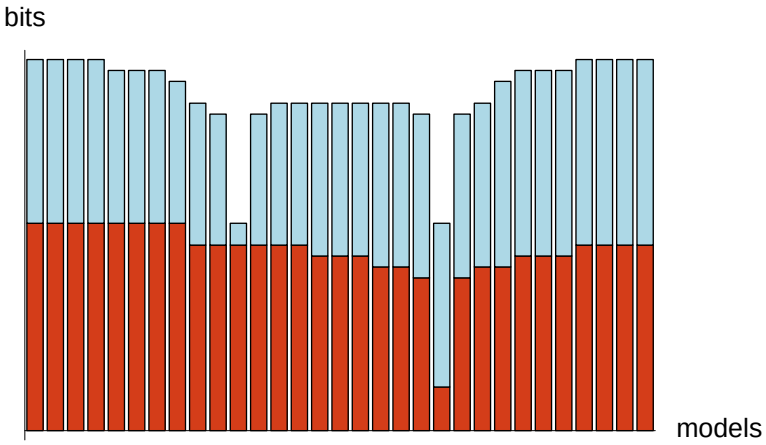
The Minimum Description Length Method with an Application to the Visual Estimation of Vehicle Trajectories

Roberto Fraile



Thesis submitted for the degree of Doctor of Philosophy
Department of Computer Science
The University of Reading
February 2003

Declaration

I confirm that this is my own work and the use of all material from other sources has been properly and fully acknowledged.

Roberto Fraile
DNI 9311230

Abstract

This thesis investigates methods for comparing models of different complexity, using the Minimum Description Length (MDL) method. Some applications to Computer Vision are described. The data are sequences of filmed images, and the models are vehicles and their trajectories. In the MDL approach, a model is favoured if it can be used to obtain good compression of the data.

A major problem in Computer Vision is the choice of adequate model complexity. Simple models might not capture the essential properties of the data, whereas complex models might be strongly affected by noise, resulting in overfitting. Model Selection methods such as Maximum Likelihood are biased towards complex models. Such overfitting can be avoided by carefully weighting the models according to their complexity.

MDL overcomes the problem of overfitting by considering models as Turing machines, and assigning to each model a prior probability. Models are favoured if they have a low prior probability and if they achieve a good compression of the data.

The MDL method is presented, from a combinatorial point of view, as an enumeration of the leaves of a binary tree. This binary tree corresponds to the input language of a Universal Turing machine. Since the enumeration, and with it the MDL method, is not computable, computable approximations to it are devised. A software package written in the language `Mathematica` is used to facilitate the development of these computable approximations to MDL.

The computable approximations to MDL have been tested with experiments in vehicle tracking and trajectory modelling and estimation. Entire image sequences are compressed using 3D models for vehicle shapes and trajectories. In separate experiments, trajectory data are compressed using a dynamics-based model for trajectories. The experiments show that the length after compression is the basis of a reliable method for choosing between trajectory models.

In addition, vehicle trajectories produced by a vehicle tracker are segmented and labelled using a Hidden Markov Model. Experimental results illustrate that a simple model is able to filter out noise and produce a simple segmentation of the trajectory based on the driver's actions.

The conclusion is that it is possible to compare models according to their ability to compress data. The data compression can be carried out with the aid of generic and standard algorithms, such as the Lempel-Ziv-Welch (LZW) compression algorithm or the logstar code.

a Conchi, Primi,
Ana Cris y Eugenia

Acknowledgements

The main thing I have learned during the research reported here is that time and memory are finite, and that this fact might not be as bad as it sounds.

Finite time. To me, having finite time and resources is essential in providing an end to the recursive repeated refinement that sometimes is confused with perfectionism, but which in reality is a form of nihilism. In fact, this finiteness might be enjoyable. Don Knuth said in his Turing Award speech [Knu92]:

> One rather curious thing I've noticed about aesthetic satisfaction is that our pleasure is significantly enhanced when we accomplish something with limited tools. For example, the program of which I personally am most pleased and proud is a compiler I once wrote for a primitive minicomputer that had only 4096 words of memory, 16 bits per word. It makes a person feel like a real virtuoso to achieve something under such severe restrictions.

Not that I have enjoyed the pleasure of having limited resources. Instead, I had sufficient computing power, and I could use a large, and expensive, software package (`Mathematica`, which is a trademark of Wolfram Research). During the past year I even had a large office, something unusual in our department. But large does not imply great.

The great thing was doing research in the area of Kolmogorov Complexity. Even though the applications are still in their infancy, Kolmogorov's last legacy is amazingly beautiful, linking together well known concepts of Information Theory with old problems that arise when modelling the world using statistical methods. And it all boils down to enumerating branches in binary trees. I thank Steve Maybank for, among other things, suggesting the use of Kolmogorov Complexity theory in my research.

My life and work was enriched by the company of people like Julio Aparicio, Raquel Álvarez, Brinda Ramasawmy, Hamdan Dammag, and Izumi Okata with whom I have shared a very finite amount of time.

Finite memory. Having finite memory is good not only because bad memories can be forgotten, but also because the ones that remain are good souvenirs, and there are many people whom I remember fondly. I thank Peter Sturm and Tom Grove for many interesting and useful discussions about Computer Vision. I also thank José María Martínez for taking the time to share with me his wonderful explanations of how things work in a world of thresholds and signal-to-noise ratios, and Manu Satpathy for preaching and practising optimism in research.

I thank Maud Poissonier and Ana Sanz for selflessly taking time to read parts of the draft and give me feedback during the last stages of the writing up, and doing so in such a delightful way.

I was reminded by the "No one forgets a good teacher" campaign by www.canteach.gov.uk to be grateful to Felicidad Paramio and Javier Finat who spend time and effort opening the eyes of their students, often in a hostile environment.

I am grateful to those who provided me with economic means. Although part of this research has been sponsored by DRA Malvern, none of the results are confidential. I have also received a couple of cheques from Don Knuth.

I thank James Ferryman, the technical support team and the administrative and academic staff at cs.rdg.ac.uk. My life would have been much harder had they not done a good job.

For three years Leonora Lee has been a great influence and support. For similar reasons, in different contexts, I am also grateful to Zayyana Al-Amri and Daniel Rodriguez-García. Some of my housemates had to cope with more downs than ups. Fran Claxton even tried to motivate my research with good food. I am glad she did not poison me when I gave up in my attempts to prove the Conjecture in page 29.

I thank George Ghinea for being there and believe in me, and for pushing ahead at the most difficult times. I am especially grateful to `nts` for giving us good things, I shall always have an open account for him in my computer, with unbounded quota.

Eugenia and my family shared hopes with me, and provided understanding. They have been crucial in helping me keep my sanity, all along the way, even from the distance. Most of this work would not have been possible without Eugenia's seemingly infinite support.

Now, if you find it sad to remember that time has an end, then follow my advice and do not think of it as *finite*, but think of it as *bounded*. Having bounded memory, but not finite, is what distinguishes Turing machines from finite state automata, as a consequence, it is what permits checking that parentheses are well paired, and, perhaps, it was what allowed us to jump between tree branches a few million years ago. Having that in mind, I invite you to put your feet on the ground, and read on.

October 8, 2003

Publications

[FM98] R. Fraile and S. J. Maybank. Vehicle trajectory approximation and classification. In Paul H. Lewis and Mark S. Nixon, editors, *British Machine Vision Conference*, 1998.

[FM99a] Roberto Fraile and Stephen J. Maybank. Building 3D models of vehicles for computer vision. In Dionysius P. Huijsmans and Arnold W. M. Smeulders, editors, *VISUAL'99: Visual Information and Information Systems Conference Proceedings*, number 1614 in Lecture Notes in Computer Science, pages 697–702. Springer, 1999.

[FM99b] Roberto Fraile and Stephen J. Maybank. Model matching using all grey levels. Technical report, Department of Computer Science, The University of Reading, 1999.

[FM00a] Roberto Fraile and Steve Maybank. Image compression for trajectory refinement. In James Ferryman, editor, *Proceedings of PETS2000*, pages 46–49, 2000.

[FM00b] Roberto Fraile and Steve Maybank. Selection of rigid models for visual surveillance. In James Ferryman and Anthony Worrall, editors, *Proceedings SIRS2000*, pages 289–294, 2000.

[FM01] Roberto Fraile and Steve Maybank. Comparing probabilistic and geometric models on lidar data. In Michelle A. Hofton, editor, *Workshop on Land Surface Mapping and Characterization Using Laser Altimetry*, pages 67–70. ISPRS, October 2001.

[MF01] S. J. Maybank and R. Fraile. Minimum description length method for facet matching. In Jun Shen, P. S. P. Wang, and Tianxu Zhang, editors, *Multispectral Image Processing and Pattern Recognition*, number 44 in Machine Perception and Artificial Intelligence, pages 61–70. World Scientific Publishing, 2001. Also published as special issue of the International Journal of Pattern Recognition and Artificial Intelligence.

[RMFB98] Paolo Remagnino, Stephen J. Maybank, Roberto Fraile, and Keith Baker. *Advanced Video-based Surveillance Systems*, chapter Automatic Visual Surveillance of Vehicles and People, pages 95–105. Kluwer Academic, 1998.

Contents

Chapter 1

Introduction

1.1 Computer Vision

Images are available from a variety of sources, many of them in digital form. Some images have a purely informative purpose. This is the case of the images from surveillance cameras. An example is shown in Figure 1.1.

Figure 1.1: A view from a surveillance camera (Courtesy of BBC London `www.bbc.co.uk /london` and Transport for London)

Digital images, considered as arrays of numbers, can convey a large amount of information captured with minimal effort. For example the image of Figure 1.1 shows the presence and position of vehicles on the road and, if a sequence of images is available, it is possible to visually estimate the motions of the vehicles.

Image understanding is the activity of extracting information from images. For example, faced with an image or a sequence of images, the Human Vision System is able to answer questions such as:

- Is object X in the image?
- What is the position of object X in the image?
- What is the motion of object X?

In the particular case of images from traffic surveillance cameras, a human operator is able to answer more specific questions about an image sequence:

- Is there a vehicle in lane n?

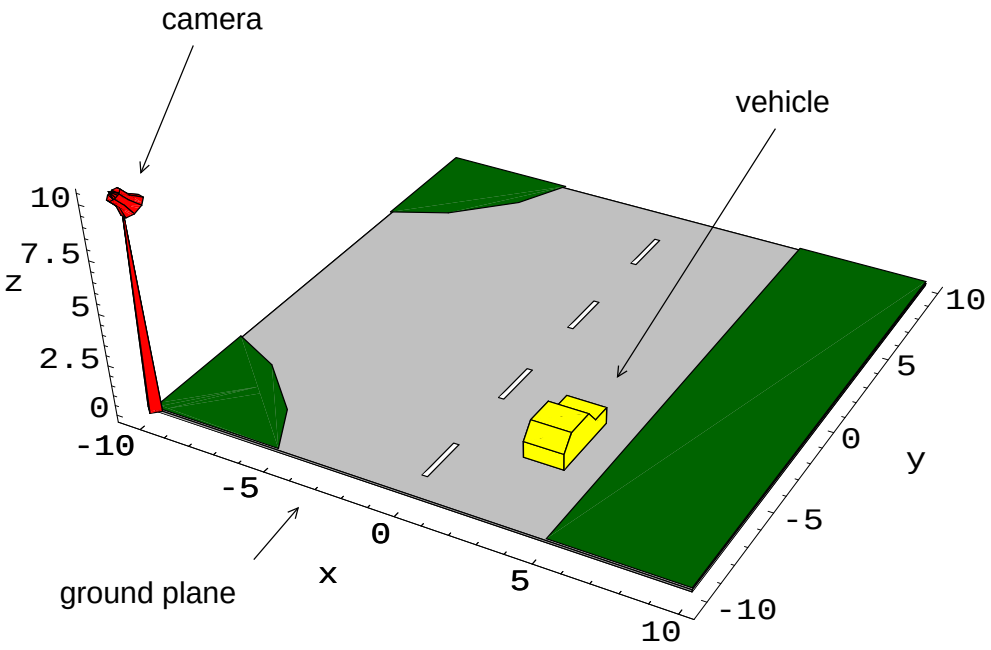


Figure 1.2: Vehicle surveillance from a fixed, calibrated camera, and a facet model of the shape of the vehicle.

- Has a vehicle followed the path or *trajectory* t?
- Has an accident occurred?
- Is there a traffic jam?

Image Understanding can be described formally by a function V that maps elements from the set of *data*, $\mathcal{D}$, which can be images or image sequences, into the set of *models*, $\mathcal{M}$, which are explanations of the data or answers to the questions above.

$$V : \mathcal{D} \longrightarrow \mathcal{M}$$

We will refer to such a function as a *Vision Function*.

Computer Vision is a research area motivated by the need for automatic methods of image understanding that can achieve performance levels similar to those of the Human Vision System. A Vision Function V is identified with a computer program that implements it. The purpose of this research is to study how to define and implement instances of Vision Functions.

1.2 Model-based vision

A fundamental question in Image Understanding is the quality and quantity of prior knowledge. A human observer often has far more prior information than an automatic vision system. For example, a security guard watching a surveillance image sequence

has prior knowledge of the shape of a double-decker bus, a car, the marks on the ground plane, etc.

Does the human observer have a detailed "model" of the shape of vehicles in mind, and uses it to recognise moving vehicles? Or is it only necessary to have a rough "model" of objects in motion? Let us consider the case of understanding images of vehicles in motion:

- A detailed *vehicle* model would be a rigid object with constant shape which moves in a highly predictable trajectory on the road.
- A less detailed and more general model would include changes in shape and a wider range of motion trajectories.

This prior knowledge is represented by *models*. For example, images taken from a typical traffic surveillance camera, as shown in Figure 1.2, can be modelled using prior knowledge in the form of:

- a *shape* model describes the shape of a vehicle,
- a *trajectory* model describes the motion of the vehicle on the ground plane and
- an *error* model describes the pixel-by-pixel difference between the projection of the shape on the image plane and the actual pixel values.

The *complexity* of a model is the quantity of prior knowledge it embodies (not to be confused with the computational cost of evaluating V). A choice in the amount of prior knowledge for a Vision Function

$$V : \mathcal{D} \longrightarrow \mathcal{M}$$

implies a choice of the complexity of the models in $\mathcal{M}$.

1.3 Computer programs as models

A formal definition of the term *model* is required in order to give meaning to the notion of model complexity, which is studied throughout this thesis. We also need to formalise how a model is used to describe data, and how to assign a cost or measure of the fit of a model to the data.

Let us define *model* and *description*. Assume that the data is a binary string d. Let M be a "computer program" or Turing machine[Tur36, Gar90] (not necessarily a Universal Turing machine). The Turing machine M can be considered as a partial function defined on a subset of the set of binary strings. For some choice of Turing machine M, it is possible that M can output the data d given appropriate input e. Let us denote this fact as

$$d = M(e) \tag{1.1}$$

In this case we will say that the Turing machine M is a *model* of the data d, and e as a *description* of the data d under M. The description e might not be unique, or, for some choice of Turing machine M, might not exist.

Turing machines are useful as models for two reasons:

- They include a wide range of models as particular cases
  - Vehicle shapes and trajectories can be modelled as computer programs that generate them given their parameters.
  - Probability distributions can be modelled by data compression algorithms.
- There is a prior probability in the space of Turing machines that can be used as a prior probability for statistical inference. This fact is a surprisingly useful application of the assumption that data has been produced by computation.

1.4 Model Selection

Define *cost* of a model M given the data d as the length of the shortest description e of d under M. This cost is a measure of the fit of the model to the data in the following sense:

- If there is a short description e for the data d under M, then we can intuitively say that M provides a good fit, because the model does not require many additional bits to produce the data.
- If, otherwise, all possible descriptions e of the data are long, then M needs many additional bits to generate the data, and therefore provides a bad fit.

From this point of view it seems reasonable to use the length, in bits, of the shortest description e as cost or measure of how well the model M fits the data $d = M(e)$. Given two models for d, M_1 and M_2, and their respective shortest descriptions e_1 and e_2,

$$d = M_1(e_1) = M_2(e_2) \tag{1.2}$$

the preferred model M_i is the one for which e_i is the shortest. In fact, Maximum Likelihood method of Model Selection is a particular case of this approach [LV97].

$$\text{cost}[M] = \underbrace{\text{length}(e)}_{\text{error}}$$

But so far we have not taken into account the complexity of each model. It might be the case that model M_1 is much more complex than model M_2, and, by most standards, model M_1 is much more unlikely than model M_2. For example, it could be the case that M_1 is a vehicle moving with constant velocity along the road, and M_2 is an elephant moving at a slow pace across the street. Under the assumption that data corresponds to a traffic surveillance camera, the model M_1, is more likely, and should be preferred against M_2 even in cases when e_2, the fit of the elephant model to the data, is shorter than e_1, the fit of the vehicle model to the data.

A Bayesian approach to statistical inference makes assumptions about the prior probability of the models, and selects a model according to Bayes' formula. In this case, the prior knowledge about the complexity of a model can be translated as a prior probability: A more complex model has a lower probability. The problem is that this prior probability might be difficult to estimate.

It has been argued [Edw92] that, in general, it makes no sense to decide about a prior probability in the space of models. (That is, no one can decide about the probability of an elephant crossing the street). This is the *frequentist* approach to probabilistic inference, embodied in the principle of Maximum Likelihood, under which all information provided by a model is given by its fit to the data. This approach fails when we need to take the model complexity into account. A dramatic case is the example above, in which a model for the vehicle and the elephant are compared. A more mundane failure occurs when training data is given to fit the model, and the model preferred by the Model Selection method not only describes the (predictable) source of data, but also the (unpredictable) noise [Die95]. As a consequence, a model of higher complexity than necessary is used, and the ability to predict future data is diminished.

Providing a solution to this type of problems of Model Selection requires, in essence, assigning a *cost* to each model, according to its complexity, and add this cost to the measure of the error between the model and the data.

$$\text{cost}[M] = \underbrace{\text{complexity}[M]}_{\text{cost of the model}} + \underbrace{\text{length}(e)}_{\text{error}}$$

In this way, a cost is obtained for each model, that avoids fitting unnecessarily complex models to the data.

Now the problem is which cost to assign to the model complexity. For this purpose, we will use the fact that models are Turing machines, and therefore can be represented as binary strings themselves.

1.5 The Minimum Description Length method (MDL)

Our initial definition of cost of a model M_i was the *description length* of the data d under M_i, the length of e_i in Equation (1.2). But we need to take model complexity into account. A weight needs to be assigned to each model according to its complexity.

The model M is a computer program that runs in some language interpreter U. The program M with input e is the same as the language interpreter U with input me, where me is the concatenation of the strings m and e. We can compare between models by choosing a language interpreter U that can interpret all models.

Define the cost of model M as the *length* of the shortest *description* of the data d under language interpreter U, with the condition that the description includes the model M. This cost takes the complexity of the model into account.

A description of the data d in the language U is the binary string me.

$$d = M(e) = U(me)$$

And the length of the description of d in U is

$$\underbrace{\text{cost}[M]}_{\text{description length}} = \underbrace{\text{length}(m)}_{\text{cost of the model}} + \underbrace{\text{length}(e)}_{\text{error}} \tag{1.3}$$

The cost of the model M, is the length of the binary string m: shorter binary strings correspond to simpler models.

The MDL method of Model Selection chooses the model that minimises the sum of:

- The length of a description m of the model M, in U.
- The length of a description e of the data d, in M.

Or, in other words, choosing the model M that minimises

- The length of a description of d in U, that begins with a description of M.

There is an alternative interpretation of the MDL method in terms of *data compression*. Consider that we want to transmit data d between two points, and we have a choice of models $M_1, \ldots, M_r$. Assume a protocol for the transmission of the models, and that each model is a protocol for the transmission of the data. The MDL method states that one should choose the model that allows the transmission of the data in the least number of bits.

- To *compress* a binary string d using a model M, is the problem of finding the shortest string e such that $d = M(e)$.
- To *decompress* the binary string e, using the model M, is simply the evaluation of $M(e)$ to obtain d.

1.6 Applying the MDL method to Vision problems

Unfortunately the MDL method is not computable in general. There is no computer program that, given data d and an arbitrary family of models $M_1, \ldots, M_r$, can return the shortest description of the data that uses one of the models. This is a consequence of the Halting problem.

The solution to the non-computability problem is to restrict the family of models, and to consider only approximations of the shortest descriptions strings m and e.

The MDL method is usually stated in the form of Equation (1.3), or an approximation of it, barring notational changes [Ris83, LV92, LV97]. This equation is valid so long as a single model is used to compress the data.

In practice, data and models are compound data structures, which need to be decomposed into individual elements before compression. That is, both models and data have *structure*. This is the case of Chapters 3, 4 and 5 of this thesis. We will see in Chapter 2 a generalisation of Equation (1.3).

The application of the MDL method to define a Vision Function for vehicle surveillance is conceptually straightforward. The Vision Function uses

- a compression algorithm that takes all data available (namely, each image of an image sequence) and
- as models, the shapes and trajectories of vehicles.

All prior knowledge is contained in the compression algorithm. The cost of the shape and trajectory models is the size of the compressed data, using those models. That is, the cost is the minimal description length available. The Vision Function selects the models that yield better compression.

1.7 Contributions and Structure of the thesis

The contributions of this thesis can be summarised in two:

- Derivation of the MDL method from first principles.
- A framework for the use of the MDL method in solving Computer Vision problems.

Chapter 2 Presents the MDL method avoiding the usual assumptions found in the literature. Combinatorial and probabilistic interpretations of the MDL method are provided.

Chapter 3 A simple implementation of the MDL method is used to define a cost function for vehicle tracking. Shape and trajectory models, together with a 3D graphics renderer, are used to create a synthetic version of the data. An off-the-self data compression program [Wel84, lGA02] is used to compress the data using the models.

Chapter 4 A simple yet powerful framework for developing computable approximations to the MDL method is presented in the form of a reusable software package written in the computer language `Mathematica`.

Chapter 5 A dynamics-based trajectory model is developed, and a compression algorithm based on it, using the software package of Chapter 4. Experiments test the usefulness of the trajectory model.

Chapter 6 A method for vehicle trajectory analysis is presented. A low curvature approximation of the trajectory is found by minimising a functional, and a Hidden Markov Model is used to model the driver's actions.

Figure 1.3 shows the areas of research covered by each chapter. Chapter 2 deals with the foundations of the MDL method. Chapter 3 uses preliminary computable approximations to the MDL method and uses them to define Vision Functions that are tested with experimental data. Chapter 4 covers the area between the MDL method and software that can be used to solve real-world problems. Chapters 5 and 6 contain experimental results.

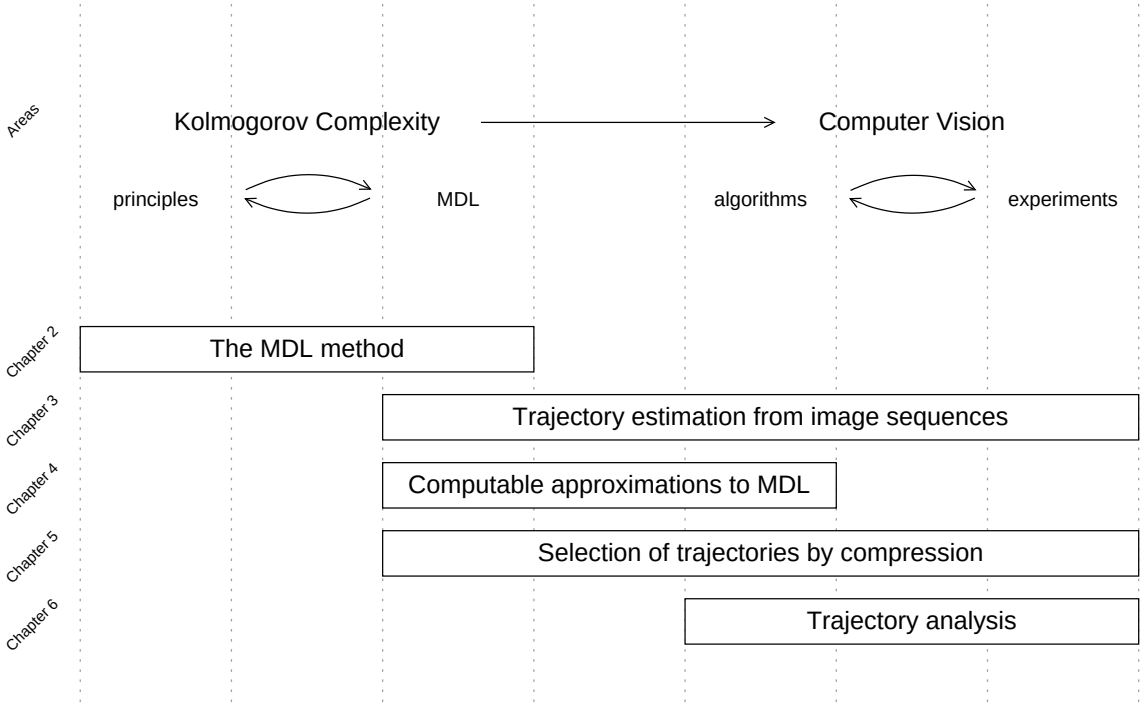


Figure 1.3: Structure of this thesis

Chapter 2

The Minimum Description Length method

2.1 Introduction

A *model*, in the context of Computer Vision, is an abstraction of a real world object which represents some information of interest. Such information can be about the shape of the object, or some statistical property concerning the object.

For solving a Computer Vision problem, in many cases, we are required to choose a model which captures best the relevant properties of data; for example, in vehicle tracking, a shape model retains the right information about a shape and ignores all other information about the vehicle. It may be required that the model be parametric. In this case, the number and the type of parameters determine the complexity of the model, and *Model Selection* is the problem of choosing a parametric model and the number and the type of parameters. A parametric model denotes a family of models. Once the parameters are instantiated with their values, we obtain a *Model instance*. We would like to consider the choice of number of parameters, their types and their values, all at the same time. Henceforth, by a model we will mean a model instance, and by Model Selection the choice of the number, the type and also the *value* of the parameters.

A central problem in Model Selection is *overfitting*, i.e. the tendency to choose a model with a large number of parameters. A common instance of overfitting happens when models are fit to a training dataset. Since it is common that model families with a large number of parameters fit the data better, the result is a model that also describes the noise and therefore is overly complex [Die95]. One solution to the problem of overfitting is the *Minimum Description Length* (MDL) method [Ris83, LV97].

The MDL method assumes that the model itself is a computer program. This computer program can be viewed as a Turing Machine. Any data can be represented by a finite binary string. A description of data d is a Turing machine M that outputs d given some input e. This Turing machine itself can be represented by a binary string m and hence it can have a definite length. The MDL method is about choosing the model M whose description of data, (m, e) is shorter.

In order to elucidate the notion of MDL, let us consider the example of transmitting a picture over a communication channel. We are given the picture and a range of formats to choose from. The formats are GIF, JPEG or PostScript which play the role

of models. The corresponding Turing machines are the decoders for those formats. A description of the picture consists of the data format M (i.e. the model) together with the encoding of the picture in that format, e. The MDL method chooses the model that allows the transmission of the picture (i.e. the description consisting of (m, e)) in fewer bits.

The main contributions of this chapter are:

- A new approach to the MDL method, which does not rely on probability distributions as existing approaches do.
- An ordering relationship is defined over Model Selection methods. It is conjectured that the MDL method is a minimal Model Selection method in relation to this ordering. This approach avoids the axiom that "shorter is better".

The MDL method of Model Selection has combinatorial and probabilistic interpretations, which are discussed in Sections 2.4 and 2.5. Finally, we review two alternative methods of Model Selection, Akaike's AIC and Schwarz's BIC in Section 2.6.

2.2 Modelling the output of computations

In order to define the MDL method of Model Selection, we need to define models and data in such way that they are comparable. We will use models and data which can be represented as binary strings, and use the length of the binary strings as an estimate of its complexity. For this purpose, models will be identified with Turing machines, and data will be the output of the Turing machines. The existence of Universal Turing machines means that models can also be represented as binary strings.

Turing machines are a formalisation of the notion of computation. Each binary string can be represented by an integer. There exist Turing machines U that, given the index of a Turing machine T, *simulate* T in the sense that U maps the same input to the same output as T does. Universal Turing machines are a formalisation of this idea.

The Turing machines that we will use in this chapter are functions defined between sets of binary strings, with the following properties:

- Their input (argument) and output (value) are binary strings
- Their input is a *prefix-free* set. This is a set of binary strings in which no element is a prefix of another element.
- As a consequence, the domain of the Turing machines is a binary tree.

Using Turing machines whose domain is a prefix-free set has several properties that are important for the foundations of the Minimum Description Length method. Those properties are discussed in [LV97, Chapter 3]. Prefix-free sets correspond to binary trees, which provide an intuitive way to understand the properties of the domains of our Turing machines. We will use this fact thorough the current chapter. Besides, a probabilistic distribution can be defined that a Turing machine T outputs a string d, thanks to the fact that the domain of T is a prefix-free set.

2.2.1 Computation

Definition 2.1 TURING MACHINE A Turing machine is a triplet $(\mathcal{S}, \Sigma, T)$ where

States $\mathcal{S}$ is a finite set of states, one of them chosen to be the initial state, another one chosen to be the final state.

Alphabet $\mathcal{A}$ is a finite alphabet.

Transition T is a partial function

$$T : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{A} \times \mathcal{M}$$

where $\mathcal{M} = \{\curvearrowleft, \downarrow, \curvearrowright\}$.

