optimal control and state trajectory $\boldsymbol{u}^*$ and $\boldsymbol{x}^*$ as the pair minimizing:

$$\boldsymbol{u}^*, \boldsymbol{x}^*, t_0^*, t_F^* = \underset{\boldsymbol{x}, \boldsymbol{u}, t_0, t_F}{\arg\min}\, J_{\boldsymbol{u}}(\boldsymbol{x}, t_0, t_F). \tag{10.13a}$$

$$\text{(Minimization subject to all constraints)} \tag{10.13b}$$

The minimization occurs over all admissible trajectory-control pairs and is therefore subject to the constraints imposed on the system, i.e. the dynamics, $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ and whatever other constraints affect the system, for instance $\boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}}$.

Minimizing over the start-time $t_0$ would be relevant if we were e.g. planning the trajectory of a Mars-mission since it would allow us to determine the optimal launch window dependent on the planets configurations. However, no problems in this course will have this property, and the start-time will always be fixed at $t_0 = 0$, and the start-position $\boldsymbol{x}_0$ will also be given[1]. In this case the control will determine $\boldsymbol{x}(t)$ for $t > t_0$ and so we will often refer to the cost-function as $J_u(\boldsymbol{x}_0, t_F)$.

[1] Although $t_0 = 0$ I have chosen to include the symbol $t_0$ in the notes and exercises because the symbol $t_0$ is more informative than just a 0.

### 10.4.1 Example 1: The pendulum

Recall that the state of the pendulum system can be described as $\boldsymbol{x} = \begin{bmatrix}\theta & \dot{\theta}\end{bmatrix}$ and the dynamics is given in eq. (10.3)

$$\dot{\boldsymbol{x}} = \begin{bmatrix} x_2 \\ \frac{g}{l}\sin(\theta) + \frac{u}{ml^2} \end{bmatrix} = \boldsymbol{f}(\boldsymbol{x}, u). \tag{10.14}$$

The control is in this case the torque $u(t)$.

The goal of the pendulum problem is to balance it upright starting from the downwards, still position $\boldsymbol{x}_0 = \begin{bmatrix}0 & 0\end{bmatrix}$ at $t_0 = 0$. As is always the case, the motor can only output a finite torque $u_{\max}$.

One way to specify the problem is to say we want the pendulum to arrive at the upwards, still position within $t_F = 4$ seconds. This gives us the following constraints:

$$-u_{\max} \leq u \leq u_{\max} \tag{10.15a}$$
$$t_0 = 0 \tag{10.15b}$$
$$\boldsymbol{x}_0 = \begin{bmatrix}0 & 0\end{bmatrix}^\top \tag{10.15c}$$
$$0 \leq t_F \leq 4 \tag{10.15d}$$
$$\boldsymbol{x}_F = \begin{bmatrix}\pi & 0\end{bmatrix}^\top \tag{10.15e}$$

As for the cost, we simply choose to penalize high control values:

$$J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = \int_0^{t_F} u(\tau)^2 d\tau \tag{10.16}$$

QUADRATIC COST

The form of cost chosen here is an instance of a **quadratic cost** function which is very popular in control theory.

#### Cost or constraint formulation?

If we consider the role of the cost and constraints we might note most problems can be formulated using constraints or, alternatively, using cost. For instance, if we wanted to specify the pendulum should end in the up-right position, we could have considered a cost-function containing the terms

$$\cos(\theta) + \lambda\|\boldsymbol{u}\|^2 \tag{10.17}$$

and no end-point constraint. The correct choice depends obviously on what we hope to accomplish with the system, but also on the solution method since some methods handle constraints quite poorly.

### 10.4.2 Example 2: The harmonic osscilator

The harmonic oscillator is one of the most important system in physics (see fig. 10.1). It consist of a spring where one end is fixed at $x = 0$ (the black square in fig. 10.1), and

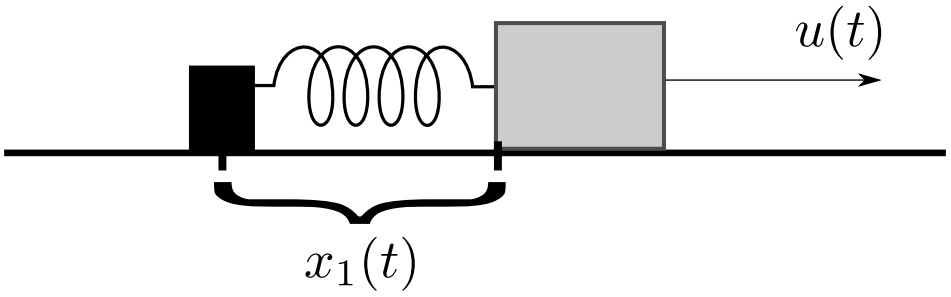


Figure 10.1: The harmonic osscilator. A frictionless ball is attached to a spring and can move back-and-forth. The ball is described by the position $x(t)$ and velocity $\dot{x}(t)$

the other end is attached to another block (the gray rectangle) at location $x(t)$ which can move without friction in the $x$-direction. It is assumed we can apply a bit of force to the gray block using a control signal $u(t)$. According to Newtons laws the equations of motion are:

$$\ddot{x}(t) = -\frac{k}{m}x(t) + \frac{1}{m}u(t) \tag{10.18}$$

Here $m$ and $k$ are the mass/spring constant of the oscillator. We can transform it into the standard representation using the new coordinates $\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$ and this allows us to write the dynamics as a linear function of $\boldsymbol{x}$ and $u$:[2]

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u. \tag{10.19}$$

The cost function is chosen as a quadratic function of $\boldsymbol{x}$ and $u$:

$$J = \int_0^{t_F} \left( \boldsymbol{x}(t)^\top \boldsymbol{x}(t) + u(t)^2 \right) dt. \tag{10.20}$$

This choice of cost will attempt to drive the system towards a state where it is standing still at $x = 0$, but due to the term $u(t)^2$ it will attempt to do so using a small control signal.

The harmonic oscillator is a special case of so-called linear-quadratic problems[3] which are very important in control theory and which we will return to in chapter 12.

### 10.4.3 Example 3: The racecar

As a final problem we will consider the problem of driving a race car through a track. The racecar, along with the track, is shown in fig. 10.2. The goal is to complete the track as quickly as possible without driving across the blue boundaries.

[2]You can verify this is true by plugging in the definition of $\boldsymbol{x}$ to get the two equations: $\dot{x}(t) = 1 \cdot \dot{x}(t) + 0$ and $\ddot{x}(t) = -\frac{k}{m}x(t) + \frac{1}{m}u(t)$, which are equivalent to eq. (10.18).

[3]The name is derived from the linear dynamics and quadratic cost-function

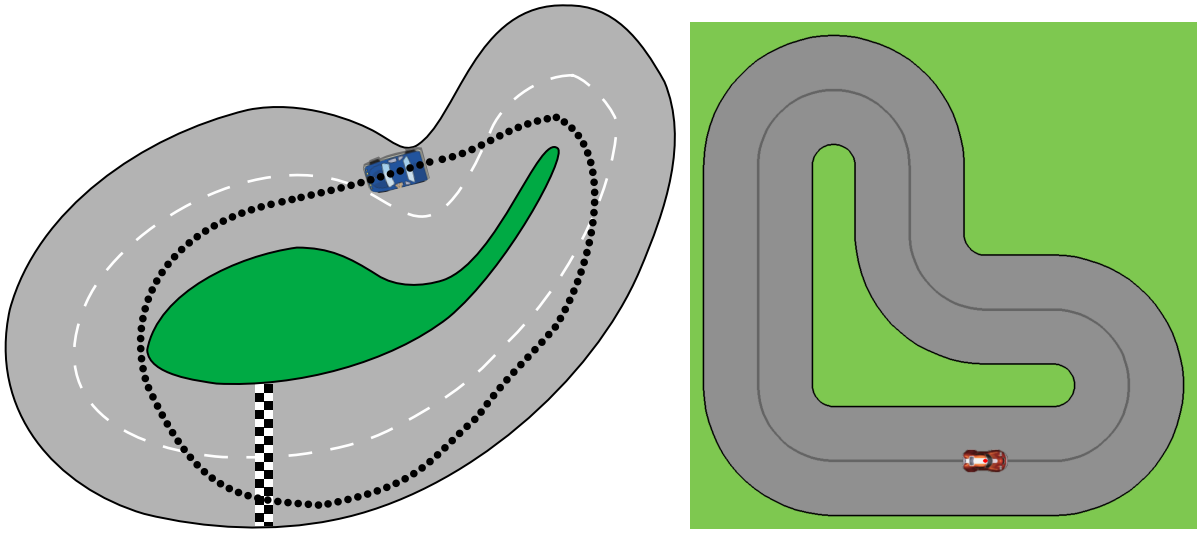

Figure 10.2: Car-environment. The car has to complete a track as quickly as possiblê. The right-pane shows the track we will consider. The track ïs designed by a centerline, and the boundaries are at a fixed distance from the centerline. The lap start is at $x = 0$.

This is a well-studied mechanical system for obvious reasons and one in which a great deal can be gained by using the right coordinate system.

The control vector is naturally chosen as:

$$\boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{10.21}$$

where $u_1$ is the steering angle (the angle the front wheel makes with respect to the car body) and $u_2$ is the throttle (applied to the rear wheels).

One way to describe the car is in terms of the $(x, y)$ location, the velocity in the $x$ and $y$-direction, as well as the global orientation of the car body with respect to the track measured in degrees. However, this representation would pose a number of problems, such as simply figuring out if the car is on the track or not, as well as measuring how far along the track the car was, which would be important to define a cost function.

CURVILINEAR ABSCISSA REFERENCE FRAME

For this reason it is common to use a what is known as a **curvilinear abscissa reference frame** to describe a car on a track. The frame is defined with respect to the center of the track (dotted red line), and represent the car as:

$$\boldsymbol{x} = \begin{bmatrix} v_x \\ v_y \\ \dot{\psi} \\ e_\psi \\ e_y \\ s \end{bmatrix} \tag{10.22}$$

here, $v_x$ is the speed in the longitudinal direction of the car (i.e., the forward-direction with respect too the card body), $v_y$ the lateral speed (perpendicular to $v_x$, i.e. if the

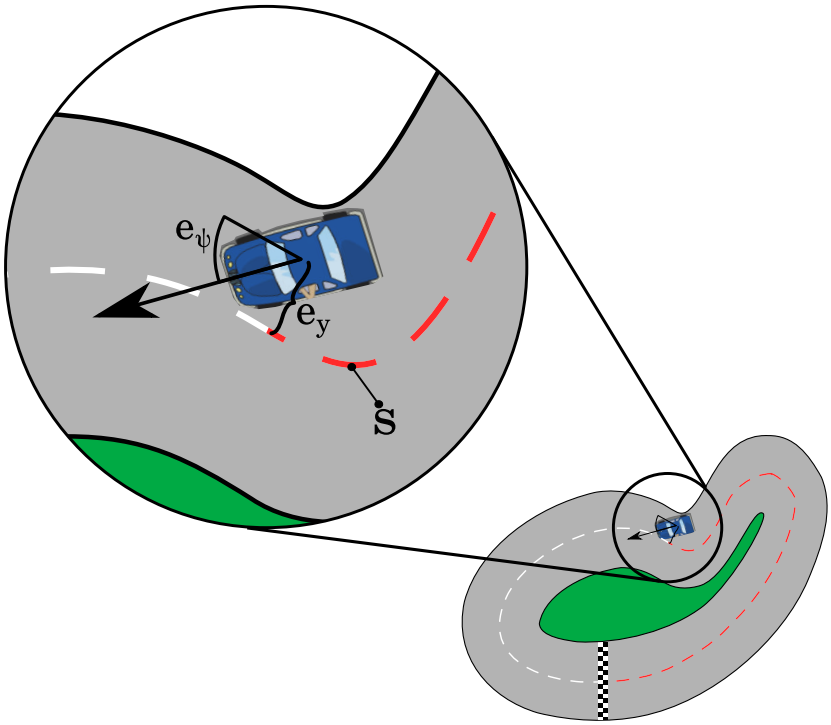


Figure 10.3: The curve linear abscissa coordinate system. The car id described by the angle wrt. the dotted centerline $e_\psi$, the distance from the centerline $e_y$, and distance it has traveled along the center-line $s$ (indicated by red).

car is slipping), $\dot{\psi}$ the cars yaw rate (rate of rotation of the car around center axis), and $e_y$ the distance to the center line, $e_\psi$ the angle between the car body and the tangent of the center line, and $s$ how far along the center-line the car is, see fig. 10.3.

This is a complicated coordinate system and computing the equations of motion in this, or even the basic $(x, y)$ coordinate system, requires a good deal of classical mechanics. However, assume that the dynamics are found, it makes our life much simpler: $s$ measures how far the car has traveled (i.e., when this reach a certain value the track is complete) and $e_y$ measures the distance to the center-line, i.e. as long as

$$|e_y| < \frac{1}{2}\{\text{track width}\} \tag{10.23}$$

the car is on the track. The cost-function can now be chosen as

$$J_{\boldsymbol{u}}(x_0, t_F) = \int_0^{t_F} 1dt = t_F \tag{10.24}$$

and minimizing this cost-function, while ensuring eq. (10.23) as well as requiring the steering angle be limited, that the throttle cannot exceed a certain value, and the car

cannot travel faster than $3m/s$, gives us four simple constraints:

$$-\frac{1}{2}\{\text{track width}\} \leq e_y \leq \frac{1}{2}\{\text{track width}\} \tag{10.25}$$

$$v_x \leq 3 \tag{10.26}$$

$$-0.5 \leq u_1 \leq 0.5 \tag{10.27}$$

$$-1 \leq u_2 \leq 1 \tag{10.28}$$

## 10.5 Implementation details

### Specifying the model

Specifying a control problem in this course is done by implementing the `ControlModel` class. As an example, the following implement the pendulum problem:

```python
# basic_pendulum.py
class BasicPendulumModel(ControlModel):
    def sym_f(self, x, u, t=None):
        g = 9.82
        l = 1
        m = 2
        theta_dot = x[1]  # Parameterization: x = [theta, theta']
        theta_dot_dot = g / l * sym.sin(x[0]) + 1 / (m * l ** 2) * u[0]
        return [theta_dot, theta_dot_dot]

    def get_cost(self) -> SymbolicQRCost:
        return SymbolicQRCost(Q=np.eye(2), R=np.eye(1))

    def u_bound(self) -> Box:
        return Box(np.asarray([-10]), np.asarray([10]), dtype=float)

    def x0_bound(self) -> Box:
        return Box(np.asarray( [np.pi, 0] ), np.asarray( [np.pi, 0]), dtype=float)
```

Further details on how you use the control model is discussed in the online documentation for this week.

# Chapter 11

# Simulation

The previous chapter defined the (idealized) continuous control-problem as the dynamics $\boldsymbol{f}$, a cost function, and a set of constraints. This chapter will consider how we simulate the effect of a policy $\boldsymbol{u}$ in such a problem and thereby obtain a cost estimate and trajectory $\boldsymbol{x}$.

## 11.1 Exactly solving the dynamics

Before we introduce numerical answers, it is perhaps instructive to consider two examples where we can obtain an exact solution to this question:

### 11.1.1 Example A: 1-d problem with no control

Consider the case where $\boldsymbol{u} = 0$, $\boldsymbol{x}$ is one-dimensional and satisfy the initial condition $x(0) = x_0$, and where the dynamics and cost-function is chosen as:

$$\dot{x}(t) = -ax(t) \tag{11.1}$$

$$J_{u=0}(x_0) = \int_0^{t_F} x(t)^2 dt \tag{11.2}$$

The dynamics is a first-order ODE which can be solved with the usual methods from high-school to get $x(t) = x_0 e^{-at}$. By substitution we can evaluate the cost to be:

$$J_{u=0}(x_0) = x_0 \int_0^{t_F} e^{-2at} dt = \frac{x_0}{2a}(1 - e^{-2at_F}).$$

As a small challenge, you can try to solve the problem when $u(t) = \alpha x(t)$.

### 11.1.2 Example B: The harmonic oscillator

As another example, consider the harmonic oscillator we encountered earlier, and lets assume that the control signal is again selected as $u(t) = 0$, that the oscillator starts in

a position where it has been displaced by $d$ from the start position: $x(0) = d$, $\dot{x}(0) = 0$. We then obtain the dynamics and cost:

$$\ddot{x} = -\frac{k}{m}x + \frac{1}{m}u = -\frac{k}{m}x \tag{11.3}$$

$$J = \int_0^{t_F} \left[x(t)^2 + (\dot{x}(t))^2\right] dt \tag{11.4}$$

This is again a familiar type of differential equation with a solution:

$$x(t) = d\cos(\Omega t), \quad \text{with} \quad \Omega = \sqrt{\frac{k}{m}} \tag{11.5}$$

The displacement $x_1(t)$ has been plotted in fig. 11.1 (black line) using $k = 0.1$, $m = 2$ and an initial displacement of $d = 1$. For completeness we can compute the cost-function explicitly[1]:

$$J(\boldsymbol{x}(0)) = d^2 \int_0^{t_F} (\cos(\Omega t)^2 + \Omega^2 \sin(\Omega t)^2) dt \tag{11.6}$$

$$= d^2 \int_0^{t_F} \left[1 + (\Omega^2 - 1)\sin(\Omega t)^2\right] dt \tag{11.7}$$

$$= \frac{d^2}{2}\left((\Omega^2 + 1)t_F - \frac{\Omega^2 - 1}{2\Omega}\sin(2\Omega t_f)\right). \tag{11.8}$$

## 11.2 Euler integration

Analytically solving ODEs will quickly become intractable, and so we need numerical recipe for simulating the future trajectories. The simplest method is **Euler integration**. Consider the general problem $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$. A Taylor expansion of $\dot{\boldsymbol{x}}(t)$ with respect to the time variable gives:

EULER INTEGRATION

$$\boldsymbol{x}(t + \Delta) = \boldsymbol{x}(t) + \dot{\boldsymbol{x}}(t)\Delta + \frac{1}{2}\ddot{\boldsymbol{x}}(t)\Delta^2 + \mathcal{O}(\Delta^3) \tag{11.9}$$

We can now simply truncating the Taylor expansion after the first order term, i.e. assume $\Delta^2$ is negligible, and plug in the definition of $\dot{\boldsymbol{x}}$. This gives us the following expression:

$$\boldsymbol{x}(t + \Delta) = \boldsymbol{x}(t) + \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\Delta$$

This tells us something important: Given we know what the system is doing at time $t$, in state $\boldsymbol{x}(t)$ and with action $\boldsymbol{u}(t)$, we can approximately know the state of the system at a later time point $t + \Delta$. This gives rise to the following simple idea:

[1] See `https://en.wikipedia.org/wiki/List_of_integrals_of_trigonometric_functions`.

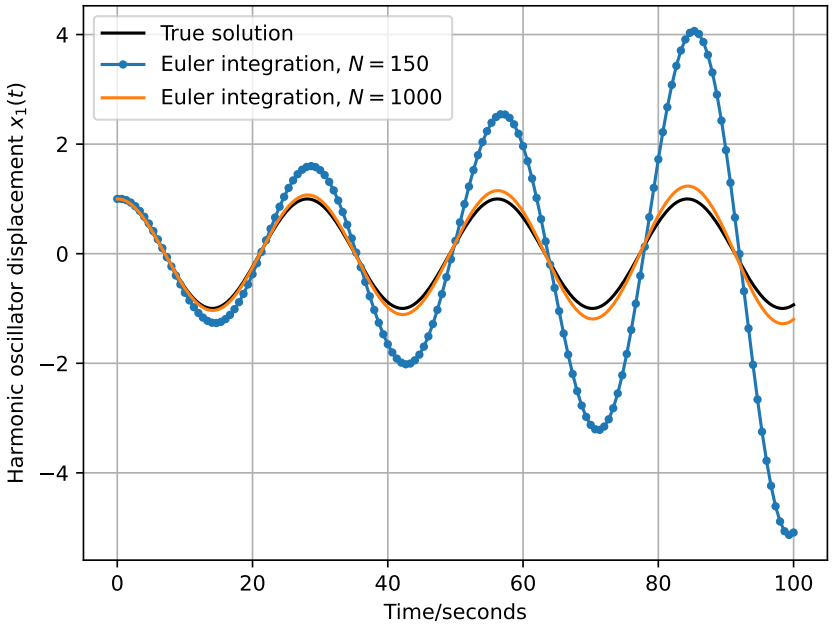


Figure 11.1: True solution to the Harmonic oscillator environment eq. (11.5) and Euler integration using a discretization of $N = 150$ and $N = 1000$. Even for an extremely fine grid Euler integration is not exact for this problem.

We first select a grid size $N$, define $\Delta = \frac{t_F - t_0}{N}$, and then introduce the $N + 1$ equidistant time points

$$\begin{aligned} t_1 &= t_0 + \Delta \\ t_2 &= t_0 + 2\Delta \\ t_k &= t_0 + k\Delta \\ t_N &= t_0 + N\Delta = t_F \end{aligned}$$

We can then simply use the above expression to obtain an expression for how the system will behave at later time points:

$$\boldsymbol{x}(t_k + \Delta) = \boldsymbol{x}(t_k) + \Delta \boldsymbol{f}(\boldsymbol{x}(t_k), \boldsymbol{u}(t_k), t_k) \tag{11.10}$$

If we introduce the shorthand

$$\boldsymbol{x}_k = \boldsymbol{x}(t_k), \quad \boldsymbol{u}_k = \boldsymbol{u}(t_k)$$

this give rise to the following method:

**Definition 11.2.1** (Euler integration)**.** *Given an ODE*

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t),$$

*initial condition $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ and an integer $N$, Euler integration approximate the solution to the ODE in the interval $t_0, t_F$ by iteratively computing*

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k) \tag{11.11}$$

*where $\Delta = \frac{t_F - t_0}{N}$ is the step-size.*

Euler integration has the benefit of being simple, easy to implement, and as $N \to \infty$ it will converge to the right trajectory.

### 11.2.1 Example A continued: Euler integration of a simple 1d system

Consider the following variant of the system we considered in section 11.1.1, but lets now include the control signal:

$$\dot{x}(t) = -ax(t) + bu(t)$$

If we apply Euler integration, we get the following simple update rule:

$$x_{k+1} = x_k - \Delta a x_k + \Delta b u_k$$

Note that for any choice actions $u_k$, this can be implemented using a simple `for`-loop.

### 11.2.2 Example B continued: Euler integration of the harmonic oscillator

As a more involved example, we can apply Euler integration to the Harmonic oscillator example section 11.1.2. In this case we must write the updates in the coordinates $\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$ (see eq. (10.19)):

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u \tag{11.12}$$

If we plug this into the Euler update we get:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \begin{bmatrix} 0 & 1 \\ -\Omega^2 & 0 \end{bmatrix} \boldsymbol{x}_k. \tag{11.13}$$

An example where the system is simulated for $t_F = 100$ seconds, and using two values of $N$, can be found in fig. 11.1. As seen, even for $N = 1000$ the result is not very accurate. In the example Euler integration fails to conserve energy, and the simulation will eventually diverge. Simply put, Euler integration is not to be trusted unless $N$ *very* large.

## 11.3 Runge-Kutta

Runge-Kutta is a family of integration methods for ordinary differential equations which are more exact than the Euler method. For our purpose, it is sufficient to focus on the classic Runge-Kutta method (RK4), which I will simply present as a numerical recipe:

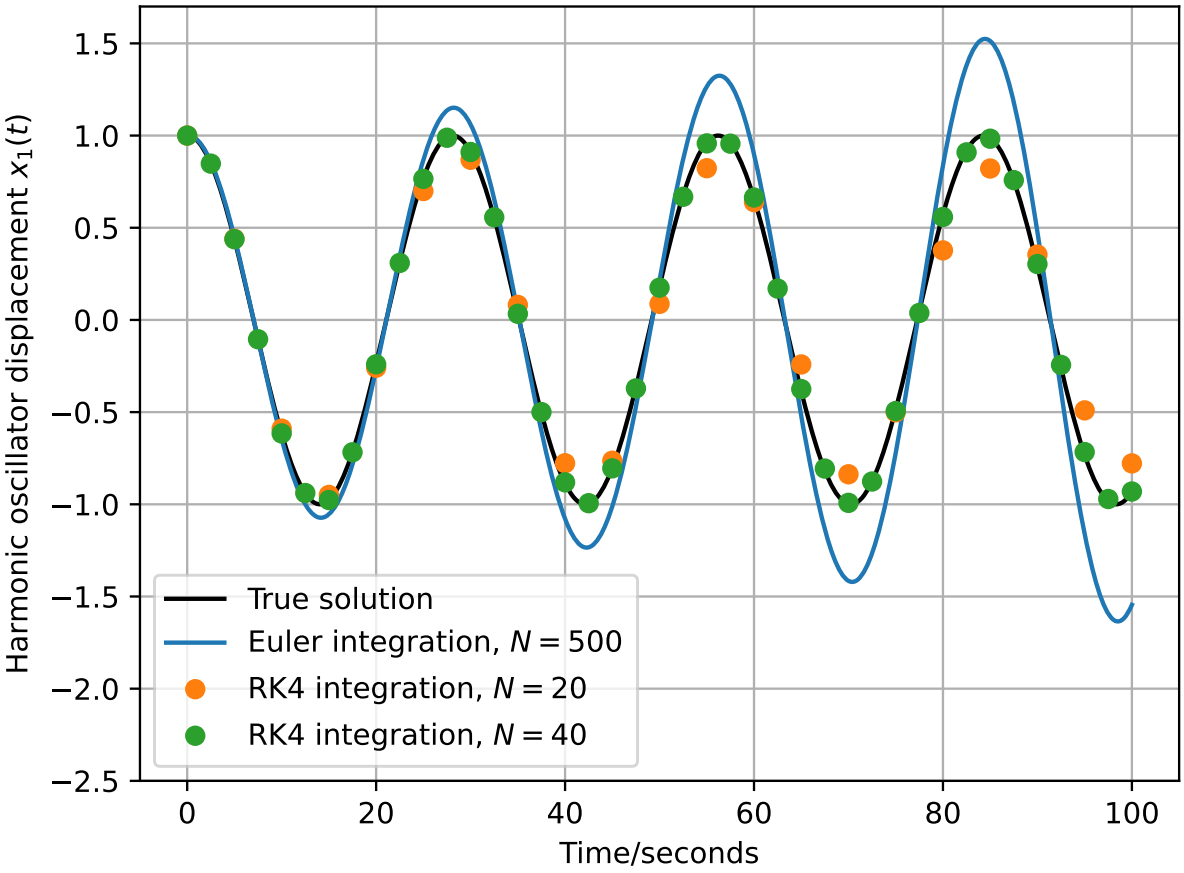


Figure 11.2: Continuing the Harmonic oscillator environment but including two RK4 solution. Even for low $N$, RK4 remains reasonably precise.

**Definition 11.3.1.** *RK4 The problem we consider is of the form:*

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \tag{11.14}$$

*Assume we discretize the problem using a step-size* $\Delta$ *and defining* $t_k = \Delta k$, $\boldsymbol{x}_k = \boldsymbol{x}(t_k)$, *we then compute* $\boldsymbol{x}_{k+1}$ *from* $\boldsymbol{x}_k$ *as*

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{1}{6}\Delta\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

*where*

$$\begin{aligned} k_1 &= \boldsymbol{f}\left(t_k, \boldsymbol{x}_k\right) \\ k_2 &= \boldsymbol{f}\left(t_k + \tfrac{\Delta}{2}, \boldsymbol{x}_k + \Delta\tfrac{k_1}{2}\right) \\ k_3 &= \boldsymbol{f}\left(t_k + \tfrac{\Delta}{2}, \boldsymbol{x}_k + \Delta\tfrac{k_2}{2}\right) \\ k_4 &= \boldsymbol{f}\left(t_k + h, \boldsymbol{x}_k + \Delta k_3\right) \end{aligned}$$

RK4 integration is much more precise than Euler integration at the same computational budget, and it is not much more difficult to implement. To use RK4 for a control problem we use the RK4 algorithm above with the substitution:

$$\boldsymbol{f}(t, \boldsymbol{x}) \leftarrow \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$$

☞ **Algorithm 18** RK4 for simulating a control problem, see theorem 11.3.1

**Require:** Dynamics $\boldsymbol{f}$, start/end time $t_0 < t_F$, grid size $N \geq 1$
**Require:** Initial state $\boldsymbol{x}(t_0)$ and control $\boldsymbol{u}(t)$, $t \in [t_0, t_F]$
**Ensure:** Approximate trajectory $\boldsymbol{x}(t_0) = \boldsymbol{x}_0, \ldots, \boldsymbol{x}_N \approx \boldsymbol{x}(t_F)$
1: $t_k \leftarrow t_0 + \frac{k}{N}(t_F - t_0)$
2: $\Delta = \frac{t_F - t_0}{N}$
3: $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}(t_0)$
4: $\boldsymbol{u}_k \leftarrow \boldsymbol{u}(t_k)$
5: **for** $k = 0, \ldots, N-1$ **do**
6: Compute

$$\begin{aligned} k_1 &= \boldsymbol{f}\left(\boldsymbol{x}_k, \boldsymbol{u}_k\right) \\ k_2 &= \boldsymbol{f}\left(\boldsymbol{x}_k + \Delta\frac{k_1}{2}, \boldsymbol{u}\left(t_k + \frac{\Delta}{2}\right), t_k + \frac{\Delta}{2}\right) \\ k_3 &= \boldsymbol{f}\left(\boldsymbol{x}_k + \Delta\frac{k_2}{2}, \boldsymbol{u}\left(t_k + \frac{\Delta}{2}\right), t_k + \frac{\Delta}{2}\right) \\ k_4 &= \boldsymbol{f}\left(\boldsymbol{x}_k + \Delta k_3, \boldsymbol{u}(t_{k+1}), t_{k+1}\right) \end{aligned} \tag{11.15}$$

7: $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \frac{1}{6}\Delta\left(k_1 + 2k_2 + 2k_3 + k_4\right)$
8: **end for**

The result can be found in algorithm 18

In fig. 11.2 we see that using just $N = 40$ grid points Runge-Kutta is in near-perfect agreement with the true solution, and it is roughly on-par with Euler integration using $N = 1000$ time steps. Whenever we wish to test a control signal we should always use RK4[2], and whenever we apply Euler integration it should always be assumed that the result is not very exact.

### 11.3.1 Evaluating the cost

The cost function is typically fairly smooth (i.e., linear or quadratic) and the choice of numerical integration scheme will be less important. We will therefore rely on the simple Riemann approximation which is:

$$\int_{t_k}^{t_{k+1}} f(t)dt \approx (t_{k+1} - t_k)f(t_k)$$

(valid when $t_{k+1} - t_k$ is small). The value of the cost function can therefore be approximated by:

$$J(x_0) = c_f(t_0, t_F, x_0, x_F) + \Delta\sum_{k=0}^{N-1} c(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k).$$

[2]Or a comparable high-order integration scheme

### 11.3.2 Comments on simulation

The Runge-Kutta method, or a similar high-order method with $N$ chosen reasonably large ($N = 1000$ will do in nearly all cases) provides an answer as to *what* the system does when we apply a particular control policy $\boldsymbol{u}$, however it does not tell us how to find $\boldsymbol{u}$.