This should be interpreted as follows: the machine consists of a head that can move step by step on a tape of cells. At each step, the head reads a symbol from the tape, modifies its internal state, and executes a movement. The machine's internal state is an element of $\mathcal{S}$. The symbol read from the tape is an element of $\mathcal{A}$. At each step, according to the transition function T, the machine changes state, prints a new symbol on the tape, and its head either moves to the left, moves to the right or remains where it is on the tape. The machine halts when it reaches the final state. (This definition is adapted from [Pra76, page 476]).

Example 2.2 SUCCESSOR A Turing machine that computes the *successor* function is illustrated in Figure 2.1. The figure shows the table of transition rules and an example of execution. The input is a natural number n encoded on the tape as n ones followed by a blank. In this example $\mathcal{A} = \{\text{blank}, 1\}$, M is the transition table represented in the figure, and $\mathcal{S}$ is the set of states S (for Start, the initial state), F (for Forward), B (for Backward), and E (for End, the final state). The example shows the computation of the successor of 2. □

The Turing machine is said to accept a string of symbols from the alphabet $\mathcal{A}$, $x \in \mathcal{A}^*$, when, being x the content of the input tape at the beginning of the execution, the machine eventually reaches the final state. The output for x is the contents $y \in \mathcal{A}^*$ of the tape when the machine halts. It is then said that the machine *computes* y given input x. A machine defines a partial function in the space of input tapes. (A machine that halts with input x can only read a finite number of symbols of x, therefore we can assume x is a finite string).

Definition 2.3 COMPUTABLE FUNCTION A partial function F defined between sets of binary strings, is said to be computable when there is a Turing machine that halts only for input x which are elements of the domain of F, and the output of the machine for x is $F(x)$. We will denote with the same letter, F, both the machine and the computable function.

We will assume that the functions considered are partial functions.
The *domain* and *range* of a computable function are defined as follows:

Domain The set of binary strings for which F is defined, denoted by $\mathcal{H}_F$. This is the set of inputs for which the corresponding Turing machine F *halts*.

Range The set of all possible outputs of the Turing machine F.

A machine *cycle* is each of the steps in which the execution of the Turing machine is divided. A computation takes a finite number of machine cycles before it halts. We will use this fact when enumerating the domain of Turing machines in Section 2.4.

Turing machines represent the notion of computation in its most general sense [Tur36]. Trivially, everything we expect to run on a computer needs to be computable. Moreover, restricting ourselves to *modelling* computable functions still includes common probabilistic models [Grü98, page 33]. The restriction from general functions to functions which are computable by a Turing machine is very useful, because we can represent computable functions using finite binary strings. We will use this property below, when a universal Turing machine is defined and chosen.

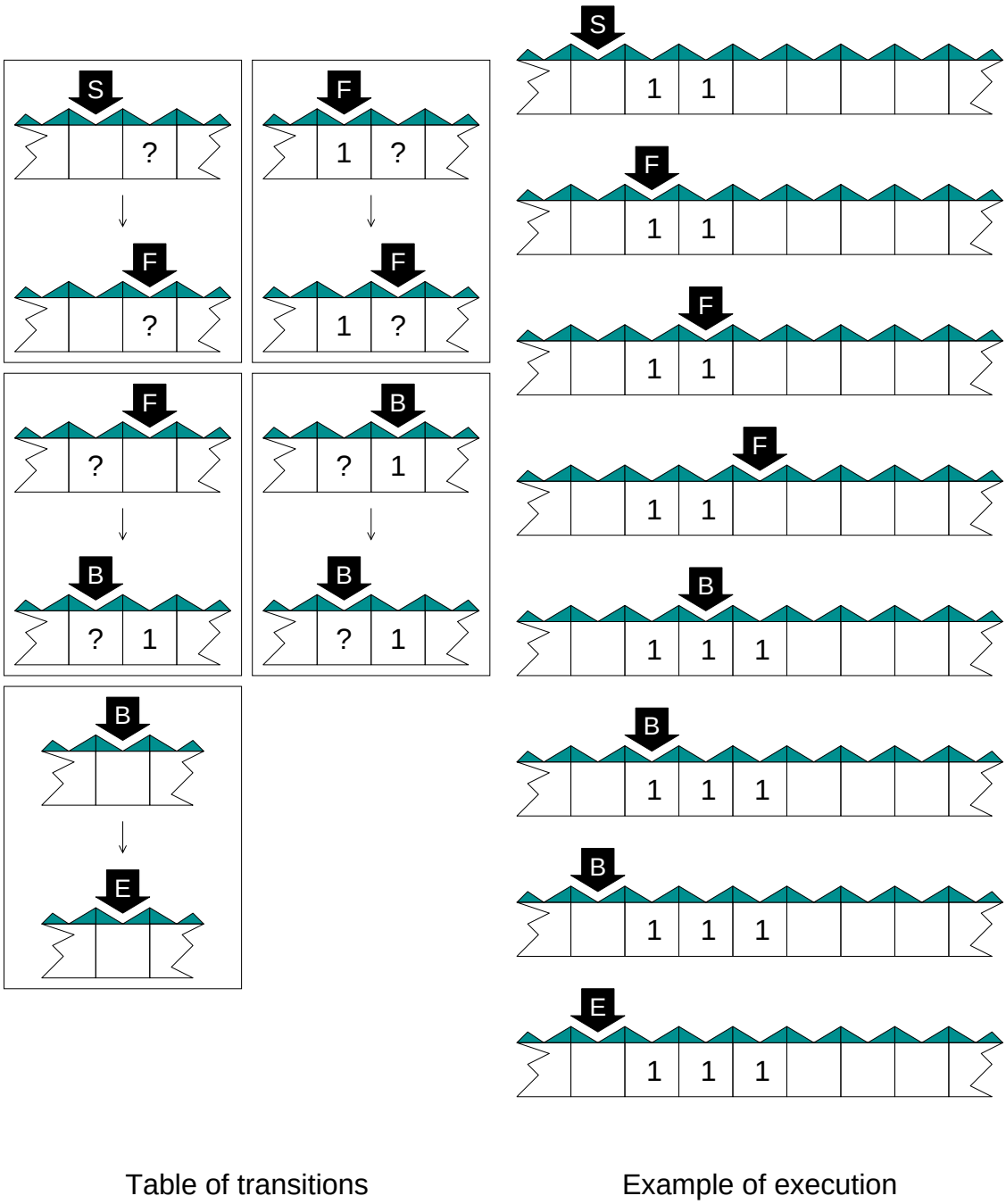


Figure 2.1: Turing machine that computes the successor function for natural numbers. Input and output are encoded in unary form: $n \mapsto 1 \overset{n}{\cdots} 1$ followed by a blank. (Modified from [Mae00])

2.2.2 Prefix-free sets

According to Definition 2.1, a Turing machine can read its input in any direction, alternating movements to the left and right if necessary. But we will consider only Turing machines that read their input from left to right. The example of Figure 2.1 read its input from left to right. For example, computer files are most frequently read in one direction (which we can assume to be from left to right, as in English text), and computer languages are designed to be read from in one direction too, at least at a first approach. The key advantage of writing strings only in one direction is that it simplifies storing in the string itself where the string ends. This takes the form of a length, which is the case of a computer file in a typical filesystem, or a character terminator, as the character "`\0`" for strings of characters. The critical property is that the input of the machine is *uniquely decodable*, that is, the string itself is all that is required to read the input with no ambiguity (and not, for example, an initial state of the machine and position of the head).

Notation. Let $\mathcal{B} = \{0, 1\}$. Let $\mathcal{B}^*$ be the set of finite binary strings. Given a string x, formed by the bits $x_1, x_2, \ldots, x_n$ we will say $x = x_1 x_2 \cdots x_n$. In general, let xy denote the concatenation of two finite strings x and y. Let us denote finite binary strings as just *strings*. Capital letters such as H denote functions. Lowercase letters such as h, x and p denote strings. Calligraphic letters such as $\mathcal{D}$ and $\mathcal{S}$ denote sets.

It is clear that a uniquely decodable set of binary strings which is read from left to right is a prefix-free set according to the following definition:

Definition 2.4 PREFIX-FREE SET A prefix-free set of strings $\mathcal{S} \in \mathcal{B}^*$ is a set such that no element z of $\mathcal{S}$ is a prefix of any other element of $\mathcal{S}$ distinct from z. In other words, for any strings x and y,

$$x, xy \in \mathcal{S} \implies x = xy$$

An example of a prefix-free set, with decimal digits instead of bits, is the set of valid phone numbers that can be dialled from any particular telephone.

A *prefix-free function* is a function whose domain is a prefix-free set. All computable functions we are going to consider are prefix-free.

2.2.3 A reference prefix-free universal Turing machine

We will need to represent Turing machines themselves as binary strings. For this purpose, we will define a special type of Turing machine that can "simulate" other Turing machines, which can be identified as binary strings. This special Turing machine is not unique, but its domain is a binary tree which is unique in a sense.

Let us consider Turing machines whose domains are prefix-free sets, and we shall call them prefix-free Turing machines. This corresponds to a modification of the original definition of Turing machines, in which the machine works with three tapes:

- an input tape in which the head can only move to the right,
- a working tape in which the head can move either to the right or to the left,
- an output tape in which the head can only move to the right.

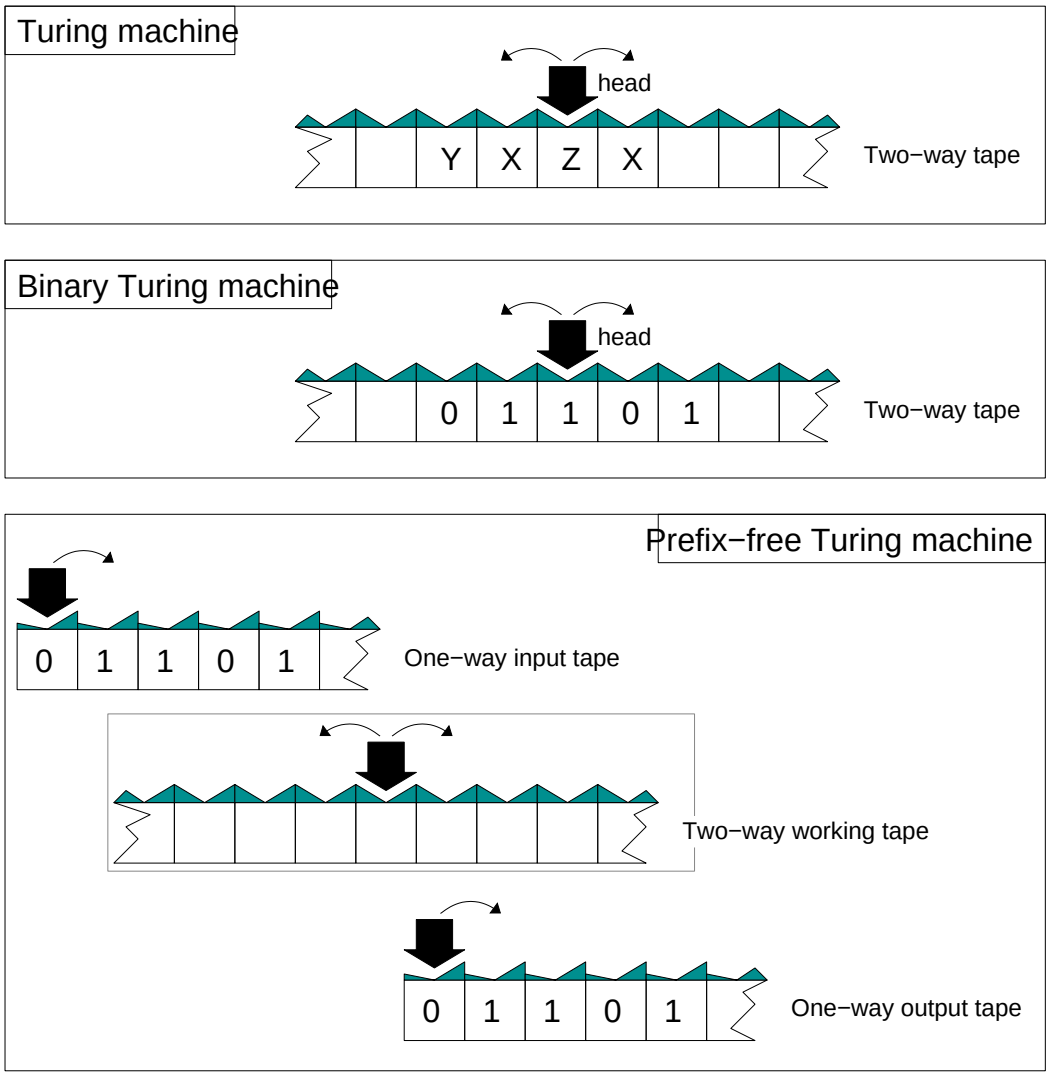


Figure 2.2: A Turing machine is usually defined with a head that can move in both directions on a tape that contains symbols from an alphabet (top). The alphabet can be assumed to be binary, without loss of generality (centre). A prefix-free Turing machine contains one-way input and output tapes, and a two-way working tape (bottom). The domain is a prefix-free set.

This idea is illustrated in Figure 2.2.

The input of the Turing machines are binary strings. We can also represent the Turing machines themselves as binary strings:

Proposition 2.5 UNIVERSAL TURING MACHINE There exists a Turing machine U such that, for each Turing machine T, there is a string t with the property

$$U(tx) = T(x)$$

for all x in the domain of T [Tur36, LV97].

This means that each Turing machine can be identified by a binary string, or a *program* for it:

Definition 2.6 PROGRAM FOR A TURING MACHINE Given a Turing machine T and a Universal Turing machine U, the program for T under U is the string t such that $U(tx) = T(x)$ for all x in the domain of T.

The domain is not computable. Let $\mathcal{H}_U$ be the domain of the Turing machine U. The characteristic function H of the set $\mathcal{H}_U$ is not computable. This is known as the *Halting Problem*. Several unsolvability results from computer science and other areas have been rephrased in terms of the unsolvability of the Halting Problem [Mel76, page 232]. The Halting Problem ultimately leads to the non-computability of the MDL method of Model Selection.

2.2.4 Data, models and descriptions

In order to model data, we need a definition of model and description. Essentially, a description of data d is Turing machine, that outputs d reading no additional input. A model is a Turing machine that *can* output d if appropriate input is provided. By using an appropriate Universal Turing machines, models can be identified with prefix of descriptions.

Models

A formal definition of *model* is needed before defining Model Selection method. The term "model" has very different meanings depending on the context, even within the areas covered by this thesis. A formal definition will solve this ambiguity:

Definition 2.7 MODEL A model M of data d is a Turing machine for which there is a binary string e such that

$$d = M(e)$$

That is, data d is in the range of the Turing machine M.

Common usage of the term "model" in Computer Vision is covered by Definition 2.7. Two main classes of models are *geometric* models and *probabilistic* models.

Geometric models. The geometric model of a 3D object d is a Turing machine M that can produce the object d given parameters e. For example, a cuboid floating in the space requires nine parameters to represents its dimensions, position and orientation. A program M that takes those nine parameters and produces an image of the cuboid is a model of the shape of the cuboid.

Probabilistic models. Most useful random variables used to model the data can be generated transforming fair random coin flips using a simple algorithm. And this algorithm corresponds to a binary tree whose leaves (or, equivalently, paths) are labelled with the elements of the event space, or "symbols". Given a binary string produced by fair coin flips, we divide the string in paths of the tree, and replace each substring by the corresponding symbol. This fact is discussed, for example, in [CT91, Section 5.21].

Therefore models can typically be represented either by parametric shapes or by binary trees that correspond to probability distributions. In practice, we will use a combination of both.

Description length

The notion of *description* and *description length* will be derived from Definition 2.7 of model. But, first of all, let us set a common ground in which all models can be compared. Since models are computer programs (Turing machines), let us agree on a language in which those computer programs are written (Universal Turing machine).

Let U be a prefix Universal Turing machine as defined in Section 2.2 The machine U can simulate any other Turing machine T by means of a string t such that

$$T(x) = U(tx) \tag{2.1}$$

for all x in the domain of T, as we have seen in Proposition 2.5.

This leads to the following definition:

Definition 2.8 DESCRIPTION A description or program for a string d under a Turing machine T is a string p such that

$$T(p) = d$$

We wanted to assign a cost to the model, so that simpler models have a lower cost, and complex models have a higher cost. The key is to consider the length of the string me.

$$\text{length}(me) = \text{length}(m) + \text{length}(e)$$

This is the *description length* of the data d using the model m.

2.3 The MDL method

Let us first state the MDL method of Model Selection in informal terms. Given a family of models $M_1, M_2, \ldots$, and data d, the model preferred by the MDL method is the one that allows coding the data in fewer bits, when the data is coded as

- a binary string that identifies the model M ("description length of the model M"), followed by

- a binary string that identifies the data d given the model M ("description length of the data d given the model M").

(See [LV97])

Note that this definition of the MDL method has two distinctive characteristics, compared to what is usually considered as a Model Selection method:

1. It is independent of the problem to solve; no assumption has been made about the models or the data.

2. It depends on the way the programs are implemented. This breaks the rule of clearly separating the definition of a method or algorithm from its implementation as a computer program.

In this section we will see how the MDL method has both characteristics at the cost of not being computable:

Data source The main assumption is that data has been produced by a Turing machine.

Universality Although its definition relies on the way models are implemented on the computer, it is independent of the choice of implementation.

First of all we need to define what we mean by compression, and what would be *optimal* compression.

Definition 2.9 COMPRESSION FUNCTION Let M be a Turing machine. A compression function C_M is a 1-1 function

$$\mathcal{B}^* \longrightarrow \mathcal{B}^*$$

with the property

$$M(C_M(b)) = b \tag{2.2}$$

for any string b in the range of M. If the description of b given by C_M is always shorter than any other description x of b,

$$\text{length}(C_M(b)) \leq \text{length}(x)$$

then C_M is an *optimal* compression function and is denoted as $\mathbf{C}_M$.

(In some cases, for simplicity, we will talk about compression functions C and $\mathbf{C}$ without explicit reference to a particular Turing machine). One way to look at the value of $C_M(d)$ is as an error between the model M and the data d. The length of the error, $\text{length}(C_M(d))$ is an upper estimate of $\text{length}(\mathbf{C}_M(d))$. In other words, when C_M is computable, it will be treated as a computable approximation of $\mathbf{C}_M$. The term *compression* is used for the function C_M (and $C = C_U$) because it carries an intention of finding short descriptions for some strings. Note that the inverse function of C_M, the *decompressor*, is M.

Now we are ready to define Kolmogorov Complexity of a binary string, which is a measure of the individual complexity of the binary string.

Definition 2.10 KOLMOGOROV COMPLEXITY The Kolmogorov Complexity of a binary string x is the length of any of the shortest descriptions of x.

$$K_U(x) = \text{length}(\mathbf{C}_U(x))$$

We will use the same idea to define complexity of a model, and complexity of data given the model.

Kolmogorov Complexity is a measure of the information content of individual binary strings. This is in contrast with Shannon's Information Theory [Sha48]. Shannon's information (or *entropy*) is defined for a set of possible messages, whereas Kolmogorov Complexity measures the information of a single string. Both notions are somehow equivalent: the Kolmogorov Complexity of a realisation x of a random variable X is approximately equal to the entropy of X [CT91].

2.3.1 Minimise description length

Let us try to formalise the MDL method using the notion of Kolmogorov Complexity. For any given data d, there exists a string $\mathbf{C}_U(d)$ with minimal length that describes d. We are given models M_1, M_2, ..., which correspond to binary strings $m_1, m_2, \ldots$, and we have to make a choice of model. One initial suggestion for the MDL method would be to estimate the Kolmogorov Complexity of the data, but *only considering those descriptions which begin with some* m_i. The model M whose string m prefixes the shortest such description would then be the preferred one.

$$m\mathbf{C}_M(d)$$

But the problem of using the length of $m\mathbf{C}_M(d)$ in order to compare between models is that we might choose a particularly long string m to represent the model M. For example, both m and um correspond to the same model M, but they differ in size by length(u) bits (where u is the program that corresponds to the Turing machine U). The same applies to uum, $uuum$ and so on.

The solution to the problem of unnecessarily long models m is to apply the function $\mathbf{C}$ also to the model, and, in this way, we will obtain a formal statement of the Minimum Description Length method.

Definition 2.11 THE MDL METHOD OF MODEL SELECTION Given a set of models $\mathcal{M} \subset \mathcal{B}^*$, and data $d \in \mathcal{B}^*$, choose the model M that minimises

$$\text{argmin}_m(\text{length}(\mathbf{C}_U(m)\mathbf{C}_m(d))) \tag{2.3}$$

where m is a program for M. This definition does not depend on which m has been chosen for M.

Formula (2.3) is usually split into

$$\text{argmin}_m(\text{length}(\mathbf{C}_U(m)) + \text{length}(\mathbf{C}_m(d)))$$

which corresponds to the intuitive definition of MDL given at the beginning of the current section.

2.3.2 Universality of the MDL method

The choice of model by the MDL method depends on the choice of Universal Turing machine U. This choice affects the length of $\mathbf{C}_U(m)$, the shortest program for a model m. Thus, the MDL method might yield different results for different Universal Turing machines. For example, it is possible that, given models m and n, and Universal Turing machines U and V, the following inequalities hold:

$$\begin{aligned} \text{length}(\mathbf{C}_U(m)) &< \text{length}(\mathbf{C}_U(n)) \\ \text{length}(\mathbf{C}_V(m)) &> \text{length}(\mathbf{C}_V(n)) \end{aligned}$$

while $\text{length}(\mathbf{C}_m(d)) = \text{length}(\mathbf{C}_n(d))$ for some data d. The implication is that the MDL method prefers to model d using m rather than n when U is the Universal Turing machine, but prefers n to m when V is the Universal Turing machine,

$$\begin{aligned} \text{length}(\mathbf{C}_U(m)\mathbf{C}_m(d)) &< \text{length}(\mathbf{C}_U(n)\mathbf{C}_n(d)) \\ \text{length}(\mathbf{C}_V(m)\mathbf{C}_m(d)) &> \text{length}(\mathbf{C}_V(n)\mathbf{C}_n(d)). \end{aligned}$$

Clearly this is not a desirable situation. Let us study how the choice of model using a particular Universal Turing machine can still be useful.

Since both U and V are Universal Turing machines, they can simulate each other. Consider v, the program of V under U. The shortest program that describes a string x under V, $\mathbf{C}_V(x)$, can be run by U if we prefix it with v. So $v\mathbf{C}_V(x)$ is a description of d under U, which means that

$$\text{length}(\mathbf{C}_U(x)) \leq \text{length}(v) + \text{length}(\mathbf{C}_V(x))$$

Note that the length of v does not depend on x, but only on V and U. Swapping U by V we obtain a similar result, albeit with a different constant, and therefore the difference of lengths of $\mathbf{C}_U(x)$ and $\mathbf{C}_V(x)$ is bounded by a constant $r > 0$.

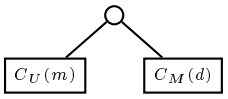
$$\mid \text{length}(C_U(x)) - \text{length}(C_V(x)) \mid < r$$

Let us now define the $n > 0$ *most preferred models* by the MDL method, given a family of models, as the preferred model m together with the $n-1$ most preferred models when m is removed from the model family. In other words, the n most preferred models are those that yield shortest description length of the data.

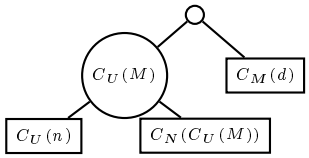
The consequence of U and V leading to minimal description lengths which are bounded by r, is that the MDL method produces consistent results across different Universal Turing machines, "modulus" a finite number r of models. This means that, given a family of models, the preferred model found using U is among the r most preferred models found using V, and r does not depend on the models or the data. In this sense the MDL method can be considered independent of the Universal Turing machine.

2.3.3 Using a fixed-point Universal Turing machine

The Minimum Description Length method is often stated in informal terms as: "Choose the model that gives the shortest description of the data" [LV92, LV97, HY01]. But



(a) A two-part description of d using model M.



(b) A multi-part description of d using models M and N.

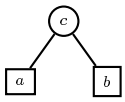
Figure 2.3: The description of data can be decomposed in many parts. The models M and N are represented by strings m and n respectively.

the Expression (2.3) in Definition 2.11 does not output d when the Universal Turing machine U is applied to it. In general, the string

$$C_U(m)C_M(d) \tag{2.4}$$

is never a description of d under the Universal Turing machine U, a fact which will be proven below. Still, the length of the string (2.4) is meaningful as the description length of d under another Universal Turing machine V.

One way to understand the role of Expression 2.4 is to find out if the expression is enough to transmit data d between two points, assuming that no additional information, other than U, is shared between the sender and the receiver. The diagrams of Figure 2.3 show how a description can be reconstructed collapsing the leaves of a tree into binary strings until only one node remains. The actual transmission are the leaves of the tree, in prefix order [Knu98]. Each node is reconstructed at the receiving end by applying U to each subnode. The transmitted data is obtained by applying U to the root node. For example, for transmitted binary strings a and b, $c = U(a)U(b)$, $d = U(c)$.



Transmitting expression (2.4) allows the receiver to reconstruct d *when the receiver knows that d is formed by two parts*: a description of the model $C_U(m)$, and a description of the data given the model $C_M(d)$. (Figure 2.3(a)). The problem arises when the model M can be modelled by another model N, as illustrated in Figure 2.3(b), and *the receiver does not know how many parts are needed to reconstruct the final data d.*

For example, the data d could be the description of a valid Turing machine under U. Should the receiver expect more data and then treat the received data as a model for further data, or should assume that the transmission has stopped? Real-world instances of this problem are dealt with in Chapter 5.

What is not a description

Before proving that Expression (2.4) is not a description of d in U, we need to look at the way Turing machines read their input.

Let T be a Turing machine, $x \in \mathcal{H}_T$, and y is any binary string. Then

$$T(xy) = (Tx)y = T(x) \tag{2.5}$$

That is, T will only read the bits up to the end of x and discard the rest. This is a consequence of the Turing machine reading their data from left to right. But when a Turing machine T is being simulated by another machine Q, and $x \in \mathcal{H}_T$, the unread bits of the input are not discarded.

$$Q(T(xy)) = Q((Tx)y) \tag{2.6}$$

which means that the unused bits of $T(xy)$, which are y, can still be used by Q.

Proposition 2.12 The string (2.4) is not, in general, a description of d in U.

Proof. Let us apply U to Expression (2.4)

$$\begin{aligned} U(C_U(m)C_M(d)) &= (U(C_U(m)))C_M(d) \quad \text{by (2.5)} \\ &= U(C_U(m)) \quad \text{by (2.2)} \\ &= m \end{aligned}$$

and, in general, $m \neq d$. □

A fixed-point Universal Turing machine

Now we will find a Universal Turing machine V so that Expression (2.4) is a description of d under V.

A *fixed-point* of a Turing machine T is a string x such that $T(x) = x$. A consequence is that, by iterating T, the output does not change. Define the Universal Turing machine V to compute the fixed-points of U as

$$\begin{aligned} V(x) &= V(U(x)) \text{ if } U(x) \neq x \quad (2.7) \\ &= x \text{ if } U(x) = x \quad (2.8) \end{aligned}$$

V is defined in the domain of U.

Expression (2.4) is a description of d under V, as shown in Proposition 2.13.

Proposition 2.13 Expression (2.4) is a description of d under V

$$V(C_U(m)C_M(d)) = d$$

for any string d in the range of V.

Proof.

$$\begin{aligned} V(C_U(m)C_M(d)) &= V(U(C_U(m)C_M(d))) \quad \text{by (2.7)} \\ &= V((UC_U(m))C_M(d)) \quad \text{by (2.6)} \\ &= V(mC_M(d)) \quad \text{by (2.6)} \\ &= V(U(mC_M(d))) \quad \text{by (2.7)} \\ &= V(M(C_M(d))) \quad \text{by (2.1)} \\ &= V(d) \quad \text{by (2.2)} \\ &= d \quad \text{by (2.8)} \end{aligned}$$

□

Data for V must be fixed-point for U

Note that the range of the Universal Turing machine V is not $\mathcal{B}^*$, but the subset of it formed by the fixed points of U.

$$U(d) = d$$

(which is not empty because there are programs that print themselves out). This constraint is necessary so that we can distinguish fixed points of V (data) from everything else (executable programs).

This constraint on the strings that can be data is not very restrictive, although it implies that both data and programs form a prefix-free set. Most data structures in modern computer languages are elements of prefix-free sets.

2.3.4 Computable approximations

In general the MDL method is not computable because the function $\mathbf{C}_U$ is not computable. This fact is ultimately a consequence of the Halting problem, the inability to compute whether a Turing machine halts or not.

As a consequence, it is not possible in general to know how close the difference between $\mathbf{C}$ and C is,

$$\text{length}(\mathbf{C}_T(x)) - \text{length}(C_T(x))$$

because it would be equivalent to knowing $\mathbf{C}$ for Turing machine T and binary string x.

In practice we will be looking for computable approximations to $\mathbf{C}$. Most of the reasoning used in this chapter only requires C to be a *compression* function, not an *optimal compression* function $\mathbf{C}$. The consequence is that most results hold while C is not the optimal compressor, but an approximation to $\mathbf{C}$. In Chapter 4 we will develop computable approximations to $\mathbf{C}$.

2.4 Combinatorial interpretation of MDL

Once the MDL method has been formalised, it needs to be interpreted in the context of other Model Selection methods. In this section Model Selection methods are defined as enumerations of the leaves of a binary tree. It is shown that the MDL method is minimal, in a sense, among those enumerations.

2.4.1 Prefix-free subsets of $\mathcal{B}^*$ are binary trees

An important property of prefix-free sets of binary strings is that they can be visualised as the leaves of a binary tree. A particular example is $\mathcal{H}_U$, the domain of a prefix-free Universal Turing machine U.

Definition 2.14 BINARY TREE A binary tree is a finite set of nodes that is either empty or is formed by two disjoint binary trees, and a root. The disjoint binary trees L and R are called the *left* and *right* subtrees of the root [Knu97].

A node is said to be the *parent* of the roots of its subtrees. A *leaf* of a binary tree is an empty node, that is, a node with no subtrees. The *path* of a leaf $\mathcal{L}$ is defined

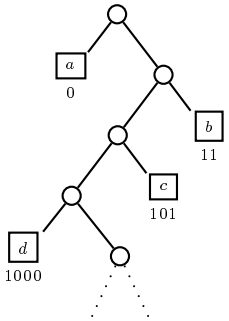


Figure 2.4: A binary tree. Each leaf is labelled with its path. Digit 0 corresponds to a left branch, digit 1 corresponds to a right branch.

recursively as the path of its parent (which is the empty set if it has no parent), followed by the digit 0 if $\mathcal{L}$ is the root of the left subtree of its parent, or the digit 1 if $\mathcal{L}$ is the root of the right subtree of its parent.

Proposition 2.15 PREFIX-FREE SETS AND BINARY TREES There is a 1-1 correspondence between prefix-free sets and binary trees.

Proof. A tree can be identified with the set of paths of its leaves. These paths form a prefix-free set:

Let $\mathcal{S}$ be a prefix-free set. In order to define a tree for $\mathcal{S}$, let us consider the set of nodes formed by all elements of $\mathcal{S}$ and their prefixes. These are the nodes that will form the tree.

For each node b, let $\mathcal{S}_b$ be the strings that contain b as a prefix. Define *left* subtree of b as the set $\mathcal{S}_{b0}$ (where $b0$ indicates the binary string b followed by the digit 0). Similarly, the *right* subtree of b is the set $\mathcal{S}_{b1}$.

The leaves of this tree are the sets $\mathcal{S}_b$ with only one element. In fact, the path of the leaf $\mathcal{S}_b$ is the binary string b. The prefix-free property guarantees that for each element b of $\mathcal{S}$ the set $\mathcal{S}_b$ has only one element, and therefore it is a leaf.

We have just defined a map $b \mapsto \mathcal{S}_b$ for b in $\mathcal{S}$. The inverse of this map is the function that maps each leaf to its path. Since the path of one leaf cannot be a prefix of the path of another leaf, the resulting set is prefix-free. □

Figure 2.4 shows an example of tree leaves and their paths.

2.4.2 $\mathcal{H}_U$ as a tree

The domain of U and the notions of model, description, and description length have a simple interpretation in terms of binary trees.

The set $\mathcal{H}_U$ corresponds to a tree, according to Proposition 2.15. We can identify the prefix-free set $\mathcal{H}_U$ with the binary tree, and use $\mathcal{H}_U$ to refer either to the prefix-free set or the tree.

The tree $\mathcal{H}_U$ is infinite. Let u be a program for U under U. For all x that halts U we have

$$U(ux) = U(x)$$

The subtree hanging from u is equal to $\mathcal{H}_U$. Therefore $\mathcal{H}_U$ contains itself as a subtree, and therefore it is infinite. This is illustrated in Figure 2.5.

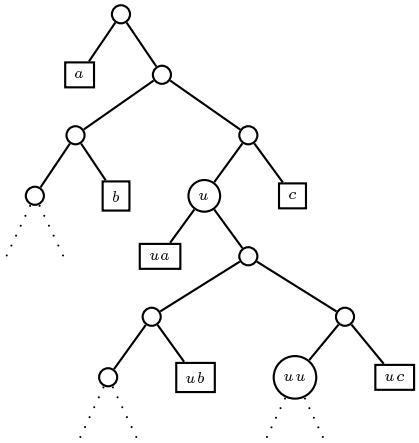


Figure 2.5: The tree $\mathcal{H}_U$ contains itself, and $U(um) = U(m)$

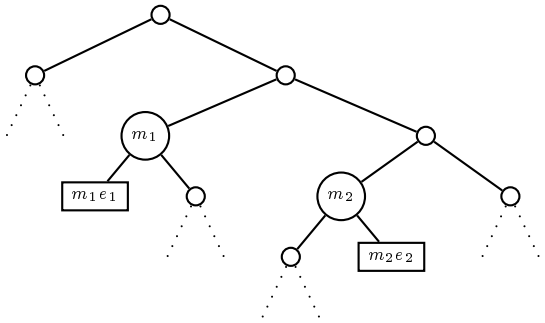


Figure 2.6: Example of $\mathcal{H}_U$. Models $m_1 = 10$ and $m_2 = 110$ and descriptions $m_1e_1 = 100$, $m_2e_2 = 1101$ of a binary string $d = U(m_1e_1) = U(m_2e_2)$.

In fact, the domain of U is, in a sense, independent of the choice of U, because Universal Turing machines can simulate each other. Therefore, a choice of U is just a choice of root node. A consequence is that the path length of each leaf is independent of the choice of root node (or Universal Turing machine U).

Descriptions are leaves of $\mathcal{H}_U$ Let the string x be a description of a string d, such that $d = U(x)$. The string x, which is an element of $\mathcal{H}_U$, corresponds to a leaf of the binary tree.

Models are nodes of $\mathcal{H}_U$ Any description of the form $x = me$ is a leaf of the subtree whose root is the node m. We have seen that a Turing machine M is simulated in U by using a string m, so that $M(e) = U(me)$. The Turing machine M corresponds to the node m.

Description length The length of a description $x = x_1x_2 \cdots x_n$ in U, where $x_i \in \mathcal{B}, 1 \leq i \leq n$, is n, the length of the path defined by x in the tree $\mathcal{H}_U$. When a description is of the form $x = me$, the lengths of m and e are added up.

These ideas are illustrated in Figure 2.6. For given data d, and models M_1 and M_2 with programs $m_1 = 10$ and $m_2 = 110$, the descriptions of d are m_1e_1 and m_2e_2.

2.4.3 Model Selection as enumeration of the tree $\mathcal{H}_U$

We now proceed to define Model Selection method and compare it with the Maximum Likelihood method using an example.

An enumeration of $\mathcal{H}_U$ is defined as a 1-1 map between $\mathcal{H}_U$ and the set of natural numbers $\mathbb{N}$. We are now defining Model Selection methods as enumerations of $\mathcal{H}_U$.

Let $\mathcal{M}$ be a class of models, and $\mathcal{D}$ a set of all possible data, for example, all possible outcomes of an experiment. A Model Selection method is a correspondence that maps a subset $\{M_1, M_2, \ldots\}$ of $\mathcal{M}$ and an element d of $\mathcal{D}$ to an element M_j of the subset $\{M_1, M_2, \ldots\}$. We will say that M_j the *preferred model* among the models $\{M_1, M_2, \ldots\}$.

$$(\text{Power}(\mathcal{M}) - \emptyset) \times \mathcal{D} \longrightarrow \mathcal{M} \tag{2.9}$$

(Where Power(X) is the power set of X.) In order to be useful, a Model Selection method needs to have the following property:

Minimal model For any finite or infinite family of models, there is a model which is preferred to any other model.

This property is necessary to avoid the situation when, for given data d and three models A, B and C, a Model Selection method prefers A to B, prefers B to C, and prefers C to A. As a consequence, a Model Selection method can be considered as an enumeration of all models that can model the data.

Model Selection as enumeration of models. A Model Selection method implies an enumeration of models that can model given data d. This is a consequence of the existence of a *minimal* model M which will be mapped into the natural number 1. The rest of the models has another minimal element, which is mapped into the natural number 2, and so on.

Let us recall that models are prefix of descriptions. Descriptions are leaves of a binary tree $\mathcal{H}_U$. If we have an enumeration of $\mathcal{H}_U$, we will be able to derive an enumeration of models, as explained below.

Model Selection as enumeration of descriptions. We can assume that a Model Selection method is an enumeration of descriptions of data d. Consider an enumeration φ of the descriptions of d. Given a model M and a description of d, such that $d = M(e)$, take the binary string m that implements M such that me is preferred under the enumeration φ. Given two descriptions of d, $d = M_1(e_1)$ and $d = M_2(e_2)$, and the binary strings m_1 and m_2, such that m_1e_1 is preferred to m_2e_2 under the enumeration φ, then we can say that the model M_1 is preferred to the model M_2.

Two questions arise in mapping an enumeration of descriptions of d to an enumeration of models of d:

- A model M can produce two different descriptions of the data, $d = M(e_1) = M(e_2)$. In this case we can ignore the description $M(e_j)$ ($j = 1$ or 2) with higher index in the enumeration of descriptions.
- Different models M_1 and M_2 can derive the same description. This is the case when there is a description of the data for which both M_1 and M_2 are a prefix.

We will avoid this situation by considering families of models such that those families are prefix-free sets.

The following definition embodies and simplifies these ideas.

Definition 2.16 MODEL SELECTION METHOD An enumeration of $\mathcal{H}_U$, or a subset of $\mathcal{H}_U$.

As we have just seen, from an enumeration of $\mathcal{H}_U$ we can obtain an enumeration of a prefix-free set of models which conforms to the requirement of equation (2.9).

The subset of $\mathcal{H}_U$ to be considered depends on the data and the models. It can be arbitrarily small but, the larger the subset, the more general the Model Selection method is because it covers more cases (more models, and more descriptions). The Model Selection method presented in this chapter is an enumeration of the whole $\mathcal{H}_U$.

Note that Definition 2.16 implies an enumeration of a subset of descriptions of each data d. Conversely, the enumerations of a set of descriptions of different data $d_1, d_2, \ldots$ can be collated into a single enumeration of a subset of $\mathcal{H}_U$ using Cantor's diagonal method. Consider an enumeration of the binary strings, $d_1, d_2, \ldots$. Then take the first model for d_1, the first model for d_2 and the second model for d_1, and so on, at each step n take the first model for d_n, the second model for d_{n-1}, etc.

Example 2.17 MAXIMUM LIKELIHOOD As an example of Model Selection method as an enumeration of $\mathcal{H}_U$, consider a Maximum Likelihood method [Edw92]. A likelihood function $L(M \mid d)$ is defined for model M given data d as a function proportional to the probability P of the data conditional to the model.

$$L(M \mid d) \propto P(d \mid M)$$

where P is a "rational measure of belief" that the data would be d [Edw92]. Assume that the value of $-\log(P(d))$ is an integer (consider all logarithms in base 2). Each of the models is a Turing machine M represented by a binary string m. For each model M, there is a string e_M such that $d = M(e_M)$ and

$$\text{length}(e_M) = -\log P(d \mid M)$$

which can be obtained using Huffman encoding based on P [Knu97]. The function that assigns a cost to the model M, $-\log L(M \mid d)$ gives us an order among models, in which the models with lower cost come first. When the cost of two models is the same, we can set an arbitrary order among them.

When all models m have the same length, the enumeration of the models according to $-\log L(M \mid d)$ is the same as the enumeration of the models according to the description length of d under U,

$$\text{length}(me_M)$$

Therefore maximum likelihood corresponds to minimal length of the description me_M when all models m have the same length. □

Example 2.18 LEAST SQUARES The Least Squares method is a particular case of Maximum Likelihood. Let E be an encoding of integers into binary strings such that, for a natural number n,

$$\text{length}(E(n)) = n^2$$

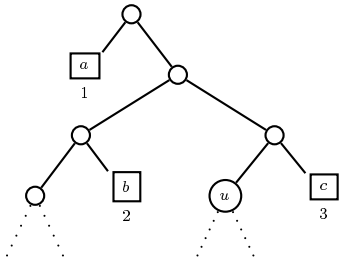


Figure 2.7: An enumeration of $\mathcal{H}_U$ assigns an index to each leaf.

The Least Squares method for a single parameter which is a natural number consists of using a model M such that $d = M(E(n))$. The encoding E can be easily generalised to several parameters $E(n_1, n_2, \ldots)$ and for other types of numbers.

□

2.4.4 Comparing Model Selection methods

Model Selection methods can be compared. This idea might seem counterintuitive since Model Selection methods are usually chosen according to the models, the data, the relation between both, and the application. But it is conceivable that, given two Model Selection methods α and β, and any two models, the choice of model by the method α is a good estimate of the choice of model by the method β, but not the other way around. We now proceed to formalise this idea.

It is possible to define an order relation between enumerations of $\mathcal{H}_U$, and therefore an order relation among Model Selection methods. Let α be an enumeration of $\mathcal{H}_U$, or a subset of it. Each leaf of $\mathcal{H}_U$ has an index given by the enumeration α, as shown in Figure 2.7. Let x be an element of $\mathcal{H}_U$. Denote with $\text{index}_\alpha x$ the index of the binary string x under the enumeration α.

Two enumerations α and β can be compared in the way they assign an index to each element of $\mathcal{H}_U$. For example, for an element x for which both α and β are defined, either $\text{index}_\alpha x \leq \text{index}_\beta x$ or $\text{index}_\alpha x > \text{index}_\beta x$.

Definition 2.19 ORDER OF ENUMERATIONS Let α and β be two partial enumerations of $\mathcal{H}_U$. Say that $\alpha \preceq \beta$ when there is a natural number k such that

$$\text{index}_\alpha x \leq \text{index}_\beta x + k$$

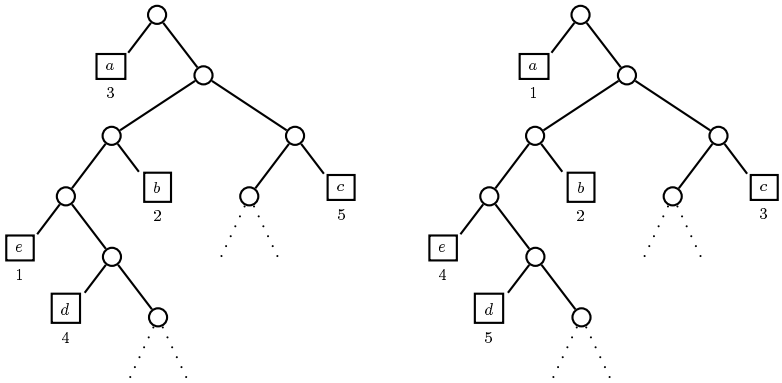
for all x in $\mathcal{H}_U$ for which $\text{index}_\alpha x$ and $\text{index}_\beta x$ are both defined.

This order can intuitively be interpreted as follows: the Model Selection α of Definition 2.19 will give a choice of model similar or better than the choice of model β, when comparing models.

From this point of view, the MDL method will be useful so long as it is minimal by the ordering $\preceq$, compared to other Model Selection methods.

2.4.5 A minimal enumeration under $\preceq$

The MDL method can be expressed as an enumeration of $\mathcal{H}_U$. The key idea is that each element b of $\mathcal{H}_U$ corresponds to an integer whose binary expansion is b. The relevance



(a) An enumeration of the tree

(b) The MDL-enumeration of the tree

Figure 2.8: Enumerations of the leaves of a binary tree

of this enumeration depends on its comparison with other enumerations using the order $\preceq$. Unfortunately, there is no Turing machine that computes this enumeration, and therefore it cannot be used directly for solving Model Selection problems by computer.

Recall that, given a model M and the Universal Turing machine U, there is a binary string m with the property that $M(x) = U(mx)$ for all x.

Definition 2.20 MDL-ENUMERATION AND MDL METHOD Let d be data and $\mathcal{M}$ a family of models of d. Each model $M \in \mathcal{M}$ is given as a binary string m. For each model M, let e_M denote the description of d under the model M, so that

$$d = M(e_M) = U(me_M)$$

Sort models according to the integer whose binary expansion is me_M. The result is the MDL-enumeration of the models of $\mathcal{M}$. The MDL method consists in selecting the model with lower index under the MDL-enumeration.

Figure 2.8 illustrates the MDL-enumeration.

Enumerating the leaves of $\mathcal{H}_U$ using the MDL-enumeration is traversing the tree completing each level before moving to the level below, starting from the root. In this way, the first description of the data d found is the shortest.

Conjecture 2.21 MDL IS MINIMAL UNDER $\preceq$. The enumeration of $\mathcal{H}_U$ implied by the MDL method is minimal under the order $\preceq$, for the enumerations which can be computed.

Thus, if we wanted to choose a universal Model Selection method under the assumption that the data has been produced by a Turing machine, MDL is the best choice.

The Halting Problem, reviewed in Section 2.2, guarantees that there is no Turing machine that states whether a binary string is an element of $\mathcal{H}_U$. As a consequence, there is no Turing machine that can compute the MDL-enumeration.

Other non-computable enumerations. It is important to note that the conjecture applies only for computable enumerations, while the MDL-enumeration (the minimal element) is not computable.

One could think of refuting the conjecture by constructing an enumeration α based on the MDL enumeration, in such a way that the MDL-enumeration is not a minimal enumeration for any set that contains α.

For example, a candidate for α could be computed from the MDL enumeration by reversing the sequence in which programs of the same length are enumerated.

But in this case, α would not be computable either, because computing it would imply knowing which are all the programs of a given length, and that would lead to a computable solution of the halting problem.

In general, we should keep in mind that it is the set of *computable* enumerations for which we are trying to find a minimal element under $\preceq$.

2.4.6 Interpretation of the conjecture

In the same way that models and descriptions correspond to nodes in a binary tree, enumerations of $\mathcal{H}_U$ correspond to enumeration of the leaves of a binary tree. But we need a less static way to represent the enumerations of $\mathcal{H}_U$.

Let us represent the non-computable MDL-based enumeration, and a computable alternative, using a Cartesian table, so we can compare their properties.

In the remaining of this section, we will draw a table such that

- Rows contain programs of length l, $\{p \in \mathcal{H}_U \mid \text{length}(p) = l\}$
- Columns contain programs that run for t cycles before halting, $\{p \in \mathcal{H}_U \mid \text{longevity}_U(p) = t\}$

where longevity of a program p in a Turing machine U,

$$\text{longevity}_U(p)$$

is the number of cycles before the Turing machine halts. We will now consider both the MDL-enumeration and a computable approximation to the MDL-enumeration.

The MDL-enumeration

Let $\mathcal{L}_l$ be the programs of length l,

$$\mathcal{L}_l = \{p \in \mathcal{H} \mid \text{length}(p) = l\}$$

The MDL-enumeration of Definition 2.20 is an enumeration of the elements of $\mathcal{L}_1$, then an enumeration of the elements of $\mathcal{L}_2$, ... and so on, up to an enumeration of $\mathcal{H}$. Figure 2.9 shows the table. Row l contains the elements of $\mathcal{L}_l$.

The enumeration restricted to each $\mathcal{L}_l$ is in general not computable due to the halting problem: there are programs that do not halt and there is no computable way to find out.

Informally, the MDL-enumeration jumps from a leaf to the next shortest leaf available.

time→ length↓	0	1	2	3	...
1	$\mathcal{L}_1$	$\mathcal{L}_1$	$\mathcal{L}_1$	$\mathcal{L}_1$	
2	$\mathcal{L}_2$	$\mathcal{L}_2$	$\mathcal{L}_2$		
3	$\mathcal{L}_3$	$\mathcal{L}_3$			
4	$\mathcal{L}_3$				
⋮	⋮				

Figure 2.9: Enumeration $\mathcal{L}_1, \mathcal{L}_2, \ldots$. Column t contains the programs that halt after t cycles. Row l contains the programs that are l bits long. $\cup_j \mathcal{L}_j = \mathcal{H}_U$.

time→ length↓	0	1	2	3	...
1	$\mathcal{J}_1$	$\mathcal{J}_2$	$\mathcal{J}_3$	$\mathcal{J}_4$	
2	$\mathcal{J}_2$	$\mathcal{J}_3$	$\mathcal{J}_4$		
3	$\mathcal{J}_3$	$\mathcal{J}_4$			
4	$\mathcal{J}_4$				
⋮	⋮				

Figure 2.10: Enumeration of $\mathcal{J}_1, \mathcal{J}_2, \ldots$. Column t contains the programs that halt after t cycles. Row l contains the programs that are l bits long. $\cup_r \mathcal{J}_r = \mathcal{H}_U$.

A computable approximation

The enumeration we are going to define now is an approximation of the MDL-enumeration, replacing the sets $\mathcal{L}_1, \mathcal{L}_2, \ldots$ by sets $\mathcal{J}_1, \mathcal{J}_2, \ldots$, with the constraint that the enumeration of each subset $\mathcal{J}_i$ is computable.

Let $\mathcal{J}_r$ be the programs of length l that stop at time $r - l$,

$$\mathcal{J}_r = \{p \in \mathcal{J} \mid \text{length}(p) + \text{longevity}(p) = r\}$$

An enumeration of $\mathcal{H}_U$ that runs each program of length $l = 1, 2, \ldots, r$ up to $t = r - l$ cycles is certainly computable. Figure 2.10 shows the table. The elements of each set $\mathcal{J}_r$ are spread along a diagonal of the table.

This is an enumeration of the whole set $\mathcal{H}_U$ because every program p that halts has a definite length length(p) and halts at time longevity(t), therefore p belongs to $\mathcal{J}_{\text{length}(p)+\text{longevity}(p)}$.

2.5 Probabilistic interpretation of MDL

Although we have defined and interpreted the MDL method from a combinatorial point of view, it is subject to a direct interpretation in terms of probabilities. This should not surprise us since a link between data compression and probability distribution of the data source has been established in Information Theory [CT91]. From the Bayesian point of view, the MDL method provides a prior for models.

2.5.1 The universal prior

We are going to see that the probability that a string x is a halting program for the Universal Turing Machine U, can be used as a drop-in replacement for any computable prior in Bayes' theorem.

Let T be a Turing machine. The probability of producing a given element e of $\mathcal{H}_T$, using a string of length(e) random coin flips, is $2^{-\text{length}(e)}$.

$$P_T(e) = 2^{-\text{length}(e)} \tag{2.10}$$

This is the probability that, when a binary string is fed into the Turing machine T, the program e is executed. The fact that $\mathcal{H}_T$ is a binary tree guarantees that P_T is a probability. In other words, this is the probability of the *input*.

A different issue is the probability of the *output*. The probability of obtaining an output d is the sum of the probabilities of each of its descriptions:

$$Q_T(d) = \sum_{e|T(e)=d} 2^{-\text{length}(e)}$$

The function Q_T is a probability measure defined in the union of $\mathcal{H}_T$ and the binary strings for which T does not halt [LV97]. When the binary tree $\mathcal{H}_T$ is formed by the paths 0 and 1, the corresponding sum $2^{-1} + 2^{-1}$ is 1. If one of the paths is missing, for example if T does not halt with input 0, the sum is less than one. When the binary tree is deeper, the same argument applies to each branch, with the depth l increased by one. By recursion, it can be proven that

$$\sum_{d \in \mathcal{H}_T} Q_T(d) \leq 1$$

and the sum is equal to one if the programs that do not halt are included (see Kraft's Inequality in [CT91] and the discussion on probability measures in [LV97, p. 244] for more details).

The measure Q_U, for a Universal Turing machine U, is in a sense independent of the choice of U. Since the path lengths between different Universal Turing machines only differ by an additive constant, the Universal Measure according to different Turing machines only differs by a multiplicative constant [LV97]. Therefore we can refer to Q_U as *the Universal Measure*.

The measure Q_U can be used as prior probability when there is no information available about the "true" prior on the space of models, or when the existence of such "true" prior is debatable.

An important theorem links Q_U and Kolmogorov Complexity. The proof, which is included in the more general "coding theorem" is given in [LV97, page 253]

Proposition 2.22 Universal Measure and Kolmogorov Complexity There is a constant r such that, for any string x,

$$| -\log Q_U(x) - K(x) | < r$$

Recall that the Kolmogorov Complexity of x is the length of the shortest description of x,

$$K_U(x) = \text{length}(\mathbf{C}_U(x))$$

The consequence is that length$(C_U(x))$ can be used to approximate $-\log Q_U(x)$.

Proposition 2.22 has established the relation between the Universal Measure Q_U and the Kolmogorov Complexity K. Now we proceed to study the use of Q_U as a prior in Bayes' theorem.

2.5.2 Bayes

MDL corresponds to using the measure Q_U as prior probability in Bayes' rule. It can be used as long as the data is "typical" of each model.

In practice, this means that we can use Q_U as a prior in Bayes rule, instead of any other computable prior (see [LV97, Theorem 5.2.1] for an upper boundary of the difference in predictions). This would release us from the burden of having to find an adequate prior for a particular problem, but at the cost of computability.

Since computability is essential, and we must deal with computable approximations, what we obtain is that the posterior calculated with Q_U *converges* towards the posterior calculated with more accurate priors, but only in the large data limit [CT91].

The consequence is that the MDL method provides a prior for Bayesian inference, but only useful when there is a large amount of data from which to "learn" the prior.

The Bayesian cost of a model M given data d is the probability of a model M conditional to data d, $P(M \mid d)$. If the prior probability of a model $P(M)$ is available, and we could calculate the probability that a model M produces data d, $P(d \mid M)$, then we could use Bayes' rule

$$P(M \mid d) = \frac{P(M)\ P(d \mid M)}{P(D)}$$

and choose the model M that minimises

$$P(M)\ P(d \mid M) \propto P(M \mid d) \tag{2.11}$$

which is equivalent to minimising the expression

$$-\log P(M) - \log P(d \mid M)$$

To obtain the Minimum Description Length principle, we need to replace the probabilities $P(M)$ and $P(d \mid M)$ by the measures Q_U and Q_M ,

$$-\log Q_U(M) - \log Q_M(d)$$

For further details about the conditions under which replacement can take place see [LV97, Section 5.5.1]. The log-probability $-\log P(M)$ is replaced by $-\log Q_U(M)$, and $-\log P(d \mid M)$ is replaced by $-\log Q_M(d)$.

Expression (2.12) becomes, applying Proposition 2.22 with any kind of Turing machine,

$$\text{length}(\mathbf{C}_U(m)) + \text{length}(\mathbf{C}_M(d)) = \text{length}(\mathbf{C}_U(m)\mathbf{C}_M(d))$$

for any program m of the model M. Finding the M that minimises the value of Expression (2.5.2) is the same as Definition 2.11 of the Minimum Description Length method.

The Universal Probability effectively includes a mixture of all computable probability distributions [CT91]. For models drawn from a computable distribution, the posterior calculated with the Universal Probability converges to the posterior calculated using the computable prior.

2.5.3 Maximum Likelihood

Given model M and data d, the Likelihood function is proportional to $P(d \mid M)$, where P is a probability defined on the space of data, and the constant of proportionality arbitrary [Edw92]. The Maximum Likelihood method prefers the model with higher Likelihood. The Likelihood function is not a probability on the models: when considering $P(d \mid M)$ as a probability, the data d is the variable and the model M is constant. But, for the Likelihood function, the data d is the constant and the model M is the variable, and the sum of $P(d \mid M)$ for all models does not need to be 1.

A disadvantage of the Maximum Likelihood method is that it does not take into account the intrinsic differences between models, and therefore it is subject to the problem of overfitting, as we have seen in page 9. For this reason, the Maximum Likelihood requires that the models being compared have been chosen carefully.

The MDL method explicitly uses a "population of models" (the binary strings that correspond to Turing machines given a Universal Turing machine U), and therefore goes beyond Maximum Likelihood in making a choice among the models on account of their intrinsic properties. In fact the MDL method includes the Maximum Likelihood method as a special case [LV97, 370], namely when all models of a family $\mathcal{M}$ have the same minimal description length, that is, for every $M \in \mathcal{M}$, $\mathbf{C}_U(M)$ is the same, and

$$P(d \mid M) = 2^{-\text{length}(\mathbf{C}_M(d))}.$$

In practice, even though models might not all have the same description length, the differences in their description length might be small compared to the minimal description length of the data. Thus the description length of the data,

$$\text{length}(\mathbf{C}_U(M)\mathbf{C}_M(d))$$

is dominated by the binary string $\mathbf{C}_M(d)$, which is the one that really makes a difference. In this situation, the MDL method is also equivalent to the Maximum Likelihood method and the cost of the model is practically not taken into account.

2.6 Alternatives to MDL

2.6.1 Minimum Message Length

A similar method to MDL is the Minimum Message Length (MML) method of model selection developed by Wallace et al. [WB68, WD99]. For most practical purposes the MML method is equivalent to the MDL method: use models to encode data into a binary string, then add the length of that binary string to the length of each model to obtain a cost function for each model.

The main difference between the MML and MDL methods is that MML explicitly uses priors about the data [vR99]. In practice both MDL and MML yield similar results because the choice of set of models to compare is generally very restricted, for practical reasons, and it is the dominant factor in the encoding of the models.

In the context of this thesis, the equivalent of the MML method would lie near the point at which we make a choice of particular Universal Turing machine, in Chapter 4.

2.6.2 AIC and BIC

Let us now consider two alternatives to the MDL method, the information criteria of Akaike (AIC) and Schwarz (BIC). The setting for these criteria is usually different from the one chosen in this chapter for MDL. Usually a Model Selection method is not a choice of individual model, based on a measure of the model complexity, but a choice of a family of models or "parametric-model", based on the dimension of its parameter space. Let us first define what we consider by parametric-model and its dimension. A *parametric-model* is a family of models $\mathcal{M}$ which are indexed by n parameters $p_1, \ldots, p_n$, for $n > 0$. A parametric-model has a *dimension* d which is the number of statistically independent parameters. The problem of parametric-model selection is, given data and a family of parametric-models $\mathcal{M}_1, \mathcal{M}_2, \ldots$, to choose which $\mathcal{M}_i$ better describes the data. This decision is based on

1. the best fit $M_i \in \mathcal{M}_i$ to the data and,
2. the dimension of the model class $\mathcal{M}_i$.