One approach, inspired by the discretization scheme above, is to simply consider $\boldsymbol{u}$ as only being defined on the grid-points $\boldsymbol{u}_k$, and then select values at the grid-points which produces trajectories with low cost and which satisfy the constraints. In this manner a problem of determining a function $\boldsymbol{u} : \mathbb{R} \to \mathbb{R}^d$ becomes one of determining a finite set of vectors $(\boldsymbol{u}_k)_k$. In general, turning a continuous-time control problem into one defined on a discrete grid is known as *discretization*.

Discretization methods for simulation, and discretization methods to create a control problem, has a large overlap, but their purpose is different. When we discretized (in order to simulate, for instance using RK4), we want to select $N$ as large as possible because we *want to know what the system actually does*. For control, each value of $\boldsymbol{u}_k$ is a new vector of variables we have to optimize over (or select using some other method), and the concern is therefore to select $N$ low enough that the problem remains feasible.

# Chapter 12

# Linear-quadratic problems

LINEAR QUADRATIC

**Linear quadratic** problems are by far the most important and well-studied class of problems in control theory. The basic feature of the linear-quadratic control problem is that the dynamics is linear:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t) + \boldsymbol{d} \tag{12.1}$$

We assume $t_0 = 0$ and that the cost-function contains linear, quadratic and constant terms. In general:

$$\begin{aligned} J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = &\frac{1}{2}\boldsymbol{x}^T(t_F) Q_F \boldsymbol{x}(t_f) + \boldsymbol{q}_F^\top \boldsymbol{x}(t_F) + q_{c,F} \\ &+ \int_0^{t_f} \left( \frac{1}{2}\boldsymbol{x}^T(t) Q \boldsymbol{x}(t) + \boldsymbol{q}^\top \boldsymbol{x}(t) + q_c + \frac{1}{2}\boldsymbol{u}^T(t) R \boldsymbol{u}(t) + \boldsymbol{r}^\top \boldsymbol{u}(t) \right) dt \end{aligned} \tag{12.2}$$

This expression is a bit overwhelming, so it is worth emphasizing that by far the most important terms are the two quadratic terms associated with $\boldsymbol{x}$ and $\boldsymbol{u}$:

$$J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = \int_0^{t_f} \left( \frac{1}{2}\boldsymbol{x}^T(t) Q \boldsymbol{x}(t) + \frac{1}{2}\boldsymbol{u}^T(t) R \boldsymbol{u}(t) \right) dt \tag{12.3}$$

For the linear-quadratic problem to be well-defined we require that $Q$ and $Q_F$ are positive semi-definite and $R$ is positive definite. If this was no so, the cost could be made arbitrarily small by choosing e.g. an $\boldsymbol{u}$ such that $\boldsymbol{u}^\top R \boldsymbol{u}$ was arbitrarily small.

**Linear terms in the cost-function:** Assuming $Q$, $Q_F$ and $R$ meet these requirements eq. (12.3) is minimized when $\boldsymbol{x} = \boldsymbol{u} = 0$. In this case the control sequence $\boldsymbol{u}^*$ will try to drive the system towards $\mathbf{0}$.

Suppose on other hand we wanted to drive the system towards goal states $\boldsymbol{x}_g$ and $\boldsymbol{u}_g$. We can modify our quadratic cost function slightly so it is at a minimum at these

new values:

$$J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = \frac{1}{2}\int \left[(\boldsymbol{x}-\boldsymbol{x}_g)^\top Q(\boldsymbol{x}-\boldsymbol{x}_g) + (\boldsymbol{u}-\boldsymbol{u}_g)^\top \boldsymbol{R}(\boldsymbol{u}-\boldsymbol{u}_g)\right] dt \tag{12.4}$$

$$= \int \left[\frac{1}{2}\boldsymbol{x}^\top Q\boldsymbol{x} + \boldsymbol{q}^\top \boldsymbol{x} + \frac{1}{2}\boldsymbol{u}^\top R\boldsymbol{u} + \boldsymbol{r}^\top \boldsymbol{u} + q_c\right] dt \tag{12.5}$$

where: $$\boldsymbol{q} = Q\boldsymbol{x}^* \tag{12.6}$$

$$\boldsymbol{r} = R\boldsymbol{u}^* \tag{12.7}$$

$$q_c = \frac{1}{2}(\boldsymbol{x}^*)^\top \boldsymbol{Q}\boldsymbol{x}^* + \frac{1}{2}(\boldsymbol{u}^*)^\top \boldsymbol{R}\boldsymbol{u}^* \tag{12.8}$$

As the calculation shows the cost-function keep the linear-quadratic form. In other words, the role of the linear terms has to do with specifying an objective different than $\boldsymbol{x} = 0$.

The linear-quadratic problem has the virtue it is one of the few problems in control theory we can solve analytically. As we will see later, most non-linear problems can be approximated by a linear-quadratic problem and this will give rise to an important approximate solution method (iLQR).

## 12.1 An exact solution to linear problems

As we saw in the example section 11.1, it was possible to exactly solve two simple linear system without applied control. The solution method can be generalized, and the resulting method, **exponential integration**, will allow us to create nearly exact, discrete, model of linear-quadratic systems in the next chapter.

EXPONENTIAL INTEGRATION

Our starting point is still the linear system dynamics:

$$\dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t) + \boldsymbol{d} \tag{12.9}$$

MATRIX EXPONENTIAL

To solve this in general it is useful to define the so-called **Matrix exponential**, which for any $n \times n$ matrix $A$ is defined using the Taylor series of the exponential function[1]:

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!} = I + A + \frac{1}{2}A^2 + \frac{1}{6}A^3 + \cdots . \tag{12.10}$$

The reason we are interested in the matrix exponential is because it satisfy the following property, which you can easily check by simply differentiating the right-hand side of eq. (12.10) with respect to $t$:

$$\frac{d}{dt}e^{At} = Ae^{At}. \tag{12.11}$$

[1]The matrix exponential can be imported as `from numpy.linalg import expm` and then simply used as `expm(A)`

We first multiply both sides of eq. (12.9) with the matrix exponential to get:

$$e^{-At}\dot{\boldsymbol{x}}(t) = e^{-At}A\boldsymbol{x}(t) + e^{-At}(B\boldsymbol{u}(t) + \boldsymbol{d}) \tag{12.12}$$

Re-arranging the terms:

$$e^{-At}(B\boldsymbol{u}(t) + \boldsymbol{d}) = e^{-At}A\boldsymbol{x}(t) - e^{-At}\dot{\boldsymbol{x}}(t) = \frac{d}{dt}\left[e^{-At}\boldsymbol{x}(t)\right] \tag{12.13}$$

We get rid of the derivative on the right-hand side by integrating from $t_0$ to $t$:

$$\int_{t_0}^{t_f}\left(\frac{d}{dt}e^{-At}\boldsymbol{x}(t)\right)dt = \int_{t_0}^{t_f} e^{-At}(B\boldsymbol{u}(t) + \boldsymbol{d})dt \tag{12.14}$$

$$\Rightarrow e^{-At}\boldsymbol{x}(t) - e^{-At_0}\boldsymbol{x}(t_0) = \int_{t_0}^{t_f} e^{-At}(B\boldsymbol{u}(t) + \boldsymbol{d})dt \tag{12.15}$$

$$\Rightarrow \boldsymbol{x}(t) = e^{A(t-t_0)}\boldsymbol{x}(t_0) + \int_{t_0}^{t} e^{A(t-\tau)}(B\boldsymbol{u}(\tau) + \boldsymbol{d})d\tau \tag{12.16}$$

This is still slightly unwieldy, however, assuming $\boldsymbol{u}$ is constant and $A$ is invertible this becomes:

$$\boldsymbol{x}(t) = e^{A(t-t_0)}\boldsymbol{x}(t_0) + \int_{t_0}^{t} e^{A(t-\tau)}(B\boldsymbol{u} + \boldsymbol{d})d\tau \tag{12.17}$$

$$= e^{A(t-t_0)}\boldsymbol{x}(t_0) + A^{-1}(e^{A(t-t_0)} - I)(B\boldsymbol{u} + \boldsymbol{d}) \tag{12.18}$$

This tells us that when we use linear dynamics, the solution at a later time $\boldsymbol{x}(t)$ can be computed exactly from $\boldsymbol{x}(t_0)$ assuming $\boldsymbol{u}$ is constant in $[t_0; t]$.

**What if $A$ is not invertible?**

The last expression $A^{-1}(e^{A(t-t_0)} - I)$ assumes that $A$ is invertible, which naturally raises the question what happens when $A$ is not invertible. If we think about this as engineers, it is easy to come up with physical systems where $A$ is not invertible. Therefore, the problems with $A^{-1}$ can be expected to go away with a little more mathematical. The solution for general matrices $A$ turn out to be simple: we can simply use the definition of the matrix exponential as a Taylor series eq. (12.10) directly, and write the integral as an infinite sum of terms that don't depend on $A^{-1}$.

### 12.1.1 Example 2: Level flight for a 747

As a more involved example of the generality of the linear-quadratic control problem we will consider the problem of controlling a Boeing 747 in level flight condition described by the coordinate system indicated in fig. 12.1.

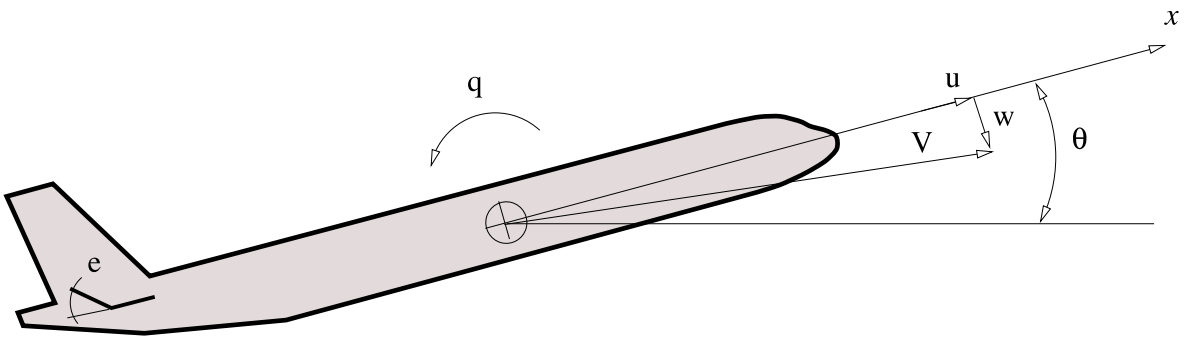


Figure 12.1: Illustration of coordinate system of Boing 747 level-flight model

A simplified model, suitable for level flight, can be described as the linear model:

$$\begin{bmatrix} \dot{u} \\ \dot{w} \\ \dot{q} \\ \dot{\theta} \end{bmatrix} = \underbrace{\begin{bmatrix} -0.003 & 0.039 & 0. & -0.322 \\ -0.065 & -0.319 & 7.74 & 0. \\ 0.02 & -0.101 & -0.429 & 0. \\ 0. & 0. & 1. & 0. \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u - u_w \\ w - w_w \\ q \\ \theta \end{bmatrix}}_{\boldsymbol{x}} + \underbrace{\begin{bmatrix} 0.01 & 1. \\ -0.18 & -0.04 \\ -1.16 & 0.598 \\ 0. & 0. \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} e \\ t \end{bmatrix}}_{\boldsymbol{u}} \tag{12.19}$$

For the state $\boldsymbol{x}$, the coordinate $u$ refers to the the speed in the longitudinal direction and $w$ is the speed in the direction perpendicular to $u$. The values $u_w$ and $w_w$ are constants, signifying the value of $u$ and $w$ relevant for the level flight conditions at which the model is derived. $\theta$ is the angle with respect to the horizontal and $q$ the angular velocity.

In other words, $x_1 > 0$ will correspond to the *increase* in airspeed (a small value) relative to the default airspeed $u_w$. This interpretation explain why $\boldsymbol{x}$ is small and the dynamics is still $\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u}$. Note the control $\boldsymbol{u}$ corresponds to the elevator angle $e$ and the throttle $t$.

The cost function is defined by first defining the output variable $\boldsymbol{y}$

$$\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \underbrace{\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & -1. & 0. & 7.74 \end{bmatrix}}_{=P} \begin{bmatrix} u(t) - u_w(t) \\ w(t) - w_w(t) \\ q(t) \\ \theta(t) \end{bmatrix} \tag{12.20}$$

The two variables $y_1$ and $y_2$ corresponds to the airspeed and climb rate respectively. The cost function is then defined using these two variables:

$$J = \int_0^{t_F} \left( \frac{1}{2} \|\boldsymbol{y}\|^2 + \frac{1}{2} \|\boldsymbol{u}\|^2 \right) d\tau \tag{12.21}$$

and is seen to have the standard LQ form with $R = 1\boldsymbol{I}$ and $Q = P^\top P$. For a realistic airplane there are obviously restrictions on the available controls and possible plane configurations, however, since the system is formulated as small deviations around the

intended behavior the autopilot will not encounter the boundary conditions. In other words, the problem is formulated without constraints, and it is assumed some other part of the autopilot checks if the computed control trajectory is in fact feasible.

# Chapter 13

# Discretization of a control problem

The previous chapter defined the (idealized) continuous control-problem as the dynamics $\boldsymbol{f}$, a cost function and a set of constraints. We call this the continuous-time formulation of the control problem. In this section, we will consider how we can transform $\boldsymbol{f}$ to an (approximate) discrete model $\boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k)$ which will add additional techniques to our toolbox.

## 13.1 Building models by discretization

Suppose we want to build a controller that can balance a physical pendulum robot. In this case there are two practical limitations:

- For the controller, computing the control signal $\boldsymbol{u}(t)$ given $\boldsymbol{x}(t)$ takes some time, and can therefore only be done a certain number of times a second
- The robot can only process a limited number of control commands $\boldsymbol{u}(t)$ in a given time period

To overcome these limitations, many practical control systems plan using a discrete model of the dynamics. Specifically, a discretization step $\Delta$ is chosen and we then assume:

- Time is discretized as $t_k = t_0 + \Delta k$, $k = 0, \ldots, N$ where $t_N = t_F$.
- The agent interacts with the world at time $t_k$, thereby observing $\boldsymbol{x}_k = \boldsymbol{x}(t_k)$, and (immediately) sends a control command $\boldsymbol{u}_k = \boldsymbol{u}(t_k)$
- It is assumed that when the Agent applies control $\boldsymbol{u}_k$, then this control is applied constantly to the environment in the time interval $[t_k; t_{k+1}[$ (sometimes called zero-hold), see fig. 13.1


Figure 13.1: Zero-hold discretization of a controller: The control signal which is actually applied to the environment is $\boldsymbol{u}(t) = \boldsymbol{u}_k$ for $u \in [t_k, t_{k+1}[$

Thus, the agent will plan its controls using a model/cost-function of the form:

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{13.1}$$

$$J_{\boldsymbol{u}=(\boldsymbol{u}_0,\boldsymbol{u}_1,\ldots,\boldsymbol{u}_{N-1})}(\boldsymbol{x}_0) = c_f(t_0, \boldsymbol{x}_0, t_F, \boldsymbol{x}_F) + \sum_{k=0}^{N-1} c_k(\boldsymbol{x}_k, \boldsymbol{u}_k). \tag{13.2}$$

where $\boldsymbol{f}_k$ and $c_k$ needs to be determined. This looks nearly identical to simulation, and indeed there will be overlap, however it should be emphasized that the model $\boldsymbol{f}_k$ is *internal to the agent* and one which we have to construct from the true dynamics $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)$ or alternatively, learn. This model will be subject to other demands than simulation; if the model is defined using a very fine grid, or it is expensive to compute, it may simply not be feasible to use it for planning. For these reasons it should from now on, as will nearly always be the case, simply be assumed the model is a fairly coarse approximation.

### 13.1.1 Example: The discrete linear-quadratic model

DISCRETE LINEAR-QUADRATIC MODEL

The **discrete linear-quadratic model**, which we will study in greater detail later, has dynamics of the form:

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d}_k \tag{13.3}$$

Where $A_k$, $B_k$ and $\boldsymbol{d}_k$ are fixed and known. The cost function is assumed to be quadratic, for instance:

$$c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) = \frac{1}{2}(\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \boldsymbol{u}_k^T R_k \boldsymbol{u}_k) \tag{13.4}$$

$$c_N(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T Q_N \boldsymbol{x}_k \tag{13.5}$$

where $Q_k \in \mathbb{R}^{n\times n}$ is positive semi-definite and $R_k \in \mathbb{R}^{m\times m}$ is positive definite for all $k = 0, \ldots, N$. The linear-quadratic model is one of the most well-studied objects in control theory, both because it is flexible and because it allows for quick simulation.

Our task, when we want to use a linear-quadratic model, is therefore to somehow define $A_k$ and $B_k$ from the true dynamics $\boldsymbol{f}$.

### 13.1.2 Discretization using Euler integration

Our main discretization procedure is to simply use Euler integration. Given a discretization time $\Delta$ we simply define:

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) = \boldsymbol{x}_k + \Delta f(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k). \tag{13.6a}$$

**Why not RK4?** It may appear odd why Eulers method and not e.g. RK4 is a prevalent discretization method. Arguments in favor of Euler discretization includes that it is simpler, meaning that if we want to compute gradients of $\boldsymbol{f}_k$ it involves less algebra, and that it is easier to perform error analysis (a subject not covered in this course). A more honest answer is that the various implementations of control methods I have looked at when planning this course all appears to use Eulers method and so I stuck to that choice.

At any rate, a consequence of using Eulers method, especially since the time constant $\Delta$ will often be fairly large, is that the discretized model should always be assumed to be approximate at best.

### 13.1.3 The special case of linear dynamics

Linear dynamics can be used to illustrate Euler discretization. Recall linear dynamics refers to

$$\dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t) + \boldsymbol{d} \tag{13.7}$$

and we consider $A$ and $B$ as independent of time. Simply inserting into eq. (13.6) we get

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \Delta f(\boldsymbol{x}_k, \boldsymbol{u}_k) && (13.8)\\ &= (I + \Delta A)\boldsymbol{x}_k + \Delta B\boldsymbol{u}_k + \Delta \boldsymbol{d} && (13.9) \end{aligned}$$

If we consider the general form of the discrete linear model $\boldsymbol{x}_{k+1} = A_k\boldsymbol{x}_k + B_k\boldsymbol{u}_k + \boldsymbol{d}_k$ we quickly see the Euler discretization is equivalent to selecting $A_k = I + \Delta A$, $B_k = \Delta B$ and $\boldsymbol{d}_k = \Delta \boldsymbol{d}$.

But we can in fact do a lot better. Recall we showed in section 12.1, eq. (12.18), that:

$$\boldsymbol{x}(t) = e^{A(t-t_0)}\boldsymbol{x}(t_0) + A^{-1}(e^{A(t-t_0)} - I)(B\boldsymbol{u} + \boldsymbol{d}) \tag{13.10}$$

If we consider the special case where $t_0 = t_k$ and $t = t_{k+1} = t_k + \Delta$, and recall that the control signal per definition is constant between $[t_k; t_{k+1}[$, then the *actual* next state $\boldsymbol{x}_{k+1}$ will be:

$$\boldsymbol{x}_{k+1} = e^{A\Delta}\boldsymbol{x}_k + A^{-1}(e^{A\Delta} - I)B\boldsymbol{u}_k + A^{-1}(e^{A\Delta} - I)\boldsymbol{d}_k \tag{13.11}$$

So if the continuous dynamics is linear, we can obtain an *exact*[1] discrete model by using the coefficient matrices[2] $A_k = e^{A\Delta}$, $B_k = A^{-1}(A_k - I)B$ and $\boldsymbol{d}_k = A^{-1}(A_k - I)\boldsymbol{d}$

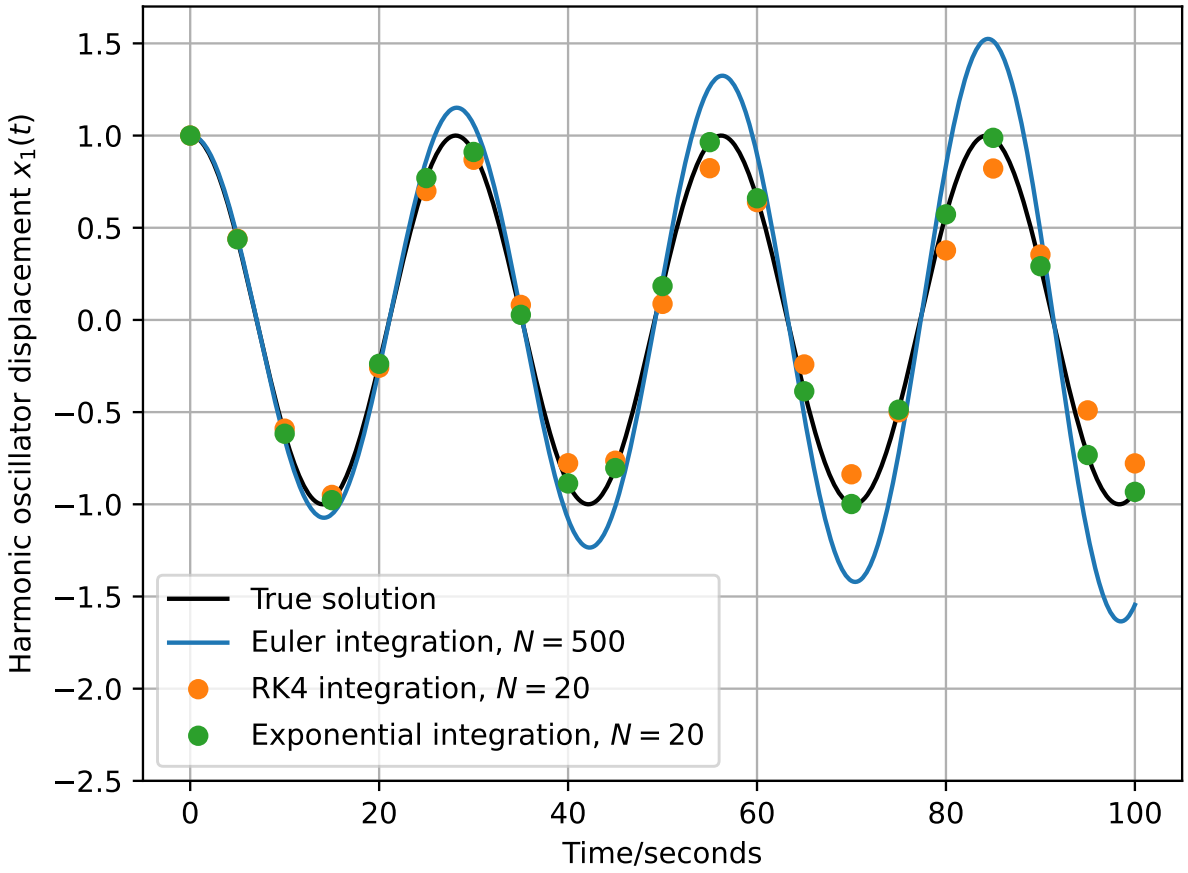


Figure 13.2: The harmonic oscillator example but also showing the result using exact integration method described in section 13.1.3. Since no force is applied, the exact integration method will agree with the true solution regardless of how low $N$ is.

EXPONENTIAL INTEGRATION

. This method is called **exponential integration**, and since the overhead involved in computing the coefficient matrices (a matrix exponential and a matrix inverse) is usually negligible, it should always be preferred over Euler integration – unfortunately it is only applicable to linear models. Figure 13.2 show exponential integration applied to the Harmonic oscillator environment using a very coarse time discretization. As we can see, because the force is $u = 0$ the simulation agrees exactly with the true solution regardless of $N$.

### 13.1.4 Coordinate transformations

In the pendulum example the state is represented as

$$\boldsymbol{x} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}.$$

[1] assuming the control signal $u(t)$ is constant on all intervals $[t_k, t_{k+1}[$

[2] Although this derivation assumes that $A$ is invertible this is not necessary, and you can apply exponential integration to systems where $A$ is not invertible such as $A = 0$

As we already saw, the angular coordinate $\theta$ has the annoying property it is periodic in $2\pi$. This will create a problem when we design cost functions since a linear or quadratic term in $\theta$, which corresponds to the same system configuration, can become arbitrarily large. This issue is not specific to to the pendulum, but is to some degree present in any mechanical system described by angles[3].

A way to get around this problem is to change the coordinates of the angle to be expressed in terms of sin and cos. In other words we apply the transformation:

$$\phi_x : \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} \mapsto \begin{bmatrix} \sin(\theta) \\ \cos(\theta) \\ \dot{\theta} \end{bmatrix}. \tag{13.12}$$

This expands the dimensionality of the state from 2 to 3, and has the benefit of making the cost function easier to express. We can do something similar with the action-vector. In the pendulum example, recall that $u$ is subject to the constraint: $-U \leq u \leq U$, however if we transform $u$ to the new coordinates:[4]

$$\phi_u : \begin{bmatrix} u \end{bmatrix} \mapsto \begin{bmatrix} \tanh^{-1} \frac{u}{U} \end{bmatrix}. \tag{13.13}$$

then the new coordinate vector will lie in $]-\infty; \infty[$ and will therefore not be constrained.

### 13.1.5 Discretization the cost

The discretization of the cost is exactly the same as for simulation:

$$J_{\boldsymbol{u}}(\boldsymbol{x}_0) = c_N(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{13.14}$$

$$c_N(\boldsymbol{x}_N) = c_f(\boldsymbol{x}_0, t_0, \boldsymbol{x}_N, t_N) \tag{13.15}$$

$$c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) = \Delta c(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k). \tag{13.16}$$

### 13.1.6 Discretization of an environment

We can now construct a discretized version of any control problem $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t)$ using the following recipe:

- Select a discretization time step $\Delta$
- (Optionally:) Select coordinate mappings $\phi_x$ and $\phi_u$

[3]Furthermore it not one we can easily overlook; my own attempts at using control methods such as iLQR on the Pendulum environment without transformations failed.

[4]Recall that tanh maps from $]-\infty, \infty[$ to $]-1; 1[$

- Let $\boldsymbol{x}^d = \phi_x(\boldsymbol{x})$ and $\boldsymbol{u}^d = \phi_u(\boldsymbol{u})$ be the coordinates used in the *discrete* model. The Euler discretization of the dynamics is:

$$\boldsymbol{x}^d_{k+1} = \phi_x\left(\phi_x^{-1}(\boldsymbol{x}^d_k) + \Delta \boldsymbol{f}(\phi_x^{-1}(\boldsymbol{x}^d_k), \phi_u^{-1}(\boldsymbol{u}_k), t_k)\right) \tag{13.17}$$

$$= \boldsymbol{f}_k(\boldsymbol{x}^d_k, \boldsymbol{u}^d_k) \tag{13.18}$$

- The total cost, as well as the cost obtained in each time step, is given in eq. (13.14).
- In the specific case of linear dynamics we use exponential integration eq. (13.11) (for simplicity variable transformations are omitted since they are not relevant):

$$\boldsymbol{x}^d_{k+1} = e^{\tilde{A}\Delta}\boldsymbol{x}^d_k + (\tilde{A})^{-1}(e^{\tilde{A}\Delta} - I)(B\boldsymbol{u}^d_k + \boldsymbol{d}_k). \tag{13.19}$$

One slight problem with this expression is what occurs when $A$ is not invertible. The true answer is the case can be addressed by an appropriate factorization of $A$, however I have adapted a much simpler approach by defining $\tilde{A} = A - 10^{-6}I$ in the case $A$ is singular, since this will ensure $\tilde{A}$ is non-singular and does not result in a meaningful change in the dynamics.

## 13.2 Notes on implementation

Please consult the online documentation for examples on how to discretize your own `ControlModel`. I have created models and environments corresponding to the systems we will often work with, for instance the Pendulum.

As an example, the following code define a continuous-time pendulum model (without coordinate transformations), simulate the effect of a policy, and plot the trajectory:

```
# chapter7contiuous/model_example_plot.py
cmodel = PendulumModel()
x0 = cmodel.x0_bound().low

def policy(x, t):
    return [3 * np.sin(2 * t)]

xx, uu, tt, cost = cmodel.simulate(x0, policy, t0=0, tF=10)
plt.plot(tt, xx[:, 0], label="$\\theta$")
plt.plot(tt, uu[:, 0], label="$u$")
```

The simulated trajectory is shown in fig. 13.3 The simulator uses RK4 on a fine grid, and can be considered the ground-truth effect of the policy. Note the policy has to return a list (or 1-d numpy array), because the action is always a vector.

### 13.2.1 Discretized model

It is easy to specify coordinate transformations (see the online documentation). This examples show the pre-made Pendulum model with the sin/cos coordinate transformation applied.

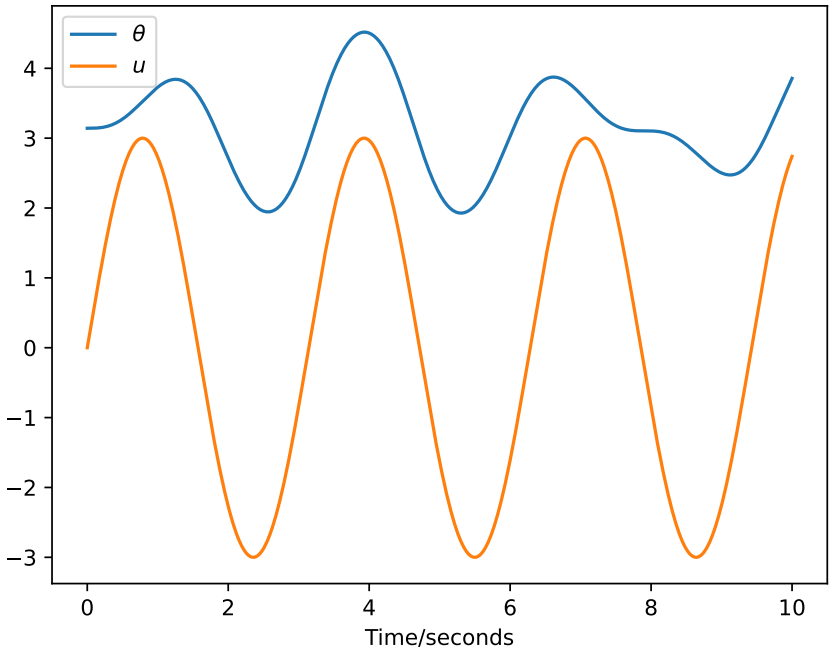


Figure 13.3: Simulated trajectory of the pendulum-environment

```
# chapter7contiuous/model_example.py
dmodel = DiscreteSinCosPendulumModel()
ud = sym.symbols(f"u0:{dmodel.action_size}")
xd = sym.symbols(f"x0:{dmodel.state_size}")
x_next = dmodel._f_discrete_sym(xd, ud, dmodel.dt)
print("First coordinate of f_k(x,u,t)", x_next[0]) # The symbolic expression for the first coordinate of the Euler upd
theta = np.pi/4
x = [np.sin(theta), np.cos(theta), 0.5] # Get a state.
u = [1]                                 # Get an action
print(dmodel.f(x,u) )
```

The script evaluates the dynamics $\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_0, u_k = 1)$ and produces the output:

```
First coordinate of f_k(x,u,t) sin(0.02*x2 + atan2(x0, x1))
[0.71414238 0.70000048 0.7531149 ]
```

Note the discretized pendulum now has an observation space of three coordinates to reflect the coordinate change.