Finding the best fit M_i within a parametric-model $\mathcal{M}_i$ can be done using a standard model fitting method, such as Maximum Likelihood, for each i. The next step would be to compare the best fits $M_1, M_2, \ldots$. But comparing them on the same grounds, as if they were elements of the same parametric-model, leads to the problem of *overfitting*, as we have seen in page 9, because the model complexity, in this case, the dimension of each parametric-model, has not been taken into account.

Akaike's AIC method, and Schwarz's BIC method are two popular methods for selecting parametric-models [HY01, NT97]. Both provide a correction factor α to the Likelihood function that depends on the dimension of the parametric-model. When the log-likelihood is maximised, the correction takes the form of an additive term.

Akaike's AIC. Akaike's method [Aka74] consists in choosing for each parametric-model $\mathcal{M}_i$, $i = 1, 2, \ldots$, the parameters that yield maximum likelihood L_i. The preferred parametric-model $\mathcal{M}_i$ is the one that maximises

$$\log L_i - d_i. \tag{2.12}$$

where d_i is the dimension of the parametric-model $\mathcal{M}_i$.

Akaike's method was initially devised as an alternative to Hypothesis Testing when selecting models for time series analysis [Aka74]. Formula (2.12) is derived from an unbiased estimate of the *mutual information* between the model M and the "true" probability distribution that produced the data [CR99].

Schwarz's BIC. Schwarz's method [Sch78] is a modification of AIC in which the function to maximise is

$$\log L_i - \frac{1}{2} d_i \log n_i. \tag{2.13}$$

The BIC method is based on the idea that, for a given dimension, Bayesian estimators tend to Maximum Likelihood estimators as the size of the data increases, regardless of which is the prior distribution (as long as it is not zero at any point) [Sch78]. Formula (2.13) is derived as an approximation to the limit Likelihood function.

The AIC and BIC methods differ when the number of observations increases [Sch78]. Schwarz's method is more biased towards small models than Akaike's method because the difference in the second term of Equations (2.12) (2.13) is

$$\frac{1}{2}\log n_i$$

Both AIC and BIC provide a way to assign a weight or cost to the dimensionality of a model M with respect to the fit of M to the data. (The letters IC of the acronyms stand for "Information Criterion", whereas "A" and "B" are index letters which accidentally correspond to "Akaike" and "Bayesian" [Aka74]). In this chapter we have presented the MDL method with a similar purpose, but in different terms:

- Instead of selecting a parametric-model $\mathcal{M}_i$ (in practice, a dimension d_i), we are using MDL to select a particular model M. The comparison is not between classes of models, but between individual models.
- Instead of considering the dimension of a parametric-model $\mathcal{M}_i$, the MDL method weights in the complexity of the model M.

In general, our definition of the MDL method is intended to make a choice between *individual models*, whereas AIC, BIC and other parametric-model selection methods are designed to provide a choice between *model classes*, given that additional methods are successful at choosing a model within a given class.

The main problem of the AIC and BIC methods is the assumption that the dimension d of a model is known. A common approach in practice [NT97] is to consider as dimension d the number of parameters n, but then it is easy to see that, by modifying the parameter space (e.g. mapping 1-1 two parameters into one) the result of the criteria is different while the models are essentially equivalent. The dimension d coincides with the number of parameters n when the parameters are statistically independent.

The MDL method can be applied to parametric-models, because each model within a parametric-model has a well defined description length. When the model complexity is considered the same for all models, MDL reduces to Maximum Likelihood, and is equivalent to Bayes with a uniform prior in the space of models. When the model complexity varies across parametric-models, but is the same within each parametric-model, the cost of a model M is a function of the parametric-model $\mathcal{M}_i$ to which it belongs. The weight that the MDL method assigns to each model M, $\text{length}(\mathbf{C}_U(M))$, depends on the way the model is mapped into a binary string. For some families of models, and for some computable approximations to $\mathbf{C}$, we might obtain the same cost for a model as in formulas (2.12) and (2.13). For this reason computable approximations to the MDL method contain AIC and BIC as particular cases [Ris83, HY01].

2.7 Conclusion

We have used the MDL method of Model Selection to avoid the problem of overfitting, and choose between models of different complexity taking not only their fit to the data, but also their complexity, into account. Overfitting occurs when complex models are systematically preferred to simpler models, and is a symptom that model complexity

is not taken into account. In our development of the MDL method, we have considered models as Turing machines, and their complexity is defined as their description length.

The MDL method, which is usually introduced from the probabilistic point of view, admits also an interpretation in terms of enumerations of the domain of the Universal Turing machine U. It has been conjectured that, under an order among enumerations, the MDL-based enumeration is minimal. It falls outside the set of computable enumerations, although it can be approximated by a computable enumeration.

Applying the MDL method has been reduced to a problem of data compression. Compression is performed by the function $\mathbf{C}_M(d)$, that finds the shortest description of d under the Turing machine M. The MDL method, in a first approach, consists in choosing the model M that minimises the length of the binary string

$$\mathbf{C}_U(m)\mathbf{C}_M(d)$$

where m is a program for M.

In other words, the MDL method states that the models that better explain the data, are the ones that compress the data more.

The compression function $\mathbf{C}_U$ is not computable for any Universal Turing machine U. Computable approximations C_M and C_U to $\mathbf{C}_M$ and $\mathbf{C}_U$ are needed. Chapters 3,4 and 5 use computable approximations to the MDL method.

The MDL method has been compared to the alternative AIC and BIC information criteria for parametric-models. The MDL method is, in a sense, a generalisation of AIC, BIC and Maximum Likelihood methods, according to which particular approximation C is used to the optimal compression $\mathbf{C}$. The MDL method has also been derived within a Bayesian framework.

The next step is to produce computable approximations to the MDL method that can be used to solve Computer Vision problems. Chapter 3 brings us to the first experiments with MDL, by taking a simpler yet well founded approach to vehicle tracking. The data d is an image sequence, the models M are vehicle trajectories, and the compression function $C_M(d)$ is an off-the-shelf data compression algorithm with a computer graphics engine.

Chapter 3

Trajectory estimation from image sequences

Two approaches are taken in this thesis to define computable approximations to the optimal compression function $\mathbf{C}$:

1. The experimental way is to define a data compressor using all information available, tailored to the application, and refine it according to the results of the experiments. This is the approach taken in the current chapter.

2. A more thorough approach is to write a software package of reusable code. Separation between the general MDL method and the particular approximation of $\mathbf{C}$ used in an application is crucial for the code to be reusable. This approach will be taken in Chapter 4.

This chapter describes an application of the MDL method to a real-world problem of Model Selection in Computer Vision. In this application, images are compressed using geometric and dynamic models for the shapes and trajectories of the objects being modelled. Experiments fit 2D and 3D vehicle models and simple trajectory models to traffic image sequences. This approach is valuable because it provides information about the performance of the MDL method.

3.1 Introduction

Visual surveillance. The traffic of many cities and major roads is routinely monitored by a network of cameras connected to central stations (see Figure 3.1). The information that can be obtained from traffic surveillance cameras [Fer02] ranges from estimates of traffic volume and speed, to the detection of unusual trajectories at intersections, estimation of vehicle occupancy, vehicle classification, detection of accidents, monitoring the effects of roadworks, detection of vehicles outside the lanes or in the wrong lane, detection of obstructions on the road, etc. Most of these applications require monitoring the trajectories of individual vehicles, or *tracking*. Tracking is one of the main applications of research in Computer Vision [IEE99, GL96, RMFB98, MRW+94, CLK+00].

Figure 3.1: Images from traffic surveillance cameras. Top images and bottom left image courtesy of BBC London (`www.bbc.co.uk/london`) and Transport for London. Bottom right image courtesy of the Welsh Assembly Government's traffic information centre (www.traffic-wales.com). Images reproduced by kind permission.

Vehicle tracking. Let us denote by *tracking* the problem of finding the position and orientation of a vehicle, in coordinates on the ground plane, at regular intervals of time. The simplest trackers model the image of the vehicle by a "blob" or connected region of pixels. Differences in pixel values are detected from one image to the next. This method can be adequate provided the camera is placed high enough to avoid occlusions [KWM94]. Tracking in 2D is not sufficient when trying to recover more detailed information needed for accurate monitoring and control of traffic. This information can only be obtained visually by taking into account the 3D nature of the scene. For example, the trackers described in [KN95, Sul92, Sul94] obtain a time sequence of measurements of the 3D position of a vehicle by matching wire-frame models against the images.

Top-down vision. Consider a Vision Function that maps data to models that represent understanding of the data.

$$\mathcal{D} \longrightarrow \mathcal{M}$$

When the Vision Function is just the verification or *matching* of models against data (or a representation of it), it is said to be a top-down Vision Function.

In some cases models are matched against data by comparing properties of image pixel values with values predicted by the models. For example [Sul92] matched 3D wire-frame models of vehicles against the images. A boolean test is used to detect the presence of an edge at one of the bounding edges of the wire-frame. The matches for each bounding edges of the model are combined to form a *cost* assigned to each model. Let $\varphi(M)$ be the cost of model M.

The Vision Function finds the model M_0 which minimises the cost φ. Defining and implementing a Vision Function requires two steps:

1. Define a cost function φ which assigns a lower cost to models which provide better understanding of the data under some criterion.

2. Define an optimisation algorithm that finds the minimum of φ.

For example, when the model is a wire-frame vehicle and the data is an image, the cost function could be a likelihood function of pixel values in the proximity of wire-frame edges [Sul92].

This chapter is solely concerned with Step 1, defining a cost function φ. Optimisation, Step 2, should be designed according to the properties of the cost function.

Edge-based cost function. The areas of an image that correspond to the projection of edges of artificial objects are usually associated with large difference of intensity between pixels. This principle has been used for detecting edges [Mar82, Can86] in bottom-up vision, and for matching edges against the data, in top-down vision

An example of vehicle tracking by matching projected edges is given by Brisdon's edge matching function [Bri90]. In [Sul92], the edge matching function is used to match 3D models against images, by projecting the 3D models, as wire-frames, into the image plane. The cost of a model depends on the number of projected edges that match perceived edges in the image, and the quality of each match. Thus a very small amount of data from the original image is used, in fact only a few pixels around the projected edges, and very few and simple arithmetic operations are performed.

- The main advantage of reducing the number of pixels being considered is the increase in speed of computation, but
- the main problem is that a good initial estimate of the vehicle's position in the image needs is needed.

The problems of edge-based methods. While edge based methods have the advantage of requiring a very small number of pixels for the matching of the model with the image, they have two main disadvantages:

- Edge models need to fit well to the data. This requires a thorough search of the space of shape models since it is not possible to know a priori the shape of a new vehicle being tracked. Work by [FWSB97] has taken this direction.
- Edges in the image may not be numerous enough to validate a match. This is the case when the scene contains any of the following:
  - highlights in the vehicle surface
  - overlapping vehicles
  - overlapping unmodeled objects

In order to avoid these problems, simplify the method, and provide a better solution than arbitrary weighting of models, we will not only model pixel values around edges, but also pixel values of the whole image. For this purpose, we will use the MDL method of model selection.

Alternative: An MDL-based cost function. The alternative suggested in this chapter is the comparison of different models according to their ability to compress image sequences, using the MDL method of Model Selection.

The procedure is to describe the data using each model in turn. The description consists of the model and the difference between the model and the data. The cost is the length in bits of that description.

Consider the case of detecting whether a vehicle is stationary or not in a given image sequence d. Let A be the model of a vehicle moving on a straight line, and let B be the model of a vehicle stopped in the middle of the road. In order to compare both models, let us compress the image sequence with each of them.

- Compressing the image sequence d using model A assumes that the image sequence is essentially static but for a small region of pixels translating across it.
- Compressing the image sequence d using model B assumes that all the images in d are identical.

Let $C_A(d)$ and $C_B(d)$ be, in the notation of Chapter 2, the binary strings that result from each compression. If the image sequence d shows a vehicle moving on the trajectory assumed by model A, $C_A(d)$ will be shorter than $C_B(d)$. Conversely, if a vehicle is present and is stopped (or is not present) then $C_B(d)$ will be shorter than $C_A(d)$.

From Definition 2.11 of the MDL method, each model M is assigned a cost

$$\varphi(M) = \text{length}(\mathbf{C}_U(M)) + \text{length}(\mathbf{C}_M(d))$$

where $\mathbf{C}_T(y)$ is the shortest description of y under model T, and U is a reference Universal Turing machine.

Note that, in the example above, there is no way to distinguish between a vehicle that is stopped and no vehicle at all. Or even between those and a traffic jam. This is both a problem and an advantage

Problem If we wanted to distinguish between no vehicle, one vehicle and traffic jam we need to increase the size of the data (that is, to use a longer image sequence), that includes vehicles in motion, and the amount of models, which will include each of those cases.

Advantage If we do not want to distinguish between those three cases, then we do not need to model them. This is a fundamental difference from an edge-based model matching system, in which additional tests are required to estimate the number of vehicles in the scene.

We will use this advantage for the detection of events in a roundabout in Section 3.3 in cluttered images.

In addition, an MDL approach has the potential to overcome the problems of edge-based methods:

- This approach relies on very few assumptions on the data. The main assumption is that the data is the output of a Turing machine which is given random input.
- All data available is used, namely all pixel values of the image, and not only pixel values around predicted edges.

Using all data available. The purpose is to compress, with the help of the model, all data available. When the data is a single image, as it is the case for edge-based cost functions, each of the pixel values of the image will be compressed. But, in our application to vehicle surveillance, more than a single image can be made available. The *data*, will be a sequence of consecutive images in time.

The models used in this chapter will describe not only the objects in a single image, but also how those objects move. In particular, the models will not only be the *shape* of the vehicle, but also the *trajectory*.

Advantages: simplicity. A main advantage of the method proposed in this chapter is its simplicity. The software used for the experiments of this chapter is written in about 4000 lines of C code. Most of the computing resources used by the software are spent on generation of 3D graphics from geometric models, and the lossless compression of image sequences. Alternative implementations of these algorithms could be obtained from standard libraries or even off-the-shelf hardware.

Advantages: The universal probability. A consequence of using the MDL principle, as in Chapter 2 is that the error between the model and the data is modelled using a computable approximation to the universal probability. The consequence of this is that if we only assume that the data has been produced by a Turing machine, and make no further assumptions, the universal probability is our best bet for a prior

distribution of that data. Of course, in practice, we do not assume that the data has been produced by a generic Turing machine, but we have a large mount of prior information about the data, and the effort is directed towards including all that information in the compression algorithm.

Previous applications of the MDL method to Computer Vision were oriented towards low-level image processing problems such as region growing [ZY96] or object segmentation [ZB95]. In [FM99b, MF01, FM01] the author studied the compression of static images, and the models described physical properties of objects in the images, such as position and rugosity. The current chapter is based on, and extends, the methods for modelling image sequences using shapes and trajectories presented in [FM00a, FM00b].

Contents of this chapter. A method to compress image sequences using geometric models for the shape and the trajectories of objects is presented in this chapter. Particular issues arising with 2D and 3D geometric models for the shape are discussed. Experiments for each case are presented.

2D models. Very different trajectories are compared. These trajectories correspond to the motion of vehicles on different areas of a roundabout (Section 3.3).

3D models. Similar trajectories are compared. These trajectories correspond to alterations in the motion of a vehicle with a time scale of 4 or 5 seconds (Section 3.4).

Before discussing compression using 2D and 3D models, a generic procedure is presented, which is common for both.

3.2 Two-part compression of traffic image sequences

This section describes a general procedure for compressing image sequences using a geometric model of the scene. The procedure is outlined in Figure 3.2. The geometric model consists of the shapes of the vehicles and their trajectories on the ground plane. This method does not depend on the particular type of geometric model being used. Two types of models will be considered in this chapter:

- The case of a 2D model of vehicles in the image is covered in Section 3.3.
- The case of a 3D model of vehicles in the space is covered in Section 3.4.

The compression method, outlined in Figure 3.2, consists of *encoding* or representing the data as a binary string formed by two main parts:

- The geometric shape model, the trajectory model, and the first and last images of the image sequence (*reference images*). This is used to create an approximation of the original data. Figure 3.3 illustrates the reference images and the way a shape model is projected on each image according to the trajectory.
- The error, pixel by pixel, between the original data and the approximation.

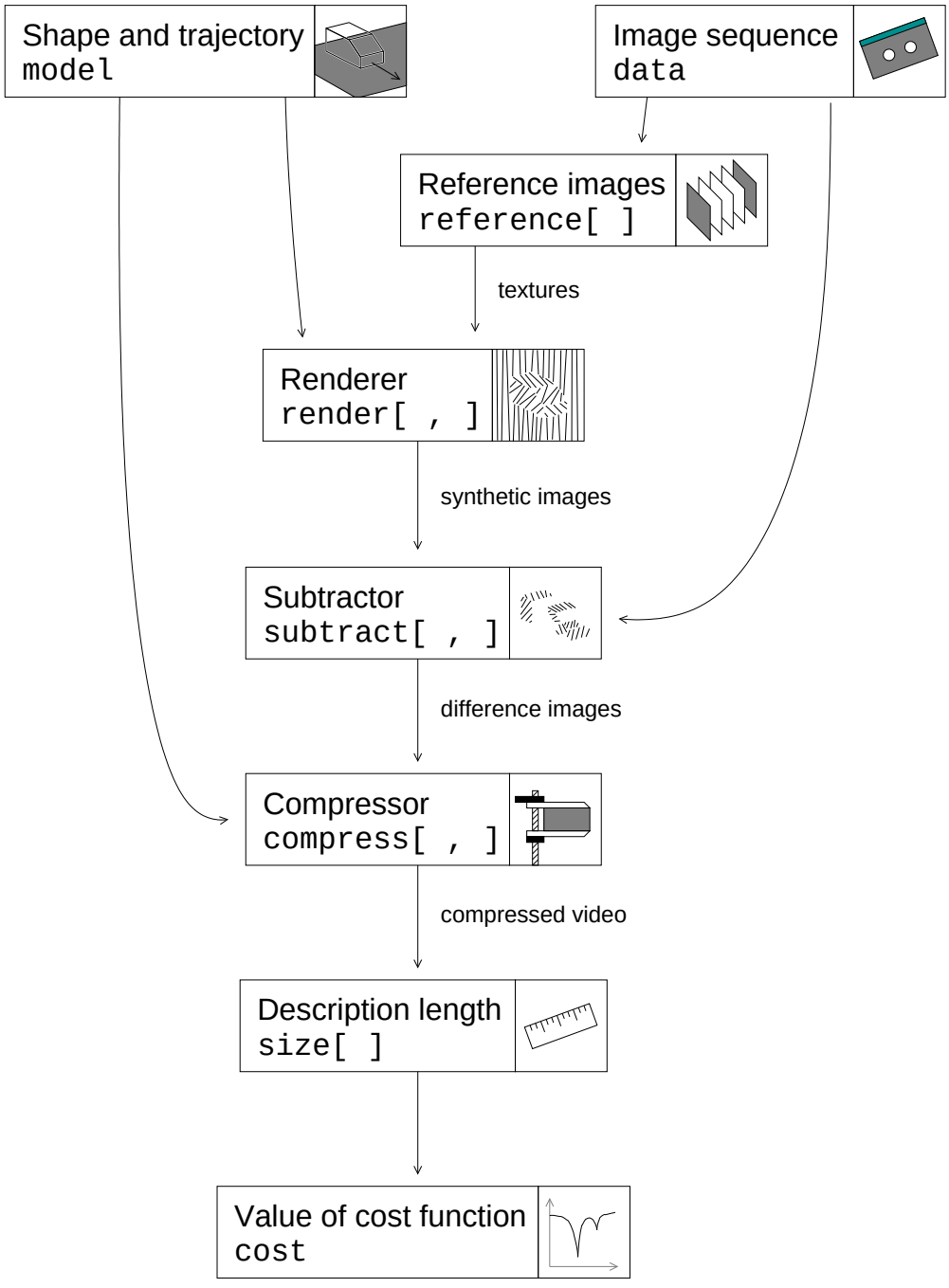


Figure 3.2: Two-part compression of traffic image sequences

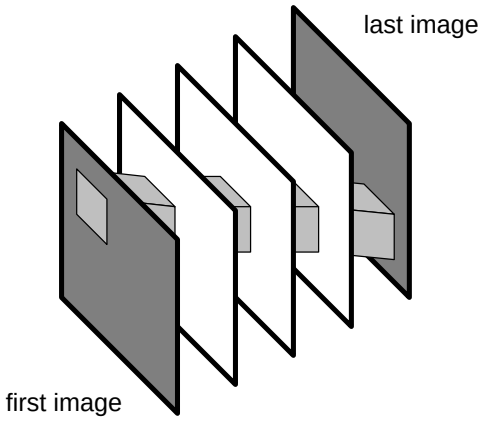


Figure 3.3: The reference images in an image sequence.

Both components together form a *two-part description* of the image sequence data. The shape, trajectory, reference images and the pixel-by-pixel errors are enough to reconstruct the original image sequence exactly.

The procedure described in Figure 3.2 consists of the following sequence of steps. Given the `data` and the shape and trajectory models, named together as `model`:

1. Take part of the original data for reference, `reference`. In this chapter it is assumed that the reference images are the first and last images of the image sequence, as in Figure 3.3. These two images should include between them all regions visible in the sequence. For example, in the case of a 3D model of a vehicle turning, those images should include all the sides of the vehicle which are visible at any time during the sequence. The pixels in the reference images will be projected later on in other parts of the image. In computer graphics, these pixels are called the pixel *textures*. Following the notation used in `Mathematica` in which a function f applied to x takes the form $f[x]$, `textures = reference[ data ]`

2. Render a synthetic version of the original data by using the shape and trajectory models, and the reference data. The result is a sequence of images of similar dimensions than the data. `synthetic = render[ model, textures ]`

3. Subtract the synthetic data from the original, to obtain difference images. If the model is correct, then the difference images should contain small values. `difference = subtract[ data, synthetic ]`

4. Compress the difference images, assuming they contain small values. A *general purpose* data compressor is used for this purpose. `description = compress[ model, texture, difference ]`

5. Measure the size in bits of the compressed description of the data, consisting of: the shape and trajectory models, the reference images, the compressed difference image. `cost = size[ description ]`

This is a summary of the steps, decomposing `model` into `shape` and `trajectory`:

1. `textures = reference[ data ]`

2. `synthetic = render[ {shape, trajectory}, textures ]`
3. `difference = subtract[ data, synthetic ]`
4. `description = compress[ {shape, trajectory}, texture, difference ]`
5. `cost = size[ description ]`

Note that just using `description` it is possible to reconstruct the original data. This is the reason why we are compressing the entire images, and not just parts of them.

The first parts of the description, `shape` and `trajectory` contain the useful information about the scene, namely the shape and the trajectory of the object. The lower the value for `cost`, the better `shape` and `trajectory` describe `data`.

Sections 3.3 and 3.4 give details of the implementations and performance for the 2D and 3D shape and trajectory models.

3.3 2D shape and trajectory models

In this section a simple type of 2D shape model is used for an implementation of the method of Section 3.2. It is applied to the detection of events in a roundabout.

One of the simplest shape models we could choose are rectangles in the image plane, aligned with the image axes. Taken together with an image, such a rectangle defines a *texture* or array of pixels taken from the image. The rectangle, together with the texture, is what we call a *sprite* [How] according to tradition in computer graphics.

Sprites are not only simple, but also fast to implement, and correspond to the intuitive idea of motion of an object in the image, in the sense that, roughly, it should be possible to cut and paste from one image to generate the next. This "cut and paste" procedure will not produce realistic synthetic images when the apparent size of the vehicle changes due to variations in the distance from the vehicle to the camera. If a full calibration of the camera is available then the problem can be solved by scaling the sprites to represent the change in size according to the distance.

Since the shape models are not very accurate, we do not expect to obtain an accurate estimate of the vehicle trajectory. The purpose of this section is to compare very different trajectories, and see if the cost function can discriminate between them. Each of those trajectories corresponds to what we will call a "traffic event" or just "event".

Therefore the models used in this section are:

Shape: rectangles in the image plane, parallel to the image axis, that can be subject to translation and scaling.

Trajectory: a sequence of points, defined by hand on the ground plane, corresponding to the events we want to detect in the image sequence.

Below is an explanation of how to implement the function `render`, and experiments using the resulting cost function to detect events.

3.3.1 Compressing a sequence of images using 2D models

We need to define the function `render` of Section 3.2 when the shape models are sprites.

The function `render` has the following prototype:

```
synthetic = render [ model, textures ]
```

where `model` is a sprite, and the trajectory is a sequence of points on the ground plane. The `textures` is the first and last images of the image sequence from which pixel values will be cut and pasted to the rest of the images.

The output of `render` is the sequence of images `synthetic`. The pixel values of `synthetic` are computed as follows.

Let p_t be the sequence of positions on the ground plane of the sprite at time t, for $t = 1, 2, \ldots$. Let r_0 be the original sprite. The *frame of reference* of the sprite is defined as follows: the origin is the bottom left corner. The unit vector $(1, 0)$ is the horizontal side of the sprite, from left to right. The unit vector $(0, 1)$ is the vertical side of the sprite, from bottom to top. A pixel is located inside the sprite if and only if its coordinates are between 0 and 1 in the frame of reference of the sprite.

1. Using the camera calibration [FvDFH90, WSB94], calculate the projection of the position p_t on the image plane, q_t, and the scaling factor s_t according to the distance between p_t and the camera.

2. Calculate r_t by scaling r_0 by s_t, and translating it in the image plane so that the centre point of the base of the sprite r_t is q_t.

3. For each pixel (x, y) at image time t,

   - If (x, y) is outside r_t, then the value of (x, y, t) is the value of the pixel with same coordinates in the first image $(x, y, 0)$, but for the exception below.
   - If (x, y) is inside r_t, let (x', y') be the coordinates of (x, y) in the frame of reference of r_t. The value of (x, y, t) is interpolated from the pixel of the first image with coordinates (x', y') with respect to r_0.
   - An exception takes place when (x, y) is outside r_t but inside r_0. In that case, the value of (x, y, t) is the value of (x, y) in the last image. This is the reason why both the first and last images are taken for reference. If this special case was not considered, a vehicle would appear twice in each image, both at r_0 and at r_t.

3.3.2 Experiments

In order to illustrate the ability to work in cluttered scenes, the cost function has been used to test which "event", from a family of events, was taking place. The family of events consists of very different trajectories that vehicles can follow in a roundabout. It includes the event "nothing is happening".

Figure 3.4: (Left) An image of the original image sequence. The sprite is marked with a rectangle. (Middle) An image of the synthetic sequence. The vehicle has been cut, scaled and pasted from the first image of the sequence. (Right) The difference image.

Events

An "event" is a trajectory defined as a sequence of points on the ground plane, corresponding to a situation we would like to detect. Seven events were defined; six correspond to trajectories of vehicles moving on a road, plus another that represents "nothing is happening".

Upper A vehicle is taking the upper exit

Left A vehicle is moving around the roundabout on the left lane (left from the point of view of the camera)

Right A vehicle is moving around the roundabout, on the right lane (right from the point of view of the camera)

Crossing A vehicle is crossing the white lines near the camera

Enter A vehicle is entering the roundabout from the left hand side of the image

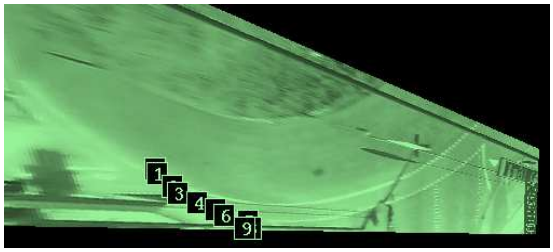
Changing A vehicle is changing lane around the centre of the image

Nothing None of the above is happening

The label attached to each "event" is only descriptive. Each event is intended to represent the trajectories which are *similar* to the trajectory given to the system under that label. The measure of *similarity* will be given by the compression achieved, using the MDL method. Figure 3.5 displays the events as points on the ground plane, as viewed from above. The homography between the ground plane and the image plane has been used to draw the background image (for illustrative purpose). Each point of the trajectory is labelled with its index in the sequence.

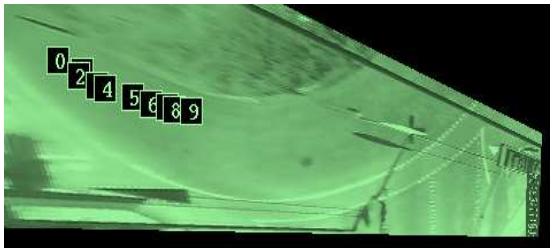
Data

Each dataset is a sequence of images labelled 00000, 00207 and 00620, according to the timestamp of the first image. Each image was 256×256 pixels in size and 8 bit

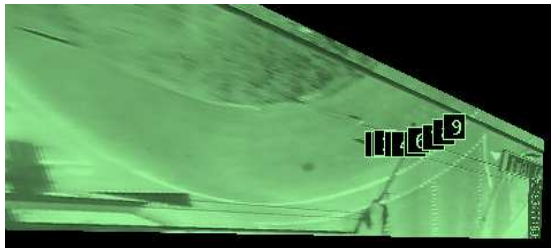


(a) Upper

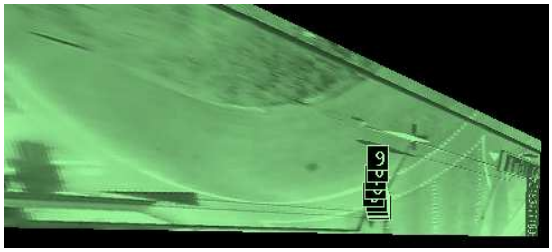
(b) Left



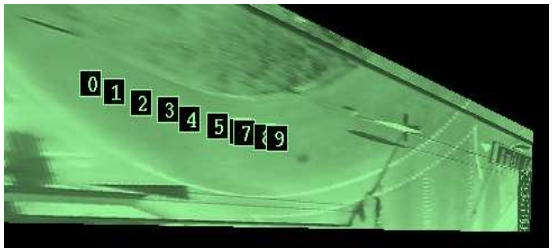
(c) Right



(d) Crossing



(e) Enter



(f) Changing

Figure 3.5: The point-per-point specification of each event as seen from above.

event→ dataset↓	Nothing	Upper	Left	Right	Crossing	Enter	Changing
00000	1 842 487	1 848 341	1 838 036	1 842 489	1 862 883	1 837 519	1 844 196
00207	1 806 240	1 818 605	1 814 821	1 805 441	1 821 152	1 818 174	1 815 395
00620	1 890 579	1 887 861	1 894 487	1 891 545	1 897 233	1 922 693	1 893 229

Table 3.1: Cost in bits of each image sequence, given each event. Each column is a event. Each row is a dataset. The test is positive for a particular event and dataset when its cost is below the cost of the event Nothing with the same dataset. Correct positives are underlined, false negatives are overlined. All other entries are correct negatives. The size of each image sequence, under no compression, when the image size is already known, is 5 242 880 bits. The size after Huffman compression, which is optimal under the simplistic assumption that all pixel values are independent, is 5 068 984, 5 107 232 and 5 222 056 bits respectively.

monochrome. The images were filmed and converted to video. Each sequence consists of 10 images, taken in steps of 0.24 seconds, that is, one frame out of six of a typical 25 frames/second video sequence.

The camera calibration was provided with the data. It was obtained using a method similar to the calibration described in page 54.

Test of events

A variety of events as described above can occur in a particular data set. The events are tested by comparing their cost to the cost of the event Nothing. If the cost of the event is higher or equal than the cost of Nothing, the test is negative. Otherwise, the test is positive. For example, testing the event Crossing with the dataset 00000 gives higher cost than the event Nothing with the same dataset, therefore the test is negative.

Performance

The performance of the event detector is shown on Table 3.1. Each row contains the results for an image sequence. Each column is an event. A human observer established the ground truth, namely, which events happened in each sequence. Underlined entries are correct positives, that is, their cost is below the cost of the event Nothing, and they were detected by the human observer. The overlined entry is a false negative, that is, its cost is above the cost of the event Nothing, and therefore the test is negative, but the human observer detected the event. There are no false positives. The rest of the entries are correct negatives.

Compression algorithm

The *general purpose* compression method used to compress the difference images is the program `gzip` [lGA02], which implements the Lempel-Ziv-Welch (LZW) method [LZ76, ZL77, Wel84]. This compression method is used in wide variety of compression applications, including text, source code, log files, etc. LZW is the compression method used in the GIF image format.

The LZW compression method has been used in these experiments because it makes very few assumptions on the data, namely that it is composed by repeated strings of

symbols of arbitrary length. This choice contrasts with a compression in the frequency domain, such as JPEG, designed for lossless compression of natural images. The differences in compression achieved with `gzip` are enough to successfully choose between the models used in these experiments.

3.3.3 Discussion of 2D models

The method presented uses models composed by

- two dimensional models for vehicles on the image plane,
- a rough camera calibration, and
- trajectories defined point by point on the ground plane.

This method is fast, simple, and can cope with occlusions. Although it uses all grey levels, the only transformations on pixel values are translation and scaling on the image plane. Occlusions are handled by the use of camera calibration, so that the base of each vehicle rectangle or *sprite* is located on the ground plane.

An advantage of this method with respect to bottom-up approaches is that a single events can be detected from a mixture of events. For example, considering events A, B and N (the null event), given an image sequence in which the events A and B take place, the test for event B will be positive, even if the event A is not being considered (and the cost function for the event A not evaluated),

$$\varphi(B) < \varphi(N)$$

where φ is the cost function.

For example, this method can give a positive test for more than one event if both events have a cost lower than the null event. That would be the case for the events A and B above,

$$\varphi(A) < \varphi(N) \tag{3.1}$$
$$\varphi(B) < \varphi(N) \tag{3.2}$$

We have so far implemented the function `render` of Figure 3.2 for 2D models. The resulting cost function has been used to detect simple events. Next section will implement `render` for 3D shape models and more detailed trajectories.

3.4 3D shape and trajectory models

This section presents an algorithm for the `render` function specified in Section 3.2, that uses 3D models for the shape of vehicles. The shape models are facets in the 3D space. For each image, the trajectory model, which consists of a position and orientation of the object, induces a rotation and translation on the shape model. The result is a set of facets located in the space for each image. There is a 1-1 map between the projection of the facets on two different images.

The main questions in the design of this function is which geometric shape to choose for the facets, and how to use the contents of the facets in order to create the synthetic image sequence.

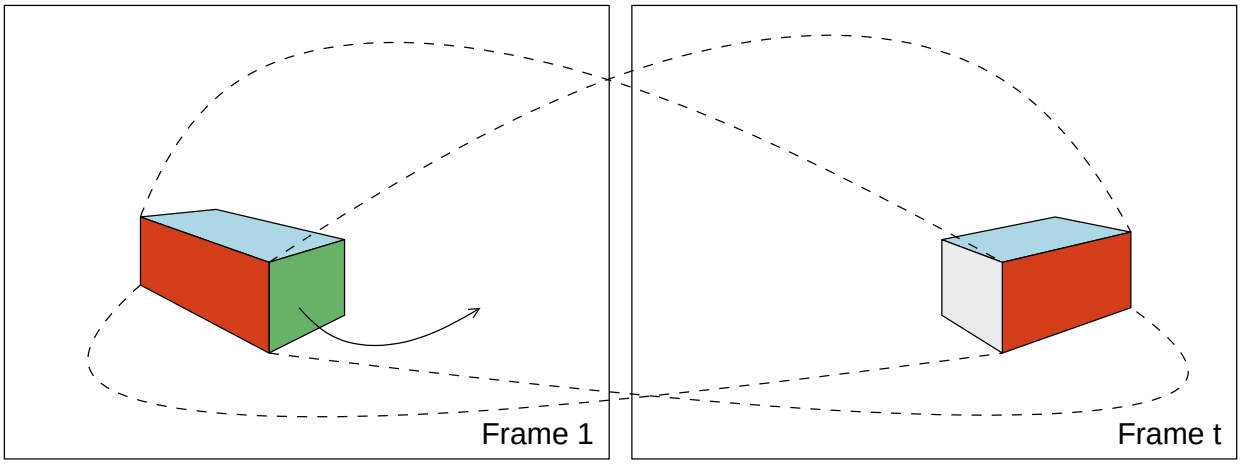


Figure 3.6:

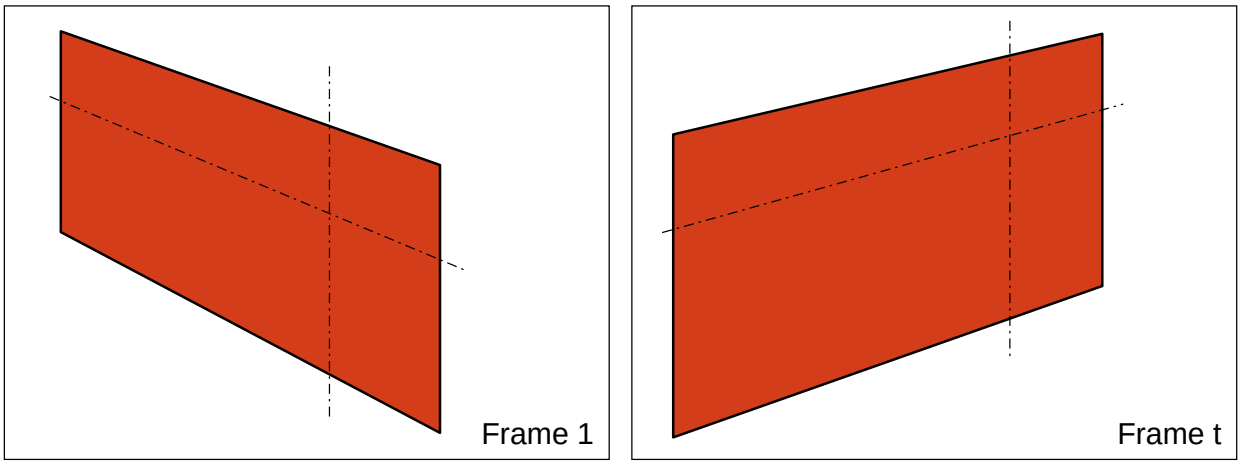


Figure 3.7:

3.4.1 Compressing a sequence of images using 3D models

Let us remember that the central idea of this chapter is to describe the data with the information included in the model, and use the description length as a cost. The more information the model contains about the data, the smaller the error between the model and the data.

The models being considered consist of the shape of the vehicle and the trajectory of the vehicle over time, given as the position and orientation of the vehicle on the ground plane, image by image. Therefore the models are geometric: a model induces a geometric description of the scene for each image. An additional consequence is that it is possible to create, using computer graphics methods, a *synthetic* version of the original image sequence, using only the model.

A simple description of the image sequence, `data`, consists of the `model` and the difference, pixel by pixel, between the synthetic image sequence, generated with the model, and the original image sequence, `data`. This difference can be considered as the *error* between the model and the data. The model, and that error, are enough to

reconstruct the original image sequence, `data`.

But a simple wire-frame geometric model (consisting only of edge segments but no intensity values, as in [Sul94]), is not enough to produce a realistic synthetic version of the data that can be subtracted from the original data. More information is needed, namely, an approximation to the facet intensity levels that form the texture of the geometric model.

The method proposed here is to take the texture from the first and last images of the image sequence. It is expected, as it happens in the data used for the experiments of this chapter, that all facets visible at any point of the image sequence can be obtained from the first and last images. Therefore, the first and last images become additional information used to create the synthetic image sequence.

Figure 3.6 displays a projective transformation between two quadrilaterals, corresponding to the same facet of the same model, but in different positions on the ground plane, in different images.

Now the question is which shape to choose for the facets of the 3D shape. The simplest shapes are triangles, extensively used in computer graphics. Another option is to use quadrilaterals.

Since the projective transformations between the images are, in general, homographies [SK52, FvDFH90], it seems reasonable to choose as facet something that can be mapped using homographies easily into any other facet. Quadrilaterals have the property that, given two quadrilaterals p and q (as an ordered list of vertices) in different images, in the image plane, there is a single homography that maps each pixel of q into a pixel of p. This is valid either in 3D coordinates, or in the image plane.

The shape model s is a list of quadrilaterals $\{q_1, \ldots, q_n\}$, in the 3D space. The centre of rotation of the object defined by the quadrilaterals should be located at $(0, 0, 0)$, so that rotation does not require a previous translation. In particular, the centre of gravity of the shape model of a vehicle is be located at $(0, 0, 0)$.

The trajectory model is a list of pairs (position, orientation), (p_t, θ_t), for each image $t = 1, \ldots, r$. Each p_t is the position of the vehicle on the ground plane. Each θ_t specifies the orientation of the vehicle in the ground plane. Let $M(p, \theta)$ be the matrix that corresponds to the rotation θ followed by the translation p. The matrix $M(p, \theta)$ represents a rigid motion.

The following steps yield the synthetic image sequence.

1. Apply $M(p_t, \theta_t)$ to each corner of the shape s for $t = 1, \ldots, r$. The result is the 3D shape s_t, moved to its 3D location at time t. To do so, represent each quadrilateral q_j as matrix whose column vectors are the quadrilateral's corners in the 3D space, then let $q_{tj} = M(p_t, \theta_t) \cdot q_j$ for each quadrilateral $j = 1, \ldots, n$.

2. Let q'_{tj} be the projection of quadrilateral q_{tj} on the image plane, calculated using the camera calibration, for each time step $t = 1, \ldots, r$ and each quadrilateral $j = 1, \ldots, n$.

3. Calculate the *visibility maps* $V_1, \ldots, V_r$. Each pixel in image V_t is the number of quadrilateral visible at the same pixel coordinates in image t, which is calculated according to the distance from the quadrilaterals to the camera. The visibility maps $V_1, \ldots, V_r$ will be used to handle occlusions.

4. Calculate the homography H'_{1tj} from q'_{tj} to q'_{1j}, and the homography H'_{rtj} from q'_{tj} to q'_{rj}. These are the pixel-by-pixel correspondences between image t and the reference images 1 and r.

5. Now we are ready to calculate the synthetic image sequence pixel by pixel. For each frame $t = 2, \ldots, r-1$

   - For each pixel p
     (a) Let q_j be the quadrilateral of s to which p belongs and that is visible in frame t. That is, j is the value of the pixel p in the visibility map V_t.
     (b) Let $p_i = H'_{itj} \cdot p$ for $i = 1, r$ be the coordinates of the pixels that correspond to p in the reference frames. We want to extract the value of p from the values of p_1 and p_2.
     (c) Consider the list of two elements $L = \{p_1, p_r\}$. Both p_1 and p_r belong to the respective projections of q_j in their frames. But they might not be visible. This is the case when the value of p_i in the visibility map V_i is not j. In that case remove element p_i from L. We will interpolate the value of p from the remaining elements in the list L.
     (d) For each remaining element p_i in the list L do the following: The coordinates of p_i are not integers, therefore use bilinear interpolation between the integer neighbours of the pixel with coordinates p_i to estimate the value of the pixel p_i in the reference frame i.
     (e) Interpolate from the values of the pixels in L.

In this way the pixel values are projected from the reference images 1 and r to each image $t = 2, \ldots, r-1$.

This definition of the function `render` together with the function `cost` of Figure 3.2 was implemented in C in order to run the following experiments.

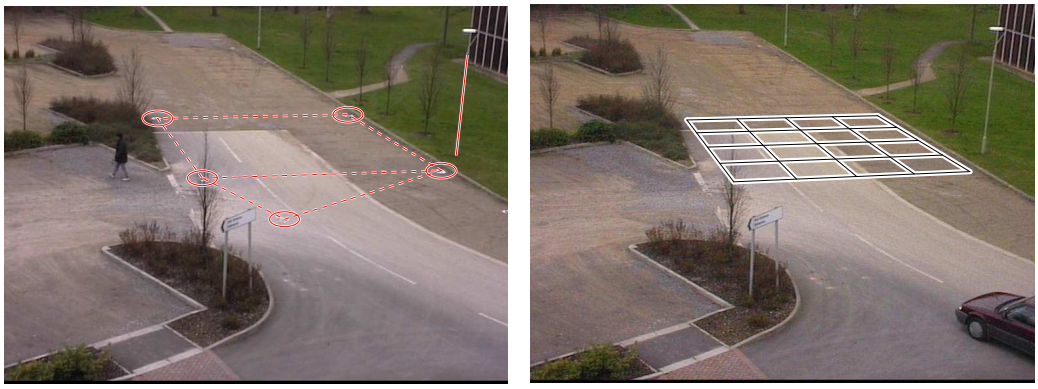
3.4.2 Experiments

The purpose of the experiments is to evaluate the MDL-based cost function `cost`. The trajectories used are relatively similar to each other: they all have the same starting point, and their end points are within a range of a few meters on the ground plane.

Setup

The data used in this experiments consist of short image sequences in which several vehicles are moving at the same time. The scenario is a car park. The camera is fixed on a building overlooking the car park. The camera was fully calibrated.

Camera calibration. Full camera calibration was obtained from measurements on site. The camera was fixed and the focal length constant, therefore the calibration was constant. The camera is modelled as a pinhole ([SHB99, page 456],[FvDFH90, Fau93]). Anecdotal evidence shows no need for a more complex model that includes radial distortion, or any accounting for a difference between the centre of projection and the centre of the image.



(a) The calibration marks: four calibration points forming a rectangle, an additional point with known position, and a vertical line

(b) The homography between the ground plane and the image plane is computed.

Figure 3.8: Camera calibration using measurements on the road.

The camera calibration was obtained using the software described in [WSB94]. This software has a visual interface to help find the transformation from world coordinates to image coordinates. The primary calibration data are

- the positions of four points on the ground plane, forming a rectangle, and
- the image of a line known to be vertical to the ground plane.

Additional calibration data were the height of the camera and the position of a fifth point on the ground plane, used for redundancy checks. Figure 3.8(a) illustrates these measurements. The user of the calibration software described in [WSB94] can display different grids overlaid on the calibration image, and move the grids in ways that correspond to changes in the projective transformation. The measurement unit is the centimetre. A consequence is that measurements are integer numbers, which will facilitate the encoding of the measurements on the ground plane in the remaining chapters of this thesis. The origin of coordinates of the world reference frame, together with the first two axes, form the ground plane in which vehicles move. In particular, the homography between the ground plane and the image plane was obtained (See Figure 3.8(b)).

Dataset. The images used for the experiments presented here are those of the PETS2001 dataset [Fer01]. These image sequences consist of 25 images per second of colour video at PAL resolution (768×576 pixels) encoded using JPEG lossy compression. The image sequences were filmed using a Canon MV1 digital camera. For the following experiments, the images were converted into eight bit greyscale and reduced to a quarter size (384×288 pixels) using the library PBMPlus [Pos02]. Thus, each image occupies 110592 bytes, assuming known image size and one byte per pixel.

Software. The software used for this experiment consists of 4000 lines of code written by the author in object oriented C, plus a library for numerical computations together with standard UNIX tools.

The shape model. The shape model is a cuboid, as shown in Figure 3.6. Although the software can render complex shapes made up of quadrangular facets, there are several reasons for using a simple cuboid model:

- Most image pixels belonging to the moving vehicle are included in the projection of the cuboid.
- The number of parameters of a cuboid is small (three integers: width, length, height) as opposed to a more detailed model such as those used by Ferryman and others [FWSB97], which requires six parameters even after Principal Component Analysis reduction of the number of parameters.
- There is no need to fit closely an edge model to the vehicle, as it is required in [FWSB97] and other methods based on wire-frame models.

(In [FM99a] the author studied the feasibility of extracting shape models from images displaying a vehicle in different positions. An estimate of the height of each vertex above the ground plane was used, in combination with the camera calibration, to obtain the 3D position of each vertex. Vertexes were combined into facets to produce the shape model which could be displayed, and corrected, in different positions on the ground plane. The images were separate frames of the image sequence. But the increased model complexity, in terms of number of facets, was not considered necessary, and therefore this method has not been used to produce the models of the following experiments).

The trajectory model. The trajectory is the variable in these experiments. In a first approach, a trajectory is defined as a sequence of points on the ground plane, one per step of the image sequence. For the purpose of these experiments, a trajectory is defined by giving the starting point in the first step and the end point in the last step. The points on the trajectory in the intermediate images are calculated under the assumption that the velocity is constant. The orientation at each step is interpolated from the orientation in the first and last steps.

Visualising the cost. Plots like Figure 3.10 show the cost function evaluated in an array of parameters. Instead of plotting the value of the cost function $\varphi(x, y)$, its negative $-\varphi(x, y)$ is displayed. Therefore a minimum in φ is displayed as a maximum of $-\varphi$, in the plot. This transformation is motivated by the fact that, viewed from above, maxima are easier to see than minima due to better contrast against the background.

The compression algorithm. The difference images are compressed using the Lempel-Ziv-Welch (LZW) method, which was discussed on page 50.

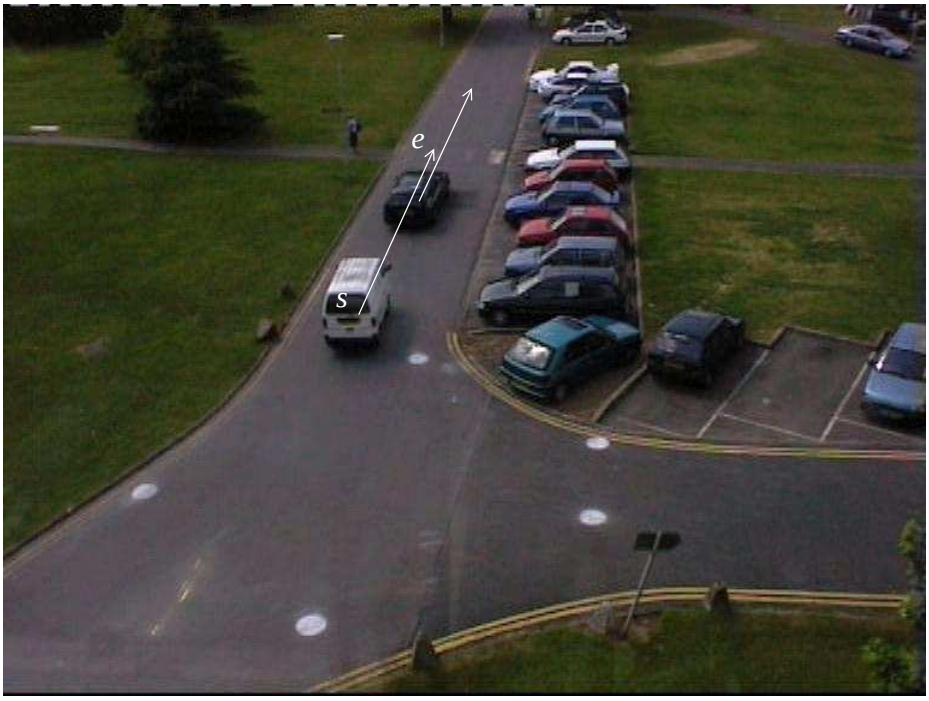


Figure 3.9: An image from the data of Experiment 1: Two vehicles moving with constant velocity in the same direction. The white van moves from s to e.

Experiment 1: Vehicles with constant velocity

In order to show that the cost function reaches a minimum at the correct trajectory, a family of trajectories was defined for a fixed constant starting point $s = (s_x, s_y)$ and a variable end point $e = (e_x + i\delta, e_y + j\delta)$ where δ corresponds to one metre, and i and j take the values $-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5$.

$$e_{ij} = (e_x + i\delta, e_y + j\delta) \tag{3.3}$$

The data of the experiments is a sequence that, to the human eye, shows a vehicle moving with constant velocity in straight line along the road, and this trajectory (s, e_0) is included in the set of models under consideration. The experiments are successful in that the minimum of the cost function is attained for the model very close to the trajectory (s, e_0).

The first image sequence, consists of ten images with timestamp increments of 7/25 seconds. It contains two vehicles moving with constant velocity, in the same direction. An image of the sequence is shown in Figure 3.9. where the ground truth for the white van's trajectory is a straight line.

Fitting. The cost function is evaluated for each of the 11×11=121 trajectories defined between the start point s in Figure 3.9, and the end points $\{e_{ij}\}_{i,j=-5}^{5}$. The trajectory with lowest cost is displayed in Figure 3.11.

The surface formed by the cost function on the 121 trajectories is shown in Figure 3.10.

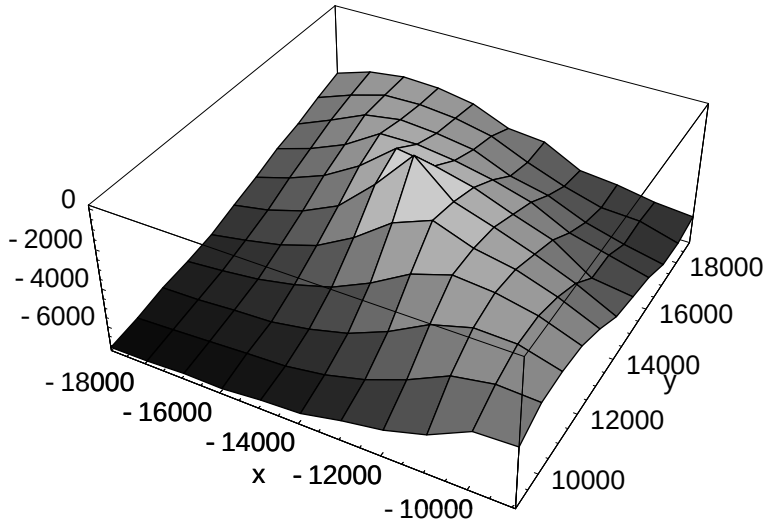


Figure 3.10: The (negative of the) cost function. The best trajectory (maximum in this plot) is displayed in Figure 3.11

Experiment 2: when initialisation is wrong

A common measure of the error between two sequences of points on the plane $p_1, \ldots, p_r$ and $q_1, \ldots, q_n$ is the sum of the square of their distances,

$$\sum_{i=1}^{r} \| p_i - q_i \|^2$$

for a norm $\| \cdot \|$. This error measure can easily be affected by outliers [Rou84]. The presence of outliers can be modelled be either a vehicle moving nearby, or by a wrong initialisation.

This experiment is performed with the same image sequence as Experiment 1. The parameters are also the same, except that a wrong starting point w is used instead of s as shown in Figure 3.12.

The main consequence of using the wrong starting point is that the *ground truth* trajectory is not in the search space.

Before looking at the results of the experiment, let us consider which could be the possible outcomes:

1. Either the correct end point $e = e_{00}$ is found, that is, the trajectory found with this method, and the ground truth, cross at their end points, trajectory joining w and e in Figure 3.12, or

2. the trajectory found crosses (s, e_{00}) at some point. For example, the trajectory joining w and d in Figure 3.12, or

3. the trajectory found does not cross the ground truth (s, e_{00}).

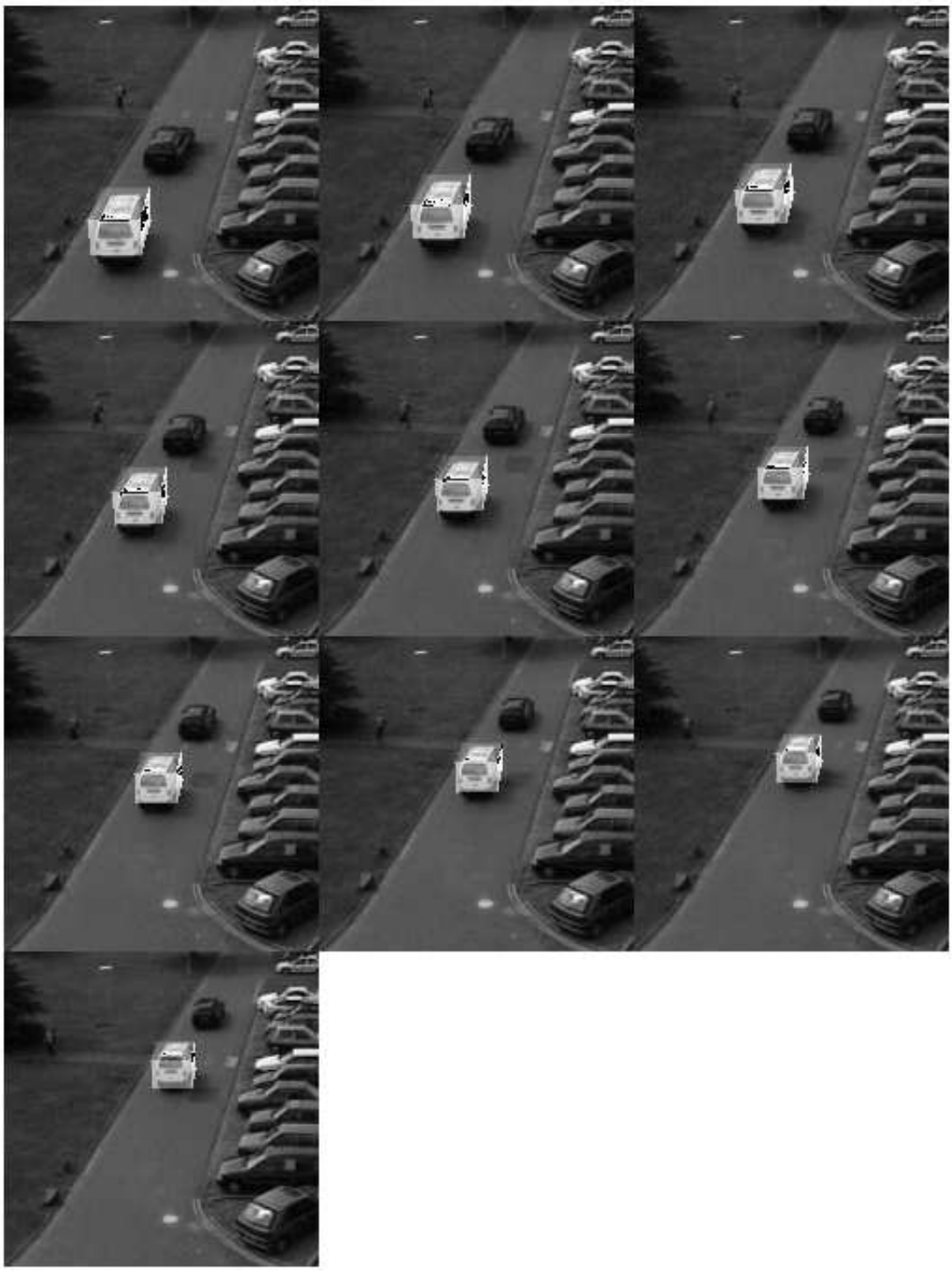


Figure 3.11: Sequence of images showing two vehicles moving on the same direction with constant velocity. The cuboid model represents the best fit according to the cost function. (Excerpts from the original images)

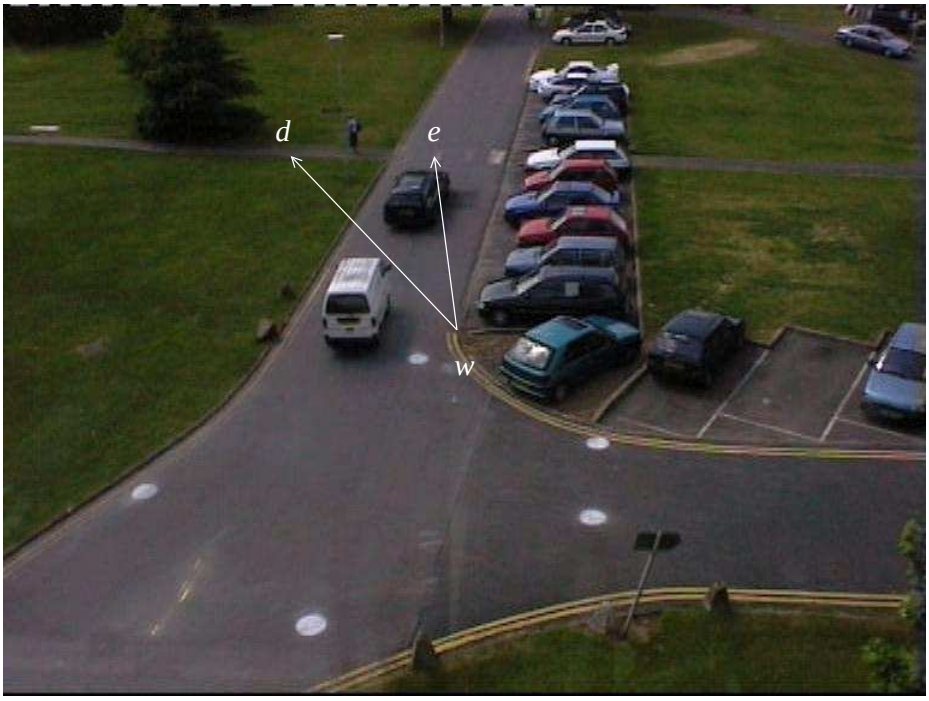


Figure 3.12: An image of the data of Experiment 2: The *wrong* starting point w. Trajectory (w, d) crosses the ground truth. Trajectory (w, e) ends at the same point as the ground truth.