### 13.2.2 Models to environments

Transforming a discrete model to an environment can also be done easily (see the online documentation). The following example shows an interaction with the Pendulum-environment corresponding to the previous model. Note that it has an action-space and the familiar `step` and `reset`-functions:

```
# chapter7continuous/model_example.py
env = GymSinCosPendulumEnvironment()
env.reset() # Reset both the time and state variable
u = env.action_space.sample()
next_state, reward, done, _, info = env.step(u)
print("Current state: ", env.state)
print("Current time", info['time_seconds'])
```

This produces output

```
Current state:  [-9.35435236e-04 -9.99999562e-01  9.35129154e-02]
Current time 0.02
```

### 13.2.3 Training control methods

The `Agent`/`train` interface does not change. All we need to remember is that the state `s` now corresponds to $\boldsymbol{x}$ and the reward is equal to minus one times the cost.

Let us try to let a random agent interact with our pendulum model. This is accomplished in the usual way:

```
    # chapter8discretization/random_agent.py
    env = GymSinCosPendulumEnvironment()
    agent = RandomAgent(env)
    stats, trajectories = train(env, agent, num_episodes=1, return_trajectory=True, verbose=False)
    print("Total cost: ", stats[0]['Accumulated Reward'])
    print("Trajectory length/steps: ", stats[0]['Length'])

    plt.plot(trajectories[0].time, trajectories[0].state[:,0], label="$\\sin(\\theta)$")
    plt.plot(trajectories[0].time, trajectories[0].state[:,1], label="$\\cos(\\theta)$")
    plt.xlabel("Time/seconds")
    plt.legend()
    savepdf("random_agent_pendulum_b")
```

With output

```
Total cost:  -30275.31835796231
Trajectory length/steps:  250
```

The environment terminated after 5 seconds (this is easy to set, see the online documentation), which for $\Delta = 0.02$ corresponds to 250 steps ($250 = \frac{5}{0.02}$). This mean the environments step-function is called 250 times, the agents policy (and train)-function is called 250, however we end up with 251 different states, since at the last step we still obtain a final state. For this reason the trajectories $x$-value as plotted above actually have length 251.

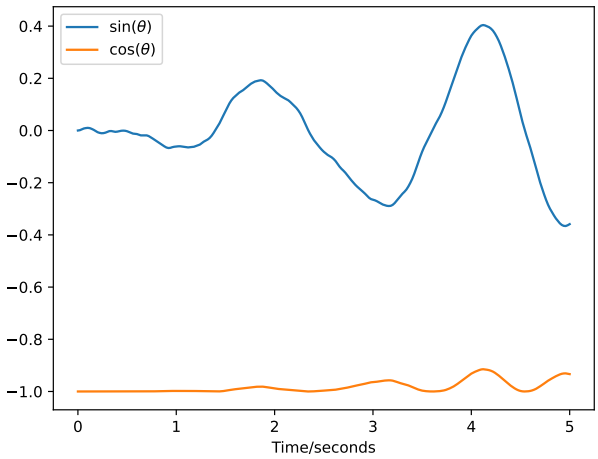


Figure 13.4: Plot of $\sin(\theta(t))$ and $\cos(\theta(t))$ arising letting a random agent interact with the pendulum environment.

# Chapter 14

# PID Control

Proportional-integral-derivative (PID) control takes a very direct approach to control, namely to simply define the control law $\boldsymbol{u}(t)$ directly from $\boldsymbol{x}(t)$ without using a model of the environment. It is therefore entirely heuristic and requires tuning by an engineer.

Despite the simplicity, it is said 90% of all control applications uses variations of PID control. This includes everything from large industrial plants to the read-head in a magnetic hard drive, and from air-condition units to the original Apollo 11 moon lander.

The principle is quite simple. Suppose we have to heat up a swimming pool using a heating element. In this case we observe the temperature $x(t)$, which is a single number, and we want to keep the temperature at a target temperature $x^*$. Our available control is a heating element where we can control the watts applied $u(t)$ (also a single number).

We could either try to come up with a model of the swimming pool, which would be difficult, or we could adopt a simple rule updating $u(t)$ in each time step:

$$u(t) \leftarrow \begin{cases} u(t) + \{\text{small amount}\} & \text{if } x(t) < x^* \\ u(t) - \{\text{small amount}\} & \text{if } x(t) \geq x^* \end{cases} \tag{14.1}$$

PID control builds on this idea, but also overcomes some of the limitations of this approach.

## 14.1 The P in PID ensures we reach our goal

The P in PID is for proportion, and is the most important term. Consider a problem analogous to the swimmingpool, namely the problem of controlling a 1d train on a track. The train has a location $x(t)$, we can apply a force $u(t)$ in the forward direction, and the train always starts at $x(0) = -1$. The goal for the train is to reach (and stand still) at the destination at $x = 1$.

The problem is formally equivalent to the Harmonic oscillator with $k = 0$, and so the train is described using the position and velocity. For simplicity we will just keep referring to $x(t)$ as the $x$-position.

Figure 14.1: A sketch of the locomotive environment. The locomotive has to drive to the target $x^* = 0$ indicated by the red triangle on the track. The locomotive starts at $x(0) = -1$ and can apply a force $u(t)$ in the forward/backward direction.

Suppose we define the goal as $x^* = 0$. The simple rule, increase force $u(t)$ if $x(t) < x^*$, can be defined by defining the error:

$$e(t) = x^* - x(t) \tag{14.2}$$

And then apply a control signal:

$$u(t) = K_p e(t) \tag{14.3}$$

The name, proportional, exactly refers to how the magnitude of the control is proportional to how far away from the goal we presently are. The full method, including two terms we have not discussed yet, can be found in algorithm 19, and the simple controller described in this section is recovered if we set $K_p > 0$ and $K_i = K_d = 0$.

If we apply the method to the locomotive problem we obtain the result in fig. 14.2, here using two values of $K_p$. We see that when $K_p$ is low, the magnitude of $u(t)$ will be low, hence the system slowly reaches the goal at $x^* = 0$, but then overshoots; after it overshoots the train begins to slowly break, and then overshoots the goal again in the opposite direction. Since there is no friction in the train system it will in fact keep oscillating/overshooting.

## 14.2 The D in PID control oscillations

The D-term stands for derivative and is intended to fix the problem of oscillations/overshooting.

Let's consider the problem focusing on the orange curve in fig. 14.2. The error starts out at $e(0) = 1$, giving rise to a control $u(0) = K_d$. The error then decreases, and so will $u(t)$ – but not fast enough to avoid overshoot! What we want is to add the

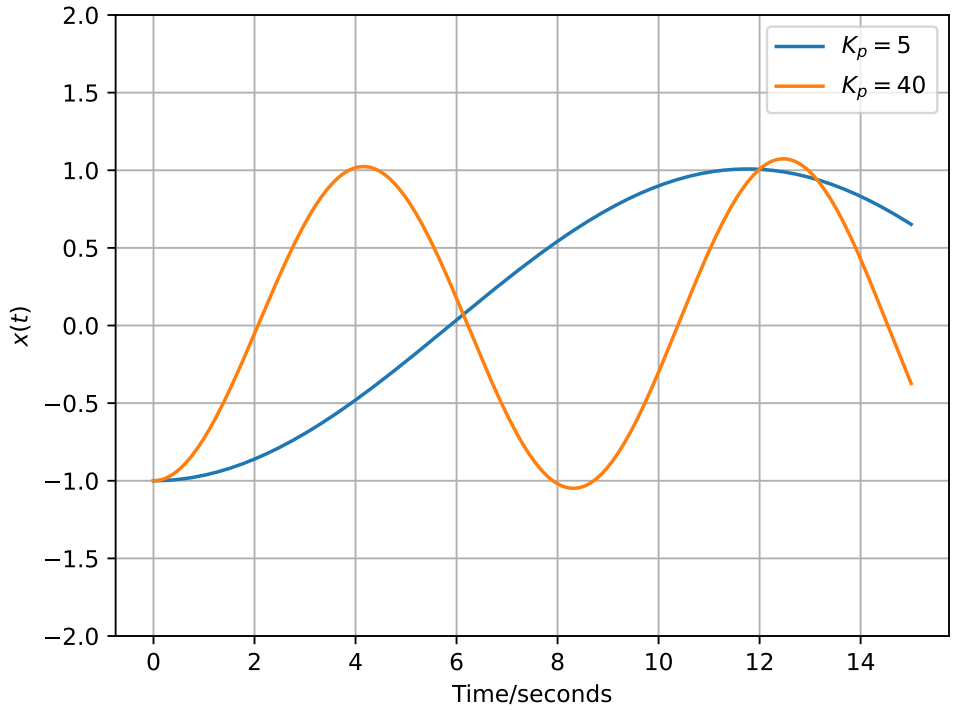


Figure 14.2: Train control using various values of $K_p$. The figure show the $x$-position of the train and the goal is to get to the position $x(t) = 0$ while the train is standing still.

following rule: If the error is positive but *decreasing*, then don't let $u(t)$ be as large as it otherwise would be. If the error is decreasing or not is measured by the derivative $\frac{de(t)}{dt}$, and so what we want is:

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} \tag{14.4}$$

We cannot compute this exactly but we can approximate it:

$$\frac{\mathrm{d}e(t)}{\mathrm{d}t} \approx \frac{e(t) - e(t - \Delta t)}{\Delta t}$$

Where $\Delta$ is the discretization time. The update rule is now:

$$u(t) = K_p e(t) + K_d \frac{e(t) - e(t - \Delta)}{\Delta} \tag{14.5}$$

So what we need is a way to keep track of $e(t)$ at the previous time step. The performance of the new controller is illustrated in fig. 14.3. If $K_d$ is relatively low, we still see the oscillating effect. If $K_d$ is increased to a high value, here $K_d = 100$, then the controller becomes hesitant and only reaches the target $x^* = 0$ very slowly. A reasonable middle-ground can be found with a bit of tuning, in this case $K_d = 50$.

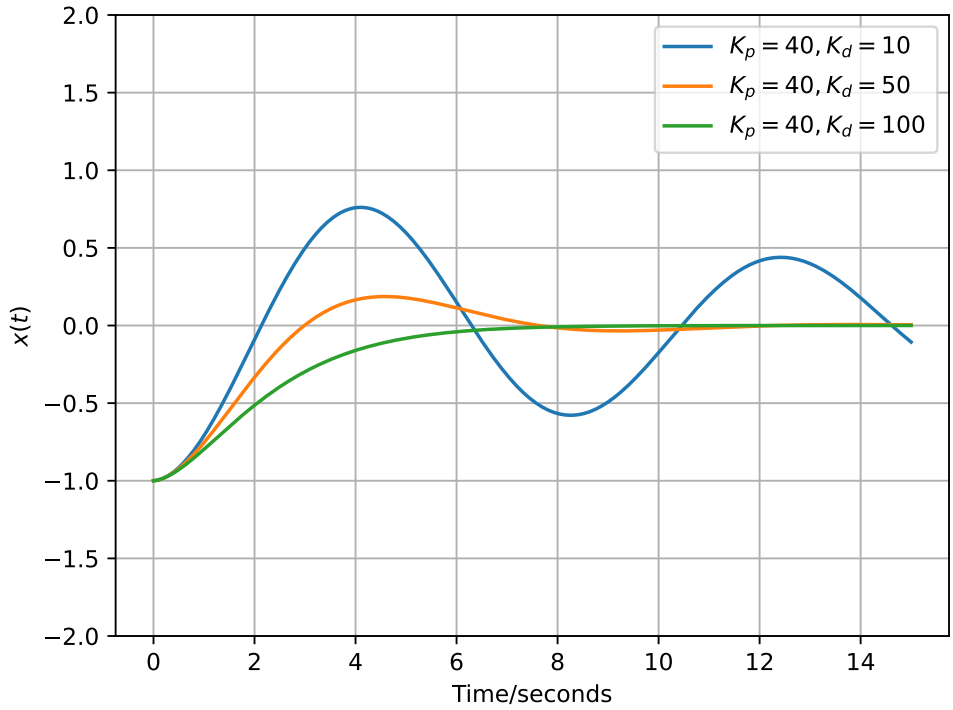


Figure 14.3: PID control for the train using $K_p = 40$ for various values of $K_d$

## 14.3 The I in PID fix droop

Lets suppose the train is put on an inclined plane as illustrated in fig. 14.4. Our $K_p = 40, K_d = 50$ controller will no longer fare well. The reason is that if the train is standing still at the target, $e(t) = 0$, then $\dot{e} = 0$ and $u(t) = 0$, in which case the slope would drag the train downwards. Thus, the $PD$-controller cannot stand still at $e(t) = 0$, but will in fact reach an equilibrium at around $x(t) = -0.6$. This effect is called droop and is illustrated in the right-hand pane of fig. 14.4: the train never reach the target $x^* = 0$.

The way to overcome this is to add a new term which, in effect, says: Whatever else is true, if $e(t) > 0$ for a long time, keep increasing $u(t)$. The meaning of *being low for a long time* can be implemented using the integral $I(t) = \int_0^t e(\tau)d\tau$ and so we obtain the ideal controller:

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{14.6}$$

To implement a discrete version of this, note the integral term can be discretized as:

$$I(t) = \int_0^{t-\Delta} e(\tau)d\tau + \int_{t-\Delta}^{t} e(\tau)d\tau = I(t-\Delta) + \Delta e(t) \tag{14.7}$$

And so to implement this method, all we need is a way to store the old value of the integral $I(t - \Delta)$ and update it in each step. As shown in fig. 14.5 this overcomes the problem of droop:

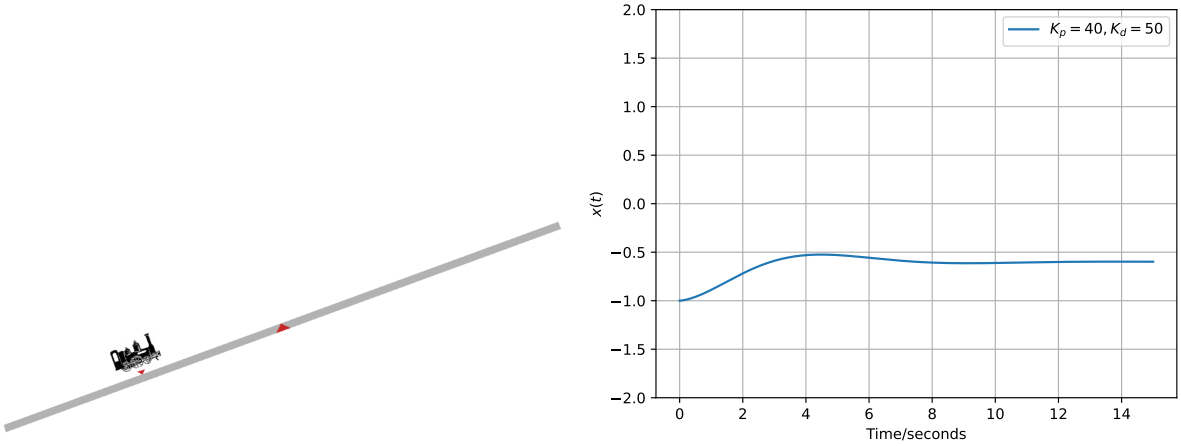


Figure 14.4: Locomotive on an inclined plane. The $K_p + K_d$-controller con no longer stand still at $x_t = x^*$ since when $e(t) = 0$, the controller will apply no force $u(t) = 0$, and so the train will slide backwards. This is called droop.

### 14.3.1 Tuning PID controllers

We can summarize these observations as follows:

- Set $K_i = K_d = 0$. Then select $K_p$ so high the system reaches the desired state reasonably quickly. Ensure the boundary conditions are not violated.
- If the system oscillates or overshoots dramatically, add a derivative term.
- If the system experience droop, add an integral term
- In both cases, consider reducing $K_p$ a little for stability.

This is only scratching the surface. There exist a large literature on tuning PID controllers and many practical issues such as multiple targets, better estimation of derivatives, stability, etc. etc.

## 14.4 Example: The car-model

The car model has two control parameters: The angle of the front wheels with respect to the body, $u_1(t)$, and the engine force $u_2(t)$. This does not exactly match our formulation since we considered the case of a single input $x$ and a single control $u$, however, we can use two PID controllers for the two output parameters.

Recall the state-vector $\boldsymbol{x}(t)$ contained some rather complicated terms, however, three of the coordinates capture our attention:

- $x_1 = v_x$: The velocity in the direction of the car frame, i.e. forward
- $x_4 = e_\psi$: The angle the car body forms with the centerline. I.e. if this angle is 0, we are driving in the same direction as the centerline, and if it is positive we are steering towards the right-most barrier.

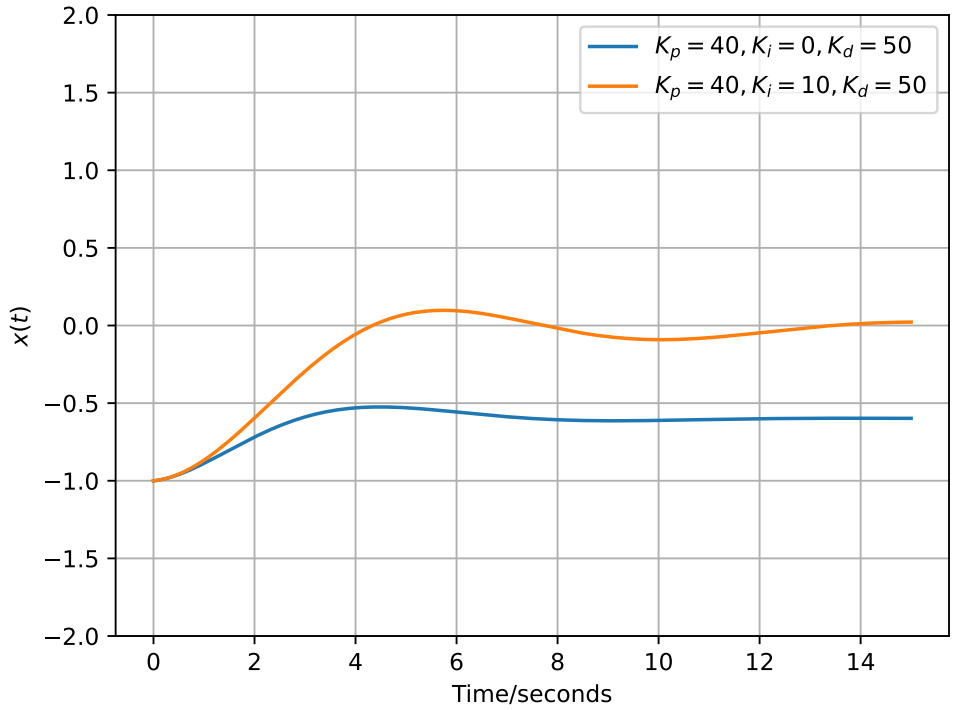


Figure 14.5: Locomotive on the inclined plane. When we set $K_i > 0$, the train will increase $u(t)$ in proportion to the total area of the error, thereby eventually overcoming the incline and reach $x(t) = x^*$.

- $x_6 = e_y$: How far, in the perpendicular direction, the car is from the centerline of the track.

The first idea is to make the car drive at a constant (slow) velocity. We can accomplish this by letting $x_1$ be the input to a PID controller, $u_2$ (engine force) the output (as computed by the PID), and then let $x^*$ be some (low!) desired velocity. It is now a simple matter of choosing $K_p, K_i, K_d$, which in this case can be done by choosing only $K_p$.

The steering wheels are a bit more tricky. One idea is to use a separate PID controller with $x_5$ be the input and $u_1$ as the output. If $x_5 > 0$, it means that we are to the right of the centerline, in which case we should steer left, in other words when the PID computes a control output of $u(t)$ and we apply $u_1(t) = u(t)$ to the car (as a practical matter, if we drove in the clockwise direction, we would apply $-u(t)$).

The result of the controller, using $K_p = 1$ (for the angle controller) and $K_p = 1.5$ (for the velocity controller) and a target of 0.3 for the velocity (all other terms are set to zero) is shown in fig. 14.6 as well as the path taken by the car (black)

We see the PID controller gets the job done, in the sense no constraints are violated and we eventually get to the goal in about 70 seconds, however it is awfully slow and not very stable, and experience oscillations in $x_5$ (distance to center-line) both due to the curvature of the track, which the controller is blissfully unaware of until it begins to go straight in a curve, as well as overshooting since when it drives towards the center-line

☞

**Algorithm 19** PID controller

$K_p$, $K_i$, and $K_d$
$\Delta$ time between observations $x_k$ (discretization)
$x^*$ Control target
$e^{\text{prev}} \leftarrow 0$ ▷ Previous value of error
**function** POLICY($x_k$) ▷ PID Controller called with observation $x_k$
    $e \leftarrow x^* - x_k$ ▷ Compute error
    $I \leftarrow I + \Delta e$ ▷ Update integral term
    $u \leftarrow K_p e + K_i I + K_d \frac{e - e^{\text{prev}}}{\Delta}$ ▷ PID control signal, including derivative term
    $e^{\text{prev}} \leftarrow e$ ▷ Save current error for next iteration
    **return** $u$
**end function**

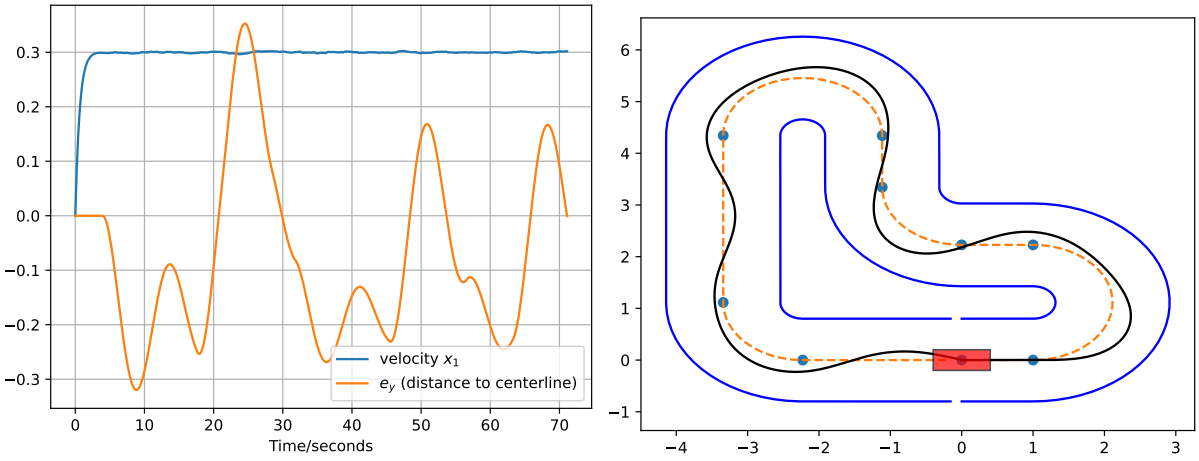


Figure 14.6: PID control of the car model using just the $K_p$ terms. The car tries (and succeeds) at driving with a constant (low) velocity and steers towards the centerline. We see the car experience oscillations/overshooting.

at an angle it will automatically overshoot.

One fix is to include a derivative term, however an alternative idea is to use the information in $x_4$, the angle with the centerline. The intuition is this angle may increase rapidly if the car enters a curve and so there may be benefit in also using this information. Doing this is quite simple: we just let the input to the PID controller could be

$$x(t) = x_4(t) + x_5(t).$$

This works better, in the sense of reducing the oscillations and allowing the car to travel at a greater speed, however I have not tried to extensively tune the parameters to the PID controllers and I am curious how fast the car could theoretically go using PID.

# Chapter 15

# Direct methods

As we in section 5.1.4, a deterministic problem can be solved using open-loop control. That is, an optimal policy is just a sequence of actions found by optimization. Since the models used in control, $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}, t)$ is deterministic one should expect optimal control is intrinsically tied to simple optimization. We will therefore explore optimization-based methods to optimal control in this chapter.

## 15.1 Optimization

Since this chapter will treat optimization, it is useful to discuss optimization from an abstract point of view before proceeding.

### 15.1.1 Non-linear optimization

NON-LINEAR PROGRAM

A general non-linear optimization problem, sometimes known as a **non-linear program**, typically take the following abstract definition for a minimization problem over $\boldsymbol{z} \in \mathbb{R}^n$:

$$\begin{array}{ll} \min_{\boldsymbol{z}} E(\boldsymbol{z}) & \text{subject to} \\ & \boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{0} \\ & \boldsymbol{g}(\boldsymbol{z}) \leq \boldsymbol{0} \\ & \boldsymbol{z}_{\text{low}} \leq \boldsymbol{z} \leq \boldsymbol{z}_{\text{upp}} \end{array} \tag{15.1}$$

For some functions $E$, $\boldsymbol{h}$ and $\boldsymbol{g}$. The goal is to find the optimal $\boldsymbol{z}^*$ which minimizes this expression.

SEQUENTIAL CONVEX PROGRAMMING

There are no general ways to solve such a problem, however, assuming the problem is not too complicated methods such as **sequential convex programming** may yield good results. We will therefore assume we have access to a way of solving such a problem in section 15.3 without understanding how it works.

What is important is to note that for these methods to work well, we need an initial guess for the solution (and the performance depends on how good that guess is; for instance if it satisfy the constraints), and it is recommended we can compute gradient/Hessian of $E$ as well as the Jacobians of $\boldsymbol{g}$ and $\boldsymbol{h}$.

### 15.1.2 Linear-quadratic optimization

Quadratic programming

**Quadratic programming** is a particular case of the above problem where the cost is quadratic, the constraints are linear, and $Q$ is symmetric $Q = Q^\top$

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\mathbf{x}^{\mathrm{T}} Q\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ \text{and} & F\mathbf{x} = \mathbf{g} \end{array} \tag{15.2}$$

In the case where $Q$ is positive definite and the problem is not very large (which is the case relevant to us) it can be assumed the problem can be solved using simply library calls.

## 15.2 Optimizing the discrete problem

Consider the case of the linear-quadratic regulator with dynamics

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d}_k \tag{15.3}$$

and a quadratic cost function

$$J_{\boldsymbol{u}_0,\ldots,\boldsymbol{u}_{N-1}}(\boldsymbol{x}_0) = \boldsymbol{x}_N^\top Q_N \boldsymbol{x}_N + \sum_{k=0}^{N-1} (\boldsymbol{x}_k^\top Q_k \boldsymbol{x}_k + \boldsymbol{u}_k^\top R_k \boldsymbol{u}_k). \tag{15.4}$$

We will also assume the system is subject to linear constraints such as $F'\boldsymbol{x} = \boldsymbol{h}'$ and $F''\boldsymbol{x} = \boldsymbol{h}''$. Since the cost function is at most quadratic it is little surprise this problem can be put into the form of a quadratic program by collecting all $N$ control vectors $\boldsymbol{u}_k$ into one large vector and consider the problem as a quadratic programming problem in this vector.

However, we can do better. A problem with this approach is that small initial changes in controls near $k = 0$ may result in large changes at subsequent times, and so the problem is not very stable. A better approach is to also consider the $N$ state-vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$ as part of the optimization problem. The new problem therefore becomes a $N(d+n)$-dimensional optimization problem over $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{u}_0, \ldots, \boldsymbol{x}_N, \boldsymbol{u}_{N-1})$ defined as:

$$\text{minimize:} \quad \boldsymbol{x}_N^\top Q_N \boldsymbol{x}_N + \sum_{k=0}^{N-1} (\boldsymbol{x}_k^\top Q_k \boldsymbol{x}_k + \boldsymbol{u}_k^\top R_k \boldsymbol{u}_k) \tag{15.5a}$$

$$\text{simple constraints:} \quad F'\boldsymbol{x} \leq \boldsymbol{h}', \quad F''\boldsymbol{x} \leq \boldsymbol{h}'', \tag{15.5b}$$

$$\text{and dynamic constraints:} \quad A_k \boldsymbol{x}_k + B_k \boldsymbol{x}_k + \boldsymbol{d}_k - \boldsymbol{x}_{k+1} = \mathbf{0} \tag{15.5c}$$

In other words, we adopt a strategy of expanding the conditionality of the optimization problem and adding new constraints to account for the dynamics in eq. (15.2), but at the same time the overall problem becomes more numerically stable.

### 15.2.1 Transcription Methods

SHOOTING
COLLOCATION

The choice we made above, between optimizing just the action values (known as **shooting**) or optimizing both state and action values (as we in fact did, known as **collocation**) is a fundamental choice in any numerical algorithm for optimizing a discretized problem. The later method, collocation, produces more numerically stable algorithms and is therefore preferred in a variety of contexts. It also easier to implement, since modern optimization software for quadratic programs allows us to specify cost-functions and constraints on a per-variable basis as in eq. (15.5), rather than requiring us to figure out what the matrices $Q$ and $\boldsymbol{q}$ are in eq. (15.2), it is easy to implement.

### 15.2.2 Comments about optimizing the discrete problem

The disadvantage of optimizing the discrete control problem as in eq. (15.5) is that if $N$ is large, the optimization may be numerically costly (to the point of becoming unfeasible), and furthermore, that although we solve the *discrete* problem exactly the *actual* environment will diverge from this solution due to the discretization error. That is, our closed-loop controller cannot be trusted after a certain number of time steps. While this might discourage us from using the method directly, both limitations can be addressed to produce powerful control methods as we will see in algorithm 29 and when we discuss the LMPC controller in section 18.3.

## 15.3 Direct collocation

In this section, we will consider an important class of solution methods which assume

- We assume we have a very good approximation to the underlying dynamical model $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$ and
- We are concerned with planning an optimal trajectory.

DIRECT COLLOCATION METHODS

Examples of such situations could be to bring a satellite into orbit or landing a space shuttle. Within this class of problems, **direct collocation methods** (or simply direct methods) is perhaps the best method for trajectory optimization, and it has the benefit of easily incorporating non-linear constraints and non-typical cost functions.

The easiest way to think about direct methods is as an extension of the simple discrete optimization approach in the previous chapter, but applied directly to the dynamical model $\boldsymbol{f}$ rather than a discrete approximation.