For the current application the first case (1) is preferable since it would convey most useful information about the real trajectory, namely its approximate direction and end point, thus allowing for an iterative algorithm to *catch up* with the moving vehicle, and recover from the initialisation error.

The result of the experiment is in fact case 1 (as above), the end point with minimal cost is the same as for the ground truth. Figure 3.13 shows the cost function evaluated in the area. Figure 3.14 shows the trajectory corresponding to the minimum cost.

Experiment 3: Curved trajectories

Curved trajectories, for this experiment, are modelled as trajectories with constant speed in which the orientation of the shape model, at each point, is linearly interpolated between the orientation in the first point and the last point of the trajectory. Chapter 5 will discuss a more realistic trajectory model.

The consequences of using linear interpolation for the orientation are:

- the interpolation algorithm is the same as for the positions of the vehicle in the first images
- the resulting trajectory assumes that the vehicle slides sideways, at this scale, a minor deviation from the physics that govern the motion of the vehicle.

Figure 3.15 shows the trajectories of the two vehicles reversing and rotating at the same time.

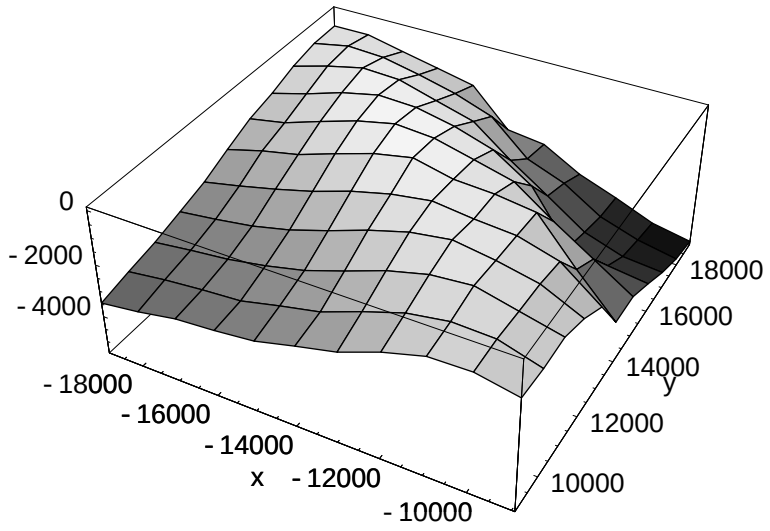


Figure 3.13: The (negative of the) cost function. The best trajectory (maximum in this plot) is displayed in Figure 3.14. The maximum of this plot is located in the centre.

The image sequence, called T, consists of 10 images, in which one image was taken out of 15 from the original image sequence. Thus, approximately 1.7 images per second were used.

The models are 121 trajectories (s, e_{ij}) for $i, j = -5, -4, \ldots, 4, 5$ (see Equation 3.3). Again the points e_{ij} and e_{i+1j} are separated by one metre.

Figure 3.16 is the cost function. The trajectory with lowest cost is pictured in Figure 3.17.

Discussion. The relevance of a cost function for trajectories is in the location of the minimum, and how easy is to reach it. This limited set of experiments is enough to illustrate the shape of the cost function, which shows a clear minimum near the ground truth (the centre of the grid in each plot). The shape of the cost function is smooth.

This method is not comparable to a working vehicle tracker since it does not operate in an image-by-image basis, and does not include the optimisation algorithm. Dynamic trajectory models, such as the Kalman Filter [Ger99], which is the standard model for vehicle tracking, are usually incorporated as part of the minimisation algorithm. We will include vehicle dynamics in the model in Chapter 5.

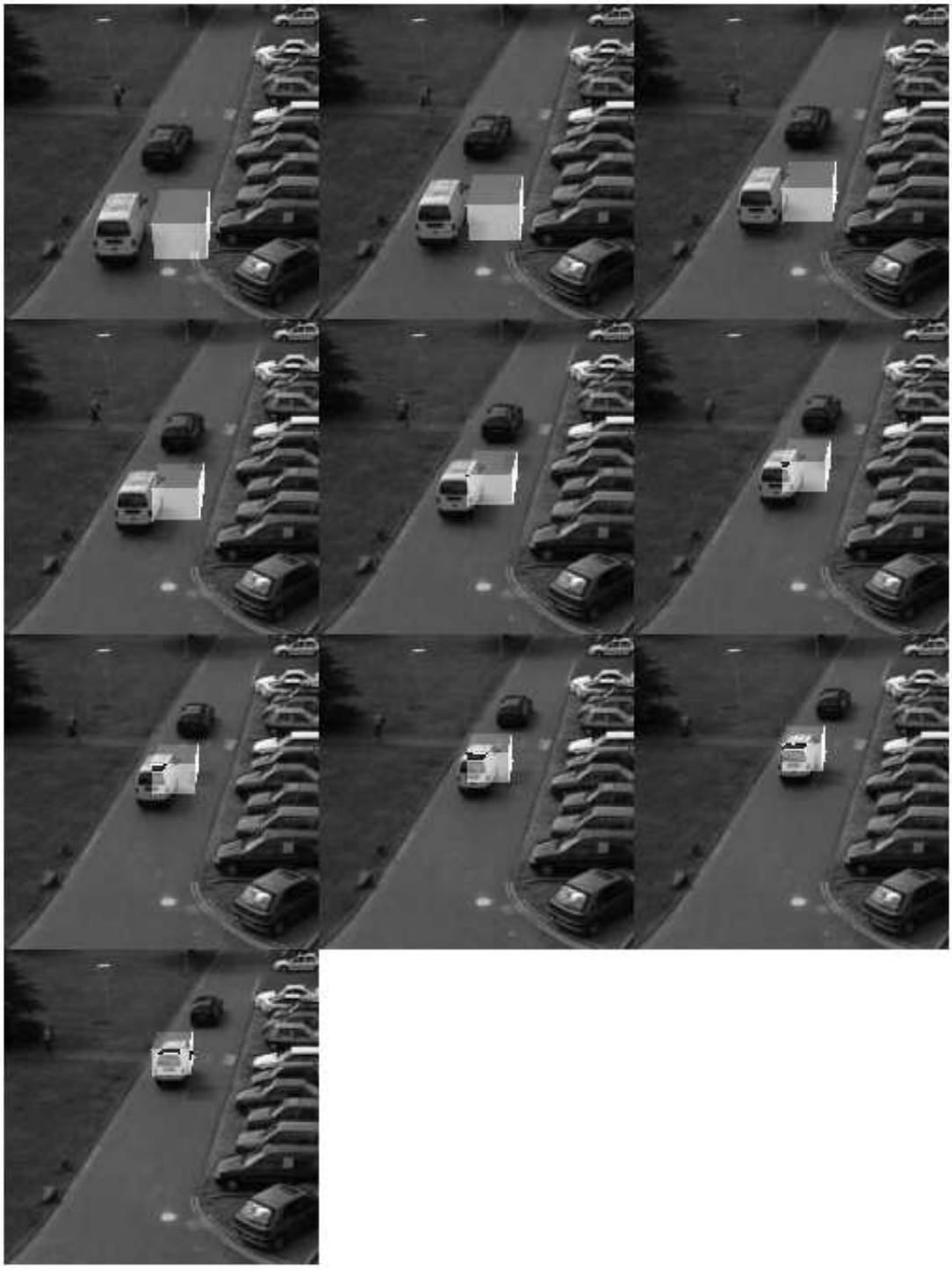


Figure 3.14: Sequence of images showing the trajectory with lowest cost in Figure 3.13. (Excerpts from the original images)

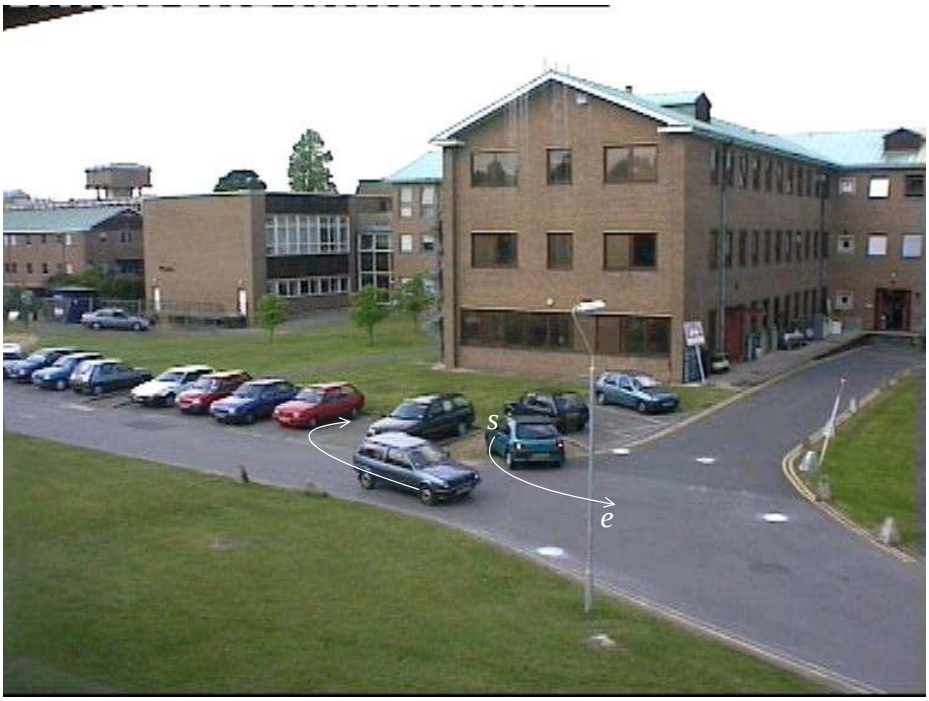


Figure 3.15: Sequence T. Two vehicles are moving at the same time. The vehicle on the left hand side is parking, while the vehicle on the right hand side is leaving a parking slot.

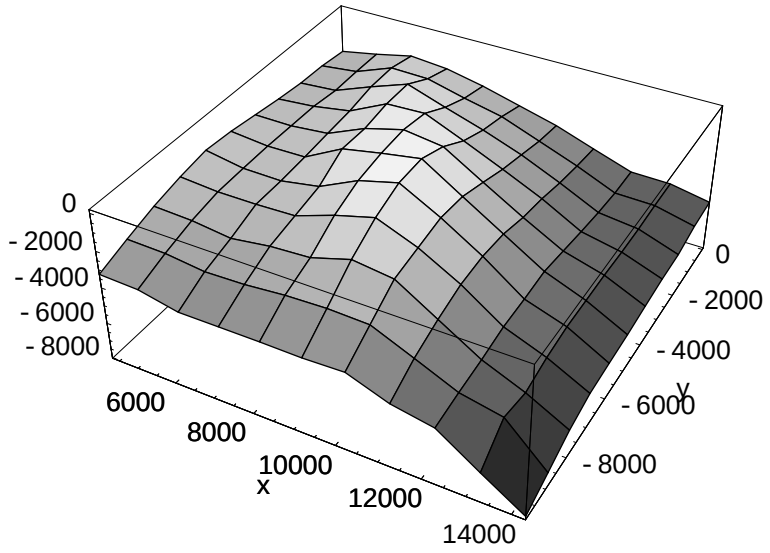


Figure 3.16: The (negative of the) cost function The best trajectory (maximum in this plot) is displayed in Figure 3.17

Figure 3.17: Sequence of images showing the trajectory with lowest cost in Figure 3.16.(Excerpts from the original images)

3.5 Conclusion

Application to 3D tracking. An application of the MDL method to vehicle tracking has been presented. The approach proposed is conceptually simple, albeit slow in its current implementation. Its main characteristics are:

- A 3D graphics engine has been used to interpolate a synthetic image sequence from part of the sequence.
- The error model implied is an approximation of the universal measure of Chapter 2, since it is an application of the MDL method.
- The compression algorithm makes few assumptions about the data. The general purpose data compression algorithm `gzip` compresses the difference between the synthetic image sequence and the original data. It is based on the LZW method, the *de facto* standard in general purpose data compression.

Alternative methods in this area of research typically use fast but conceptually complex algorithms. In particular, among model-based approaches, models based on edges depend on arbitrary thresholds and rely on exact matches. The method presented in this chapter does not require an exact match between the model and the image.

Application to 2D event detection. Besides, simple event detector, based on the same principle and 2D models, has been used to successfully discriminate between different vehicle trajectories in a roundabout.

In both 2D and 3D cases a simple shape model was good enough for the application. The trajectory models were overly simple, and did not contain prior knowledge about the dynamics of vehicles in motion.

The next step is to develop a more realistic trajectory model. We will do this in Chapter 5 within an application of the MDL method. But, before that, we will take in Chapter 4 a closer look at the way computable approximations to the MDL method can be defined.

Chapter 4

Computable approximations to MDL

4.1 Introduction

In the previous chapters we have seen how a problem in Computer Vision can be solved using data compression. The models that better explain the image data are the models that help compress the data more.

In Chapter 2 we saw that the *optimal* compression function $\mathbf{C}$ of Definition 2.9, which is used by the MDL method, is not computable. We need to define computable approximations to $\mathbf{C}$. In Chapter 3 a hands-on approach with models and experimental data was taken to define computer approximations C, tailored to a particular class of models and data, but based on a standard data compression algorithm.

This chapter defines a compression function `Compress` that approximates the optimal compression function $\mathbf{C}$. The function `Compress` is grouped together with other useful functions in a package called `Thesis‘Compress‘`. These additional functions perform decompression, verification, and extend the definition of `Compress` for different data types.

Defining a computable approximation to $\mathbf{C}$ consists in constructing a computer program. This chapter presents a definition of `Compress` general enough to include the non-computable functions used in the MDL method, together with a wide variety of computable approximations; this is done in Sections 4.3 and 4.4. Then, in Section 4.5, the function `Compress` is defined for some basic data types such as natural numbers, integers, lists and vectors.

A general procedure to define other computable approximations `Compress` is given in Section 4.6.

The implementation presented in this chapter is written in the computer language `Mathematica`. We start with an introduction to this computer language in the following Section.

4.2 The programming language

The computer language used in this chapter is `Mathematica`. In this section the language is introduced. Further details of `Mathematica` can be found in the language manual Wolfram [Wol99] and in [Mae89, Mae96]. A critique of some features of the language can be found in [Fat92].

The name `Mathematica` is used commercially to refer to both a system for scientific computation and the programming language that supports it. In this chapter we will use the term `Mathematica` to mean the programming language. `Mathematica` has a simple syntax to represent a variety of data structures and other mathematical objects as *symbolic expressions*, and the relations between them as *rewrite rules*.

Symbolic expressions or simply "expressions", are Lisp-style lists [Knu98] in which the first element or "head" has a special meaning. The head is used, for example, to identify a function, or tag a data structure. Expressions in `Mathematica` either are *atoms* or have the form

$$b = b_0[b_1, b_2, \ldots, b_n]$$

where each b_j is an expression, a *subexpression* of b. The *head* of $b_0[\ldots]$ is b_0. An *atom* is identified by having the head `Symbol`, and is distinguished by not having subexpressions.

Rewrite rules. Programs are transformation rules defined for expressions. Execution of a program or expression is called *evaluation*, and consists of applying the transformation rules until the expression does not change [DJ90]. The final expression is said to be *reduced*. This is similar to the notion of graph reduction for functional languages [Jon87].

There is an interesting similarity between *reduction* of expressions, and *reducing* the size of a description as we have done in Chapter 2. Both uses of the term *reduce* are different, but we make them coincide when a compression function is implemented as a set of rewrite rules.

The implementation described in this chapter uses special features foreign to popular imperative languages such as C and C++. This section describes the subset of `Mathematica`'s syntax and semantics that we will need in this chapter. There follows a discussion on the convenience of using `Mathematica` expression syntax to represent data and models. The section concludes with a summary of the advantages of `Mathematica` for the implementation of computable approximations to the MDL compression function **C**.

Notation for the program examples. The examples of this chapter have been generated directly by `Mathematica`, in order to guarantee correctness. The typed text of this thesis contains only the lines beginning with `In[...]`.

```
In[line number]:= input expression
```

A `Mathematica` interpreter added the output of each evaluation, the lines beginning with `Out[...]`.

```
Out[line number]= output expression
```

4.2.1 Language properties

`Mathematica` implements several programming paradigms, including [Fat92, Mae96]

- Functional programming, like Miranda.

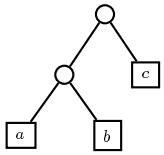


Figure 4.1: The tree {{a,b},c}.

- Imperative programming, like C, C++, Java and Pascal.
- List processing, like Lisp.
- Term rewriting or rule-based programming [DJ90].

Some properties of `Mathematica` make it particularly useful for our application:

- Use of symbolic expressions to represent both data and programs. Facilities for higher order functions (functions whose parameters are other functions).
- Weakly typed, in the sense that data types are not enforced by the syntax. At most, the head of an expression is used to represent its contents. We will use heads to store *models*.
- Polymorphism, as a consequence of weak types and higher order functions. This will facilitate code reuse.
- It is very simple to cache the result of computations, therefore speeding up computations which are repeated many times for the same input.

A function is *polymorphic* when a single definition works with parameters of different types [Jon87]. For example, an addition function `Plus` can be defined for numbers, and then for a vector space defined on those numbers. The function `Plus` would then be polymorphic. Both instances of the definition share, for example, the property of commutativity.

4.2.2 Expressions

Expressions can be used to represent a wide range of data structures, functions, operands, etc. Section 2.2.1 of [Wol99] describes how a symbolic expression can be used to represent functions, commands, operators and object types, according to the meaning of the expression's head. Our main uses are

Function for example, `f[x]` is used to represent the function `f` applied to the parameter `x`.

Data structure for example, `RGBColor[.2,.2,.5]` means the colour with given components red, green and blue, and, in this chapter, `Vector[x, y, z]` means the vector with components `x`, `y`, `z`. A tree can be represented as nested lists, such as `List[ List[ a, b ], c]`, also written as `{{a,b},c}`, as in Figure 4.1

Data type for example `Natural[n]` in this chapter means a natural number `n`> 0.

There is a special function, `Head`, that returns the head of a symbolic expression.

4.2.3 Functions

We can identify evaluation with computation. The value of an expression α is the expression that results from the evaluation of α. `Mathematica` processes expressions by applying transformation rules to parts of the expressions. In `Mathematica` syntax, a function definition

```
In[1]:= f[x_] := x + 1;
```

is in fact a rewrite rule which, when evaluated, will transform any expression of the form `f[`α`]` into $\alpha + 1$, and, if α is a number, will evaluate $\alpha + 1$.

```
In[2]:= f[a]
Out[2]= 1 + a
In[3]:= f[3]
Out[3]= 4
```

The underscore `_` in `x_` means that `x` is a variable, in the usual interpretation of functions, or a pattern to match, in `Mathematica`'s way of understanding function application. The symbol `:=` is used for function definition, as in α`:=`β. It can also be interpreted as a *delayed rule*, a rule that is applied only when an expression is found that matches the left hand side α.

4.2.4 Models are heads of expressions

Data, descriptions, models. In Chapter 2 we fixed a reference Universal Turing Machine U and defined description of data d as any program p that outputs d (Definition 2.8):

$$d = U(p)$$

A model was defined as any Turing machine M that, given suitable input e, would generate d (Definition 2.7):

$$d = M(e)$$

Since U was universal, there is a string m such that $M(x) = U(mx)$, for any string x, and therefore

$$d = U(me)$$

Conversely $U(m)$ is a Turing machine for any string m. Therefore a model of d is any prefix of any description of d.

Now the question is how to implement models in terms of symbolic expressions. The proposed answer is to identify a model with the head b_0 of an expression $b_0[b_1, b_2, \ldots, b_n]$.

The identification between heads of expressions and models is natural. In `Mathematica`, as in Lisp, a function f with arguments $b_1, \ldots, b_n$ is represented by an expression with head f and body $b_1, \ldots, b_n$. Thus, thinking of f and b_j, for all j, as binary strings, the concatenation of those binary strings yields a model as in Definition 2.7.

A consequence is that concatenation of binary strings translates into composition of symbolic expressions. Thus, in the notation of Chapter 2, a binary string d with description me where m is a model of d, corresponds to a symbolic expression `d`, with

a description `m[e]` (a program that, when executed, outputs `d`). Again, the model `m` is the head of the expression `m[e]`.

The left hand side pattern `x_` in a function definition can be a single variable such as

```
f[x_] := x + 1
```

or a compound expression with several variables. For example, the function

```
In[4]:= f[(x_)[y_]] := x + y;
```

is defined for two variables, the head `x` and the body `y`.

```
In[5]:= f[3[4]]
Out[5]= 7
In[6]:= f[a[b]]
Out[6]= a + b
```

so long as `a` and `b` are left undefined. The MDL method in Section 4.4 provides an example in which integer numbers are modelled with other integer numbers.

4.2.5 Advantages

The advantages of using `Mathematica` for the implementation in this chapter and Chapter 5 are summarised here:

- Conciseness: the most important definitions fit in one or two lines.
- Models are heads of expressions: the head of an expression can be interpreted as the model of the rest of the expression, or as a function that applies to the rest of the expression. This corresponds to the definition of model given in Chapter 2.
- Polymorphism: a definition can be given in abstract terms, relying only on certain properties of the data, but not on the data type. Thus the same definition can be used in different contexts.

4.2.6 Summary of `Mathematica`'s notation

Although ultimately everything in `Mathematica` can be represented as an expression, the notation used can be very idiosyncratic and topped with a large amount *syntactic sugar* (which is notation intended to make the language more readable for human beings [How]). Most of the `Mathematica` expressions used in this chapter are one-liners. The following list summarises the prerequisites to understand them:

- Square brackets are used to separate the name of a function (the *head*) from its arguments, as in `Sin[x]` and the type of a data structure (the *head*, again), from its components, as in `BinaryString[0,1,1,0]`.
- A function is defined using "`:=`". The arguments are denoted on the left hand side by variable names followed by underscore, as in `f[x_]:=x+1`. The value replaces the variable name on the right hand side.

- The function `Head[`α`]` returns the head of the expression α.
- The command `Needs` is used to load software packages.
- Other operators and functions are explained wherever they appear in the text.

4.3 The cost function

Our purpose is to implement the MDL method by defining computable approximations to the function **C**, not in the domain of binary strings, but in the domain of `Mathematica`'s expressions. In this section we will define a cost function for an expression, named `Cost`. The value of `Cost[`α`]` will be an approximation of the Kolmogorov Complexity of the expression α, defined as the Kolmogorov Complexity of a binary string that encodes α.

The objective is to implement a cost function `Cost` that depends on a compression function `Compress`. The function `Compress` corresponds to a compression function C of Chapter 2. When `Compress` is the non-computable optimal compressor **C**, then the cost will be the Kolmogorov Complexity. But, since `Compress` must be computable, the `Cost` will be a computable approximation of the Kolmogorov Complexity. The intention is to allow `Compress` to get arbitrarily close to the optimal compressor, so that the cost function can be arbitrarily close to the Kolmogorov Complexity given enough computational effort. The following definition provides the initial "frame" to which detail will be added in the following sections.

Definition 4.1 COST FUNCTION The cost function is defined, for an expression α, as the size, in bits, of the output by a function `Compress[`α`]`.

```
Cost[ α_ ] := Size[ Compress[ α ] ]
```

The function `Compress` must have the following properties:

- There exists a Universal Turing Machine `Decompress` so that `Compress` is the inverse of `Decompress` in its domain.
- The function `Size` is defined in the domain of the Universal Turing Machine `Decompress` (which by definition coincides with the range of the function `Compress`).

Note that the requirements in the functions `Compress` and `Size` are not very restrictive.

At this point we could define `Decompress` to be the `Mathematica` function `Evaluate`, which implements a Universal Turing Machine, and the function `Size` could be based on the `Mathematica` function `ByteCount`, which returns an upper estimate of the number of bits needed to store a particular expression.

But, seeking a closer control on the algorithms being used and the possibility to repeat experiments, and keeping the similarities with Chapter 2, we rather define `Decompress` and `Size` for binary strings instead of general expressions.

Binary strings. Let us consider expressions that represent binary strings. For this purpose, the head `BinaryString` will be used. Such expressions have the form

$$\texttt{BinaryString[}b_1\texttt{, }b_2\texttt{, ..., }b_n\texttt{]}$$

where b_j is a binary digit. The Universal Turing Machine U takes expressions of the form `BinaryString[...]` as input, and outputs expressions of any kind, not only expressions of the form `BinaryString[...]`.

Size. The `Size` function is defined as the length of the binary string. In `Mathematica` this can be implemented as

```
Size[expr_]:=Length[expr]
```

where `Length` is a built-in function that returns n given an expression of the form

$$b_0[b_1, b_2, \ldots, b_n]$$

If `Compress[`α`]` is defined to yield the shortest program that computes α, then `Cost[`α`]` is the *Kolmogorov Complexity* of α.

Since no computable definition of `Compress` will produce the shortest description of all expressions for any specific reference Universal Turing Machine, the value of the cost function will only be an upper bound of the Kolmogorov Complexity.

The degree in which `Cost` is a computable approximation to the Kolmogorov Complexity depends on how `Compress` is defined. From the point of view of program construction, we have so far defined the function `Cost` and we still have to define `Compress`; the possibilities include functions that give results arbitrarily close to the Kolmogorov Complexity.

4.4 The compression function

We have seen that the value of `Cost[`α`]` is an upper bound of the Kolmogorov Complexity of the expression α. By improving the function `Compress` to produce shorter descriptions, the upper bound is lowered towards the Kolmogorov Complexity.

We are free to define `Compress` in whichever way we like, so long as it holds the requirements of Definition 4.1. These requirements are that, when the reference Universal Turing machine is given `Compress[`α`]`, the output should be α.

The Minimum Description Length method, as we have seen in Definition 2.11, consists in defining the function `Compress` in two steps: first, compress the model, then compress the data given the model. In practice, this amounts to compressing the pair (model, data) in a way that, in a first step, decompression will yield the model, and further decompression, given the model, will yield the data.

This section proposes a definition of the function `Compress` following the two steps of the MDL method. The resulting compression function is a similar to the two-part descriptions used in Chapter 3. This section also specifies how the function `Decompress`, which is the reference Universal Turing Machine, should be defined, and how it relates to `Compress`.

4.4.1 Preliminaries

Use of Expressions

In Chapter 2 we have seen how our programs and data are binary strings that can be concatenated. The evaluation of program x, given a reference Universal machine U, with input y, is represented by $U(xy)$, that is, U applied to the concatenation of x and y.

In this chapter the building blocks are not binary strings, but expressions. Thus we need an equivalent operation to the concatenation of binary strings.

Definition 4.2 COMPOSITION OF EXPRESSIONS In the space of expressions `BinaryString[...]`, the composition of expressions is equivalent to the concatenation of the corresponding binary strings. The rule is written in `Mathematica` as follows:

```
BinaryString[a___][ BinaryString[b___] ] :=
   BinaryString[a]~Join~BinaryString[b]
]
```

Conditional description

In Chapter 2 we saw that the functions C_T are defined for any Turing machine T, not only for Universal Turing machines. Thus, if $d = T(e)$ for binary strings d and e, e is a description of d under T.