### 15.3.1 Problem formulation

The problem we consider is the general optimal control problem discussed in chapter 10. Recall that $t_0$ and $t_F$ refer to the start and end-time of the control and the functions $\boldsymbol{x}(t)$

and $\boldsymbol{u}(t)$ are defined for $t \in [t_0, t_F]$. Given this, we consider the familiar cost-function:

$$c_f\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right) + \int_{t_0}^{t_F} c(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau)) d\tau \tag{15.6}$$

Note this cost function depends on $t_0$, $t_F$ and $\boldsymbol{u}$, but also $\boldsymbol{x}$ since we are using a collocation method. The cost function is subject to a number of constraints. Most importantly that $\boldsymbol{x}$ and $\boldsymbol{u}$ must obey the system dynamics

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \quad t \in [t_0, t_F] \tag{15.7}$$

as well as whatever constraints the system may be subject to. These can either relate to what the system does on the path, i.e. for $t \in ]t_0, f_F[$,

$$\begin{array}{ll} \boldsymbol{x}_{\text{low}} \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}}\,, & \text{path bound on state,} \\ \boldsymbol{u}_{\text{low}} \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upp}}\,, & \text{path bound on control.} \\ \boldsymbol{h}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \leq \boldsymbol{0}, & \text{path constraint.} \end{array} \tag{15.8}$$

Or the constraints can be applied to the start and end-times of the system:

$$\begin{array}{ll} t^0_{\text{low}} \leq t_0 \leq t^0_{\text{upp}}\,, & \text{bounds on initial time,} \\ t^F_{\text{low}} \leq t_F \leq t^F_{\text{upp}}\,, & \text{bounds on final time,} \\ \boldsymbol{x}_{0,\,\text{low}} \leq \boldsymbol{x}\left(t_0\right) \leq \boldsymbol{x}_{0,\,\text{upp}}\,, & \text{bound on initial state,} \\ \boldsymbol{x}_{F,\,\text{low}} \leq \boldsymbol{x}\left(t_F\right) \leq \boldsymbol{x}_{F,\,\text{upp}}\,, & \text{bound on final state.} \end{array} \tag{15.9}$$

Conceptually, the goal is simple: Define the free variables as $Z = (t_0, t_F, \boldsymbol{x}, \boldsymbol{u})$, then maximize the non-linear function eq. (15.6) subject to the constraints eq. (15.7), eq. (15.8) and eq. (15.9). We want to do this using a non-linear program in the form described in eq. (15.1), and so our main challenge is to discretize $\boldsymbol{u}$ and $\boldsymbol{x}$ which are functions.

### 15.3.2 Collocation

All discretization methods take the same starting point. We assume the time coordinate $t$, which goes from $t_0$ to $t_F$, has been discretized. The discretization is assumed to contain $N$ time points as follows[1]:

$$t_0 < t_1 < t_2 < \cdots < t_k < \cdots < t_{N-1} \tag{15.10}$$

In the simplest case, which we will follow here, this is done by *defining* each time point as:

$$t_k = \frac{k}{N-1}(t_F - t_0) + t_0, \quad k = 0, \ldots, N-1 \tag{15.11}$$

The main observation is that the time points are functions of $t_0$ and $t_F$. Given this we also define $h_k = t_{k+1} - t_k$ for $k = 0, \ldots, N-1$.

[1]There was a misprint in an earlier version which used $N$ instead of $N-1$. The current version is changed to be compatible with the implementation which has not been changed.

Next, we represent the functions $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ by their values in $t_0, \ldots, t_{N-1}$. I.e. we introduce the $N$ decision-variables $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{N-1} \in \mathbb{R}^n$ and the $N$ variables $\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1} \in \mathbb{R}^d$. These $2N$ variables have to be optimized over and will be part of $\boldsymbol{z}$ in the non-linear program (see eq. (15.1))[2].

What remains is two things: We have to represent the cost function eq. (15.6) in terms of $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$, and we have to translate the system dynamics constraint eq. (15.7) as well as the other constraints eq. (15.8) and eq. (15.9) into constraints on the variables $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$.

### Trapezoid collocation

TRAPEZOID COLLOCATION

The simplest way to accomplish this is using **trapezoid collocation**. Trapezoid collocation is the geometrically obvious approximation of an integral

$$\int_{x_k}^{x_{k+1}} f(x)dx \approx \frac{1}{2}(x_{k+1} - x_k)(f(x_{k+1}) + f(x_k)) \tag{15.12}$$

which implies:

$$\int_{x_0}^{x_N} f(x)dx \approx \frac{1}{2}\sum_{k=0}^{N-1}(x_{k+1} - x_k)(f(x_{k+1}) + f(x_k)) \tag{15.13}$$

to all continuous aspects of the optimization problem. For instance, the cost function eq. (15.6) becomes a sum:

$$c_f\left(t_0, t_F, \boldsymbol{x}\left(t_0\right), \boldsymbol{x}\left(t_F\right)\right) + \int_{t_0}^{t_F} c(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau))d\tau \tag{15.14}$$

$$\approx c_f\left(t_0, t_F, \boldsymbol{x}_0, \boldsymbol{x}_N\right) + \sum_{k=0}^{N-1} \frac{h_k}{2}\left(c_{k+1} + c_k\right) \tag{15.15}$$

$$c_k = c(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k) \tag{15.16}$$

The system dynamics constraints $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ from eq. (15.7) can be discretized by integrating both sides from $t_k$ to $t_{k+1}$ and using the trapezoid quadrature rule:

$$\int_{t_k}^{t_{k+1}} \dot{\boldsymbol{x}}(t)dt = \int_{t_k}^{t_{k+1}} \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)dt \tag{15.17}$$

$$\approx \frac{h_k}{2}(\boldsymbol{f}(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{k+1}, t_{k+1}) + \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k)) \tag{15.18}$$

The left-hand side is the integral of a derivative which can be solved

$$\int_{t_k}^{t_{k+1}} \dot{\boldsymbol{x}}(t)dt = \boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \tag{15.19}$$

Combining these, and using $\boldsymbol{f}_k = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{t}_k)$ as a shorthand, we have derived the $N-1$ **collocation constraints**:

COLLOCATION CONSTRAINTS

[2]Note two small changes in the notation: (i) there are as many $\boldsymbol{x}_k$-decision variables as there are $\boldsymbol{u}_k$-decision variables which is different than the formulation in the discretization section and (ii) the role of $N$ has accordingly been changed slightly so that the last decision variable is still $\boldsymbol{u}_{N-1}$.

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \frac{h_k}{2}(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k). \tag{15.20}$$

**Other constraints**

The remaining constraints are easily dealt with simply by translating them to apply at the discrete time-points:

$$\begin{aligned}
\boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}} \quad &\rightarrow \quad \boldsymbol{x}_k \leq \boldsymbol{x}_{\text{upp}} && \text{(15.21a)}\\
\boldsymbol{h}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \leq \boldsymbol{0} \quad &\rightarrow \quad \boldsymbol{h}(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k) \leq \boldsymbol{0} && \text{(15.21b)}\\
\boldsymbol{x}\left(t_F\right) \leq \boldsymbol{x}_{F,\text{upp}} \quad &\rightarrow \quad \boldsymbol{x}_N \leq \boldsymbol{x}_{F,\text{upp}}\,. && \text{(15.21c)}
\end{aligned}$$

### 15.3.3 Constructing the solution

In summary, the problem of optimizing the trajectory $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ has been re-cast as minimizing the cost function eq. (15.15) with respect to all free variables $(\boldsymbol{x}_k)_k$, $(\boldsymbol{u}_k)_k$ and $t_0, t_F$. Once this minimization has been accomplished we are left with the optimal value of the knot-points $\boldsymbol{x}_k^*, \boldsymbol{u}^*$, and from these we have to re-construct the corresponding optimal trajectory $\boldsymbol{x}^*(t)$ and $\boldsymbol{u}^*(t)$. The control trajectory is recovered using simple linear interpolation. To keep the mathematics simple suppose $t \in [t_k, t_{k+1}]$ and define $\tau = t - t_k$ then:

$$\boldsymbol{u}(t) \approx \boldsymbol{u}_k + \frac{\tau}{h_k}\left(\boldsymbol{u}_{k+1} - \boldsymbol{u}_k\right). \tag{15.22}$$

To recover $\boldsymbol{x}(t)$ from $\boldsymbol{x}_k$, recall the only approximation affecting the systems trajectory has been the collocation constraint eq. (15.18). If we define $\boldsymbol{f}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ as a shorthand, this approximation is exact if $\boldsymbol{f}$ vary linearly between $t_k$ and $t_{k+1}$, and so we have implicitly made this assumption. We can therefore assume $\boldsymbol{f}(t)$ can be linearly interpolated for $t \in [t_k, t_{k+1}]$:

$$\boldsymbol{f}(t) \approx \boldsymbol{f}_k + \frac{\tau}{h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right) \tag{15.23}$$

To find $\boldsymbol{x}(t)$ for $t \in [t_k, t_{k+1}]$ we can simply integrate the above to get the quadratic approximation while using that $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t)$:

$$\begin{aligned}
\boldsymbol{x}(t) &= \int_{t_k}^{t} \dot{\boldsymbol{x}}(t) d\tau + \boldsymbol{x}(t_k) && \text{(15.24)}\\
&= \int_{t_k}^{t} \left(\boldsymbol{f}_k + \frac{t - t_k}{h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right)\right) dt + \boldsymbol{x}_k && \text{(15.25)}\\
&= \boldsymbol{x}_k + \tau \boldsymbol{f}_k + \frac{\tau^2}{2h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right). && \text{(15.26)}
\end{aligned}$$

☞ **Algorithm 20** Direct solver

**function** DIRECT-SOLVE($N$, GUESS=$(t_0^g, t_F^g, \boldsymbol{x}^g, \boldsymbol{u}^g)$ )
  Define $z \leftarrow (\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{x}_{N-1}, \boldsymbol{u}_{N-1}, t_0, t_F)$ as all optimization variables
  Define grid time points $t_k = \frac{k}{N-1}(t_F - t_0) + t_0, \quad k = 0, \ldots, N-1$ ▷ eq. (15.11)
  Define $h_k$, $\boldsymbol{f}_k = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k)$ and $c_k = c(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k)$.
  Define $I_{\text{eq}}$ and $I_{\text{ineq}}$ as empty lists of inequality/equality constraints
  **for** $k = 0, \ldots, N-2$ **do**
    Append constraint $\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \frac{h_k}{2}(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k)$ to $I_{\text{eq}}$ ▷ eq. (15.20)
    Add all other path-constraints eq. (15.21) to $I_{\text{ineq}}$ and $I_{\text{eq}}$
  **end for**
  Add possible end-point constraints on $\boldsymbol{x}_0, \boldsymbol{x}_F$ and $t_0$, $t_F$ to $I_{\text{eq}}$ and $I_{\text{ineq}}$
  Build optimization target $E(\boldsymbol{z}) = c_f(t_0, t_F, \boldsymbol{x}_0, \boldsymbol{x}_{N-1}) + \sum_{k=0}^{N-2} \frac{h_k}{2}(c_{k+1} + c_k)$
  Construct guess time-grid: $t_k^g \leftarrow \frac{k}{N-1}(t_F^g - t_0^g) + t_0^g$
  Construct guess states $\boldsymbol{z}^g \leftarrow (\boldsymbol{x}^g(t_0^g), \boldsymbol{u}^g(t_0^g), \cdots, \boldsymbol{x}^g(t_{N-1}^g), \boldsymbol{u}^g(t_{N-1}^g), t_0^g, t_F^g)$
  Let $\boldsymbol{z}^*$ be minimum of $E$ optimized over $\boldsymbol{z}$ subject to $I_i$ and $I_{eq}$ using guess $\boldsymbol{z}^g$
  Re-construct $\boldsymbol{u}^*(t), \boldsymbol{x}^*(t)$ from $\boldsymbol{z}^*$ using eq. (15.22) and eq. (15.26)
  Return $\boldsymbol{u}^*, \boldsymbol{x}^*$ and $t_0^*$, $t_F^*$
**end function**

### A combined method

With this information we can state the direct solution method. There are three main steps:

- Define all free variables/quantities $t_0$, $t_F$ and $\boldsymbol{x}_k$, $\boldsymbol{u}_k$.
- Solve the optimization of eq. (15.15) subject to all constraints using a non-linear program solver eq. (15.1)
- Re-construct the solution $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$

As mentioned in the beginning, a non-linear solver is sensitive to the initial guess. We will therefore assume we have access to guesses $t_0^g$ and $t_F^g$ of $t_0$ and $t_F$ as well guesses for the controls and trajectories $\boldsymbol{x}^g$ and $\boldsymbol{u}^g$. We will return to this problem in section 15.3.4. But assuming a guess is found, the method is sketched in algorithm 20

## 15.3.4 Guesses and the iterative method

Whether the simple collocation method algorithm 20 works in practice depends critically on the goodness of the initial guess of the trajectory $\boldsymbol{x}^g$ and $\boldsymbol{u}^g$. The most primitive way of obtaining such a guess is by guessing the values of the end-points $t_0^g$, $t_F^g$, guess a value of $\boldsymbol{x}$ and $\boldsymbol{u}$ at the end-points, and then linearly interpolate:

$$\boldsymbol{x}^g(t) = \boldsymbol{x}^g(t_0^g) + (t - t_0^g)(\boldsymbol{x}^g(t_F^g) - \boldsymbol{x}^g(t_0^g)). \tag{15.27}$$

☞ **Algorithm 21** Iterative direct solver

**Require:** An initial guess $\boldsymbol{z}_0^g = (\boldsymbol{x}^g, \boldsymbol{u}^g, t_0^g, t_F^g)$ found using simple linear interpolation
**Require:** A sequence of grid sizes $10 \approx N_0 < N_1 < \cdots < N_T$
1: **for** $t = 0, T$ **do**
2: $\quad \boldsymbol{x}^*, \boldsymbol{u}^*, t_0^*, t_F^* \leftarrow$ DIRECT-SOLVE$(N_t, \boldsymbol{z}_t^g)$
3: $\quad \boldsymbol{z}_{t+1} \leftarrow \boldsymbol{x}^*, \boldsymbol{u}^*, t_0^*, t_F^*$
4: **end for**
5: Return $\boldsymbol{u}^*, \boldsymbol{x}^*$ and $t_0^*$, $t_F^*$

| Parameter | Value |
|---|---|
| Gravitational constant | 9.82 |
| Pole mass | 0.80 |
| Pole length | 1.00 |
| Max torque | 6.00 |

Table 15.1: Pendulum model parameters

A full discussion can be found in [Kel17a]. Whether one uses linear interpolation or a more sophisticated method, a much better initialization strategy is obtained by first running the method using a very small value of $N$ and a poor guess (our implementation we use linear interpolation) to obtain $\boldsymbol{x}^*, \boldsymbol{u}^*$ and $t_0^*, t_F^*$, and then use *these* values as a *new* guess for the method using a higher value of $N$. The reason why this work is that when $N$ is low, the optimizer is more likely to overcome a poor initial guess than when $N$ is higher and the optimization problem is correspondingly harder. Pseudo-code for this approach can be found in algorithm 21.

Direct methods using trapezoid collocation provides a powerful methodology for trajectory optimization and according [Bet10] compares favorable to the (theoretically more sophisticated and not covered by this course) class of indirect methods. Since it uses the system dynamics $\dot{\boldsymbol{x}} = \boldsymbol{f}$ directly, the only approximations are in the approximation of the cost-function eq. (15.15), which is less important, and in the trapezoid collocation constraint eq. (15.18).

### 15.3.5 Example: Pendulum swingup

Consider once again the Pendulum problem in the $\theta$, $\dot{\theta}$ parameterization. The goal is to bring the pendulum upright and still exactly at time $t_F$. The parameters can be found in table 15.1

$$J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = \frac{1}{2}\int_0^{t_F} \|u(t)\|^2 dt. \tag{15.28}$$

We applied the method using grid refinement of $N = 10$ and then $N = 60$, and it is instructive to look at the intermediate results. If we first consider the grid-refinement

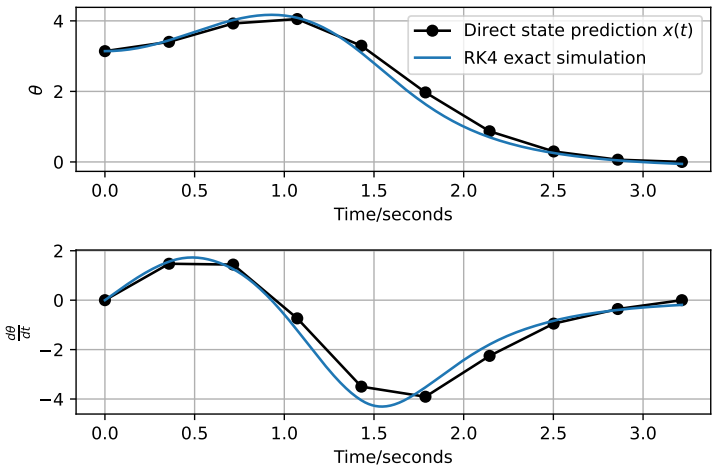


Figure 15.1: Pendulum solver using iteratively refined Grid. Here shown using initial $N = 10$ grid

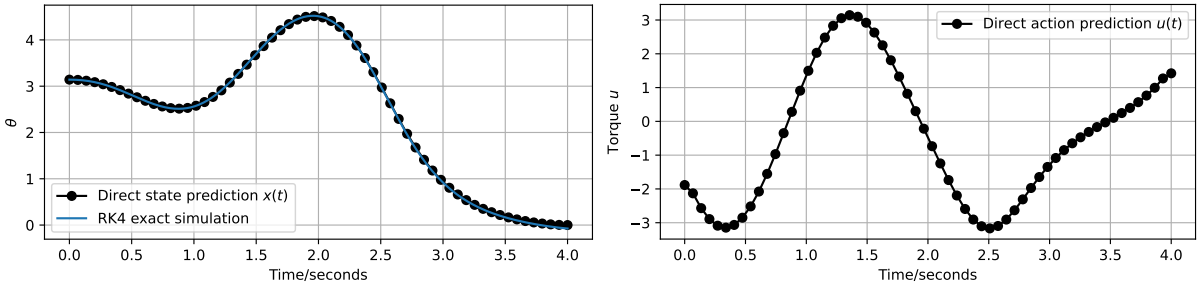


Figure 15.2: Pendulum solver using a refined grid of $N = 60$. Right-most figure also shows the action $u$

(see fig. 15.1 ) we notice the predictions of the direct method (black) agree with the boundary conditions, but to not agree with the result of simulating the resulting policy using RK4 and the exact dynamics; despite this the policy nevertheless manage to bring the actual system to an upright state. The experiment is repeated using the $N = 10$ solution as initialization and using $N = 60$ gridpoints and we obtain almost perfect agreement between simulation and prediction (see fig. 15.2) The right-pane of the figure also shows the action vector. As we see, there is a near-perfect agreement between prediction and outcome (except for the very end of the simulation) and the actions are in this case so small that the constraint is not active. To examine the effect of the constraint $t_F$ was reduced to 2 seconds and the experiment was repeated. This gives the state/action trajectory shown in fig. 15.3

### 15.3.6 Example: Cartpole swingup

In the cartpole problem, a pendulum is affixed to a cart on a track and the goal is to swing the pendulum upright. The coordinates of the system is the cart location $x$ and velocity $\cdot x$ as well as the angle the pendulum form with the up-right direction $\theta$ and

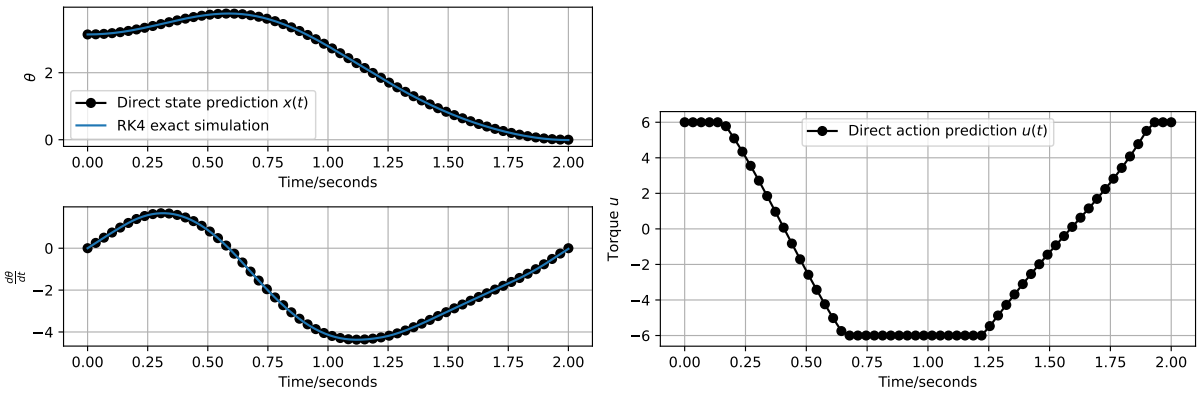


Figure 15.3: Pendulum solver using a refined grid of $N = 60$ and a final time of $t_F = 2$. The action constraint is now active because more torque has to be applied.

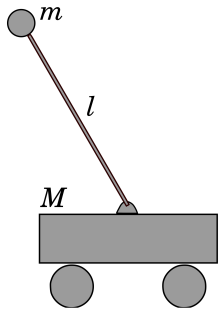


Figure 15.4: Illustration of the cartpole system

| Parameter | Value |
|---|---|
| Gravitational constant | 9.81 |
| Cart mass $m_c$ | 1.000000 |
| Pole mass $m_c$ | 0.300000 |
| Pole length | 0.500000 |
| Maximum force | -20.000000 |

Table 15.2: Cartpole model parameters

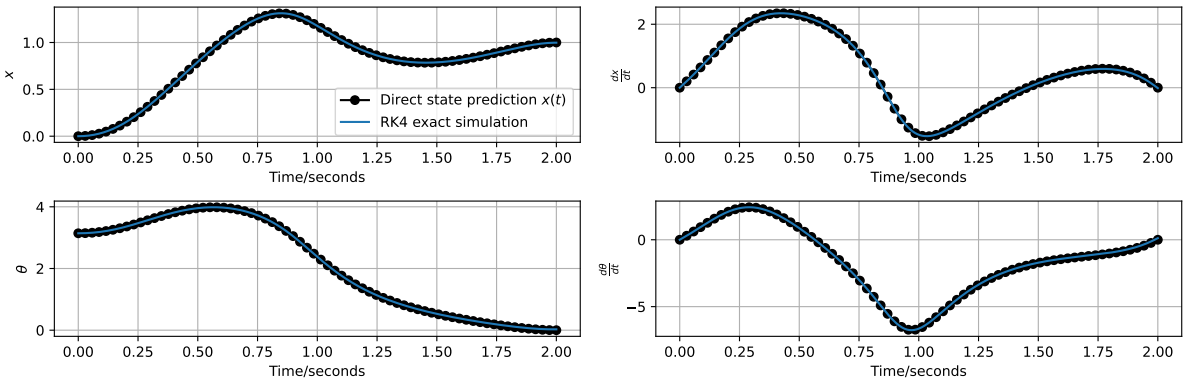


Figure 15.5: Clockwise: Position, velocity, angle, angular velocity of cart in the cartpole swingup task described in [Kel17b].

angular velocity, see fig. 15.4. The available control is the force (applied to the cart) in the forward/backward direction.

We will consider the same cartpole problem described in [Kel17b][3] (see table 15.2), where the goal is to swing the pole from a stationary downwards position $\boldsymbol{x} = \begin{bmatrix} 0 & 0 & \pi & 0 \end{bmatrix}^\top$ to the upwards position at a distance of $x = d$ from the start position, i.e. to $\boldsymbol{x} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^\top$ at exactly $t_F = 2$ seconds. These requirements are implemented as constraints, and we add a square penalty for the applied force $u$ as for the pendulum. I used a grid refinement scheme of $N = 10, 20, 70$.

**Example: Cartpole swingup minimum time**

Finally, we will consider a minimum-time formulation of the cartpole swingup problem, where the cost function is just $J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = {t_F}^4$. the swingup task is much harder to optimize since the minimum-time formulation means the constraints are more active, however the grid-refinement scheme $N = 8, 16, 32, 70$ produce reasonable results. The task completes in just under 1.3 seconds and the state trajectory is visually quite interesting and adapts a different strategy (see fig. 15.6, bottom).

### 15.3.7 Example: Brachistochrone ✸

The history of the brachistochrone problem goes back to the later 17th century when it was first posed by the mathematician Johann Bernoulli. The problem is easy to describe: Suppose we want to build a track, starting in the position $(x, y) = (0, 0)$, and

[3]Notice [Kel17b] use a different parameterization of the system such that down is $\theta = 0$. The difference is due to there being two versions of the cartpole dynamics, one is wrong and use $\theta = 0$ and the other is correct and use $\theta = \pi$. We use the correct dynamics.

[4]The example is adapted from an online source `https://github.com/MatthewPeterKelly/OptimTraj/blob/master/demo/cartPole/MAIN_minTime.m` which can be consulted for the exact parameters.

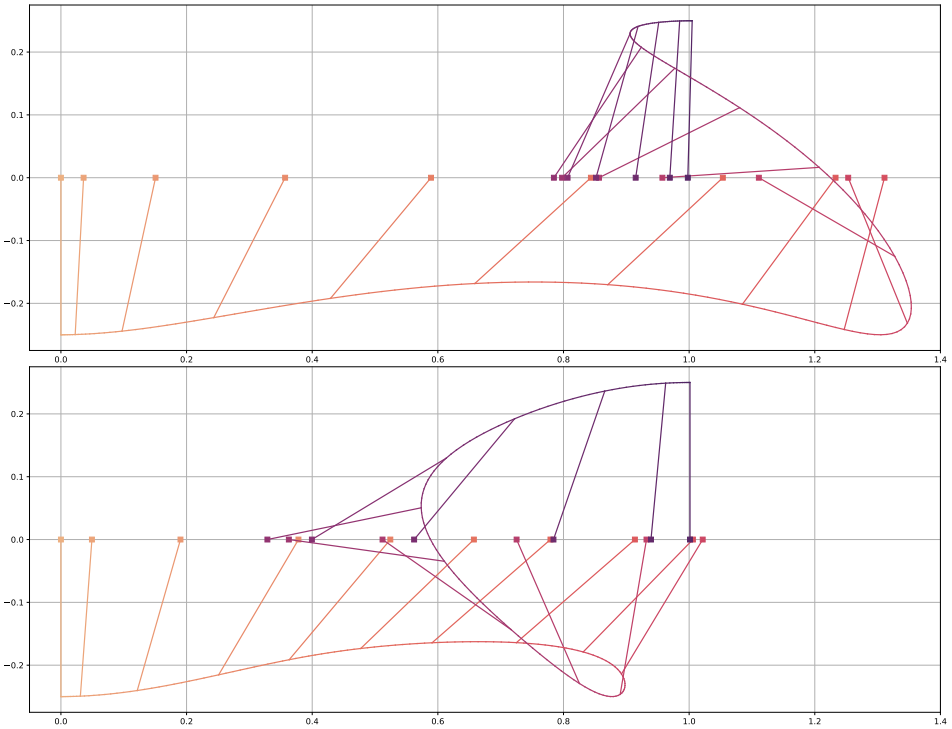

Figure 15.6: Top: Illustration of the pendulum trajectory in the cartpole example from [Kel17b]. Bottom: Cartpole trajectory when using a minimum-time cost function.

ending when $x = x_B$. Assuming that the bead slide friction-less along the track, only influenced by gravity, what shape should the track be to minimize the travel time?

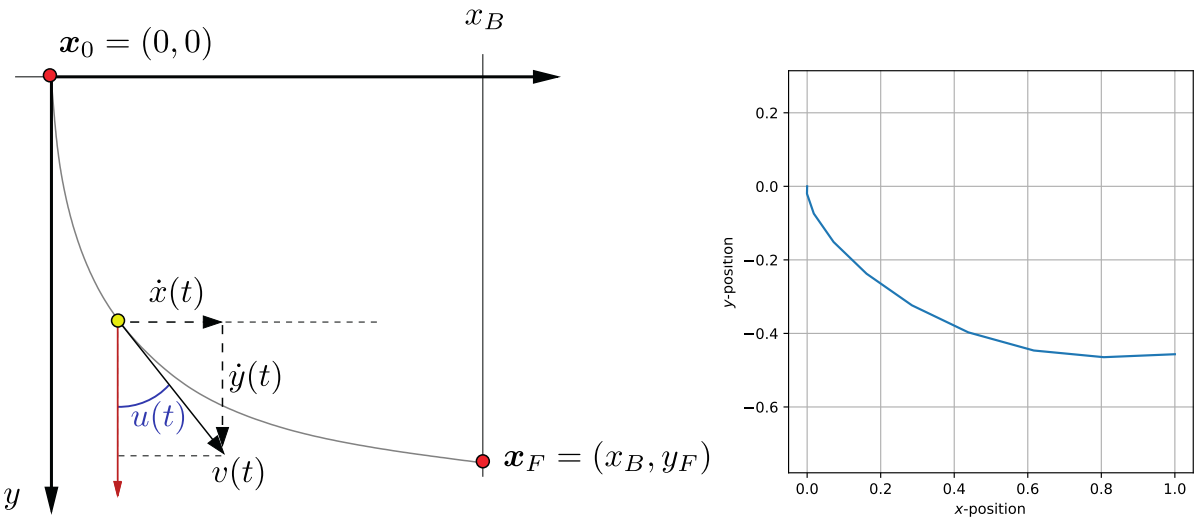


Figure 15.7: Right: Sketch of brachistochrone curve and coordinate system. Right: Solution found using direct methods.

The problem can be solved using variational calculus, however we will treat it as a

control problem. The dynamics can be described using the speed of the bead $v$, the current angle of the bead with vertical $u(t)$, and the $x, y$ position of the bead $x(t)$ and $y(t)$. The state is therefore:

$$\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ y(t) \\ v(t) \end{bmatrix}. \tag{15.29}$$

We can consider $u(t)$ as the control, since specifying $u$ is equivalent to specifying the track. Therefore, the problem of finding the optimal track is equivalent to solving a control problem with the familiar cost $J_{\boldsymbol{u}}(\boldsymbol{x}_0, t_F) = t_F$, initial state $\boldsymbol{x}_0 = \mathbf{0}$ and final state constraint affecting the single coordinate $x(t_F) = x_B$. The dynamics can be found using high-school physics as:

$$\dot{x} = v \sin u, \quad \dot{y} = -v \cos u, \quad \dot{v} = g \cos u. \tag{15.30}$$

Where $g$ is the gravitational constant. The optimal solution can be found in fig. 15.7 (right); I use $N = 10, 30$ but the problem is comparably easier to solve.

### Brachistochrone with a dynamical constraint ✸✸

We will now consider a more challenging variant of the Brachistochrone problem where an angled slab with equation $y(t) = -\frac{1}{2}x(t) - h$ has been inserted, blocking the path of the bead (see fig. 15.8). We can still solve the problem, all we have to do is to add the dynamical path-constraint of the form $\boldsymbol{h}(\boldsymbol{x}) \leq 0$ and then translate that constraint to be active in each knot-point, i.e. add the $N$ constraints: $\boldsymbol{h}(\boldsymbol{x}_k) \leq 0$, $k = 0, \ldots, N$ to the problem. The specific form of the constraint is obviously that

$$y(t) \geq -\frac{1}{2}x_1(t) - h. \tag{15.31}$$

## 15.4 Additional issues ✸✸

The direct optimization method can be modified and expanded upon in a number of ways.

**Hermite Simpson quadrature** Throughout we have used the trapezoid quadrature rule to approximate integrals. A higher-order method can be obtained by using Hermite-Simpson quadrature which is the rule:

$$\int_{t_0}^{t_F} w(\tau) d\tau \quad \approx \sum_{k=0}^{N-1} \frac{h_k}{6} \left( w_k + 4w_{k+\frac{1}{2}} + w_{k+1} \right) \tag{15.32}$$

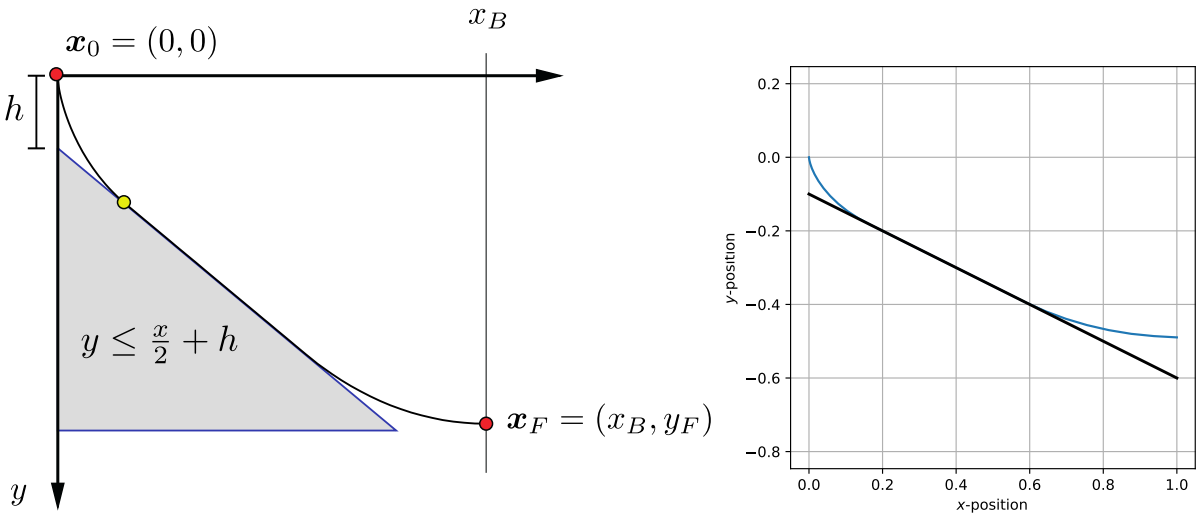


Figure 15.8: Sketch of restricted Brachistochrone problem and solution found with Direct methods.

The notation $\boldsymbol{w}_{k+\frac{1}{2}}$ means $\boldsymbol{w}$ has been evaluated at the mid-point of $t_k$ and $t_{k+1}$. Applied similar to the trapezoid rule we obtain the new collocation constraint:

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \frac{1}{6} h_k \left( \boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1} \right) \tag{15.33}$$

Again the notation $\boldsymbol{f}_{k+\frac{1}{2}}$ means $\boldsymbol{f}$ evaluated at the mid-point of $\boldsymbol{t}_k$ and $\boldsymbol{t}_{k+1}$. This can in turn be approximated as:

$$\boldsymbol{x}_{k+\frac{1}{2}} = \frac{1}{2} \left( \boldsymbol{x}_k + \boldsymbol{x}_{k+1} \right) + \frac{h_k}{8} \left( \boldsymbol{f}_k - \boldsymbol{f}_{k+1} \right) \tag{15.34}$$

Which can then form the basis of a computational procedure by simply replacing the collocation constraint in line 7 of algorithm 20. Note other tweaks are required when re-covering the paths $\boldsymbol{u}(t)$ and $\boldsymbol{x}(t)$ after optimization. For more details see [Kel17a]. Note the Hermite-Simpson collocation procedure trades accuracy for a more complex optimization task and so trapezoid collocation is still superior in some cases, particular for large problems (see [Bet10]).

**Error analysis** A second simple tweak is to compute the approximation error in the trapezoid constraint and use this to determine if $N$ should be increased. Methods for doing so are quite simple. For instance, suppose $\boldsymbol{x}$ and $\boldsymbol{u}$ is the solution path found by the direct method, then we can define the error as:

$$\varepsilon(t) = \dot{\boldsymbol{x}}(t) - \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \tag{15.35}$$

which will be zero if the solution path in fact satisfy the system dynamics (can you see why?). We can therefore define a local interpolation error as: $\eta_k = \int_{t_k}^{t_{k+1}} |\varepsilon(\tau)| d\tau$. See [Kel17a] and [Bet10] for a further discussion.

**Intelligent mesh refinement** Whether the found trajectory will in fact be representative of what the system does depends on $N$, however a too large $N$ makes the optimization intractable. An important extension to the method is obtained by computing the local error $\eta_k$ for each interval $[t_k, t_{k+1}]$, and then rather than increasing $N$ globally in algorithm 21, we simply split those intervals in two where the error exceed some threshold (for instance, we split a certain number of time intervals and choose those where the error is the largest). This is a fairly simple-to-implements but important technique which is further discussed in [Kel17a] and [Bet10].

## 15.5 Bibliographic Notes

A broad and accessible introduction to direct methods for trajectory optimization is presented in [Kel17c]. This tutorial also features a discussion of trajectory optimization for hybrid systems, which we have not discussed in this section, as well as numerical solver features. For a more comprehensive review of direct methods for trajectory optimization by the same author with an emphasis on collocation methods, see [Kel17a], and for the readers who wish to delve into the state-of-the-art of direct solvers with a focus on large problems I would recommend [Bet10].

# Chapter 16

# Linear-quadratic regulator

In this section we will address an important subclass of continuous state and action space problems for which dynamic programming can be applied exactly. In this setting, we assume linear dynamics and quadratic costs, and the problem setting is referred to as the *linear quadratic regulator* (LQR) problem. This LQR setting is important for several reasons. First, as a local stabilizing controller, it is a core tool that is often a first (effective) approach for a wide variety of problems. Second, as we build up open-loop trajectory optimization methods later in the class, the LQR approach will often be used to provide local tracking of these trajectories. Finally, tracking LQR paired with a forward rollout step will form the basis of one of the most powerful nonlinear trajectory optimization methods that we will see in this class[1].

## 16.1 The Linear Quadratic Regulator in Discrete Time

We will fix the dynamics of the system to be discrete time (possibly time-varying) linear,

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k \tag{16.1}$$

and the cost function as quadratic

$$c(\boldsymbol{x}_k, \boldsymbol{u}_k) = \frac{1}{2}(\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \boldsymbol{u}_k^T R_k \boldsymbol{u}_k) \tag{16.2}$$

$$c_N(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T Q_N \boldsymbol{x}_k \tag{16.3}$$

where $Q_k \in \mathbb{R}^{n \times n}$ is positive semi-definite and $R_k \in \mathbb{R}^{m \times m}$ is positive definite for all $k = 0, \ldots, N$. Importantly, we assume $\boldsymbol{x}_k$ and $\boldsymbol{u}_k$ are unconstrained for all $k$. To perform DP recursion, we initialize

$$J_N^*(\boldsymbol{x}_N) = \frac{1}{2}\boldsymbol{x}_N^T Q_N \boldsymbol{x}_N := \frac{1}{2}\boldsymbol{x}_N^T V_N \boldsymbol{x}_N. \tag{16.4}$$

[1]The presentation in this chapter is based on `https://github.com/StanfordASL/AA203-Notes`.

Then, applying the DP algorithm eq. (6.7a) directly, we have

$$J^*_{N-1}(\boldsymbol{x}_{N-1}) = \frac{1}{2}\min_{\boldsymbol{u}_{N-1}\in\mathbb{R}^m}\left\{\boldsymbol{x}^T_{N-1}Q_{N-1}\boldsymbol{x}_{N-1} + \boldsymbol{u}^T_{N-1}R_{N-1}\boldsymbol{u}_{N-1} + \boldsymbol{x}^T_N V_N \boldsymbol{x}_N\right\} \tag{16.5}$$

By inserting the dynamics eq. (16.1) this becomes:

$$\begin{aligned} J^*_{N-1}(\boldsymbol{x}_{N-1}) = \frac{1}{2}\min_{\boldsymbol{u}_{N-1}\in\mathbb{R}^m}\Big\{&\boldsymbol{x}^T_{N-1}Q_{N-1}\boldsymbol{x}_{N-1} + \boldsymbol{u}^T_{N-1}R_{N-1}\boldsymbol{u}_{N-1} \\ &+ (A_{N-1}\boldsymbol{x}_{N-1} + B_{N-1}\boldsymbol{u}_{N-1})^T V_N (A_{N-1}\boldsymbol{x}_{N-1} + B_{N-1}\boldsymbol{u}_{N-1})\Big\}. \end{aligned} \tag{16.6}$$

Rearranging, we have

$$\begin{aligned} J^*_{N-1}(\boldsymbol{x}_{N-1}) = \frac{1}{2}\min_{\boldsymbol{u}_{N-1}\in\mathbb{R}^m}\Big\{&\boldsymbol{x}^T_{N-1}(Q_{N-1} + A^T_{N-1}V_N A_{N-1})\boldsymbol{x}_{N-1} \\ &+ \boldsymbol{u}^T_{N-1}(R_{N-1} + B^T_{N-1}V_N B_{N-1})\boldsymbol{u}_{N-1} \\ &+ 2\boldsymbol{u}^T_{N-1}(B^T_{N-1}V_N A_{N-1})\boldsymbol{x}_{N-1}\Big\}. \end{aligned} \tag{16.7}$$

We can find the optimum by setting the derivative equal to zero[2]

$$\frac{\partial J^*_{N-1}}{\partial \boldsymbol{u}_{N-1}}(\boldsymbol{x}_{N-1}) = (R_{N-1} + B^T_{N-1}V_N B_{N-1})\boldsymbol{u}_{N-1} + (B^T_{N-1}V_N A_{N-1})\boldsymbol{x}_{N-1} \tag{16.8}$$

and setting this to zero yields

$$\boldsymbol{u}^*_{N-1} = \underbrace{-(R_{N-1} + B^T_{N-1}V_N B_{N-1})^{-1}(B^T_{N-1}V_N A_{N-1})}_{=L_{N-1}}\boldsymbol{x}_{N-1} \tag{16.9}$$

which we write

$$\boldsymbol{u}^*_{N-1} = L_{N-1}\boldsymbol{x}_{N-1} \tag{16.10}$$

which is a time-varying linear feedback policy. Plugging this policy into (16.6),

$$\begin{aligned} J^*_{N-1}(\boldsymbol{x}_{N-1}) =&\boldsymbol{x}^T_{N-1}(Q_{N-1} + L^T_{N-1}R_{N-1}L_{N-1} \\ &+ (A_{N-1} + B_{N-1}L_{N-1})^T V_N (A_{N-1} + B_{N-1}L_{N-1}))\boldsymbol{x}_{N-1}. \end{aligned} \tag{16.11}$$

Because the optimal policy is always linear, and the optimal cost-to-go is always quadratic, the DP recursion may be recursively performed backward in time and the minimization may be performed analytically[3]

[2]More formally, $R_{N-1} + B^T_{N-1}V_N B_{N-1} > 0$, and therefore, any local minima is a global minima

[3]Again a technical note: The cost-to-go remains a positive semi-definite quadratic function of the state.

**Basic discrete LQR:** Continuing with the DP recursions to $k = 0$ we obtain the discrete-time LQR controller, which simply consist of:

1. $V_N = Q_N$
2. Loop for $k = N-1, \ldots, 0$:
   (a) $L_k = -(R_k + B_k^T V_{k+1} B_k)^{-1}(B_k^T V_{k+1} A_k)$
   (b) $V_k = Q_k + L_k^T R_k L_k + (A_k + B_k L_k)^T V_{k+1}(A_k + B_k L_k)$
   (c) $\boldsymbol{u}_k^* = L_k \boldsymbol{x}_k$
   (d) $J_k^*(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T V_k \boldsymbol{x}_k$

There are several implications of this recurrence relation. First, even if $A, B, Q, R$ are all constant (not time-varying), the policy is still time-varying. Why is this the case? Control effort invested early in the problem will yield dividends over the remaining length of the horizon, in terms of lower state cost for all future time steps. However, as the remaining length of the episode becomes shorter, the tradeof shift in the favor of the immediate control effort, and so the actions will tend to become smaller.

However, for a time-independent system, if the feedback gain $L_k$ approach a constant as $N \to \infty$. This time-invariant policy is practical for long horizon control problems, and may be approximately computed by running the DP recurrence relation until approximate convergence.

### 16.1.1 Example: Double integrator

Our first problem will concern a well-studied electronic component namely the double integrator. The system evolves as:

$$\boldsymbol{x}_{k+1} = \underbrace{\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}}_{=A} \boldsymbol{x}_k + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{=B} \boldsymbol{u}_k \tag{16.12}$$

and, assuming $\boldsymbol{x}_= \begin{bmatrix} x_{k,1} \\ x_{k,2} \end{bmatrix}$, it has the cost function:

$$J(\boldsymbol{x}_0) = \sum_{k=0}^{N} \frac{1}{2\rho} x_{k,1}^2 + \sum_{k=0}^{N-1} \frac{1}{2} u_k^2 \tag{16.13}$$

This corresponds to the familiar QR cost with $Q_k = Q_N = \begin{bmatrix} \frac{1}{\rho} & 0 \\ 0 & 0 \end{bmatrix}$ and $R = 1$. If we plug this into the LQR algorithm we obtain, at each time step $k$, the policy:

$$\pi_k(\boldsymbol{x}_k) = L_k \boldsymbol{x}_k, \tag{16.14}$$

and we can use these to simulate the optimal trajectory, starting in $\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, computing controls using $\boldsymbol{u}_k = \pi_k(\boldsymbol{x}_k)$, and determining $\boldsymbol{x}_{k+1}$ using eq. (16.12). For different values of $\rho$ and planning for $N = 20$ steps we obtain the trajectory seen in fig. 16.1.

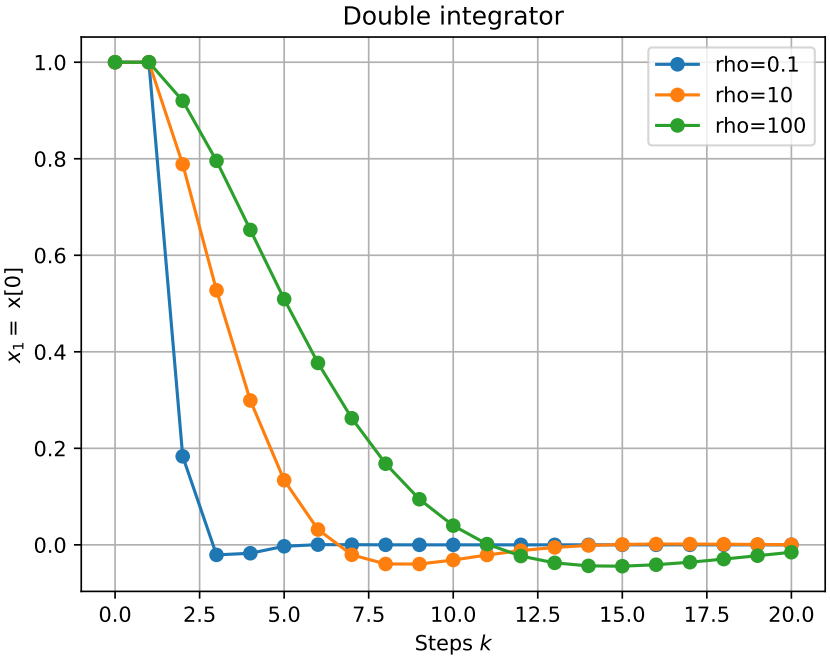


Figure 16.1: Trajectory of $x_{k,1}$ for the double integrator trajectories with different cost-functions determined by $\rho$. The system attempts to control $x_{k,1}$ to 0, and $\rho$ represents a trade-of between the action-term and the cost-term in the cost-function.

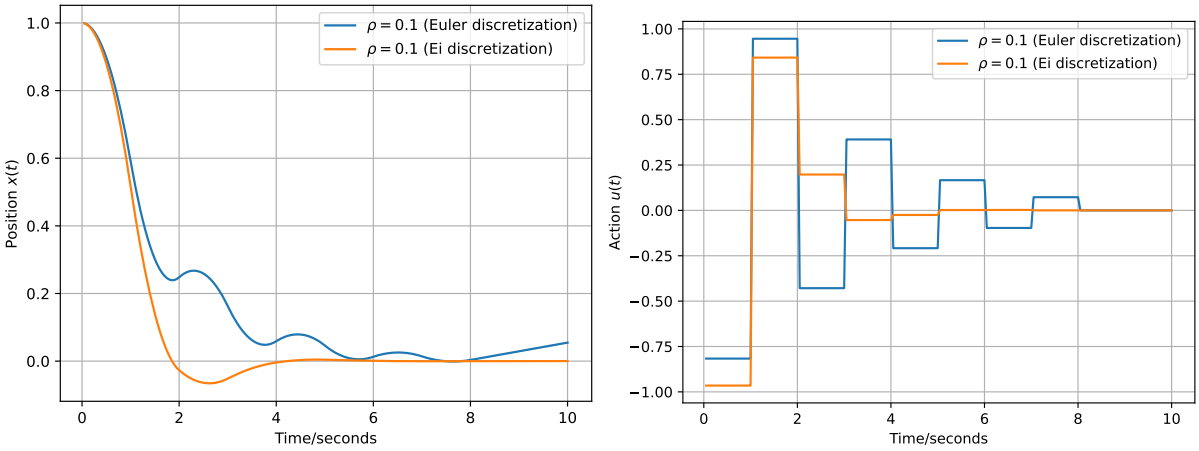


Figure 16.2: Double-integrator example. Result of applying the discrete controller, computed using the Euler-discretized dynamics in section 16.1.1 ($\Delta = 1$), to an environment governed by the actual dynamics eq. (16.16). Euler discretization produces such a bad model the optimal LQR controller is in fact quite bad. In this case the problem can be solved by using exact exponential discretization.

### 16.1.2 Example: Double integrator revisited

The double-integrator problem from section 16.1.1 is perhaps a bit abstract, however it is in fact an example of the harmonic oscillator from section 10.4.2 using $k = 0$ and

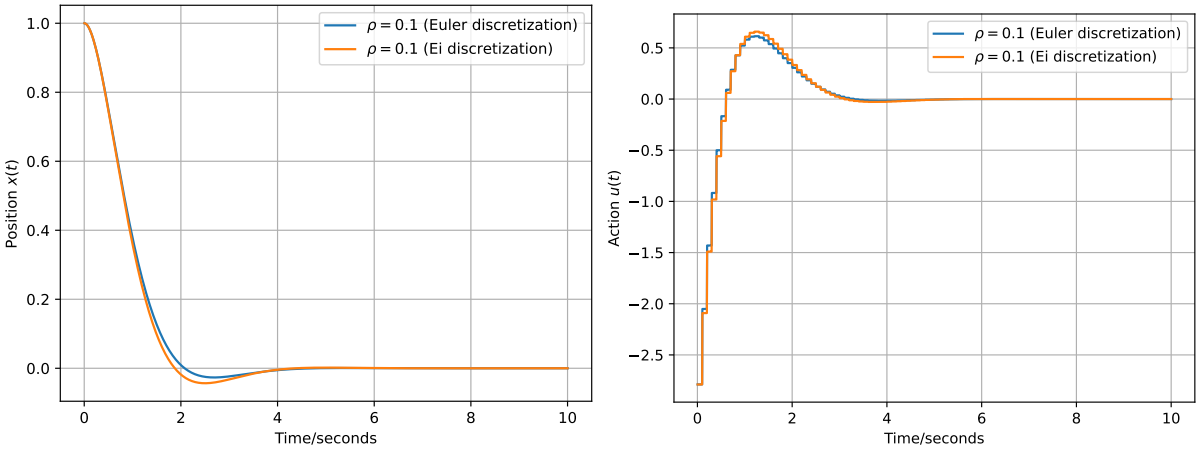


Figure 16.3: Same experiment as in fig. 16.2 but using $\Delta = 0.1$. In this case the Euler discretization is more exact and produces a much better controller.

$m = 1$, in which case the dynamics is described by the $x$-position as:

$$\ddot{x}(t) = u(t) \tag{16.15}$$

or using the $\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix}$ representation:

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \boldsymbol{u}(t) \tag{16.16}$$

and cost function:

$$J(\boldsymbol{x}_0) = \frac{1}{2}\boldsymbol{x}(t_F)^\top Q \boldsymbol{x}(t_F) + \int_0^{t_F} \left( \frac{1}{2}\boldsymbol{x}(t)^\top Q \boldsymbol{x}(t) + \frac{1}{2}\boldsymbol{u}(t)^\top R \boldsymbol{u}(t) \right) \tag{16.17}$$

with $Q = \begin{bmatrix} \frac{1}{\rho} & 0 \\ 0 & 0 \end{bmatrix}$ and $R = 1$. The discrete double-integrator example is recovered using $t_F = 20$, $\Delta = 1$ and Euler discretization. Since Euler discretization is known to be sub-optimal, we should ask what occurs if we apply the controller computed from the Euler discretized model in section 16.1.1 to a system governed by the actual dynamics in eq. (16.16) (**Euler**) versus an equivalent controller which uses exponential integration to obtain the $A_k$ and $B_k$ matrices as described in section 13.1.3.

The resulting state-trajectory $x(t)$ and actions $u(t)$ are shown in fig. 16.2. Clearly, the Euler integration trajectory is bad, and we stress this is simply what would have occurred had we applied the optimal controller calculated in the previous section to the actual system dynamics. As shown in the example, exponential integration, which is exact, completely solves this problem, however it is not an option which will be generally available. In this case the only solution is to reduce $\Delta$. Comparable result using $\Delta = 0.1$ are shown in fig. 16.3, and in this case Euler integration fare much better.

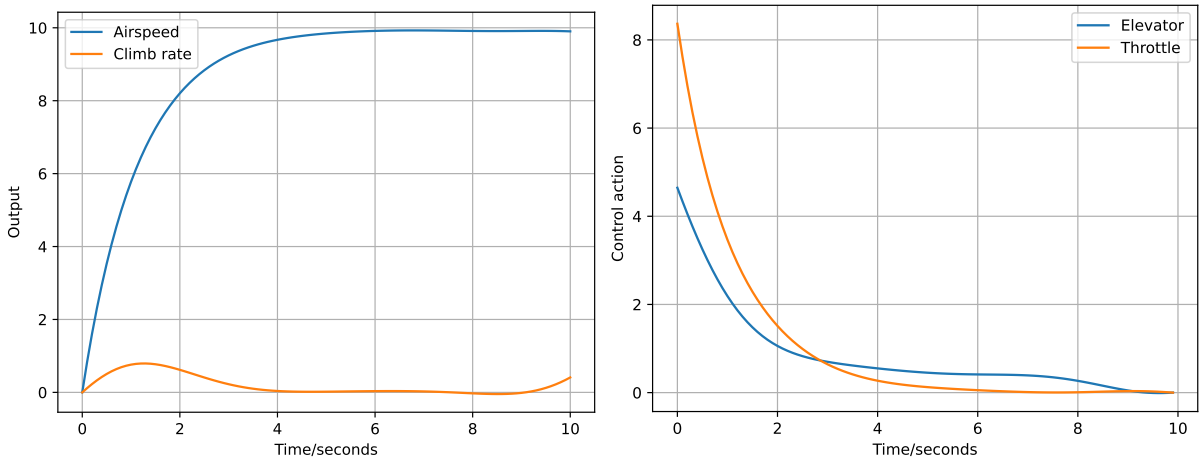


Figure 16.4: Boing level flight when the desired output corresponds to an extra airspeed of 10 ft/sec and a climb rate of 0

### 16.1.3 Example: Boing 747 flight

Recall the Boing 747 autopilot example discussed in section 12.1.1. Suppose we want the autopilot to change from level flight $\boldsymbol{x}(t) = \mathbf{0}$ to flight corresponding to a new airspeed and climb rate corresponding to $\boldsymbol{y}^*$; we assume we plan over a horizon of 10 seconds with a time discretization of $\Delta = 0.1$ seconds.

Recall the Boing problem defines the airspeed $y_1$ and climb rate $y_2$ as:

$$\boldsymbol{y}_k = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & -1. & 0. & 7.74 \end{bmatrix}}_{=P} \boldsymbol{x}_k \tag{16.18}$$

One way to implement the change is therefore to ensure the cost function is at a minimum at $\boldsymbol{y}^*$:

$$J(\boldsymbol{x}_0) = \sum_{k=0}^{N-1} \left( \frac{1}{2} \|\boldsymbol{y}_k - \boldsymbol{y}^*\|^2 + \frac{1}{2} \|\boldsymbol{u}_k\|^2 \right) \tag{16.19}$$

and simply apply LQR using $N = 100$. The result of changing the airspeed to 10 and maintaining the climb rate, i.e. $\boldsymbol{y}^* = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$, can be found in fig. 16.4. We observe that since the problem plan on a finite horizon it chooses to set the actions $\boldsymbol{u} = \mathbf{0}$ near the end of the simulations since at that point subsequent small deviations from the target $\boldsymbol{y}^*$ are relatively less important than the cost of the actions themselves, but asides that the controller easily manage the maneuver.

### 16.1.4 LQR with Additive Noise

We have so far considered LQR without disturbances. We will now extend the LQR controller to the setting in which additive Gaussian noise disturbs the system. The

system dynamics are

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{\omega}_k \tag{16.20}$$

where $\boldsymbol{\omega}_k \sim \mathcal{N}(0, \Sigma_\omega)$, and the stage-wise cost is

$$c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) = \frac{1}{2}(\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \boldsymbol{u}_k^T R_k \boldsymbol{u}_k). \tag{16.21}$$

with terminal cost $\frac{1}{2}\boldsymbol{x}_N^T Q_N \boldsymbol{x}_N$. We wish to minimize the expected cost. The cost-to-go is

$$J_k^*(\boldsymbol{x}_k) = \boldsymbol{x}_k^T V_k \boldsymbol{x}_k + v_k. \tag{16.22}$$

where $V_k$ is a positive definite matrix as in the deterministic case, and $v_k$ is an additive constant term. We leave the proof of this cost-to-go to the reader. Plugging into the Bellman equation, we have

$$\begin{aligned} J_k^*(\boldsymbol{x}_k) = &\min_{\boldsymbol{u}_k \in \mathbb{R}^m} \mathbb{E}\Big[\frac{1}{2}\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{u}_k^T R_k \boldsymbol{u}_k + v_{k+1} \qquad (16.23)\\ &+ \frac{1}{2}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{\omega}_k)^T V_{k+1}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{\omega}_k) + v_{k+1}\Big] \\ = &\min_{\boldsymbol{u}_k \in \mathbb{R}^m} \Big\{\frac{1}{2}\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{u}_k^T R_k \boldsymbol{u}_k + v_{k+1} \qquad (16.24)\\ &+ \mathbb{E}\Big[\frac{1}{2}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{\omega}_k)^T V_{k+1}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{\omega}_k)\Big]\Big\}. \end{aligned}$$

Following the same minimization procedure as for LQR, we see that the policy is identical to that in Section 16.1. Then, plugging the policy back in to the dynamic programming recursion, we have

$$J_k^*(\boldsymbol{x}_k) = \boldsymbol{x}_k^T (Q_k + L_k^T R_k L_k + \mathbb{E}[(A_k + B_k L_k + \boldsymbol{\omega}_k)^T V_{k+1}(A_k + B_k L_k + \boldsymbol{\omega}_k)])\boldsymbol{x}_k + v_{k+1} \tag{16.25}$$

$$= \boldsymbol{x}_k^T (Q_k + L_k^T R_k L_k + (A_k + B_k L_k)^T V_{k+1}(A_k + B_k L_k))\boldsymbol{x}_k + \mathrm{tr}(\Sigma_\omega V_{k+1}) + v_{k+1} \tag{16.26}$$

where $\mathrm{tr}(\cdot)$ denotes the trace. The equality between (16.25) and (16.26) holds as

$$\mathbb{E}[(A_k + B_k L_k)^T V_{k+1} \boldsymbol{\omega}_k] = 0 \tag{16.27}$$

for zero-mean $\boldsymbol{\omega}_k$, and $\mathbb{E}[\boldsymbol{\omega}_k^T V_{k+1} \boldsymbol{\omega}_k] = \mathrm{tr}(\Sigma_\omega V_{k+1})$. Note that this is identical to the noise-free DP recursion, with the exception of the added trace and constant terms which capture the role of the additive noise. Thus, we have two recursive update equations

$$V_k = Q_k + L_k^T R_k L_k + (A_k + B_k L_k)^T V_{k+1}(A_k + B_k L_k) \tag{16.28}$$

$$v_k = v_{k+1} + \mathrm{tr}(\Sigma_\omega V_{k+1}) \tag{16.29}$$

where the first is the standard Riccati recursion, and the second captures the additive constant term.

In summary, we have reached the surprising outcome that with additive Gaussian noise, we obtain the same optimal policy as in the deterministic case. The total cost has increased, but it is typical to not store the constant term in the DP recursion, as it does not impact the policy.

### 16.1.5 LQR with (Bi)linear Cost and Affine Dynamics

The previous two subsections have presented the most common formulation of the LQR setting. In this subsection, we will derive the discrete time LQR controller for a more general system with bilinear/linear terms in the cost and affine terms in the dynamics. This derivation will be the basis of algorithms we will build up in the following subsections. More concretely, we consider systems with stage-wise cost

$$c(\boldsymbol{x}_k, \boldsymbol{u}_k) = \frac{1}{2}\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{u}_k^T R_k \boldsymbol{u}_k + \boldsymbol{u}_k^T H_k \boldsymbol{x}_k + \boldsymbol{q}_k^T \boldsymbol{x}_k + \boldsymbol{r}_k^T \boldsymbol{u}_k + q_k, \tag{16.30}$$

terminal cost

$$c_N(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T Q_N \boldsymbol{x}_k + \boldsymbol{q}_N^T \boldsymbol{x}_k + q_N, \tag{16.31}$$

and dynamics

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + d_k. \tag{16.32}$$

The cost-to-go will take the form

$$J_k(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T V_k \boldsymbol{x}_k + \boldsymbol{v}_k^T \boldsymbol{x}_k + v_k. \tag{16.33}$$

Repeating our approach from the last subsection, we have

$$\begin{aligned} J_k^*(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k \in \mathbb{R}^m} \Big\{ & \frac{1}{2}\boldsymbol{x}_k^T Q_k \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{u}_k^T R_k \boldsymbol{u}_k + \boldsymbol{u}_k^T H_k \boldsymbol{x}_k + \boldsymbol{q}_k^T \boldsymbol{x}_k + \boldsymbol{r}_k^T \boldsymbol{u}_k + q_k \\ & + \frac{1}{2}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d}_k)^T V_{k+1}(A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d}_k) \\ & + \boldsymbol{v}_{k+1}^T (A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d_k}) + v_{k+1} \Big\}. \end{aligned} \tag{16.34}$$

Rearranging, we have

$$\begin{aligned} J_k^*(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k \in \mathbb{R}^m} \Big\{ & \frac{1}{2}\boldsymbol{x}_k^T (Q_k + A_k^T V_{k+1} A_k)\boldsymbol{x}_k + \frac{1}{2}\boldsymbol{u}_k^T (R_k + B_k^T V_{k+1} B_k)\boldsymbol{u}_k \\ & + \boldsymbol{u}_k^T (H_k + B_k^T V_{k+1} A_k)^T \boldsymbol{x}_k + (\boldsymbol{q}_k + A_k^T V_{K+1} \boldsymbol{d}_k + A_k^T \boldsymbol{v}_{k+1})^T \boldsymbol{x}_k \\ & + (\boldsymbol{r}_k + B_k^T V_{k+1} \boldsymbol{d}_k + B_k^T \boldsymbol{v}_{k+1})\boldsymbol{u}_k + (v_{k+1} + \frac{1}{2}\boldsymbol{d}_k^T V_{k+1} \boldsymbol{d}_k + \boldsymbol{v}_{k+1}^T \boldsymbol{d}_k) \Big\}. \end{aligned} \tag{16.35}$$

Solving this minimization problem, we see that our optimal controller takes the form

$$\boldsymbol{u}_k^* = \boldsymbol{l}_k + L_k \boldsymbol{x}_k. \tag{16.36}$$

Where the matrices used to compute $\boldsymbol{L}_k$ and $\boldsymbol{l}_k$ are defined in algorithm 22.