There is an equivalent notion of description for expressions. Let `data` be any expression, which corresponds to the binary string d, and `model` be the binary string m, which is of the form `BinaryString[...]`. A description of `data` under `model` is an expression of the form `model[error]` where `error` is an expression also of the form `BinaryString[...]`.

```
model[error] = BinaryString[...][BinaryString[...]]
```

which, by the rule of Definition 4.2, becomes a single binary string.

Now we are ready for the definition of compression as an encoding of expressions into binary strings. First, the model is mapped to a binary string b. Then, the data is mapped to a binary string `error`, that when evaluated using `Decompress`, prefixed by b, produces the data. Both the map of the model into a binary string, and the map of the data into a binary string need to be injective (no two elements have the same image) and prefix-free, that is, their range must be a prefix-free set.

4.4.2 Compression and decompression

The functions `Compress` and `Decompress` are paired: when `Compress` maps an expression α into a binary string b, the function `Decompress` maps any binary string of the form bx into the expression α and the remaining bits x.

One instance of the function `Compress` is, in the notation of Chapter 2, the function $\mathbf{C}_U$ that returns the shortest description of its argument x, so that $\mathbf{C}_U(x)$ is the shortest description of x under U.

The function `Decompress` corresponds to the reference Universal Turing Machine U of Chapter 2. The input of `Decompress` is an expression of the form `BinaryString[...]` and the output is an expression, not restricted to binary strings.

Compression

We are now ready to define the compression function based on Definition 2.11 (in page 19) of the MDL method.

Definition 4.3 COMPRESS Compression of a non-atomic expression $\alpha = \alpha_0[\alpha_1]$ is defined as the compression of the head α_0, which is the *model*, concatenated to the compression of the body α_1 using α_0

```
Compress[α_] := Compress[ Head[α] ] [ Compress[α, KnownModel] ]
```

The function `Compress[`α`]` must have an inverse, and the function `Compress[`α`, KnownModel]` must have an inverse when the head of α is known.

In other words, the function `Compress` returns the compressed head concatenated with the compressed expression assuming the head is known.

The fact that `Compress` must have an inverse, and that the binary string `Compress[`α`]` will be decompressed from left to right, implies that the range of `Compress` needs to be a prefix-free set.

The compression function is recursive. For each expression α the function `Compress` is called again on the head of α. The recursion ends when the model is a symbol, which in `Mathematica` has a head `Symbol`. A special instance of the function `Compress` is defined when the model is a symbol.

The function `Compress[`α`[`β`], KnownModel]` is defined for each particular model or head α. Examples of `Compress` for given models are given in Section 4.5.

Decompression

The counterpart of `Compress` is `Decompress`. The function `Decompress` takes a binary string and produces the corresponding expression. It is not exactly the inverse of `Compress` because `Decompress` returns also the remaining part of the input which has not been decoded, so that it can be applied recursively.

The definition of `Decompress[`α`]`, for a `BinaryString` α, is elegantly short in `Mathematica`.

```
Decompress[b_] := FixedPoint[Decompress @@ # &, {b, Symbol}];
```

which means that the output is a fixed-point of the function `Decompress`. This definition needs `Decompress` being defined for each specific data type. The function `FixedPoint[`α`, `β`]` evaluates α`[`β`]`, α`[`α`[`β`]]`, ... and so on until a fixed-point of α is obtained. It corresponds to the evaluation of the fixed-point Universal Turing machine V in Chapter 2.

Note that we have implemented `Decompress` using explicitly the built-in `FixedPoint` function of `Mathematica`, and it corresponds to Definition 2.11 in Chapter 2, in which the expression $\mathbf{C}_U(m)C_M(d)$ is a description of d under a fixed-point Universal Turing machine V. The Universal Turing machine V is a fixed-point function in the same way that the function `Decompress` is a fixed-point function. This is necessary in this chapter in order to justify the recursive compression (and decompression) of the models.

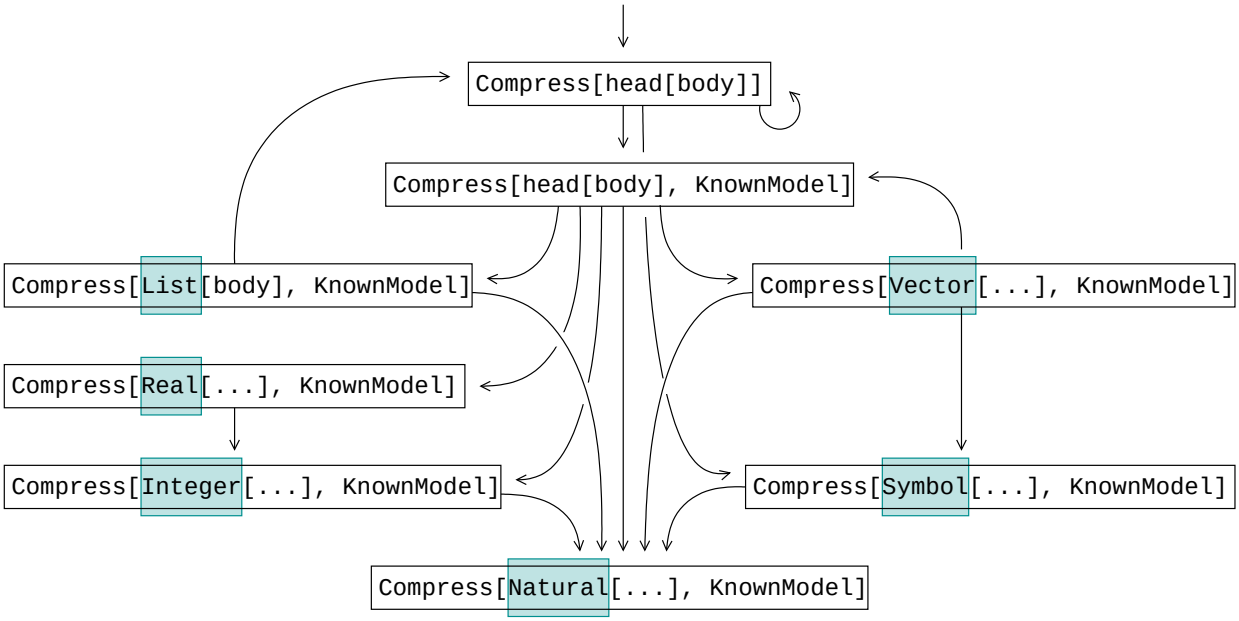


Figure 4.2: Function calls for compression of basic data types.

The computation of `Decompress[`α`]` is recursive. At each step of the recursion, the head of the resulting object is extracted, by decoding the leftmost part of the remaining binary string. The first head is extracted assuming it is a symbol.

The recursion in the definition of `Decompress` ends when a fixed point is reached.

```
Decompress[b_, expr_] := {b, expr};
```

This definition means that the function `Decompress`, in any undefined case, is the identity.

4.5 Compressing specific data types

The definitions of `Compress` for some basic data types are given below, thus filling in the gaps to in the definition of the `Cost` function. These definitions are illustrated in Figure 4.2. In this diagram, the entry point is the function call `Compress[`α`[`β`]]` where α is the head of the input expression and β is the body. This function calls either `Compress[`α`]`, if the model is not an atom or symbol, or `Compress[Symbol[...], KnownModel]` if the model is a symbol, α =`Symbol[...]`, which marks the end of the recursion.

4.5.1 Natural numbers

Let us define natural numbers as integers strictly greater than zero.

$$\mathbb{N} = \{1, 2, \ldots\}$$

The head `Natural` will identify natural numbers such as `Natural[1]`, `Natural[2]`, etc. Note that the language `Mathematica` does not have natural numbers as a in-built type. Therefore `Natural[5]` is actually the `Integer` 5 "wrapped" by `Natural[ ]`.

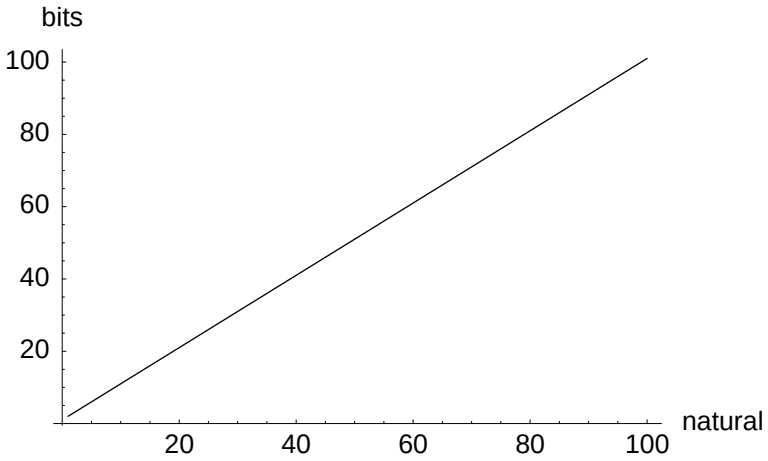


Figure 4.3: Code length for unary encoding of natural numbers.

Recall that the domain of the function `Decompress` needs to be a prefix-free set for the same reasons that the reference Turing machine U of Chapter 2 needed a prefix-free domain. In this way the argument of `Decompress`, a binary string, can be decoded from left to right.

We need to define the function `Compress` to map natural numbers into short binary strings which form a prefix-free set, so they can be decoded by a function `Decompress`.

The number of finite strings with length l is finite, for any $l > 0$. Therefore `Compress` needs to map the integers to binary strings of different lengths. In Section 2.5 such an map or *encoding* was linked to a probability distribution. Choosing such an encoding is equivalent to deciding about a prior probability for the natural numbers. The question is now which encoding to use.

Unary encoding

If we map the natural number n to the binary string of n ones followed by zero,

$$\underbrace{11\cdots1}_{n)}0$$

and plot the length of the binary string corresponding to each natural number, the result is Figure 4.3.

It is clear that this encoding will not very good at compressing data, in the sense that the binary tree that forms the input language of the Turing machine that decodes the integers will be unnecessarily sparse.

Squares encoding

Let us encode a natural number n with n^2 ones followed by zero,

$$\underbrace{11\cdots1}_{n^2)}0$$

and plot the corresponding lengths, as in Figure 4.4.

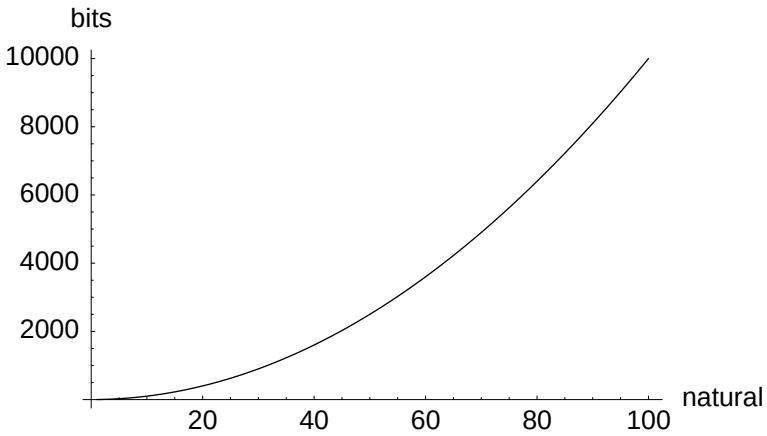


Figure 4.4: Code length for squares encoding of natural numbers.

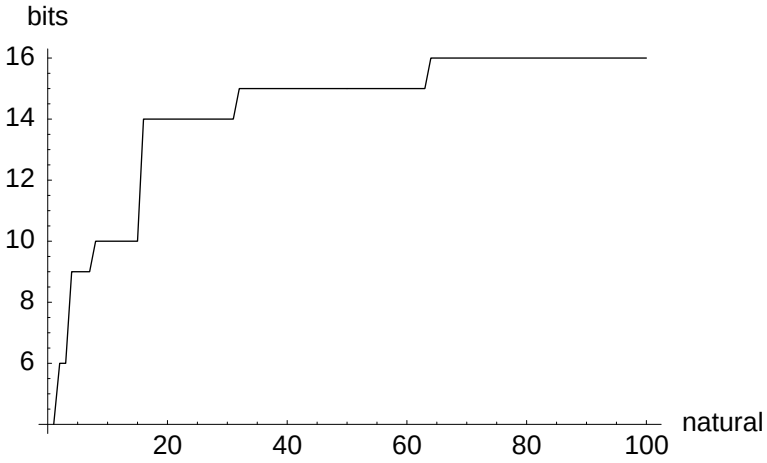


Figure 4.5: Number of bits required to encode natural numbers using $\log^*$.

Note that least squares fitting consists in minimising this cost function, where m_j are components of the model, d_j are components of the data,

$$LS((m_1, m_2 \ldots, m_n) \mid (d_1, d_2, \ldots, d_n)) = \sum_{j=1}^{n} (m_j - d_j)^2$$

and where $(m_j - d_j)^2$ is the same as `Cost[Natural[Abs[`m_j`-`d_j`]]]`.

The probability distribution implied by this encoding, which is $-\log$ of the plot of Figure 4.4 assigns higher probability to zero, and produces with extremely low probability numbers away from zero. This is the reason why the least squares model fitting method is so sensitive to outliers.

Compared to the unary encoding, this encoding wastes even more leaves of the binary tree in which each codeword is a leaf.

Logstar encoding

We have so far considered encoding each natural number n using sequences like

$$\overset{n)}{11\cdots 10} \text{ and } \overset{n^2)}{11\cdots 10}$$

Since we are usually interested in short binary strings, it seems reasonable to avoid the waste of bits involved in these encodings. We need a more compact encoding.

When we are given a finite set of natural numbers $\{n_1, \ldots, n_r\}$, Huffman encoding [Knu98] provides an optimal encoding that minimises the total size, computed from the frequency each natural number appears in the set. But in our case we do not have a set of integers to play the role of "training set", but only the prior knowledge that those integers have been produced by a Turing machine. Note that there is no upper bound in the value of the natural numbers we need to encode.

It might seem odd that from the sole assumption that the numbers are produced by a Turing machine it is possible to obtain an encoding which is optimal in any way. But, for this purpose, Rissanen [Ris83] proposed the following encoding:

Definition 4.4 LOGSTAR ENCODING The function $\log^*$ between natural numbers and binary strings is defined as

$$\begin{aligned}
\log^*(n) &= \log_R(n) \cdot 0 && (4.1)\\
\log_R(1) &= \varepsilon \text{ (the empty string)} && (4.2)\\
\log_R(n) &= \text{digits}(n) \text{ if } n = 2, 3 && (4.3)\\
\log_R(n) &= \log_R(\text{length}(\text{digits}(n)) - 1) \cdot \text{digits}(n) \text{ if } n > 3 && (4.4)\\
& && (4.5)
\end{aligned}$$

where $\log_R$ denotes the recursive part of the definition of $\log^*$, the binary string $\text{digits}(n)$ is the binary expansion of the integer n, and a dot $\cdot$ is used to emphasise string concatenation.

The final 0 is added to ensure that the image set of the function $\log^*$ is a prefix-free set, so the function can be inverted by reading the binary string from left to right.

Since $\log^*$ maps the natural numbers into a prefix-free set of binary strings, it implies a probability distribution on the natural numbers, as we have seen in Section 2.5. Rissanen [Ris83] used the function $n \mapsto 2^{-\log^*(n)}$ as an approximation to the Universal Measure defined on the natural numbers.

Interpretation. Jaynes [Jay96] suggested to use $1/n$ as prior for integers in the range $[1, n]$. This prior is based on the Maximum Entropy principle. Of course $1/n$ is not a probability, because the sum

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

does not converge. (In terms of codes, there is no code which assigns a string of length $2^{-\frac{1}{n}}$ to each integer n).

The log* code is one possible extension of Jaynes' approach, to an infinite set. It is not possible to assign the same probability to all integers, but it is possible to assign the same probability to all integers in the interval $I_k = [2^{k-1}, 2^k - 1]$ for each $k > 0$, which is what the log* code does. The binary tree of the log* code is built up by "descending" in the tree until there are enough leaves to allocate the elements of the interval I_n.

Figure 4.5 shows the code length for natural numbers. Note how the function has a clear minimum at 1, and the slope tends to zero as n increases.

The function `Compress` can now be defined for natural numbers:

```
Compress [Natural[n_], KnownModel]:=log*[Natural[n]];
```

Note that the second argument, `KnownModel`, is an undefined symbol that serves as a tag to state that the head `Natural` has already been encoded, and should be already known by the decoder when it reaches the encoded binary string. When the function `Compress[...]` is evaluated, the chain of transformed expressions is

`Compress[...]` $\mapsto$ `Compress[..., KnownModel]` $\mapsto$ `BinaryString[...]`

Therefore we are ready to approximate the cost of an integer.

Example

The following sequence of commands are a real session with the language interpreter. Let us first load the packages `Cost` and `Compress`

```
In[7]:= Needs["Thesis`Cost`"];
```

Let us compute the cost of an natural number.

```
In[8]:= Cost[Natural[5]]
Out[8]= 9
```

This cost is the length of the binary string in which `Natural[5]` is encoded. Note that we use the function `Compress` which calls

```
In[9]:= b = Compress[Natural[5]]
Out[9]= BinaryString[1, 1, 0, 1, 0, 1, 0, 1, 0]
```

The function `Decompress` will give us the remaining string (an empty binary string), together with the original expression.

```
In[10]:= Decompress[b]
Out[10]= {BinaryString[], Natural[5]}
```

The function `CompressVerify`, also defined in the package `Compress`, computes the compression–decompression cycle and returns `True` only if the functions are inverse of each other at a particular argument, when no additional input is given.

```
In[11]:= CompressVerify[Natural[5]]
Out[11]= True
```

4.5.2 Integers

The definition of `Compress` for integers can be derived from the definition for natural numbers by means of a simple 1-1 map between integers z and natural numbers n

$$z \mapsto \begin{cases} 2z & \text{if } z > 0 \\ -2z+1 & \text{if } z \leq 0 \end{cases}$$

The head used by `Mathematica` is `Integer`. Thus, for example, the integer 3 has head `Integer`

```
In[12]:= Head[3]
Out[12]= Integer
```

Let us encode the integer `3` assuming the head is known, and also encode the `Natural[6]`, also assuming the head is known. Both binary strings are the same because the integer 3 is mapped into the natural number 6:

```
In[13]:= Compress[3, KnownModel]
Out[13]= BinaryString[1, 0, 1, 1, 0, 0]
In[14]:= Compress[Natural[6], KnownModel]
Out[14]= BinaryString[1, 0, 1, 1, 0, 0]
```

But, since their models `Integer` and `Natural` are encoded in different ways,

```
In[15]:= Compress[Integer]
Out[15]= BinaryString[0]
In[16]:= Compress[Natural]
Out[16]= BinaryString[1, 1, 0]
```

when the model will not be known in advance to the decoder, and therefore it has to be encoded with the number, the encoded integer 3 and natural 6 differ in the prefix

```
In[17]:= Compress[3]
Out[17]= BinaryString[0, 1, 0, 1, 1, 0, 0]
In[18]:= Compress[Natural[6]]
Out[18]= BinaryString[1, 1, 0, 1, 0, 1, 1, 0, 0]
```

Figure 4.6 shows the cost of each integer using $\log^*$ encoding.

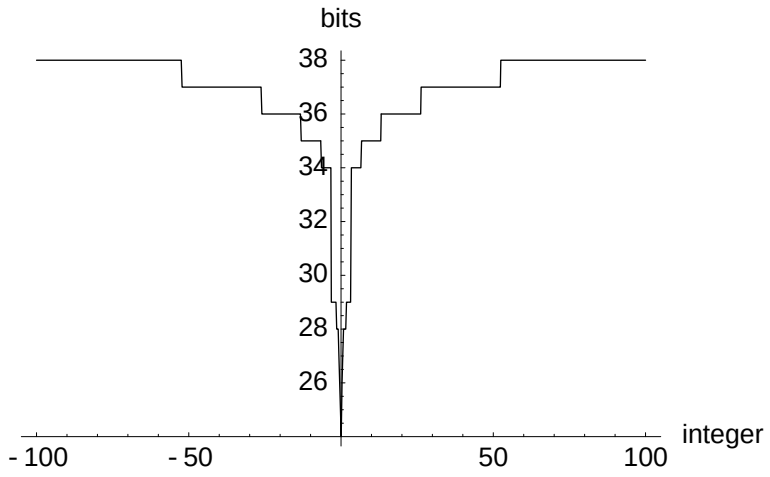


Figure 4.6: Number of bits required to encode integers using the log* code

4.5.3 Vectors and Lists of any type

Vector. A vector is a sequence of n expressions with the same `Head` or type. We will use the head `Vector` to group them up. A vector has the form

`Vector[`e_1`, `e_2`, ..., `e_n`]`

where the expressions e_j have the same head h.

These are the steps required to compress the vector:

1. encode the common head h, using `Compress[`h`]`.
2. encode the number of elements in the vector. This number is a natural n, using `Compress[Natural[`n`]]`.
3. encode each of the expressions e_j, assuming the head h is known, using `Compress[`e_j`, KnownModel]`.

List. A list is a sequence of n expressions. They do not need to have the same `Head` or type, and therefore we cannot compress them as a vector. The language `Mathematica` has a built in type `List`, and therefore we do not need to define one. A list has the form

`List[`e_1`, `e_2`, ..., `e_n`]`

These are the steps to compress the list:

1. encode the number of elements in the list, n, using `Compress[Natural[`n`]]`.
2. encode each of the expressions e_j, assuming their heads are not known, using `Compress[`e_j`]`.

Compare the last step when encoding a vector and a list. When encoding a *vector*, the head of each expression that forms the vector is already given in the first step, thus it is only necessary to compress each expression and not its head. Therefore the

last step of encoding a *vector* calls the function `Compress[`e_j`, KnownModel]`. However, when encoding a *list* each expression e_j is compressed assuming the model is not known, therefore encoding a list calls the function `Compress[`e_j`]` for the last step.

For example, let us compress a vector of two integers

```
In[19]:= Compress[Vector[1, 2]]
Out[19]= BinaryString[1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
   0, 1, 0, 1, 0, 0, 0]
```

Both numbers `1` and `2` have the head `Integer`. Since they have the same head we could compress them as a vector. But we could also compress them as a list of two elements.

```
In[20]:= Compress[{1, 2}]
Out[20]= BinaryString[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
   0, 1, 0, 0, 0]
```

Note that not only the prefix of the string is different, but each element is compressed in a different way.

4.5.4 Real numbers

We now proceed to define the compression function for approximations of real numbers, which will be required, for example, by the trajectory models of Chapter 5.

Physical measurements are usually real numbers. The representation of real numbers in the computer is an approximation. The way this approximation is done is relevant here, since different approximations of the same number will give different sizes in bits. Let us consider three ways to approximate real numbers:

Floating point Numbers of the form $m10^e$ where m is the mantissa and e is the exponent. Both m and e are integers. This approximation is useful to represent numbers over a wide range of orders of magnitude.

Fixed point Numbers of the form $m10^k$ for some constant k and an integer mantissa m. This representation is equivalent to a change of scale to convert the measurements to integers.

Fraction Numbers of the form a/b, where a and b are integers. This representation is useful when algebraic simplifications are possible.

Examples of each representation of real numbers abound, such as the modern use of decimal measurements for currency (fixed point) versus the deprecated division in thirds, halves, quarters and fifths (fractional). Rissanen [Ris83] uses the fractional representation to encode errors between models and data. The approach taken in Chapter 5 is to use *fixed point* numbers. This is a consequence of choosing centimetres as the unit for the camera calibration in Chapter 3.

The function `Compress` for numbers of types `Real` and `Rational` is defined in the package `Thesis‘Compress‘`. The type `Real` is built into `Mathematica` as a floating point type with variable size (of mantissa and exponent), and `Rational` as a fraction. For our purposes, it is enough to approximate `Rational` with a floating point `Real`.

In the package `Thesis`Compress``, a floating point number r, with `Real` is first truncated to 4 decimal digits, then multiplied by 10^4, and then encoded as an integer.

For example,

```
In[21]:= r = 3.49827
Out[21]= 3.49827
In[22]:= Head[r]
Out[22]= Real
In[23]:= b = Compress[r]
Out[23]= BinaryString[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1,
   0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1,
   1, 0, 0, 0]
```

Since no special head such as `FixedPointNumber` was created for the approximation of real numbers, the function `Decompress` is not the exact inverse of `Compress`, note that the last digit is lost by the four-decimal rounding included in `Compress`

```
In[24]:= Decompress[b]
Out[24]= {BinaryString[], 3.4982}
```

and therefore `CompressVerify` returns `False`.

```
In[25]:= CompressVerify[r]
Out[25]= False
```

The function `Compress` is used with numbers of type `Real` in Chapter 5. If we want `Decompress` to be able to recover the original data, then we should define the new type `FixedPointNumber` and define `Compress` and `Decompress` for that type, instead of the type `Real`. It was not considered necessary for this application since `CompressVerify` is only used for test purposes.

4.6 General procedure

This section explains, step by step, how to define `Compress` for a new data type. An example is given, using an integer number to model other integer numbers.

4.6.1 Defining new data types

A data type is identified by an expression head which must be an atom (that is, have the head `Symbol`). Examples of built-in data types are `Integer`, `List`, `Real` and `Rational`. Other data types used in Section 4.5 are `Natural` and `Vector`.

Each new data type needs to be "loaded" in a list, called `AcceptedTypes`, which allows the functions `Compress` and `Decompress` respectively to encode and decode the symbol. Let us look at the list of currently accepted types and the encoding of each type as a binary string.

```
In[26]:= AcceptedTypes
Out[26]= {Integer, List, Natural, Rational, Real, Vector}
In[27]:= Compress[Integer]
```

```
Out[27]= BinaryString[0]
In[28]:= Compress[Natural]
Out[28]= BinaryString[1, 1, 0]
In[29]:= Compress[Vector]
Out[29]= BinaryString[1, 0, 1, 1, 0, 0]
```

Each type is encoded using $\log^*$ for natural numbers. The natural number corresponding to each type is its position in the list of accepted types.

4.6.2 The procedure

The sequence of steps needed to define and use `Compress` with a new data type are:

1. Choose a name for the new data type, α. This should be a symbol not used already in the system.

2. Add the data type α to the list of accepted types. This is done by the function `CompressedTypeAppend[`α`]`.

3. Define the function `Compress[`α`[`β`], KnownModel]` for our new type α, that returns a description of β when the head α is known.

4. If decompression is required, define `Decompress[`β`, `α`]` where $\beta = \beta_p\beta_s$ is an expression with head `BinaryString` formed by prefix β_p and suffix β_s. This function should return a pair $\{\beta_s, \gamma\}$ where γ is the original, decompressed expression corresponding to β_p, which has head α, and β_s is the remaining binary string.

If `Decompress[`β`, `α`]` is defined, then `CompressVerify` should return `True` for expressions of our new type.

4.6.3 Example: integer models

A simple way to show the general procedure describe above is to use the MDL method to find the "best" integer that models other integers, according to our computable approximation of the MDL method. An integer z, or a vector of integers $(z_1, z_2, \ldots, z_n)$, can be modelled by another integer m by considering the differences between m and either z or each z_i.

For this purpose, we should add a definition of `Compress` that evaluates, for example, `Compress[5[8]]`. But given the integer `5`, there is no way to know whether it is an integer on its own, or whether we want to use it as a model for another integer. Thus we need to define a new type, `IntegerModel`, which "wraps" the number.

Thus, we have two new models:

- `IntegerModel` will be the *type* or *model* of m.

- `IntegerModel[`m`]` will be the *type* or *model* of the data.

and we need to define the function `Compress` for each of them.

Therefore we will be compressing

`IntegerModel[`m`][`z`]`

and

`IntegerModel[`m`][Vector[`z_1`, `z_2`, ..., `z_n`]]`

The description of the integer z given by m will be the difference $m - z$, which is an integer. The description of the vector of integers $v = (z_1, z_2, \ldots, z_n)$ given by m will be the vector $(m - z_1, m - z_2, \ldots, m - z_n)$, of integer components. That is, the data is represented by the model and the difference between the model and the data.

Let us first define an auxiliary function `subtract` first for integers

```
In[30]:= subtract[a_Integer, b_Integer] := a - b;
```

then for vectors, in which m is subtracted from each component z_j.

```
In[31]:= subtract[v_Vector, m_Integer] := (m - #1 & ) /@ v
```

Now, let us follow the steps in the list of page 84. First of all, we need to append the type `IntegerModel` to the list of accepted types.

```
In[32]:= CompressedTypeAppend[IntegerModel]
Out[32]= {Integer, IntegerModel, List, Natural, Rational,
   Real, Vector}
```

Then we need to define `Compress[IntegerModel[`m`], KnownModel]`

```
In[33]:= Compress[IntegerModel[m_], KnownModel] :=
   Compress[m, KnownModel]
```

Now we need to define `Compress[IntegerModel[`m`][`β`], KnownModel]` where β is anything for which `subtract[`β`, `z`]` is defined (i.e. an integer or a vector of integers). Essentially, the problem reduces to compressing the arithmetic difference between the model and the data.

```
In[34]:= Compress[IntegerModel[m_][expr_], KnownModel] :=
   Compress[subtract[expr, m]]
```

Modelling an integer

Let us see how `Compress` works when modelling the integer 3 by another integer. Note that the cost increases with the difference.

```
In[35]:= Cost[IntegerModel[3][3]]
Out[35]= 11
In[36]:= Cost[IntegerModel[4][3]]
Out[36]= 14
In[37]:= Cost[IntegerModel[5][3]]
Out[37]= 17
```

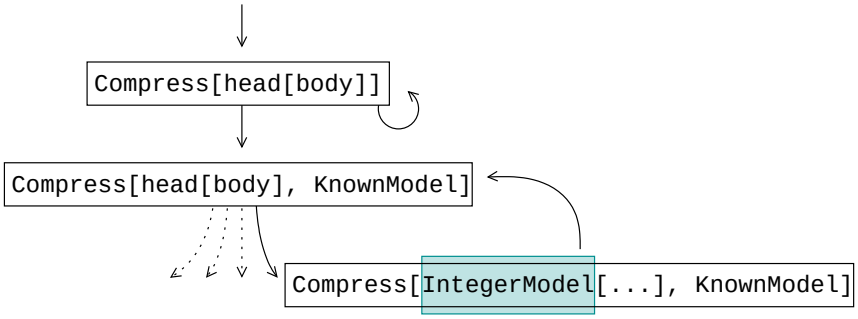


Figure 4.7: Additional function calls for the compression of integer models (compare with Figure 4.2).