### 16.1.6 Regularization

Assuming the matrices $Q_k$ are positive semi-definite and the matrices $R_k$ are positive definite the discrete LQR controller is guaranteed to not diverge. However, if the

☞ **Algorithm 22** Discrete LQR

**Require:** A problem of horizon length $N$ and a problem $A_0, A_1, \dots$ $B_k$, $Q_k$, etc. **return** An optional regularization parameter $\mu$ (default $\mu = 0$)

1: Initialize $V_N = Q_N$; $\boldsymbol{v}_N = \boldsymbol{q}_N$; $v_N = q_N$
2: **for** $k = N-1$ to $k = 0$ **do**
3: Compute intermediary matrices

$$S_{\boldsymbol{uu},k} = R_k + B_k^T(V_{k+1} + \mu I)B_k \tag{16.37a}$$
$$S_{\boldsymbol{ux},k} = H_k + B_k^T(V_{k+1} + \mu I)A_k \tag{16.37b}$$
$$S_{\boldsymbol{u},k} = \boldsymbol{r}_k + B_k^\top \boldsymbol{v}_{k+1} + B_k^\top V_{k+1}\boldsymbol{d}_k. \tag{16.37c}$$

4: Compute control matrix/vector:

$$L_k = -S_{\boldsymbol{uu},k}^{-1} S_{\boldsymbol{ux},k} \tag{16.38}$$
$$\boldsymbol{l}_k = -S_{\boldsymbol{uu},k}^{-1} S_{\boldsymbol{u},k} \tag{16.39}$$

5: Compute quadratic approximation of cost-to-go

$$V_k = Q_k + A_k^T V_{k+1} A_k - L_k^T S_{\boldsymbol{uu},k} L_k \tag{16.40}$$
$$\boldsymbol{v}_k = \boldsymbol{q}_k + A_k^T(\boldsymbol{v}_{k+1} + V_{k+1}\boldsymbol{d}_k) + S_{\boldsymbol{ux},k}^T \boldsymbol{l}_k \tag{16.41}$$
$$v_k = v_{k+1} + q_k + \boldsymbol{d}_k^T \boldsymbol{v}_{k+1} + \frac{1}{2}\boldsymbol{d}_k^T V_{k+1}\boldsymbol{d}_k + \frac{1}{2}\boldsymbol{l}_k^T S_{\boldsymbol{u},k} \tag{16.42}$$

6: $V_k \leftarrow \frac{1}{2}(V_k + V_k^\top)$ ▷ Ensure symmetric for numerical stability
7: **end for**
8: In a state $\boldsymbol{x}_k$, the control law is $\boldsymbol{u}_k^* = \boldsymbol{l}_k + L_k \boldsymbol{x}_k$
9: The cost-to-go is $J_k(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T V_k \boldsymbol{x}_k + \boldsymbol{v}_k^T \boldsymbol{x}_k + v_k$.

matrices are poorly conditioned (meaning they are close to not being positive semidefinite/positive definite) the method can be numerically unstable, and this will be a practical issue in the next section. For this reason we follow [TET12] and include a regularization parameter $\mu \geq 0$ such that $\mu = 0$ corresponds to the deviation above. We will return to this parameter in chapter 17.

## 16.2 Bibliographic Notes

A comprehensive coverage of linear quadratic methods for optimal control is Anderson and Moore [AM07]. LQG is covered in discrete time in [BBBB95].

# Chapter 17

# Iterative LQR

In the previous chapter, we saw that a generic linear/quadratic control problem in which the dynamics had the form:

$$\boldsymbol{x}_{k+1} = A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \boldsymbol{d}_k \tag{17.1}$$

could be solved using LQR to yield a controller of the form $\boldsymbol{u}_k = L_k \boldsymbol{x}_k + \boldsymbol{l}_k$. Despite LQR being a powerful approach to optimal control, it suffers from a handful of limitations. First and foremost, it assumes the dynamics are (possibly time-varying) linear, and the cost function is quadratic. In this chapter we will expand this method to include (some) non-linear problems of the form

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{17.2}$$

and a general cost function $c_k$. The approach will be typical for designing feedback controllers, namely to linearize around some operating point. This is an effective method for designing regulators, which aim to control the system to some particular state.

## 17.1 Linearization

Suppose we want to balance the pendulum model upright. Recall in our sin, cos, $\dot{\theta}$ coordinate system this corresponds to driving the pendulum towards the up-right state:

$$\bar{\boldsymbol{x}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{17.3}$$

at which we will apply no force $\bar{\boldsymbol{u}} = 0$. Many non-linear control problems can be defined as finding such as special state, and the rest of the discussion will apply to any such desired state $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$. Assume the system is actually in the state $\boldsymbol{x}_k$ and that we apply control $\boldsymbol{u}_k$. The dynamics can now be Taylor expanded to first order to give us the

approximate dynamics:

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \tag{17.4}$$

$$\approx \boldsymbol{f}_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) + \underbrace{\frac{\partial \boldsymbol{f}_k}{\partial \boldsymbol{x}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}_{A_k}(\boldsymbol{x}_k - \bar{\boldsymbol{x}}) + \underbrace{\frac{\partial \boldsymbol{f}_k}{\partial \boldsymbol{u}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}_{B_k}(\boldsymbol{u}_k - \bar{\boldsymbol{u}}) \tag{17.5}$$

Where $A_k$ and $B_k$ are the Jacobians of the dynamics. We can re-arrange this expression to yield:

$$\boldsymbol{x}_{k+1} \approx A_k \boldsymbol{x}_k + B_k \boldsymbol{u}_k + \underbrace{\boldsymbol{f}_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) - A_k \bar{\boldsymbol{x}} - B_k \bar{\boldsymbol{u}}}_{\boldsymbol{d}_k} \tag{17.6}$$

Since the dynamics is independent of time, meaning $\boldsymbol{f}_k = \boldsymbol{f}_0$, we can therefore simply compute the two matrices $A_0$ and $B_0$ and the vector $\boldsymbol{d}_0$ and we have reduced the non-linear problem to a linear control problem. The cost-function can also be Taylor expanded, here written in condensed form using $\boldsymbol{z}_k = \begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{u}_k \end{bmatrix}$ and $\bar{\boldsymbol{z}} = \begin{bmatrix} \bar{\boldsymbol{x}} \\ \bar{\boldsymbol{u}} \end{bmatrix}$ to give the *quadratic* cost function:

$$c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \approx c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) + \left(\nabla_{\boldsymbol{z}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})\right)^\top (\boldsymbol{z}_k - \bar{\boldsymbol{z}}) + \frac{1}{2}(\boldsymbol{z}_k - \bar{\boldsymbol{z}})^\top H_{\bar{\boldsymbol{z}}}(\boldsymbol{z}_k - \bar{\boldsymbol{z}}) \tag{17.7}$$

To bring this cost-function into a more familiar format we define the terms:

$$c_k = c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) \tag{17.8a}$$

$$c_{\boldsymbol{x},k} = \nabla_{\boldsymbol{x}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}), \quad c_{\boldsymbol{u},k} = \nabla_{\boldsymbol{u}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) \tag{17.8b}$$

$$c_{\boldsymbol{xx},k} = H_{\boldsymbol{x}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}), \quad c_{\boldsymbol{uu},k} = H_{\boldsymbol{u}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) \tag{17.8c}$$

$$c_{\boldsymbol{ux},k} = J_{\boldsymbol{x}} \nabla_{\boldsymbol{u}} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) \tag{17.8d}$$

The last term is the Jacobian of the gradient, i.e. the $i, j$ entry of $c_{\boldsymbol{ux},k}$ is $\frac{\partial^2}{\partial u_i \partial x_j} c_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$. The cost-function can now be written in the standard quadratic format:

$$c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \approx \frac{1}{2}\boldsymbol{x}_k^\top c_{\boldsymbol{xx},k} \boldsymbol{x}_k + \left(c_{\boldsymbol{x},k} + \bar{\boldsymbol{x}}^\top c_{\boldsymbol{xx},k}\right)^\top \boldsymbol{x}_k + \boldsymbol{u}_k^\top c_{\boldsymbol{ux},k} \boldsymbol{x}_k \tag{17.9a}$$

$$+ \frac{1}{2}\boldsymbol{u}_k^\top c_{\boldsymbol{uu},k} \boldsymbol{u}_k + \left(c_{\boldsymbol{u},k} + \bar{\boldsymbol{u}}^\top c_{\boldsymbol{uu},k}\right)^\top \boldsymbol{u}_k \tag{17.9b}$$

$$+ c_k - c_{\boldsymbol{x},k}^\top \bar{\boldsymbol{x}} - c_{\boldsymbol{u},k}^\top \bar{\boldsymbol{u}} + \frac{1}{2}\bar{\boldsymbol{x}}^\top c_{\boldsymbol{xx},k} \bar{\boldsymbol{x}} + \frac{1}{2}\bar{\boldsymbol{u}}^\top c_{\boldsymbol{uu},k} \bar{\boldsymbol{u}} + \bar{\boldsymbol{u}}^\top c_{\boldsymbol{ux},k} \bar{\boldsymbol{x}} \tag{17.9c}$$

Note in these terms all the gradients/Hessians are evaluated in the expansion point $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$ and are therefore independent of the current state/control $(\boldsymbol{x}_k, \boldsymbol{u}_k)$. This gives rise to the method defined in algorithm 23

Note we use the first control law at all future time points. This is because the method is usually intended to stabilize the system around $\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}$, and therefore just using the first law, which is intended to control the system on the longest horizon, is a

☞ **Algorithm 23** Linearized LQR

**Require:** Given a problem horizon $N$ (for instance $N = 50$), and an expansion point $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$ corresponding to where the system is expected to be

1: Compute $A, B, \boldsymbol{d}$ by linearly expanding around $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$ using eq. (17.5) and eq. (17.6). Note these matrices/vectors are constant since we are expanding around a single fixed point.
2: If cost is *not* quadratic, construct quadratic approximation of cost function using eq. (17.9). Else just use the cost matrices.
3: Use algorithm 22, with constant linear dynamics $A, B, d$ and cost matrices $Q_k, R_k, \boldsymbol{q}_k$ (typically also constant), to obtain controller $L_k, \boldsymbol{l}_k$ for $k = 0, \ldots, N-1$.
4: In a state $\boldsymbol{x}_k$, the control law is $\boldsymbol{u}_k^* = \boldsymbol{l}_0 + L_0 \boldsymbol{x}_k$
5: The cost-to-go is $J_k(\boldsymbol{x}_k) = \frac{1}{2}\boldsymbol{x}_k^T V_k \boldsymbol{x}_k + \boldsymbol{v}_k^T \boldsymbol{x}_k + v_k$

reasonable simplification. For the same reason $N$ should just be chosen as reasonably large.

Linearized LQR can be expected to work only when the states $\boldsymbol{x}_k, \boldsymbol{u}_k$ are close to the expansion point $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$. Furthermore, since we linearly approximate the (already approximate) discrete model $\boldsymbol{f}_k$, it is an approximation on top of an approximation, which means there is no guarantee it will work. In practice, it stabilize e.g. the pendulum model, and is a good starting point for more powerful methods.

### 17.1.1 LQR Tracking around a Nonlinear Trajectory

In the previous linearization method, we assumed the expansion point was fixed and selected beforehand as a point which we wished to stabialize around. We can turn this into a more powerful iterative method by letting the expansion point reflect the actual state the system is in.

NOMINAL TRAJECTORY

As a first step, assume we are given a **nominal trajectory** which satisfy our discrete dynamics:

$$\bar{\boldsymbol{x}}_{k+1} = \boldsymbol{f}_k(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k), \text{ for } k = 0, \ldots, N-1. \tag{17.10}$$

In reality, such a trajectory is easily computed by setting $\bar{\boldsymbol{x}}_0$ equal to the initial state of the system and then simulating the system using for instance $\bar{\boldsymbol{u}}_k = \boldsymbol{0}$.

We proceed as the linerized case by simply Taylor expanding around the nominal trajectory:

$$\boldsymbol{x}_{k+1} \approx \underbrace{\boldsymbol{f}(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)}_{=\bar{\boldsymbol{x}}_{k+1}} + \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)}_{A_k}(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k) + \underbrace{\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)}_{B_k}(\boldsymbol{u}_k - \bar{\boldsymbol{u}}_k) \tag{17.11}$$

The only difference is the matrices $A_k$ and $B_k$ will now depend on time, since they are expanded around the nominal trajectory.

We now make the following observation. Suppose we *define* the deviations in state/-control as:

$$\delta \boldsymbol{x}_k = \boldsymbol{x}_k - \bar{\boldsymbol{x}}_k, \quad \delta \boldsymbol{u}_k = \boldsymbol{u}_k - \bar{\boldsymbol{u}}_k. \tag{17.12}$$

We can then move the term $\bar{\boldsymbol{x}}_{k+1}$ to the left-hand side in eq. (17.11) which allows us to rewrite the system in terms of deviations, to get

$$\delta \boldsymbol{x}_{k+1} = A_k \delta \boldsymbol{x}_k + B_k \delta \boldsymbol{u}_k \tag{17.13}$$

which is linear in $\delta \boldsymbol{x}_k, \delta \boldsymbol{u}_k$. We can similarly quadratically expand an arbitrary cost function to obtain the usual quadratic form:

$$c_k(\delta \boldsymbol{x}_k, \delta \boldsymbol{u}_k) = \frac{1}{2}\delta \boldsymbol{x}_k^\top c_{\boldsymbol{xx},k} \delta \boldsymbol{x}_k + c_{\boldsymbol{x},k}^\top \delta \boldsymbol{x}_k + \frac{1}{2}\delta \boldsymbol{u}_k^\top c_{\boldsymbol{uu},k} \delta \boldsymbol{u}_k + c_{\boldsymbol{u},k}^\top \delta \boldsymbol{u}_k + \delta \boldsymbol{u}_k^\top c_{\boldsymbol{ux},k} \delta \boldsymbol{x}_k + c_k \tag{17.14a}$$

$$c_N(\delta \boldsymbol{x}_N) = \frac{1}{2}\delta \boldsymbol{x}_N^\top c_{\boldsymbol{xx},N} \delta \boldsymbol{x}_N + c_{\boldsymbol{x},N}^\top \delta \boldsymbol{x}_N + c_k \tag{17.14b}$$

In other words, considered as a function of $(\delta \boldsymbol{x}_k, \delta \boldsymbol{u}_k)$, the entire problem is of the linear-quadratic form. We can therefore apply LQR to the problem to obtain the optimal controller $(L_k, \boldsymbol{l}_k)$. When this controller is in (deviation) state $\delta \boldsymbol{x}_k$, it computes (deviation) optimal control:

$$\delta \boldsymbol{u}_k^* = L_k \delta \boldsymbol{x}_k + \boldsymbol{l}_k. \tag{17.15}$$

If we re-insert the definition of $\delta \boldsymbol{x}_k, \delta \boldsymbol{u}_k$ we obtain the optimal response in state $\boldsymbol{x}_k$ as:

$$\boldsymbol{u}_k^* = \bar{\boldsymbol{u}}_k + L_k(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k) + \boldsymbol{l}_k. \tag{17.16}$$

To summarize, given any nominal trajectory $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$, we can compute a response $u_k^*$ in each state using the linearized model. Since the nominal trajectory should ideally be close to the optimal path, we can therefore compute a new nominal trajectory by setting $\bar{\boldsymbol{x}}_0$ equal to the initial state $\boldsymbol{x}_0$, and then compute the following states by letting $\bar{\boldsymbol{u}}_k = \boldsymbol{u}_k^*$ computed using eq. (17.16) and using eq. (17.10) to compute $\bar{\boldsymbol{x}}_{k+1}$. The procedure is then repeated using the new nominal trajectory until (presumed) convergence, see algorithm 24.

☞ **Algorithm 24** Basic iLQR

**Require:** Given initial state $\boldsymbol{x}_0$

Set $\bar{\boldsymbol{x}}_k = \boldsymbol{x}_0$, $\bar{\boldsymbol{u}}_k = \mathbf{0}$ (or a random vector), $L_k = \mathbf{0}$ and $\boldsymbol{l}_k = \mathbf{0}$
$\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k \leftarrow$ FORWARD-PASS$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k, L_k, \boldsymbol{l}_k)$ ▷ Compute initial nominal trajectory using eq. (17.10) .
**for** $i = 0$ to a pre-specified number of iterations **do**
    $A_k, B_k, c_k, c_{\boldsymbol{x},k}, c_{\boldsymbol{u},k}, c_{\boldsymbol{xx},k}, c_{\boldsymbol{ux},k}, c_{\boldsymbol{uu},k} \leftarrow$ GET-DERIVATIVES$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    $L_k, \boldsymbol{l}_k \leftarrow$ BACKWARD-PASS$(A_k, B_k, c_k, c_{\boldsymbol{x},k}, c_{\boldsymbol{u},k}, c_{\boldsymbol{xx},k}, c_{\boldsymbol{ux},k}, c_{\boldsymbol{uu},k}, \mu)$
    $J^{(i)} \leftarrow$ COST-OF-TRAJECTORY$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    $\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k \leftarrow$ FORWARD-PASS$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k, L_k, \boldsymbol{l}_k)$
**end for**
Compute control law $\pi_k(\boldsymbol{x}_k) = \bar{\boldsymbol{u}}_k + \boldsymbol{l}_k + L_k(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k)$
**return** $\{\pi_k\}_{k=0}^{N-1}$
**function** FORWARD-PASS$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k, L_k, \boldsymbol{l}_k)$ ▷ Forward-simulation of dynamics
    Set $\boldsymbol{x}_0 = \bar{\boldsymbol{x}}_0$
    **for all** $k = 0, \ldots, N-1$ **do**
        $\boldsymbol{u}_k^* \leftarrow \bar{\boldsymbol{u}}_k + L_k(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k) + \boldsymbol{l}_k$ ▷ see eq. (17.16)
        $\boldsymbol{x}_{k+1} \leftarrow f_k(\boldsymbol{x}_k, \boldsymbol{u}_k^*)$
    **end for**
    **return** $\boldsymbol{x}_k, \boldsymbol{u}_k^*$
**end function**
**function** GET-DERIVATIVES$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    Obtain $A_k, B_k$ and $c_k, c_{\boldsymbol{x},k}$, etc. by expanding around $\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k$ using eq. (17.10).
    **return** $(A_k)_k$, $(B_k)_k$, $(c_k)_k$, $(c_{\boldsymbol{x},k})_k$, $(c_{\boldsymbol{u},k})_k$, $(c_{\boldsymbol{xx},k})_k$, $(c_{\boldsymbol{ux},k})_k$, $(c_{\boldsymbol{uu},k})_k$
**end function**
**function** BACKWARD-PASS$(A_k, B_k, c_{\boldsymbol{x},k}, c_{\boldsymbol{u},k}, c_{\boldsymbol{xx},k}, c_{\boldsymbol{ux},k}, c_{\boldsymbol{uu},k}, \mu)$ eq. (17.14)
    Compute $L_k, \boldsymbol{l}_k$ using dLQR with $\mu$, algorithm 22 ▷ Obtain control law
**end function**
**function** COST-OF-TRAJECTORY$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    **return** $c_N(\bar{\boldsymbol{x}}_N) + \sum_{k=0}^{N-1} c_k(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
**end function**

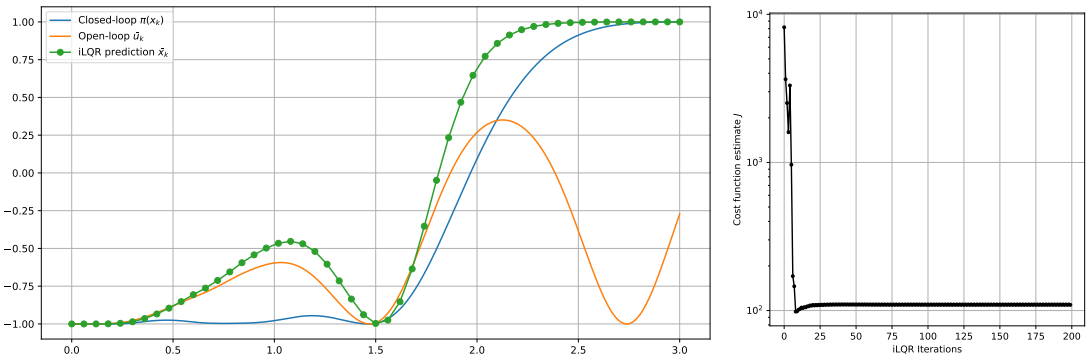


Figure 17.1: Basic ILQR algorithm 24 applied to pendulum problem. Plots show $\cos(\theta)$ and includes the iLQR predictions $\bar{\boldsymbol{x}}_k$ (dotted) as well as result of using an open-loop control with $\boldsymbol{u}_k$ and closed loop controller $\pi(\boldsymbol{x}_k)$. Right pane shows the estimated cost function.

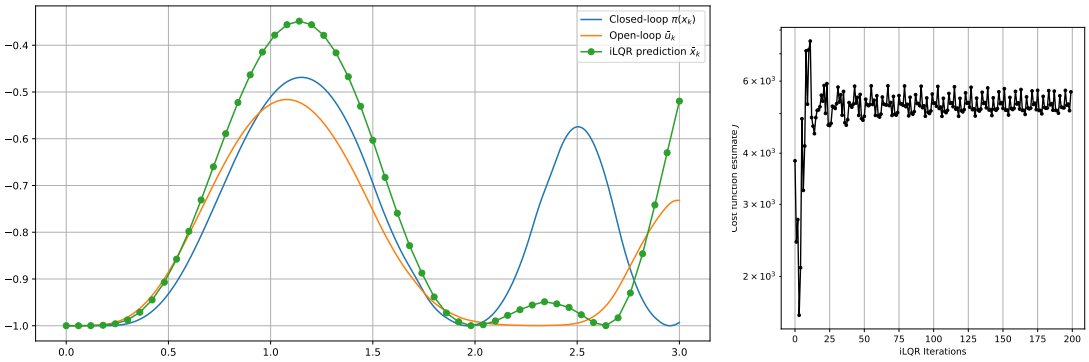


Figure 17.2: Pendulum evaluated similar to fig. 17.1 but using a different seed

### 17.1.2 Example: Pendulum and basic ILQR

We applied the basic ILQR algorithm to the pendulum environment using $N = 50$. Recall the pendulum environment is parameterized in terms of $\sin(\theta)$ and $\cos(\theta)$ such that $\cos(\theta) = 1$ is the upright position, and we included a term in the cost function proportional to $-\cos(\theta)$ to encourage the standing-up position (the other terms are quadratic and selected from the defaults from the openai gym implementation but scaled). As an experiment, we plot both the (predicted) nominal trajectory $\bar{\boldsymbol{x}}_k$, as well as the outcome of using either open-loop control using the nominal actions $\bar{\boldsymbol{u}}_k$ as well as proper closed-loop control using the control matrices eq. (17.16). We also included a plot of the cost function $J(\bar{\boldsymbol{x}}_0)$ for the nominal trajectory during training, and the result can e found in fig. 17.1.

We see the closed-loop is superior to the open-loop controller since the system eventually deviate from the nominal trajectory. The cost function behaves fairly nicely, however it does increase during some basic ILQR iterations. Note that while this result is encouraging, it is also quite seed dependent, and it is instructive to include a trajectory for which the simulation fails (the only difference is the seed is changed

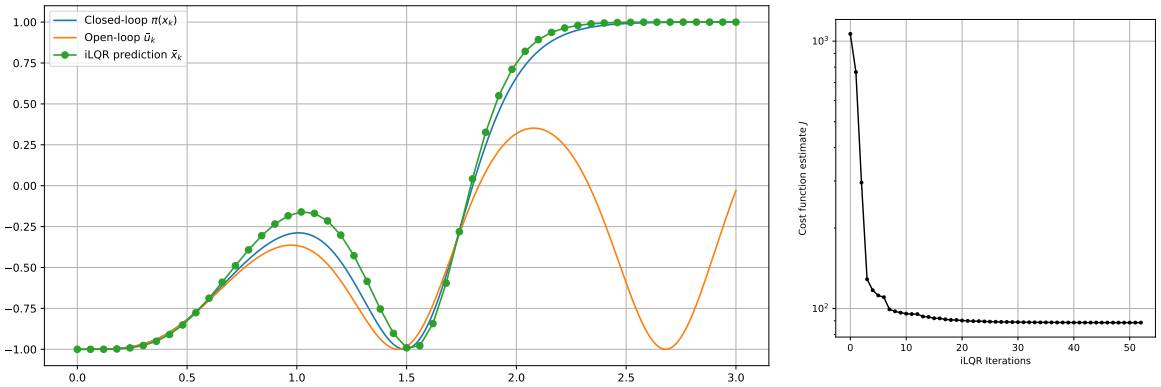


Figure 17.3: Same as fig. 17.1 but using ILQR algorithm 25

from 1 to 2), see fig. 17.2. As seen in the figure the trajectory is very poor, and the cost-function oscillates at a high value. What happens is that the nominal trajectory flip between two configurations.

## 17.2 Iterative LQR

Algorithm 24 is an improvement on the simple linearization procedure, however, it leaves out several details which are critical for obtaining a good performance. The first is it would benefit from using a convergence criteria. In [TL05], the authors stop when the update to the nominal control action sequence is sufficiently small. In [LK14], the authors iterate until the cost of the trajectory (with some additional penalty terms) increases. Finally, a variety of convergence criteria are based on expected trajectory improvement, computed via line search [JM70, TET12].

Furthermore, as the example showed, the method is prone to making too large updates to the nominal trajectory which gives rise to oscillating behavior. An additional complication is due to the quadratic expansion of the cost-function. For general cost-functions, the matrices $S_{\boldsymbol{uu},k}$ may become ill-conditioned, and we may experience divergence when computing the control laws $L_k = S^{-1}_{\boldsymbol{uu},k} S_{\boldsymbol{ux},k}$.

We will describe one approach for fixing these issues proposed by [TET12] (note the article includes additional details and some nice experiments). The idea is quite simple: Firstly, the problem of oscillatory behavior is fixed by replacing eq. (17.16) with the modified update:

$$\boldsymbol{u}^*_k = \bar{\boldsymbol{u}}_k + L_k(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k) + \alpha \boldsymbol{l}_k, \quad 0 \leq \alpha \leq 1 \tag{17.17}$$

To understand this update, if $\alpha$ is small, $\alpha \boldsymbol{l}_k$ will vanish, and in this case $\boldsymbol{u}^* = \bar{\boldsymbol{u}}_k$, and therefore lowering $\alpha$ will tend to eliminate oscillatory behavior.

The second change is the introduction of an regularization parameter $\mu$ in the dLQR algorithm algorithm 22 to avoid numerical underflow. The procedure can now be outlined as follows:

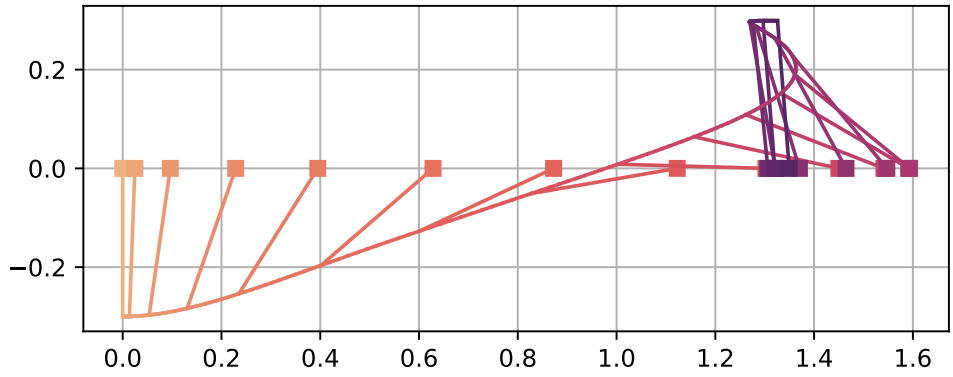


Figure 17.4: Trajectory of cartpole using ILQR from algorithm 25. Cart starts to the left, drives quickly to the right and stabilize. See section 17.2.1

- Initialize regularization parameter to a fairly low value $\mu$
- In the forward pass Line 7 of algorithm 24, use eq. (17.17), but try a range of $\alpha$-values, starting at $\alpha = 1$ and reducing $\alpha$ to 0
- For each $\alpha$-value check if the cost $J^{(i)}$ decreases relative to $J^{(i-1)}$. If so, *accept* this $\alpha$ and decrease the regularization parameter $\mu$ by a small amount
- If no $\alpha$-value works, increase the regularization parameter $\mu$ by a small amount

In [TET12] the *small amount* above is dynamically tuned using a third variable $\Delta$. The update equations are

**Increase $\mu$**

$$\begin{aligned} \Delta &\leftarrow \max\left(\Delta_0, \Delta \cdot \Delta_0\right) \\ \mu &\leftarrow \max\left(\mu_{\min}, \mu \cdot \Delta\right) \end{aligned} \tag{17.18}$$

**Decrease $\mu$**

$$\begin{aligned} \Delta &\leftarrow \min\left(\tfrac{1}{\Delta_0}, \tfrac{\Delta}{\Delta_0}\right) \\ \mu &\leftarrow \begin{cases} \mu \cdot \Delta & \text{if } \mu \cdot \Delta > \mu_{\min} \\ 0 & \text{if } \mu \cdot \Delta < \mu_{\min} \end{cases} \end{aligned} \tag{17.19}$$

The full method can be found in algorithm 25. Note the parameter $\alpha$ is passed to the FORWARD-PASS method, and the only change is the method update $\boldsymbol{u}_k^*$ using eq. (17.17). Similarly, in the BACKWARDS-PASS method, the regularization parameter $\mu$ is simply passed along to the discrete LQR method.

If we re-do the experiment from section 17.1.2 we get the result shown in fig. 17.3. The method seems to converge every time and to a better solution.

Both iLQR, and the extension DDP (see [TET12]) are local methods. Full dynamic programming approaches yield globally optimal feedback policies. In contrast, iLQR

☞ **Algorithm 25** iLQR

**Require:** Given initial state $\boldsymbol{x}_0$

$\mu_{\min} \leftarrow 10^{-6}$, $\mu_{\max} \leftarrow 10^{10}$, $\mu \leftarrow 1$, $\Delta_0 \leftarrow 2$ and $\Delta \leftarrow \Delta_0$
Initialize $\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k$ as before
**for** $i = 0$ to a pre-specified number of iterations **do**
    $A_k, B_k, c_k, c_{\boldsymbol{x},k}, c_{\boldsymbol{u},k}, c_{\boldsymbol{xx},k}, c_{\boldsymbol{ux},k}, c_{\boldsymbol{uu},k} \leftarrow$ GET-DERIVATIVES$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    $L_k, \boldsymbol{l}_k \leftarrow$ BACKWARD-PASS$(A_k, B_k, c_k, c_{\boldsymbol{x},k}, c_{\boldsymbol{u},k}, c_{\boldsymbol{xx},k}, c_{\boldsymbol{ux},k}, c_{\boldsymbol{uu},k}, \mu)$
    $J' \leftarrow$ COST-OF-TRAJECTORY$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    **for** $\alpha = 1$ to a very low value **do**
        $\hat{\boldsymbol{x}}_k, \hat{\boldsymbol{u}}_k \leftarrow$ FORWARD-PASS$(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k, L_k, \boldsymbol{l}_k, \alpha)$
        $J^{\text{new}} \leftarrow$ COST-OF-TRAJECTORY$(\hat{\boldsymbol{x}}_k, \hat{\boldsymbol{u}}_k)$
        **if** $J^{\text{new}} < J'$ **then**
            **if** $\frac{1}{J'}|J^{\text{new}} - J'| <$ a small number **then**
                Method has converged, terminate outer loop and return
            **end if**
            $J' \leftarrow J^{\text{new}}$
            $\bar{\boldsymbol{x}}_k \leftarrow \hat{\boldsymbol{x}}_k$ and $\bar{\boldsymbol{u}}_k \leftarrow \hat{\boldsymbol{u}}_k$
            $\alpha$ **accepted**: Update $\Delta$ and $\mu$ using eq. (17.19) ▷ Reduce regularization
            Break loop over $\alpha$
        **end if**
    **end for**
    **if No $\alpha$-value was accepted then**
        Update $\Delta$ and $\mu$ using eq. (17.18) ▷ Increase regularization
    **end if**
**end for**
Compute controller $\{\pi_k\}_{k=0}^{N-1}$ as before from $L_k, \boldsymbol{l}_k$

and DDP yield nominal trajectories and local stabilizing controllers. However, these local controllers are often sufficient for tracking the trajectory. As they are local method, choice of initial control sequence is important, and poor choice may result in poor convergence. Additionally, we have not considered constraints on either state or action in the derivation of iLQR or DDP. This is currently an active area of research [XLH17, TMT14, GB17].

### 17.2.1 Example: Cartpole

Our last example will concern the more challenging cartpole environment. We consider a fairly short-term trajectory optimization problem where the distance from top position is penalized heavily. The non-linesearch variant of ILQR failed completely on this task, whereas iLQR finds a good trajectory using $N = 50$ as shown in fig. 17.5. The trajectory is also illustrated in fig. 17.4

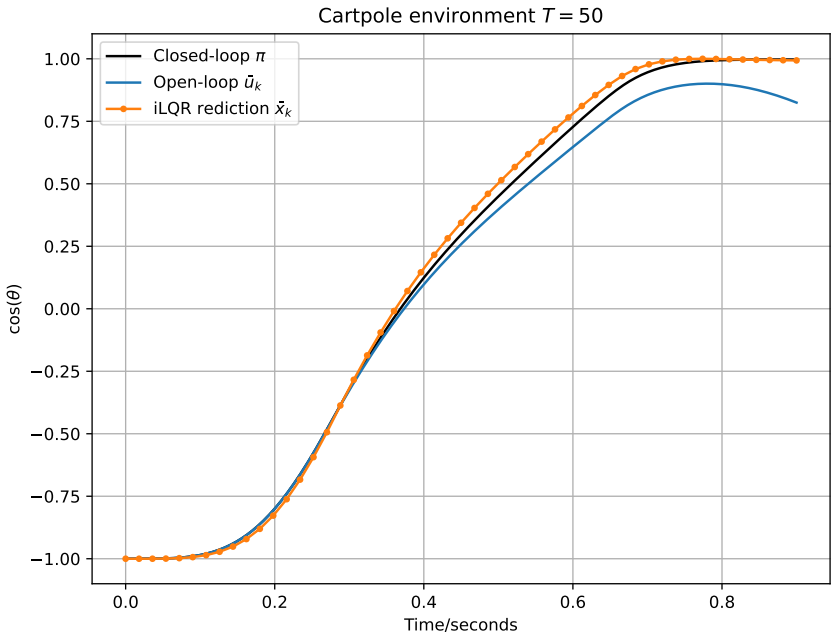


Figure 17.5: ILQR solution to the Cartpole problem. The figure shows $\cos(\theta)$ such that $\cos(\theta) = 1$ corresponds to the upright position

## 17.3 Bibliographic Notes

A comprehensive coverage of linear quadratic methods for optimal control is Anderson and Moore [AM07]. LQG is covered in discrete time in [BBBB95]. The original, comprehensive reference on DDP is [JM70], but a large body of literature on the method has been produced since then. The original papers on iLQR are [TL05, LT04].

# Chapter 18

# System estimation

System estimation is a topic in control theory where we wish to apply model-based planning, but the model is not fully known so we have to learn it from observations.

This setup is fairly close to reinforcement learning in spirit, however, since the methods are focused on learning a model (rather than a value function, as will be the case in reinforcement learning) the resulting methods will be quite dissimilar.

Model based learning is often difficult to get to work well and has found most success for fairly low-dimensional problems, however, when it works the methods are far superior to e.g. reinforcement learning.

## 18.1 Introduction

Lets consider a setup similar to the linear quadratic regulator, but in a simplified setting to avoid needless clutter:

$$\boldsymbol{x}_{t+1} = A\boldsymbol{x}_t + B\boldsymbol{u}_t + \boldsymbol{d} \tag{18.1}$$

and as usual we have to determine the controls $\boldsymbol{u}_t$ to minimize the $N$-step cost function:

$$J = \sum_{t=0}^{N-1} \left[\boldsymbol{x}_t^\top Q \boldsymbol{x}_t + \boldsymbol{u}_t R \boldsymbol{u}_t\right] + x_N^\top Q \boldsymbol{x}_N \tag{18.2}$$

Obviously, if $A, B, Q$ and $R$ were all known this problem could be solved using the standard LQR, however the setup we are interested in is one where $Q$ and $R$ is known (this is a realistic assumption since they denote what *we* think is desirable behavior), however $A$ and $B$ are *not* known.

Let us assume we have simulated the system a number of times, and thereby obtained several observed trajectories $r = 1, \ldots, R$ of the form:

$$(\boldsymbol{x}_t^{(r)})_{t=0}^{N^{(r)}}, (\boldsymbol{u}_t^{(r)})_{t=0}^{N^{(r)}-1}. \tag{18.3}$$

From these trajectories, we want to estimate $A$, $B$ and $\boldsymbol{d}$ in eq. (18.1). Note that since the observed trajectories are from the physical system, they may not exactly obey

eq. (18.1) due to noise or because the model is not completely linear. In other words we assume the trajectories obey:

$$\boldsymbol{x}_{t+1}^{(r)} = A\boldsymbol{x}_t^{(r)} + B\boldsymbol{u}_t^{(r)} + \boldsymbol{d} + \boldsymbol{\epsilon}_t^{(r)}, \quad t = 0, \ldots, N^r - 1, r = 1, \ldots, R \tag{18.4}$$

for (small) noise factor $\boldsymbol{\epsilon}_t^{(r)}$. The problem can now be stated as follows: Determining $A$, $B$ and $\boldsymbol{d}$ such that eq. (18.4) is true for all $i, r$ and while minimizing the cost:

$$\sum_{r=1}^{R}\sum_{t=0}^{N^r} \|\epsilon_t^{(r)}\|^2. \tag{18.5}$$

### 18.1.1 Solving the problem

A few simplifications will help us greatly. The first is the observation that which trajectory $r$ and time index $t$ a set of observations $\boldsymbol{x}_t^{(r)}$, $\boldsymbol{u}_t^{(r)}$ and $\boldsymbol{x}_t^{(r)}$ are from is irrelevant; all that matters is the transition took place. We can therefore consider all $D = N^{(r)} - 1 + \cdots N^R - 1$ transitions as one large dataset indexed as:

$$\boldsymbol{x}_i = \boldsymbol{x}_t^{(r)} \tag{18.6}$$

$$\boldsymbol{u}_i = \boldsymbol{u}_t^{(r)} \tag{18.7}$$

$$\boldsymbol{x}_i' = \boldsymbol{x}_{t+1}^{(r)} \tag{18.8}$$

In which case eq. (18.4) can be written as

$$\boldsymbol{x}_i' = A\boldsymbol{x}_i + B\boldsymbol{u}_i + \boldsymbol{d} + \boldsymbol{\epsilon}_i, \quad i = 1, \ldots, D. \tag{18.9}$$

We can simplify the problem even further by observing that in these $n$ equations the rows of $A$, $B$, $\boldsymbol{d}$ and $\boldsymbol{\epsilon}_i$ can be specified independently of each other; changing one row does not alter whether the equations relating to another row are true or not. We will therefore focus on just a particular row $\ell$ in eq. (18.9). Assuming

$$\boldsymbol{x}_i' = \begin{bmatrix} x_{i,1}' \\ \vdots \\ x_{i,n}' \end{bmatrix}, \quad A = \begin{bmatrix} \boldsymbol{a}_1^\top \\ \vdots \\ \boldsymbol{a}_n^\top \end{bmatrix}, \quad B = \begin{bmatrix} \boldsymbol{b}_1^\top \\ \vdots \\ \boldsymbol{b}_n^\top \end{bmatrix}, \qquad \boldsymbol{\epsilon}_i = \begin{bmatrix} \epsilon_{i,1} \\ \vdots \\ \epsilon_{i,n} \end{bmatrix}. \tag{18.10}$$

then eq. (18.9) can be written as

$$x_{i,\ell}' = \boldsymbol{a}_\ell^\top \boldsymbol{x}_i + \boldsymbol{b}_\ell^\top \boldsymbol{u}_i + c_\ell + \epsilon_{i,\ell} \tag{18.11}$$

$$= \begin{bmatrix} \boldsymbol{a}_\ell^\top & \boldsymbol{b}_\ell^\top & c_\ell \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \\ 1 \end{bmatrix} + \epsilon_{i,\ell} \tag{18.12}$$

This is perhaps beginning to look familiar. In fact if we define

$$y_i = x_{i,\ell}', \quad \boldsymbol{z}_i = \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \\ 1 \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} \boldsymbol{a}_\ell \\ \boldsymbol{b}_\ell \\ c_\ell \end{bmatrix} \tag{18.13}$$

☞ **Algorithm 26** Linear dynamics estimation

**Require:** A sequence $(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{x}'_i)_{i=1}^D$ such that $\boldsymbol{x}'_i$ immediately follows $\boldsymbol{x}_i$ when taking action $\boldsymbol{x}_i$ ▷ see eq. (18.4) and eq. (18.9)

**Require:** $\lambda \geq 0$ and weights $k_i$ ▷ Default: $k_i = 1$

1: Construct $\boldsymbol{Z}$ using eq. (18.15) and eq. (18.13)
2: **for** $\ell = 1, \ldots, n$ **do** ▷ $n$ is the dimensions of $\boldsymbol{x}_i$
3: Construct $\boldsymbol{y}$ using eq. (18.13)
4: $\boldsymbol{w} \leftarrow \left(\boldsymbol{Z}^\top K \boldsymbol{Z} + \lambda I\right)^{-1} \boldsymbol{Z}^\top K \boldsymbol{y}$. ▷ eq. (18.21)
5: $\begin{bmatrix}\boldsymbol{a}_\ell^\top & \boldsymbol{b}_\ell^\top & c_\ell\end{bmatrix} \leftarrow \boldsymbol{w}^\top$ ▷ Unpack by matching dimensions
6: **end for**
7: Gather $A = \begin{bmatrix}\boldsymbol{a}_1^\top \\ \vdots \\ \boldsymbol{a}_n^\top\end{bmatrix}$, $B = \begin{bmatrix}\boldsymbol{b}_1^\top \\ \vdots \\ \boldsymbol{b}_n^\top\end{bmatrix}$
8: **return** $A$, $B$, $\boldsymbol{d}$

We get that (18.12) is simply:

$$y_i = \boldsymbol{z}_i^\top \boldsymbol{w} + \epsilon_i \tag{18.14}$$

or in vector/matrix notation:

$$\boldsymbol{y} = \boldsymbol{Z}\boldsymbol{w} + \boldsymbol{\epsilon}, \quad \boldsymbol{Z} = \begin{bmatrix}\boldsymbol{z}_1^\top \\ \boldsymbol{z}_2^\top \\ \vdots \\ \boldsymbol{z}_D^\top\end{bmatrix} \tag{18.15}$$

(we have dropped the subscript $\ell$ from $\epsilon_i$ for simplificty). The objective is still to minimize $\epsilon_i$, i.e. the error function $\sum_{i=1}^D \epsilon_i^2$, however, for reasons that will be apparent later we generalize this error function slightly to be of the form:

$$E(\boldsymbol{w}) = \sum_{i=1}^D k_i \epsilon_i^2 + \lambda \|\boldsymbol{w}\|^2 \tag{18.16}$$

for a sequence of non-negative weights $k_i$ and a regularization term $\lambda$. This problem is exactly equivalent to the simple, linear regression problem except for the presense of the weights $\lambda_i$. Solving the problem is not hard. If we define the diagonal weight-matrix

$$K_{ij} = \begin{cases} k_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}. \tag{18.17}$$

we can re-write the cost function eq. (18.16) to

$$E(\boldsymbol{w}) = \boldsymbol{\epsilon}^\top K \boldsymbol{\epsilon} + \lambda \boldsymbol{w}^\top \boldsymbol{w} \tag{18.18}$$

☞ **Algorithm 27** Linear dynamics estimation and MPC

**Require:** $\lambda \geq 0$
**Require:** Control horizon $N$ ▷ Typically quite short
BUFFER $\leftarrow ()$ ▷ an empty list
**for** $t = 0, 1, \ldots$ **do**
  $\boldsymbol{x}_t \leftarrow$ Current environment state
  **if** BUFFER is too small **then** ▷ Buffer too small for estimation
    $\boldsymbol{u}_t \leftarrow$ random action
  **else**
    Get all transitions $(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{x}'_i)_i$ in BUFFER.
    Estimate $A, B, \boldsymbol{d}$ using algorithm 26 with $\lambda$
    Obtain control laws $(L_k, \boldsymbol{l}_k)_{k=0}^{N-1}$ using LQR algorithm 22
    $\boldsymbol{u}_t \leftarrow L_0 \boldsymbol{x}_t + \boldsymbol{l}_0$
  **end if**
  Take action $\boldsymbol{u}_k$, observe next state $\boldsymbol{x}_{t+1}$
  Append $(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1})$ to BUFFER.
**end for**

Inserting eq. (18.15), differentiating with respect to $\boldsymbol{w}$, and setting the derivative equal to zero gives:

$$\mathbf{0} = \nabla_{\boldsymbol{w}} E(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} \left[ (\boldsymbol{y} - \boldsymbol{Z}\boldsymbol{w})^\top K (\boldsymbol{y} - \boldsymbol{Z}\boldsymbol{w}) + \boldsymbol{w}^\top \boldsymbol{w} \right] \tag{18.19}$$

$$= \boldsymbol{Z}^\top K \boldsymbol{Z}\boldsymbol{w} - \boldsymbol{Z}^\top K \boldsymbol{y} + \boldsymbol{w} \tag{18.20}$$

Solving this gives the near-familiar expression;

$$\boldsymbol{w} = \left( \boldsymbol{Z}^\top K \boldsymbol{Z} + \lambda I \right)^{-1} \boldsymbol{Z}^\top K \boldsymbol{y}. \tag{18.21}$$

Pseudo-code is provided in algorithm 26

### 18.1.2 Using the linear dynamical model

The method for estimating the linear dynamical system algorithm 26 can be used by simply collecting a lot of data, estimate $A$, $B$ and $\boldsymbol{d}$, and then applying LQR to obtain a controller. However, it is instructive to consider a model which tries to control the system *while* it is learning the dynamics. We can easily do this using MPC as illustrated in algorithm 27. This algorithm converges fairly quickly, however it suffers from the problem that it estimate the matrices $A$, $B$, $\boldsymbol{d}$ based on all data in each step. Obviously the matrices (and therefore control laws) will change very little with a lot of data, and so it may make sense to stop training once buffer reaches a particular size.

☞ **Algorithm 28** MPC and local Linear estimation

**Require:** Control horizon $N$ ▷ Typically quite short
**Require:** Number of linerarization points $K$
**for** $t = 0, 1, \ldots$ **do**
  $\boldsymbol{x}_t \leftarrow$ Current environment state
  **if** Buffer is too small **then**
    $\boldsymbol{u}_t \leftarrow$ random action
  **else**
    **if** $t = 0$ **then**
      $\bar{\boldsymbol{x}}_k \leftarrow \boldsymbol{x}_0, \quad \bar{\boldsymbol{u}}_k \leftarrow$ random action ▷ Initialization
    **end if**
    $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)_{k=0}^{N-1} \leftarrow (\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)_{k=1}^{N}$ ▷ Shuffle since one time step has elapsed
    **for** $k = 0, \ldots, N-1$ **do**
      Compute local neighborhood $\mathcal{N}_K(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
      $A_k, B_k, \boldsymbol{d}_k \leftarrow$ Linear regression algorithm 26 using $\mathcal{N}_K(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
    **end for**
    $\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k \leftarrow$Solve$((A_k, B_k, \boldsymbol{d}_k)_{k=0}^{N}, \boldsymbol{x}_t)$
    Take action $\bar{\boldsymbol{u}}_0$, observe next state $\boldsymbol{x}_{t+1}$
    Append $(\boldsymbol{x}_t, \bar{\boldsymbol{u}}_0, \boldsymbol{x}_{t+1})$ to Buffer.
  **end if**
**end for**
**function** Solve$((A_k, B_k, \boldsymbol{d}_k)_{k=0}^{N}, \boldsymbol{x}_t)$
  Obtain control laws $(L_k, \boldsymbol{l}_k)_{k=0}^{N-1}$ using LQR using $A_k, B_k$ and $\boldsymbol{d}_k$ ▷ algorithm 22
  $\bar{\boldsymbol{x}}_0 \leftarrow \boldsymbol{x}_0$
  **for** $k = 0, \ldots N$ **do**
    $\bar{\boldsymbol{u}}_k \leftarrow L_k \bar{\boldsymbol{x}}_k + \boldsymbol{l}_k$
    $\bar{\boldsymbol{x}}_{k+1} \leftarrow A_k \bar{\boldsymbol{x}}_k + B_k \bar{\boldsymbol{u}}_k + \boldsymbol{d}_k$
  **end for**
  **return** $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)_k$
**end function**

## 18.2 Non-linear problems

Recall the iLQR algorithm in it's basic form algorithm 24 solves a non-linear control task using a linear controller. It does so by maintaining a path $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)_k$ it linearises the system around, i.e. approximate the dynamics as:

$$\boldsymbol{x}_{k+1} \approx A(\boldsymbol{x}_k - \bar{\boldsymbol{x}}_k) + B(\boldsymbol{u}_k - \bar{\boldsymbol{u}}_k) + \boldsymbol{d} \tag{18.22}$$

iLQR then proceeds using discrete LQR. Thus, for us to do something similar, all we need are good approximations of $A$, $B$, $\boldsymbol{d}$ which are approximately valid near $\bar{\boldsymbol{x}}_k$ and $\bar{\boldsymbol{u}}_k$. We can find these using **local linear regression**, which simply means that

LOCAL LINEAR REGRESSION

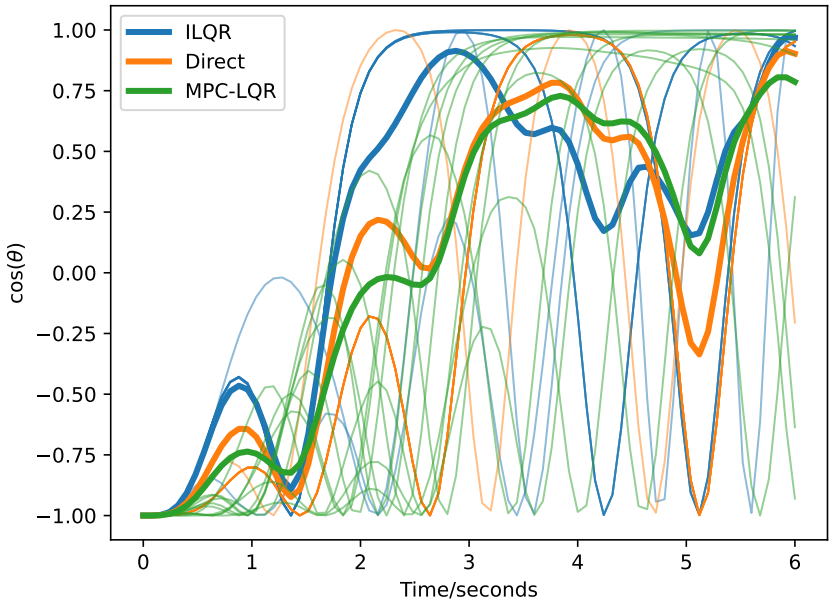


Figure 18.1: Non-linear MPC on pendulum environment

instead of considering the full dataset

$$(\boldsymbol{z}_1, y_1), \ldots, (\boldsymbol{z}_D, y_D) \tag{18.23}$$

from the buffer we only use those observations where $\boldsymbol{z}_i$ is close to $(\bar{\boldsymbol{x}}_i, \bar{\boldsymbol{u}}_i)$. Concretely, define the $K$-nearest neighborhoood as

$$\mathcal{N}(\bar{\boldsymbol{z}} = (\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})) = \{K \text{ nearest } (\boldsymbol{z}_i, y_i) \text{ measured by } d(\boldsymbol{z}_i; \bar{\boldsymbol{z}}) \} \tag{18.24}$$

Where we have introduced a general distance $d$. The method can then be summarized as follows:

- Initialize $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$ over a planning horizon $L$
- For each $k$
  - Compute neighborhood $\mathcal{N}(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$
  - Obtain linearized problem $A_k$, $B_k$, $\boldsymbol{d}_k$ based on neighborhood
- Solve LQR problem of horizon length $L$ using dynamics $(A_k, B_k, \boldsymbol{d}_k)_{k=0}^{L}$ and cost matrices defined by problem
- Update $(\boldsymbol{x}_k, \boldsymbol{u}_k)_{k=0}^{L}$ to agree with trajectory defined above

There are a few outstanding details, but the full procedure can be found in algorithm 28.

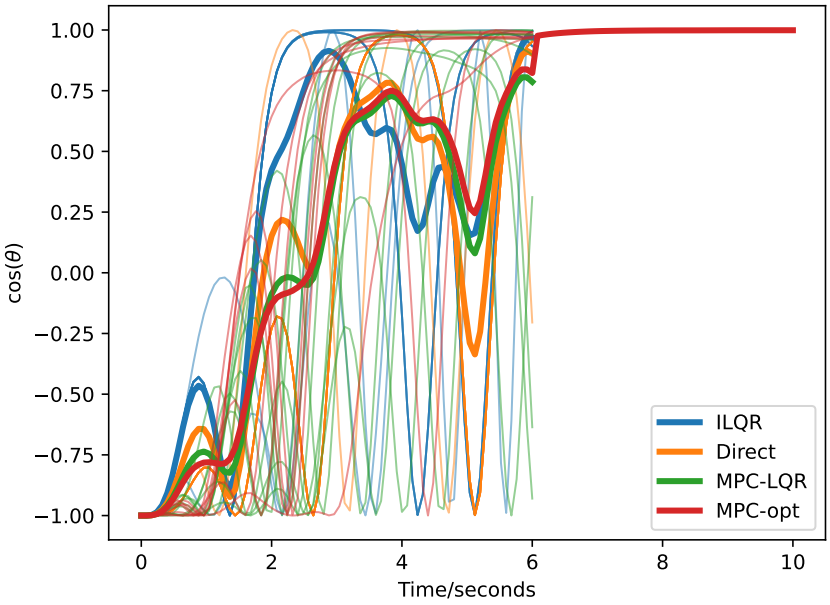


Figure 18.2: Non-linear MPC on pendulum environment

### 18.2.1 Example: Pendulum swingup

We apply the method to the pendulum swingup task. The environment is simulated over a slightly longer time horizon and the method is trained on data from 8 trajectories using a time discretization of $\Delta = 0.08$ seconds, MPC horizon length of $L = 12$ and a local neighborhood size of 40. The time discretization and horizon length had to be tweaked for the method to work, as had to the cost function. We compared against a ILQR solver and an open-loop direct trajectory planner. The result can seen in fig. 18.1, where we have repeated the experiment 10 times to get an idea about the variability.

The methods all achieve swingup, but have some problems stabilizing the pendulum thereafter (this is why the direct solver appears to fare badly, even though it does in fact plan well until stabilization).

It is interesting iLQR and MPC appear to work roughly equally well on average. I think this can be attributed to the planners being approximately similar except one uses MPC and (approximated) matrices which are however quite stable. ILQR would likely benefit from better initialization and I suspect all controllers tend to be too conservative when estimating the applied action signal, i.e. boosting $\boldsymbol{u}$ slightly may be of benefit (I suspect this is due to the Euler discretization).

### 18.2.2 MPC and optimization

In algorithm 28 we used LQR for simplicity. An alternative approach is to simply find the optimal controls by optimizing the $N$-step control problem. Since we re-compute the control trajectory once we use the first control $\boldsymbol{u}_t = \bar{\boldsymbol{u}}_0$ the resulting controller will still be closed-loop, and the use of an optimizer allows us to take constraints into account. Doing this is simply a matter of changing the $\textsc{Solve}(\cdots)$ method as indicated

☞ **Algorithm 29** Alternative solve method

1: **function** SOLVE($(A_k, B_k, \boldsymbol{d}_k)_{k=0}^N$, $\boldsymbol{x}_0$)
2: Build cost-function as

$$J(\boldsymbol{x}_0) = c_f(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} \left(c_k(\boldsymbol{x}_k, \boldsymbol{u}_k)\right) \tag{18.25}$$

3: Determine $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)$ by minimizing $J(\boldsymbol{x}_0)$ according to $(\boldsymbol{x}_k, \boldsymbol{u}_k)$ subject to $\boldsymbol{x}_{k+1} = A_k\boldsymbol{x}_k + B_k\boldsymbol{u}_k + \boldsymbol{d}_k$ and whatever other constraints affect the problem
4: **return** $(\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{u}}_k)_k$
5: **end function**

in algorithm 29

### 18.2.3 Example: Pendulum swingup and optimization

We repeated the pendulum swingup task but included result for the optimization-based controller algorithm 29. The optimizer appears to be on-par with the other methods. It is unclear why the optimizer appears to favor larger action values.

## 18.3 Learning-MPC and the racecar ✸

Learning-MPC (LMPC) can be considered a state-of-the-art approach to real-time learning and control which treats the challenging racecar problem we discussed in section 10.4.3. It has been the subject of a handful of recent papers and the reader is encouraged to look at the video online[1] of how it can quickly learn to control a racecar [RZB18, RB17b, RB17a]. We will follow the implementation available online[2] since the description in the references disagree in minor ways and does not fully match the implementation.

LMPC is essentially algorithm 28 using the optimality-based solver in algorithm 29, but with a few tweeks.

### 18.3.1 Problem formulation

Recall the car-environment (fig. 18.3) is about completing a small track as quickly as possible (i.e., lowest lap time). As we saw in section 10.4.3 it consist of a rather complex 6-dimensional coordinate system $\boldsymbol{x}$ where the car is parameterized according to how much of the lap it has completed, $x_6(t)$, and how far it is from the centerline, $x_5(t)$. The two actions is the steering angle and the acceleration. The car is subject to three

[1] `http://www.mpc.berkeley.edu/research/adaptive-and-learning-predictive-control`
[2] `https://github.com/urosolia/RacingLMPC`

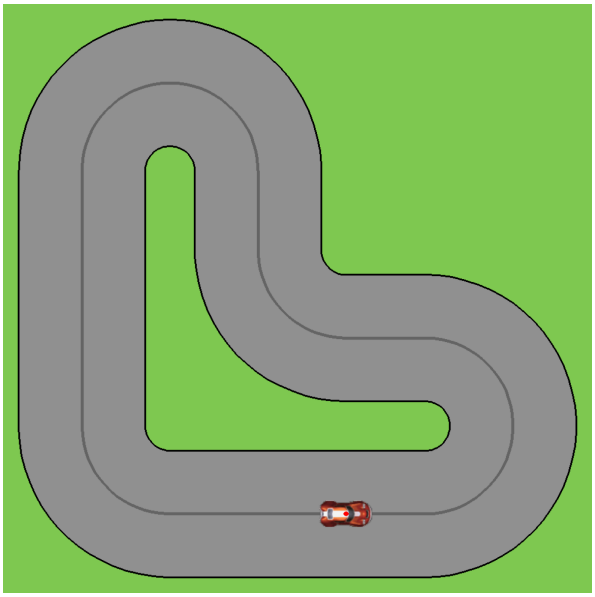

Figure 18.3: Car-environment. Reproduced from fig. 10.2

constraints, namely that it cannot drive over the sides (simple linear constraint on $x_5$) and simple linear constraints on $\boldsymbol{u}(t)$.

The interaction with the environment is in a single episode consisting of a small number $\approx 10$ laps. The coordinate $x_6(t)$ tracks the distance traveled *within the lap* and is therefore reset every time it pass start.

The environment is formulated as a minimum-time problem, i.e. at each time-step until a lap is completed the system simply obtains a cost of

$$c_k(\boldsymbol{x}_t, \boldsymbol{u}_t) = 1 \tag{18.26}$$

Therefore, the cost of lap is the number of time-steps it took to complete the lap.

As a final comment, before the method is applied to the system it is assumed data from two completed tracks have been recorded into the buffer using another method than LMPC. In practice, this occurs using a PID controller (which we have already seen) and a different, simple procedure which will not be discussed here.

### 18.3.2 Linearization

The first noteworthy observation is in how the system is linearized. Firstly, instead of considering data from all laps, the method only use data from the last two completed laps (hence why it is initialized with two laps above).

To determine each $A_k, B_k$ and $\boldsymbol{d}_k$, the method use a combination of analytical approximations and regression. Specifically, recall from eq. (17.6) we can always obtain a linear approximation of the discrete model of the form

$$\boldsymbol{x}_{k+1} \approx A_k^a \boldsymbol{x}_k + B_k^a \boldsymbol{u}_k + \underbrace{\boldsymbol{f}_k(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) - A_k^a \bar{\boldsymbol{x}} - B_k^a \bar{\boldsymbol{u}}}_{\boldsymbol{d}_k^a} \tag{18.27}$$

This model can then be combined with the local linear regression model; i.e. we take some rows from the analytical approximation above and some rows of the analytical approximation model. The reason we would want to do this is because some coordinates of the curve-linear coordinate system depend on each other in a complicated way dictated by the track geometry which is assumed known; others, such as how the motor acceleration affects the speed, depends on car-specific parameters which should be estimated by regression.

These changes are very simple in practice. When we in eq. (18.24) compute the neighborhood, we do so by first computing the scaled distance:

$$d(\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{u}), \bar{\boldsymbol{z}} = (\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})) = \frac{(x_1 - \bar{x}_1)^2}{100} + (x_2 - \bar{x}_2)^2 + (x_3 - \bar{x}_3)^2 + \|\boldsymbol{u} - \bar{\boldsymbol{u}}\|^2. \quad (18.28)$$

Then we select (no more than) 40 points where this distance is less than a critical value of $h = 5$.

Given this distance to each of the at most 40 points in the neighborhood we also compute the kernel weights in the cost function eq. (18.16) using the **Epinichikow** kernel:

Epinichikow

$$k_i = \left(1 - \frac{d(\boldsymbol{z}_i, \bar{\boldsymbol{z}}_i)^2}{h^2}\right) \frac{3}{4}, \quad \boldsymbol{z}_i \in \mathcal{N}(\bar{\boldsymbol{z}}). \quad (18.29)$$

Since the distances are capped at $h$ this is always non-negative.

This brings us to the linear regression. Recall that in the ordinary case, see eq. (18.12), we determine each row $\boldsymbol{a}_\ell$, $\boldsymbol{b}_\ell$ and $c_\ell$ of the relevant matrices through the linear regression problem

$$y_\ell = \boldsymbol{a}^\top \boldsymbol{x} + \boldsymbol{b}_\ell^\top \boldsymbol{u} + c_\ell \quad (18.30)$$

where $\boldsymbol{y}$ is the state immediately proceeding $\boldsymbol{x}$. As mentioned, we only solve this for the first three rows $\ell = 1, 2, 3$, and we do not consider all the input features: once more this is done on a consideration only the first three states should be used for estimation, as well as the natural intuition that the throttle of the car, $u_1$, only affect the velocity in the $x$-direction $x_1$. Specifically for $\ell = 1, 2, 3$ we consider the reduced regression problems:

$$\ell = 1: \quad y_1 = \begin{bmatrix} A_{11} & A_{12} & A_{13} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + B_{12} u_2 + c_1 \quad (18.31)$$

$$\ell = 2: \quad y_1 = \begin{bmatrix} A_{21} & A_{22} & A_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + B_{21} u_1 + c_2 \quad (18.32)$$

$$\ell = 3: \quad y_1 = \begin{bmatrix} A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + B_{21} u_1 + c_2 \quad (18.33)$$

All these regression problems can be solved using the method algorithm 26. This is done for each $k$, and the estimated part of $A_k, B_k$ and $\boldsymbol{d}_k$ above are used to over-write corresponding entries in the linearized version of the system matrices eq. (18.27).

### 18.3.3 The terminal cost approximation

The final point we need to cover is the definition of optimization target eq. (18.25) from algorithm 29.

$$J(\boldsymbol{x}_0) = \sum_{k=0}^{N-1} \left(c_k(\boldsymbol{x}_k, \boldsymbol{u}_k) + c_f(\boldsymbol{x}_N)\right) \tag{18.34}$$

Since the (true) objective, as given in eq. (18.26), is that the car obtains a cost of 1 unit until it completes the lab this function presents the problem that the cost-terms $c_k$ are all constant (and hence independent of the state) as long as the planning horizon is shorter than it takes to complete the lab.

To fix this we need a non-trivial definition of $c_f$, and recall from the finite-horizon formulation section 6.3.3 that $c_f$ should ideally be the optimal cost-to-go starting in $\boldsymbol{x}_N$ and taking optimal actions. Naturally we don't know what this is, but it can be approximated.

To do this, LMPC finds the $K = 44$ points from the last two laps closest to the predicted end-point $\bar{\boldsymbol{x}}_N$, from now denoted by $\boldsymbol{x}_i \in \mathcal{S}_N$.

For all observations $\boldsymbol{x}'_i \in \mathcal{S}_N$ we know how long it actually took to drive to the finish starting from $\boldsymbol{x}'_i$, and we denote that number by $Q_i$. We could approximate $c_f(\boldsymbol{x}_N)$ as $Q_i$ if $\boldsymbol{x}_N = \boldsymbol{x}'_i$ for some $i$ and otherwise $\infty$. This has the benefit of also ensuring the planner plans to a state with a known future trajectory, but it has the disadvantage of making the optimization very difficult. A smarter approach is to introduce the $K$ variables $\alpha_1, \ldots, \alpha_K$ restricted so that

$$\alpha_i > 0, \quad \sum_{i=1}^{N} \alpha_i = 1 \tag{18.35}$$

and assume that

$$\boldsymbol{x}_N = \sum_{i=1}^{N} \alpha_i \boldsymbol{x}'_i + \boldsymbol{\rho} \tag{18.36}$$

$$c_f(\boldsymbol{x}_N) = \sum_{i=1}^{N} \alpha_i Q_i \tag{18.37}$$

The first line says that $\boldsymbol{x}_N$ should be a weighted average of the points we have already seen, i.e. be somewhere inside the convex hull of the set $\mathcal{S}_N$. The variable $\boldsymbol{\rho}$ is a slack-variable restricted to be very small. The second line implies that once we know which $\boldsymbol{x}'_i$ are the most relevant, the expected remaining cost should be a (weighted average) of their expected costs $Q_i$. In addition to these changes, the optimization problem is also changed to add a cost for large controls $\|\boldsymbol{u}_k\|^2$ as well as a new term which accounts for controls not being allow to differ too much $\|u_{k+1} - \boldsymbol{u}_k\|^2$. The later is common in robotics applications to encourage smooth control trajectories; as an example, fig. 18.4

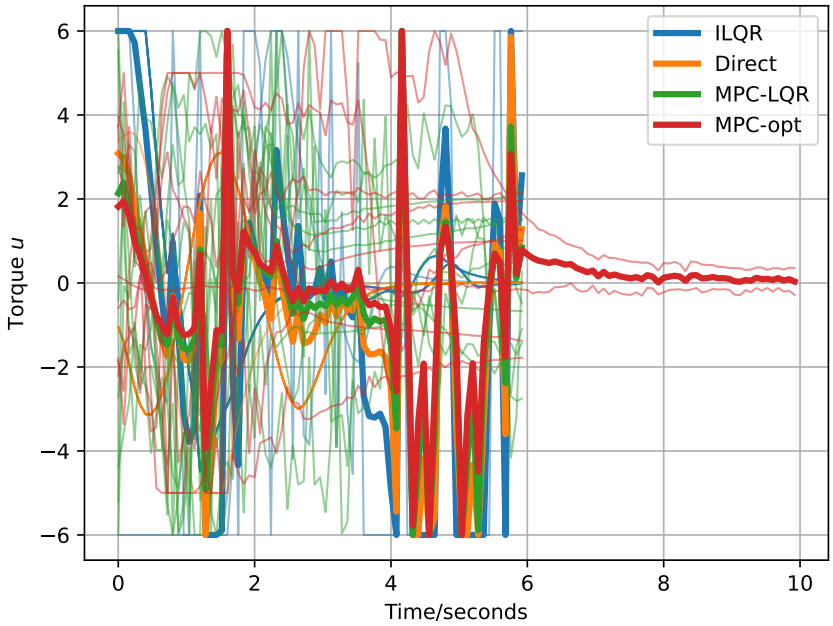


Figure 18.4: Actions in non-linear MPC on pendulum environment

shows the action $u(t)$ for the various methods applied to the pendulum environment, and for all methods except the direct method we see quite erratic action paths.

With these changes, the final cost minimized in eq. (18.25) is therefore:

$$J(\boldsymbol{x}_0) = \sum_i^K \alpha_i \boldsymbol{x}_i' + \boldsymbol{\rho}^\top Q_\rho \boldsymbol{\rho} + 5 \sum_{k=0}^{N-2} \|\boldsymbol{u}_{k+1} - \boldsymbol{u}_k\|^2 + \sum_{k=0}^{N-1} \|\boldsymbol{u}_{k+1}\|^2 \tag{18.38}$$

where $Q_\rho$ is a diagonal matrix such that $Q_{\rho,11} = Q_{\rho,5} = 100$ and $Q_{\rho,22} = Q_{\rho,33} = Q_{\rho,44} = Q_{\rho,66} = 5$.

### Conclusion

There are many details in the above and one can easily get lost in that the method turns out to be fairly simple: First we perform the linear regression to get $(A_k, B_k, \boldsymbol{d}_k)_k$, then we perform the optimization of the target eq. (18.38) subject to the constraints eq. (18.35), eq. (18.36), and the usual linear constraints on the inputs $\boldsymbol{u}_k$. Note that in this problem we also optimize over $\boldsymbol{\rho}$ and $\boldsymbol{\alpha}$, however asides that it is simply a linear-quadratic problem (with the small details it is combined with a linearized version of the model and the specific choices of how the closest neighbors are computed).

Many of these details could likely be done differently. One example is that the implementation computes $\mathcal{S}_N$ as those observations closest to *the current state* $\boldsymbol{x}_t$ rather than $\bar{\boldsymbol{x}}_N$; this is hard to make sense of, since the current state seems to have little to do with the tail cost $N$ steps in the future, whereas $\bar{\boldsymbol{x}}_N$ has the advantage of being our current prediction of where the car will be in $N$ steps.

# Chapter 19

# Preparing for RL

This chapter will serve as a bridge between the notation of the previous chapters and the Markov decision process (MDP) notation used in *"Reinforcement learning: An introduction"* [SB18]. The section assumes the reader has read the first few chapters in this references and will attempt to clarify a few points that might seem vague.

## 19.1 Markov Decision Process

The description of an MDP in [SB18] follows a familiar scheme. Time is indexed as $t = 0, 1, 2, \dots$ and the state at time $t$ is denoted $s_t$ and the action the agent took as $a_t$. The agent then transition to $s_{t+1}$ and obtains a reward $r_{t+1}$ for the transition $(s_t, a_t, s_{t+1})$. Note the shift in time index.

### Random variables ✸

As the reader might have noticed the above description does not mention the equivalent of a random disturbance $w_k$. This is because [SB18] choose, as is common in the MDP literature, to describe all quantities in terms as random variables. Recall a random variable signifies the outcome of an experiment, for instance for a roll of two dies we can define a random variable $S$ to signify the sum of the eyes, and then let $S = 3$ denote the event their sum is 3. There are advantages and disadvantages to this choice:

- It creates a more compact notation which is compatible with the technical literature of MDPs
- It makes it a bit easier to get confused when thinking in-depth about what the notation actually mean. Recall a random variable is *defined* as a function from the sample-space to the set the quantity can lie in. In the die-example, the sample space was all pairs $\{(i, j)|i, j = 0, \dots, 6\}$ and $S((i, j)) = i + j$. In reinforcement learning the sample space is what is technically known as a filtration, and it is much more difficult to actually define $S_t$ beyond it's intuitive meaning.

[SB18] follows the convention of using upper-case letters such as $S_t$ to signify a random variable and $s_t$ it's outcome.

Obviously everyone who reads the algorithms in [SB18] understands that when they see $S_t$ in an algorithm, they should insert the value $s_t$ when they implement it, and everything just works. However, in my view the $f_k$, $g_k$ notation easier to understand since it is easier to relate to a randomly generated number $w_k$ and that the function involved, $f_k$ and $g_k$, are indeed just ordinary functions.

**Transition probability**

Rather than using noise disturbances, [SB18] describe the problem using transition probabilities:

$$p\left(s', r \mid s, a\right) \doteq \Pr\left[S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right] \tag{19.1}$$

Importantly, these transition probabilities obey the Markov property, meaning the future only depends on the present and not the past.

$$\Pr\left[(R_{t+1}, S_{t+1}) \mid (S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0)\right] = \Pr\left[(R_{t+1}, S_{t+1}) \mid (S_t, A_t)\right]. \tag{19.2}$$

### 19.1.1 The terminal time

Most environment we will consider will be episodic, meaning they eventually terminate at a time $T$, giving rise to an episode (or in our language, trajectory):

$$S_0, A_0, R_1, S_1, A_1, R_1, S_2, \ldots S_{T-1}, A_{T-1}, R_T, S_T. \tag{19.3}$$

This gets us to a technical point which [SB18] does state but in my view benefit from emphasis: What is it *specifically* that determines $T$ in each time step? Since $T$ is defined as a random variable, it could in principle depend on all sorts of things such as:

- The state $S_{T-1}$ and $A_{T-1}$
- The state $S_{T-1}$ and $A_{T-1}$ and $S_T$
- Just $S_T$

The first option would for instance mean the agent could take an illegal action in a certain state and therefore the environment terminates (regardless of whether the state $S_T$ is a legal state), and the middle option is in fact what is implemented in OpenAI Gym's `Discrete` environment.

The correct way to define $T$ is fortunately the simplest: First we define the set of all states as $S_t \in \mathcal{S}$ and assume it is divided into two sets $\mathcal{N}$ and $\mathcal{T}$

$$\mathcal{S} = \mathcal{N} \cup \mathcal{T} \tag{19.4}$$

which are non-overlapping $\mathcal{N} \cap \mathcal{T} = \emptyset$. The set $\mathcal{T}$ are the terminal states, i.e. the terminal time $T$ is defined as the first time $t$ the environment enters $\mathcal{T}$:

$$T = \min \{t \mid S_t \in \mathcal{T}\}. \tag{19.5}$$

Therefore, $S_T \in \mathcal{T}$ and $S_t \in \mathcal{N}$ for $t = 0, \ldots, T-1$. Since the environment is now terminated, there are no subsequent states, rewards and actions. This means that the transition probability density $p(s', r|s, a)$, considered as a function $p$ of $(s, a, r, s')$, is not defined for $s \in \mathcal{T}$ since there can in fact be no $s'$. In other words it is a function:

$$p : \mathcal{N} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S} \to [0, 1]. \tag{19.6}$$

This has some important consequences. When we define a policy as the state-action probability:

$$\pi(a|s) = \Pr [A_t = a \mid S_t = s] \tag{19.7}$$

Then it is only defined for $s \in \mathcal{N}$, i.e. $\pi : \mathcal{N} \times \mathcal{A} \to [0, 1]$; whenever you try to call the policy on a state $s_t \in \mathcal{T}$, your code should give an error.

This also means that the value and action-value functions, defined using $G_t = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1}$:

$$V_\pi(s) = \mathbb{E} [G_t \mid S_t = s], \quad Q_\pi(s, a) = \mathbb{E} [G_t \mid S_t = s, A_t = a] \tag{19.8}$$

are *not* defined for $s \in \mathcal{T}$, i.e.

$$v_\pi : \mathcal{N} \to \mathbb{R}, \quad Q_\pi : \mathcal{N} \times \mathcal{A} \to \mathbb{R}. \tag{19.9}$$

This is an important point, since many reinforcement learning algorithms are typeset in such a way it seems like we should use $V_\pi(S_T)$. Consider TD Learning:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha [R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)] \tag{19.10}$$

When $t = T-1$ it indeed seems like we should use $V_\pi(S_T)$. A common recommendation is to define $V_\pi(s) = 0$ for $s \in \mathcal{T}$ (which produce the right result for TD learning), however this idea runs into problems when we use function approximators which all of a sudden needs to be evaluated in $S_T$. The right way to solve the problem is to consider what actually occurs at $t = T - 1$ and realize that, by the derivation of TD learning, the update rule is actually:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha [R_{t+1} - V_\pi(S_t)] \tag{19.11}$$

and then implement this update rule. The above points might appear too exotic to be of real importance, however, I have seen serious implementation of research code where the authors try to access a neural approximation of $V_\pi(S_T = s)$ for $s \in \mathcal{T}$ only to get in troubles, and then try to fix the problem in various ways presented as special optimizations/adaptations of the method.

In fact the question has a very clear answer if we simply consider the definition of the value function and realize the code was buggy the moment we tried to compute $V_\pi(S_T)$. It is for the same reason the course software often define terminal states as something silly like `terminal_state = "terminal state"`; it is a good habit to think of the terminal states as something which truly look nothing like the regular states in $\mathcal{N}$.

As a final note, [SB18] does make this distinction in [SB18, Section 3.3], using $\mathcal{S}^+$ for all states and $\mathcal{S}$ for non-terminal states $\mathcal{N}$, however I think it is a point which is easily overlooked, since this distinction is only introduced *after* the transition probabilities are introduced in [SB18, Section 3.1] (see e.g. definition $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$). These two definitions are not compatible, since the definition of $p$ would imply the environment never terminates.

### 19.1.2 Implementation of an MDP

It is perhaps useful to see a practical example of this distinction. An MDP is implemented as the `MDP` class. I have made convenient functionality to convert a Gym environment to an MDP, and in this case we will convert the Frozen Lake environment. The states/non-terminal states can be computed using:

```
    # mdp.py
    mdp = GymEnv2MDP(gym.make("FrozenLake-v1"))
    print("N = ", mdp.nonterminal_states)
    print("S = ", mdp.states)
    print("Is state 3 terminal?", mdp.is_terminal(3), "is state 11 terminal?", mdp.is_terminal(11))
```

Which produces output

```
N =  [0, 1, 2, 3, 4, 6, 8, 9, 10, 13, 14]
S =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Is state 3 terminal? False is state 11 terminal? True
```

Compare to the first exercises where we worked with the frozen lake environment. The MDP class itself is relatively straight-forward. The only slightly tricky part is that because we want to be able to define MDPs over infinite state sets, then rather than explicitly defining $\mathcal{S}$ and $\mathcal{N}$, we define a `mdp.is_terminal(state)` function to check if a state is terminal or not. The above lists are then computed only if they are requested. The following example sums up the remaining functionality we will need, namely a way to compute the transition-probabilities $p(s', r|s, a)$:

```
    # mdp.py
    state = 0
    print("A(S=0) =", mdp.A(state))
    action = 2
    mdp.Psr(state, action)  # Get transition probabilities
    for (next_state, reward), Pr in mdp.Psr(state, action).items():
        print(f"P(S'={next_state},R={reward} | S={state}, A={action} ) = {Pr:.2f}")
```

Which computes the probability of transitioning to states $4, 1, 0$ given we take action 2 in state 0 (the initial state):

```
A(S=0) = [0, 1, 2, 3]
P(S'=4,R=0.0 | S=0, A=2 ) = 0.33
P(S'=1,R=0.0 | S=0, A=2 ) = 0.33
P(S'=0,R=0.0 | S=0, A=2 ) = 0.33
```

# Appendix A

# Proof of principle of optimality

## A.1 Principle of optimality

Recall that the principle of optimality (PO) claims that:

**Definition A.1.1** (Principle of optimality)**.** *Let $\pi^* = \left\{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\right\}$ be an optimal policy for the basic decision problem, and assume that when following $\pi^*$ there is a positive probability we will find ourselves in state $x_i$ at time $i$. Consider the subproblem where we start in $x_i$ at time $i$ and which to minimize the tail subproblem from $i$ to $N$:*

$$\mathbb{E}\left\{g_N\left(x_N\right) + \sum_{i=k}^{N-1} g_i\left(x_i, \mu_i\left(x_i\right), w_i\right)\right\} \tag{A.1}$$

*Then the truncated policy $\pi^{*,k} = (\mu_i^*, \mu_{i+1}^*, \ldots, \mu_{N-1}^*)$ is optimal for this subproblem:*

$$J_k^*(x_k) = J_{k,\pi^k}(x_k) \tag{A.2}$$

The other proof of the PO I am aware of makes use of certain inequalities and analysis to establish the claim [Ber05], however I thought it would be useful to provide a more direct proof which makes use of the same intuition as the Copenhagen-Berlin example. Before we proceed, we need a few definitions:

- The PO assumes that $x_0$ is fixed. In other words, imagine $\mathcal{S}_0 = \{x_0\}$ for simplicity
- We say a state $x_k$ occurs under $\pi$ if there is a positive chance of reaching $x_k$, starting in $x_0$ and following $\pi$. Equivalently, we say $x_k$ is **reachable** from $x_0$ (under $\pi$)

REACHABLE

- Given a policy $\pi$ and a reachable state $x_k$, a trajectory starting in $x_k$, is simply the sequence of states, actions and disturbances that arose from a roll out starting in $x_k$, and is denoted by $\tau$. Since the policy is fixed, this can be fully specified by stating which disturbances occurred:

$$\tau_{x_k} = (x_k, w_k, w_{k+1}, \ldots, w_{N-1})$$

because from $x_k$ we can compute the actions as $\pi(x_k)$, and given actions and disturbances $w_k$ we can compute $x_{k+1}$ and so on

- We can compute the probability of a trajectory as simply

$$P(\tau_{x_k}) = \prod_{\ell=k}^{N-1} P_k(w_\ell \mid x_\ell, \mu_\ell(x_\ell))$$

- Given a reachable state $x_k$ and a policy $\pi$, we can define the set of all possible trajectories starting in $x_k$ as:

$$T(x_k) = \{\tau_{x_k} | P(\tau_{x_k}) > 0\}$$

- The cost-function is just an average over all possible trajectories. We can therefore re-structure it as simply

$$J_\pi(x_0) = \sum_{\tau \in T(x_0)} P(\tau) \left[ g_N(x_N) + \sum_{k=0}^{N-1} g(x_k, \mu_k(x_k), w_k) \right] \tag{A.3}$$

*Proof.* The proof of the PO is by induction over $N$, where the claim is the PO holds for any $k \geq 1$. We prove this claim by induction over $N$, starting at $N = 1$; in this case all tail policies are empty and therefore by definition optimal.

**Induction step:** For the induction step, assume that $N \geq 2$ and the claim holds for all smaller values of $N$. We consider $\pi^*$ fixed and assume a $x_k$ is reachable from $x_0$ under $\pi^*$, and have to show eq. (A.2). To this end, assume the minimum of the left-hand expression is achieved using a policy $\pi^{k,\text{opt}} = (\mu_k^{\text{opt}}, \mu_{k+1}^{\text{opt}}, \ldots, \mu_{N-1}^{\text{opt}})$, i.e.

$$\pi^{k,\text{opt}} = \arg\min_{\pi^k} \mathbb{E} \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \mid x_k \right\} \tag{A.4}$$

The claim is therefore equivalent to showing

$$J_{k,\pi^{k,\text{opt}}}(x_k) = J_{k,\pi^{k,*}}(x_k). \tag{A.5}$$

We prove this by contradiction. Assume this was not the case, and the optimal tail policy was in fact an improvement on the tail of the optimal policy:

$$J_{k,\pi^{k,\text{opt}}}(x_k) < J_{k,\pi^{k,*}}(x_k). \tag{A.6}$$

Intuitively, what we want to do is stitch together $\pi^{k,\text{opt}}$ and $\pi^*$ and show that $\pi^*$ was in fact not optimal. To do this, consider all trajectories starting in $x_k$ which are possible when following $\pi^{k,\text{opt}}$ $T_{\pi^{k,\text{opt}}}(x_k)$. Given this set, we can define the set of states we might reach starting from $x_k$:

$$R = \{x_\ell \mid x_\ell \text{ is on a trajectory } \tau \in T_{\pi^{k,\text{opt}}}(x_k)\} \tag{A.7}$$

We can then stick together the policies by defining a new policy $\hat{\pi}$ as

$$\hat{\mu}_\ell(x_\ell) = \begin{cases} \mu_\ell^{\text{opt}}(x_\ell) & \text{if } x_\ell \in R \\ \mu_\ell^*(x_\ell) & \text{otherwise} \end{cases} \tag{A.8}$$

This new policy simply follows the old optimal policy unless it reaches a state in $R$ in which the new optimal tail policy is defined in which case it follows $\pi^{k,\text{opt}}$. We derive a contradiction by showing this policy is in fact better than the optimal policy $\pi^*$.

**The Contradiction:** Recall the optimal cost can be written as

$$J_{\pi^*}(x_0) = \sum_{\tau \in T_{\pi^*}(x_0)} P(\tau) \underbrace{\left[ g_N(x_N) + \sum_{k=0}^{N-1} g(x_k, \mu_k(x_k), w_k) \right]}_{=C(\tau)} \tag{A.9}$$

The set of paths $\tau \in T_{\pi^*}(x_0)$ can be divided into a partitioning according to when they first intersect $R$: A set of $T_0$ will never intersect $R$, and otherwise for each $x_r \in R$ we define

$$T_{x_r} = \left\{ \tau \in T_{\pi^*}(x_0) \mid \begin{array}{l} x_r \text{ is on the path } \tau \text{ and there is no preceeding} \\ \text{state } x_0, \ldots, x_{r-1} \text{ on } \tau \text{ which is also in } R \end{array} \right\} \tag{A.10}$$

Obviously, $T_{\pi^*}(x_0) = T_0 \cup \left( \bigcup_{x_r \in R \setminus \{x_k\}} T_{x_r} \right) \cup T_{x_k}$. We can use this set to decompose the cost function eq. (A.9) as follows:

$$J_{\pi^*}(x_0) = \sum_{\tau \in T_{\hat{\pi}}(x_0)} P(\tau) \left[ g_N(x_N) + \sum_{k=0}^{N-1} g(x_k, \mu_k(x_k), w_k) \right] \tag{A.11}$$

$$= \sum_{\tau \in T_0} P(\tau) C(\tau) + \sum_{\tau \in T_{x_k}} P(\tau) C(\tau) + \sum_{x_r \in R \setminus \{x_k\}} \left[ \sum_{\tau \in T_{x_r}} P(\tau) C(\tau) \right] \tag{A.12}$$

The first term will be the same for both policies. Consider the average

$$\sum_{\tau \in T_{x_r}} P(\tau) C(\tau)$$

We can divide the path $\tau$ into the part before $x_r$ and the part after. In other words:

$$= \sum_{\tau \in T_{x_r}} P(\tau) \left[ \sum_{k=0}^{r-1} g_k(x_k, \mu_k(u_k), w_k) + \sum_{k=r}^{N-1} g_k(x_k, \mu_k(u_k), w_k) + g_N(x_N) \right]$$

$$= \sum_{\tau \in T_{x_r}} P(\tau) \left[ \sum_{k=0}^{r-1} g_k(x_k, \mu_k(u_k), w_k) \right] + \sum_{\tau \in T_{x_r}} P(x_r) P(\tau | x_r) \left[ \sum_{k=r}^{N-1} g_k(x_k, \mu_k(u_k), w_k) + g_N(x_N) \right]$$

$$= C_{x_r} + P(x_r) \mathbb{E} \left[ \sum_{k=r}^{N-1} g_k(x_k, \mu_k(u_k), w_k) + g_N(x_N) \right]$$

$$= C_{x_r} + P(x_r) J_{\pi^*, k}(x_r) \tag{A.13}$$

This expression is computed for $\pi^*$, however note that the terms $C_{x_r}$ will be the same for policy $\pi^*$ and the stitched policy $\hat{\pi}$ as they only differ on the set $R$, and we will absorb them into a constant $K$. Using this in eq. (A.12) we obtain

$$J_{\pi^*}(x_0) = K + \underbrace{P(x_k)J_{k,\pi^{*,k}}(x_k)}_{> P(x_k)J_{k,\pi^{k,\text{opt}}}(x_k) \text{ by eq. (A.5)}} + \sum_{x_r \in R \setminus \{x_k\}} [P(x_r)J_{r,\pi^{*,r}}(x_r)] \quad \text{(A.14)}$$

For the terms $J_{r,\pi^{*,r}}(x_r)$ we have to use our induction hypothesis in the following manner: By definition of $R$ in eq. (A.7), we know $x_r$ was a state obtained by following $\pi^{k,\text{opt}}$ from $x_k$, and therefore $r > k$. We can therefore consider $J_{r,\pi^{*,r}}(x_r)$ as a tail cost of the $N-k$-long DP problem obtained from eq. (A.4) and apply the induction hypothesis: We know any tail policy is optimal, and therefore the tail policy of $\pi^{k,\text{opt}}$, starting in $x_r$, must be optimal for the trail-subproblem. Since it is optimal it must have lower (or equal) cost than any policy started in $x_r$, specifically $J_{r,\pi^{*,r}}(x_r)$. Hence,

$$J_{r,\pi^{*,r}}(x_r) \geq J_{r,\pi^{k,\text{opt}}}(x_r)$$

Inserting this eq. (A.14) becomes

$$J_{\pi^*}(x_0) > K + P(x_k)J_{k,\pi^{k,\text{opt}}}(x_k) + \sum_{x_r \in R \setminus \{x_k\}} \left[P(x_r)J_{r,\pi^{k,\text{opt}}}(x_r)\right] \quad \text{(A.15)}$$

$$= J_{\hat{\pi}}(x_0) \quad \text{(A.16)}$$

The last equality sign follows by applying the exact same argument as in eq. (A.12) and eq. (A.13) to $\hat{\pi}$, noting we obtain the same constant, and the remaining terms involving states in $R$ are all computed as averages ovver $\pi^{k,\text{opt}}$ by the definition of the stitched policy eq. (A.8). Finally, simply note that the last equation is in contradiction to $\pi^*$ being an optimal policy, and therefore by contradiction the induction hypothesis is true. □

# Bibliography


[AM07] Brian DO Anderson and John B Moore. *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

[BBBB95] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 1995.

[Ber05] D.P. Bertsekas. *Dynamic Programming and Optimal Control*. Number v. 1 in Athena Scientific optimization and computation series. Athena Scientific, 2005.

[Bet10] John T Betts. *Practical methods for optimal control and estimation using nonlinear programming*, volume 19. Siam, 2010.

[BGJM11] Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. Chapman & Hall, 2011.

[GB17] Markus Giftthaler and Jonas Buchli. A projection approach to equality constrained iterative linear quadratic optimal control. In *IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*, 2017.

[JM70] David Jacobson and David Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

[Kel17a] Matthew Kelly. An introduction to trajectory optimization: how to do your own direct collocation. *SIAM Review*, 2017.

[Kel17b] Matthew Kelly. An introduction to trajectory optimization: How to do your own direct collocation. *SIAM Review*, 59(4):849–904, 2017. (See **kelly2017.pdf**).

[Kel17c] Matthew Kelly. Transcription methods for trajectory optimization: a beginners tutorial. *arXiv:1707.00284*, 2017.

[LK14] Sergey Levine and Vladlen Koltun. Learning complex neural network policies with trajectory optimization. In *International Conference on Machine Learning (ICML)*, 2014.

[LT04] Weiwei Li and Emanuel Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *International Conference on Informatics in Control, Automation, and Robotics*, 2004.

[MRR+53] A. W. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[MU49] N. Metropolis and S. Ulam. The monte carlo method. *J. Am. Stat. Assoc.*, 44:335, 1949.

[RB17a] Ugo Rosolia and Francesco Borrelli. Learning model predictive control for iterative tasks: A computationally efficient approach for linear system. *IFAC-PapersOnLine*, 50(1):3142–3147, 2017.

[RB17b] Ugo Rosolia and Francesco Borrelli. Learning model predictive control for iterative tasks. a data-driven control framework. *IEEE Transactions on Automatic Control*, 63(7):1883–1896, 2017.

[RN09] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.

[Ros05] Jeffrey S. Rosenthal. *A first look at rigorous probability theory*. World Scientific, Singapore [u.a.], reprinted edition, 2005.

[RZB18] Ugo Rosolia, Xiaojing Zhang, and Francesco Borrelli. Data-driven predictive control for autonomous systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 1:259–286, 2018.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).

[TET12] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[TL05] Emanuel Todorov and Weiwei Li. A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *American Control Conference (ACC)*, 2005.

[TMT14] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-limited differential dynamic programming. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[XLH17] Zhaoming Xie, C Karen Liu, and Kris Hauser. Differential dynamic programming with nonlinear constraints. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.