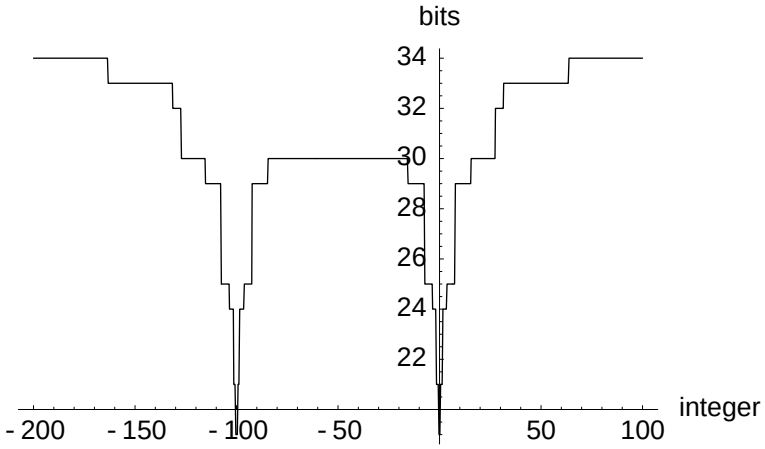


Figure 4.8: Description length of the integer -100 when compressed using the models ranging between -200 and 100. See also Figure 4.9.

The relation between `IntegerModel` and the functions defined before is illustrated in Figure 4.7, where the arrows correspond to function calls. The function `Compress[IntegerModel[`α`][`β`], KnownModel]` calls `Compress[`$\alpha - \beta_i$`,KnownModel]` for each component β_i of the vector β.

Figure 4.8 is the cost of compressing the integer -100 when the models range between -200 and 100. Two features are noticeable:

- The two minima: one for the "real" model, -100, another one for the simplest model 0. This is an example of MDL's tradeoff between model size and fitness. We will discuss this fact in Section 4.7
- The similarity between the shape of this cost function and the cost function for individual integers, in Figure 4.6.

Modelling a vector

Let us calculate the cost of the vector of integers $(3, 4, 5)$ when modelled by single integers

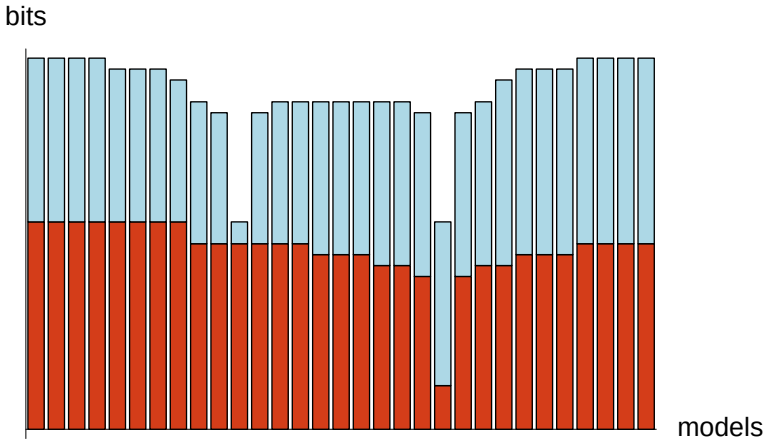


Figure 4.9: Description length of the integer −100 when compressed using the models ranging between −200 and 100. The red portion of the bar is the description length of each model. The blue portion of the bar is the description length *of the data given each model*. The total height is the same value as in Figure 4.8, and is the description length *of the data*. Note the local minima at the simplest model (0) (minimum on the right) and at the "true" model (-100) (minimum on the left).

```
In[38]:= Cost[IntegerModel[3][Vector[3, 4, 5]]]
Out[38]= 29
```

The cost is higher when the model is further away from 4

```
In[39]:= Cost[IntegerModel[4][Vector[3, 4, 5]]]
Out[39]= 27
In[40]:= Cost[IntegerModel[5][Vector[3, 4, 5]]]
Out[40]= 30
In[41]:= Cost[IntegerModel[6][Vector[3, 4, 5]]]
Out[41]= 35
```

The cost of a more complex vector, when modelled by integers, is displayed in Figure 4.10. A set of random integers drawn from a uniform random variable in the interval [200,220],

```
Vector[220, 205, 206, 216, 204, 214, 212, 212, 215, 202]
```

is modelled by each integer in the range [-100,500]. The two minima are at 0 and 210.

More complex data can be obtaining by drawing integers from a random distribution in two different intervals of the same width. Ten integers are drawn from the interval [200, 220], and five integers from the interval [350, 370]

```
In[42]:= Vector[220, 205, 206, 216, 204, 214, 212, 212,
   215, 202, 354, 358, 355, 366, 366]
Out[42]= Vector[220, 205, 206, 216, 204, 214, 212, 212,
   215, 202, 354, 358, 355, 366, 366]
```

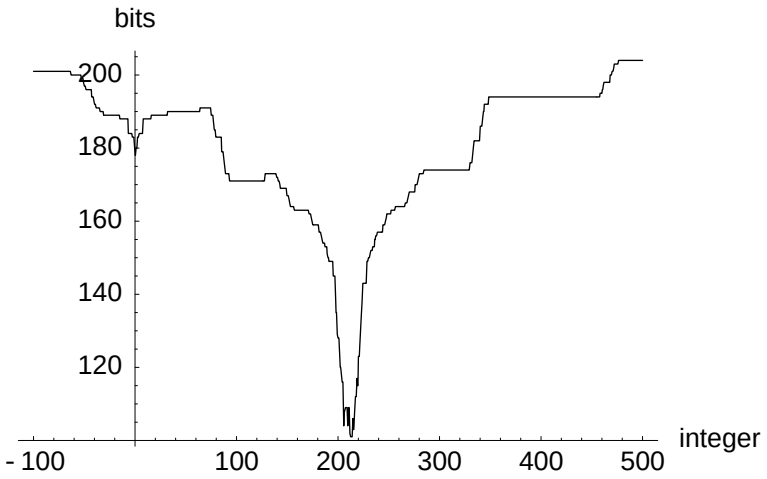


Figure 4.10: Cost function when the model is an integer in the range [-100, 500], and the data is a vector of ten integers drawn from a uniform random variable in the interval [200,220].

The cost for integer models in the range [-100, 500] is displayed in Figure 4.11. There are two distinctive local minima, corresponding to the two clusters. The lower minium corresponds to the cluster of ten elements, in the interval [200, 220]. The non-absolute minimum is in the cluster of five elements.

4.7 Discussion

This section discusses the properties of the MDL method, under the light of examples with integers.

4.7.1 Shape of the cost function

Using the functions described in Section 4.5, it is possible to display examples of the cost function for simple data formed by integer numbers, and illustrate their properties.

The cost function is simply a function that maps natural numbers into binary strings and returns the length of the binary string. Other data types, such as ordered sets of integers, can be mapped to integers and then to natural numbers.

The minimum. The cost of a single integer z is the number of bits required to represent it. Using the $\log^*$ to encode integers, the cost function is displayed in Figure 4.6. Note the symmetric shape, the sharp minimum at zero, and the fact that the cost function can be approximated by a horizontal line as z tends to infinity or to minus infinity. A similar shape is displayed in Figure 4.10.

These properties affect the optimisation of the cost function and the influence that large values have in the cost:

1. Optimisation near the minimum is easier because the difference in cost between consecutive models is high. In other words, a continuous approximation has a

high derivative near the minimum. On the other hand, optimisation that starts away from the minimum is difficult because the approximated slope is near zero.

2. Optimisation far away from the minimum is much harder, because the slope of the cost function is low. But the influence of outliers is also small: the cost function $z \mapsto \log^*(z)$ is, in the limit, below any nonzero polynomial [Ris83]. Compare with the cost implied by the least squares method, $z \mapsto z^2$ which is polynomial.

The $\log^*$ method is more robust against *outliers* than the least squares method because the $\log^*$ cost of an outlier is a low-valued function of the distance to the minimum, and the influence of outliers in the shape of the cost function is local (see paragraph Handling of outliers, below).

Tradeoff between model and data. The cost of an integer d when modelled by another integer M is given as a stacked bar chart in Figure 4.9. In this case there is a clear tradeoff between model size and error size. The heights of the lower bars are model sizes. The heights of the upper bars are error sizes, which is the number of bits added to the model in order to describe the data; in other words, it is the description length *of the data given the model*

$$\text{length}(C_M(d))$$

The total cost is the sum of the model size and the error size, or the description length *of the data*

$$\text{length}(C_U(M)C_M(d)) = \text{length}(C_U(M)) + \text{length}(C_M(d))$$

The model "0" is particularly interesting in that it shows minimal model complexity. In this case, when the "true" model is a small integer, say -100, the model "0" is comparable to the "true" model (the number -100) which has higher model complexity but lower error. As the amount of data increases in relation to the model size, the size of the model becomes irrelevant and thus the method degenerates into a Maximum Likelihood method. This is the case when modelling sets of integers using a single integer, as in the following examples (See Figure 4.12).

Adaptability. The shape of the cost function adapts to the data. For example, when there are two valid models, the cost function displays two local minima. The plot in Figure 4.11 shows such minima about the centre of the two sets of random points used as data. The breakup between model and error size is shown in Figure 4.12. This property was used in Section 3.3.3 to choose an event for a traffic image sequence when the image sequence is formed by the combination of various events.

Handling of outliers. All values have an approximately local influence on the shape of the cost function, as illustrated in Figure 4.11. This means that adding an outlier to the data does not change much the shape of the cost function, except in the region near the outlier. This is a consequence of the cost function $z \mapsto \log^*(z)$ having a low value away from the minimum, and that the total cost is the sum of the cost of each integer.

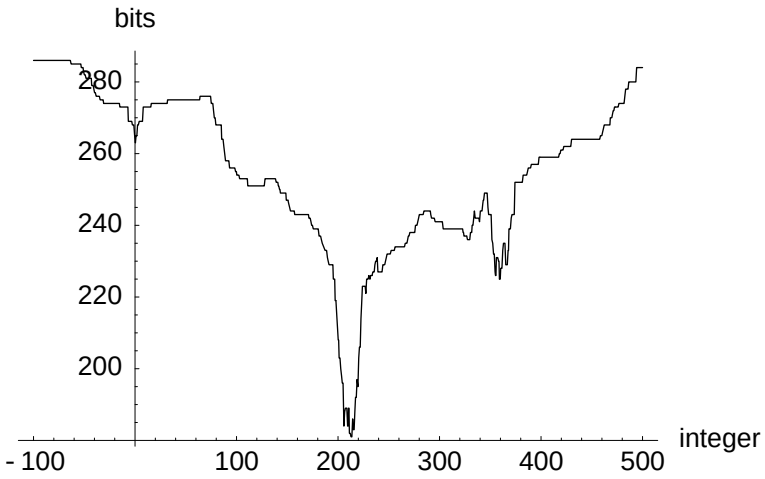


Figure 4.11: Description length when the model is a single integer varying from -100 to 500, and the data consists of two fixed sets of integers drawn from uniform distributions. The first set is the same as in Figure 4.10, ten integers drawn from the interval [200,220]. The second set consists of five integers drawn from the interval [350,370].

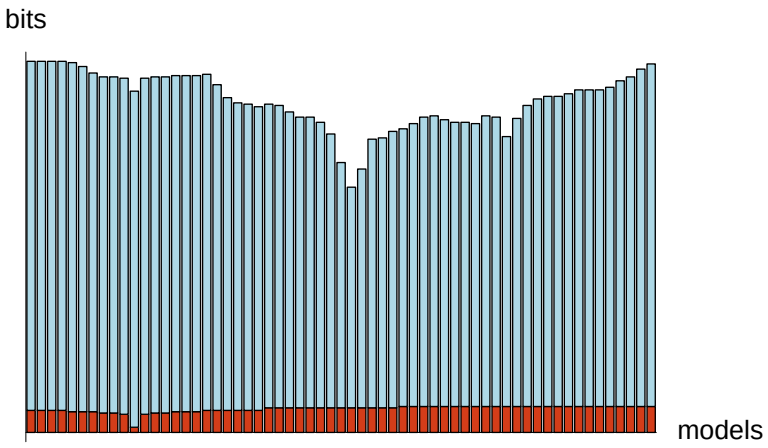


Figure 4.12: Breakup of description length corresponding to model and data in the plot of Figure 4.11 for models between -100 (left) and 500 (right). The lower part of each bar is the description length of each model. The upper part is the description length of the data given each model. Note the model with shortest description length (shorter red bar) is 0 whereas the model that yields shortest data description length is near 200.

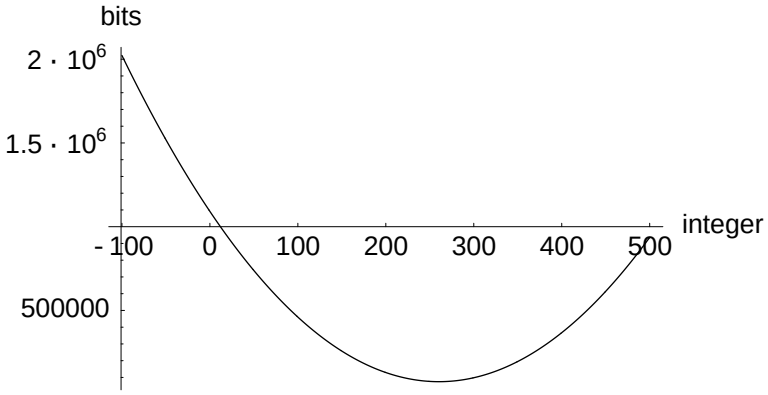


Figure 4.13: The same model and data as in Figure 4.11 but with the least squares error function as cost function.

Universal Measure

The log* function has a probabilistic interpretation. The correspondence between prefix-free sets and probability distributions of Section 2.5 can be used to assign a probability to the log* encoding. This probability is a computable approximation to the universal measure as defined in Chapter 2 [Ris83]. The universal measure, in this context, gives the probability of obtaining $\log^*(z)$ by producing strings of the form `BinaryString[...]` choosing each bit using coin flips.

General principle includes others as particular cases

Least squares. A common alternative way to fit models to data samples, in the Euclidean space, is the least squares method. This method is optimal under the assumption that the data contains additive Gaussian errors with respect to the model. In practice, the least squares method is used because of its simplicity. Its drawbacks are its sensitivity to outliers (which is a low probability assigned to large errors) and the fact that sometimes the error cannot be modelled properly as an additive Gaussian distribution.

We have seen in Figure 4.11 how the cost function has two minima roughly at the centre of the clusters of data. Figure 4.13 shows the least squares cost function, often called *error function*, which has a completely different minimum.

Least Squares gives us a single minimum and a high gradient to guide the search for the minimum. In contrast, our `Cost` function, which is the log* description length, adapts to the data and shows different minima for different plausible explanations, but its optimisation is difficult because of the small gradient far from each local minimum.

Maximum Likelihood. The Least Squares method is a particular case of Maximum Likelihood [Edw92] method. Maximum Likelihood methods are a particular case of MDL when the model size is not taken into account [Ris83, LV97].

A Maximum Likelihood method is therefore a method in which only the upper part of the bars in Figures 4.9 and 4.12 is considered as the cost value. As we have seen,

this is the situation when the difference between model sizes are negligible, such as the case of a single integer modelling a large vector of integers.

4.7.2 Simplification

Rissanen [Ris83] gives a formula which approximates the value of the cost function, the length after compression, and does not require the evaluation of the compression function. The package provided here can serve as a testbed for such approximations. The meaning of the function `Compress` can be redefined for different data types, and in different ways, thus allowing approximations arbitrarily close to ideal MDL, for `Compress`, and Kolmogorov Complexity, for `Cost`.

The evaluation time of `Cost` can be reduced by avoiding the evaluation of the function `Compress`. This evaluation might be lengthy in time, or require a lot of memory.

Simplification rules are implemented as transformations that are applied automatically whenever a pattern is found. For example, when we considered unary encoding of integers

$$\overbrace{11\cdots}^{n)}10$$

we could have defined the function `Compress`, for a natural number n, to produce the output:

```
BinaryString[1, 1, ... 1, 0]
```

By adding the following definition, which just adds up the square of the differences between `n` and each component of `v`, the computation of `Cost` is simplified to adding up integers.

```
Cost[IntegerModel[n_][v_]] :=
   Plus @@ (Map[(n - #)^2 &, v])
```

Which means that the cost of data vector v with model n is the sum of the square of the difference between each component of v and n. No encoding (as evaluation of the function `Compress`) is necessary in this case. The resulting cost function is the least squares cost function. No further changes need to be made, since the interpreter of `Mathematica` will use this rule instead of evaluating its arguments and computing the expression

```
Cost[ Size [Compress[...] ] ]
```

This simplification of the function `Cost` is feasible when there is an "analytic" or "closed-form" formula for the cost, as in the case of the Least Squares method.

4.8 Conclusion

In this chapter we have defined a cost function `Cost`, which is based on the Minimum Description Length method for model selection. The implementation of `Cost` in `Mathematica` reflects the ideas behind the Minimum Description Length principle, namely, that estimating the cost of a model given the data is the cost of the model plus the cost of the data given the model (Definition 2.11).

Contributions

Short implementations. The description length cost function has been implemented in one line

```
Cost[expr_]:=Size[Compress[expr]]
```

When `Compress` is the non-computable optimal compressor **C**, the function `Cost` is the Kolmogorov Complexity.

The Minimum Description Length principle is spelled out in a few lines of `Mathematica`. Encoding in two-parts is explicitly done in Definition 4.3.

```
Compress[model_[data_]]:=Compress[model][
   Compress[model[data], KnownModel]
]
```

Use of symbolic expressions. While Chapter 2 developed the theory using binary strings, this chapter uses symbolic expressions, albeit a subset that maps into binary strings. Symbolic expressions have the crucial property of being human-readable. Furthermore, there is a large amount of programming constructs and languages available to manipulate symbolic expressions.

Preliminary examples of MDL. Examples have been given of a working software package. These examples illustrate

- the ability of the MDL method to choose between models of different complexity, and
- the potential to fit models that represent only part of the data sources, with high tolerance to influence from the other sources (low sensitivity to outliers).

In order to ensure the examples are correct, the outputs have been computed by a `Mathematica` interpreter: the source document of this chapter was given to the `Mathematica` interpreter, which evaluated the expressions and added the output.

Future work

The implementation of the functions `Compress` presented in this chapter can be subject to a variety of optimisations in order to reduce the computation time.

- The compression algorithms can be improved by including, when appropriate, assumptions about the data, such as correlation between consecutive values.
- Additional assumptions can be laid down as simplification rules that the interpreter automatically uses to avoid further computation.

The next chapter uses the package `Thesis‘Compress‘` to apply the MDL method to compress vehicle trajectories using prior knowledge about vehicle dynamics.

Chapter 5

Selection of trajectories by compression

This chapter provides a class of models for trajectories based on vehicle dynamics, and a definition of the function `Compress` for those models. The performance of the MDL method in choosing among trajectory models of different complexity is examined with experiments.

5.1 Introduction

Vehicles have a constant mass, and the motion of vehicles on the ground plane is regulated by specific laws describing how forces, applied to that mass, affect the acceleration and, as a consequence, the position of the vehicle over time. The forces depend on the actions performed by the driver, especially the way the steering wheel and the pedals are used [MWS96a, MWS96b]. A prior probability on the force, e.g. that it is minimal, induces a prior probability on the trajectories that a vehicle might follow. The fact that cars are easier to drive in a straight line with constant velocity rather than turning with changes in the pedals means that acceleration tends to be minimised, and therefore the force is minimised. Thus, a higher probability is assigned to a trajectory in a straight line with constant speed, and the probability decreases with changes in acceleration and as curvature increases.

In this chapter the package `Thesis`Compress`` is extended to compression of trajectories, based on the assumption that the trajectories are measurements of a vehicle's motion. The data are points on the ground plane. The resulting cost, which is a function of the models, is used to compare models with different complexities, according to their relative complexity and fit to the data. This complexity is related to the number of parameters required to identify the model within the model class.

The advantage of this approach is that the complexity of the trajectory models is taken into account when selecting the trajectory model that best explains the data. This is a departure from methods that only consider the fit of each model to the data [Ger99]. The consequence, as we have seen in Chapter 2, is a method that can potentially avoid overfitting. In particular, when data has been produced by a combination of sources, such as the vehicle dynamics and noise, deciding which part of the model is noise and which part is measurement amounts to choosing the model

complexity. The balance between model complexity and the fit is provided by the MDL method.

This chapter addresses the problem of defining realistic trajectory models, the compression functions derived from those models, and an optimisation algorithm to select models. In order to do so, we need the following elements:

A realistic model for trajectories in which more probable trajectories have shorter descriptions (Section 5.3).

A simpler cost function that uses points on the plane instead of images (Section 5.4).

An optimisation algorithm within the class of trajectory models (Section 5.4).

Experimental results illustrate the ability of the MDL cost function to find the best model in the presence of noise and occlusions.

5.2 A simpler dataset

Before we proceed to define the class of trajectory models, we need to decide to which type of data are we going to fit our trajectory models. Data in Chapter 3 were image sequences. These were very large datasets compared to the size of the trajectory models, and the cost function is influenced by many factors other than the trajectory. This section justifies the use of a simpler dataset that facilitates the study of the MDL method.

5.2.1 Points on the ground plane

The trajectory models will be tested with a cost function that compresses points on the plane. Implementing and testing the realistic trajectory models with the cost function of Chapter 3 would have two disadvantages.

- Using images would add difficulty to the study the properties of the trajectory models, because additional factors would influence the final outcome of the experiments. The main factor is that the size in bits of the image data is very large compared to the size in bits of the trajectory models.
- The computation of the cost function with image sequences is very slow.

There is another reason for the use of a cost function based on points on the ground plane. An optimisation algorithm that works for the cost function used in the current chapter will be a useful starting point for an optimisation algorithm for the cost function of Chapter 3, with the trajectory models of Section 5.3, because the parameter spaces are the same.

Therefore the data considered in this chapter will be a list of points on the ground plane, one per time step. Trajectories on the ground plane will be represented as the list of points, joined by segments, which are not part of the trajectory, but indicate which is the next point in the sequence.

5.2.2 A cost function based on points on the plane

We have seen in Chapter 4 that the cost function `Cost` depends entirely on the definition of the function `Compress`.

Note that this cost function which is defined for sequences of points on the ground plane is not the same as the cost function for image sequences of Chapter 3. The underlying principle is the same, though: create a synthetic version of the data that can be compared to the data point per point.

- pixel by pixel, in the case of image sequences, Chapter 3,
- point per point, in the case of trajectories on the ground plane.

then differences are stored assuming they are small (i.e. assuming they are drawn from a specific probability distribution that assigns high probability to small values) and the final size in bits is measured.

5.3 A class of models for vehicle trajectories

In this section we will define a class of trajectory models based on vehicle dynamics. The basic idea is to minimise the number of parameters by using prior knowledge about the physical properties of the motion of a vehicle.

The function `Compress` will be defined for trajectory models and data. The `Cost` function of Chapter 4 will be used to assign a cost to each trajectory model.

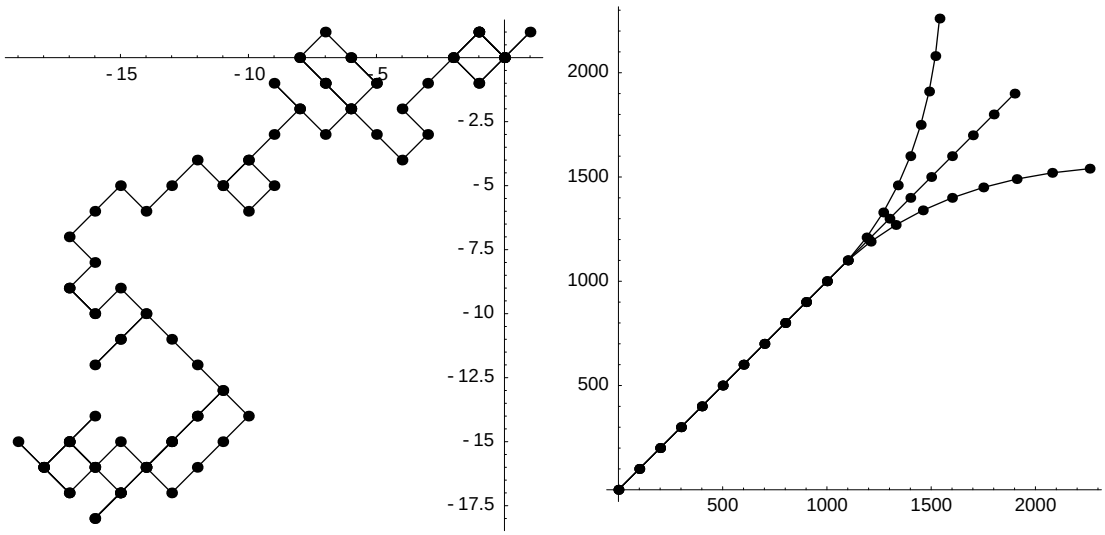
5.3.1 Realistic trajectory models

The trajectories of vehicles as they move on the road are not random (See Figure 5.1). There are definite restrictions regulating the way a vehicle moves [MWS96b]. The main factors are:

- Limited manoeuvrability of a typical vehicle.
- Regulations governing road traffic.
- Driver's preference for minimum effort.
- Dangers of making sudden changes in motion when speed is high.

These constraints can be approximated and summarised by requiring that the position at time t is a function of the position at some initial time t_0 and the acceleration values in the interval (t_0, t). The acceleration changes according to the driver's actions on the pedals and the steering wheel. Therefore we will also require that the changes in acceleration are small.

The key to defining a cost function is to use a parameter space in which realistic trajectories require shorter descriptions, while less realistic trajectories require longer descriptions. Trajectories described in such a parameter space will be referred to as *compressed* trajectories (regardless of whether the size of the data has actually been reduced or not with respect to some other parameter space).



(a) A low probability trajectory: random walk.

(b) High probability trajectories.

Figure 5.1: Probability of trajectories. The trajectory is the sequence of points on the ground plane. The segments link consecutive points. The units are arbitrary.

For example, if we know that the position of a vehicle over time, $p(t)$, depends on the second derivative, $\ddot{p}$, then we should use, as model parameter, the second derivative. The first derivative, $\dot{p}$ and the position p can be computed from the second derivative and the initial conditions $(p(1), \dot{p}(1), \ddot{p}(1))$. When using discrete approximations, the representation of the motion in terms of the initial conditions together with an approximation of $\ddot{p}$ is more compact, for realistic trajectories, than the finite sequence of points on the ground plane $(p(1), p(2), \ldots)$.

5.3.2 Parametric trajectories

Let us consider the problem of finding an adequate parameter space for vehicle trajectories, in discrete form. The alternatives considered here are, for a given length of the trajectory in time:

- Sequences of points (or *positions*) on the ground plane, a *ground plane positions model.*
- An initial point and a final point. The rest of the positions can be obtained by linear interpolation. This is the *constant velocity model.*
- A sequences of changes in the acceleration, together with the initial state of the vehicle. This is a *driver's actions model.*

Ground plane positions model

In a first approach, a vehicle trajectory is a set of points on the ground plane

$$(p(1), \ldots, p(n)),$$

where $p(t)$ is the position of the vehicle at time step t. The sequence $(p(1), \ldots, p(n))$ can be thought as a vector of *parameters* of the trajectory. We will refer to this parameter space as *ground plane positions*. This representation of trajectories is general enough to cover all possible types of motion. But the number of parameters to describe such a trajectory is large: two integers per time step. Therefore we can claim that this representation is too general.

Probability of each trajectory. Since the number of bits required to represent a trajectory x depends only on its length in time, size(x), the probability of each trajectory, defined as

$$P(x) = \mu \, 2^{-\text{size}(x)}$$

(where μ is a normalisation factor) is the same for all trajectories. For example, the same probability would be assigned to all the trajectories of Figure 5.1 if they were the same length, regardless of whether they correspond to realistic trajectories or not. We would ignore the prior knowledge available about the likely motions of an object with mass. Therefore, these are the properties of trajectory models formed by ground plane positions:

Distribution The probability distribution implied in the space of trajectories of size l is the uniform distribution.

Optimisation The number of parameters is very high, therefore optimisation will be difficult.

Constant velocity model

The number of parameters can be reduced if only trajectories with constant velocity are considered. Such a trajectory needs only a start point and an end point as parameters. This model class was used in the experiments of Chapter 3. It does not include a set of trajectories wide enough to be useful in complex cases.

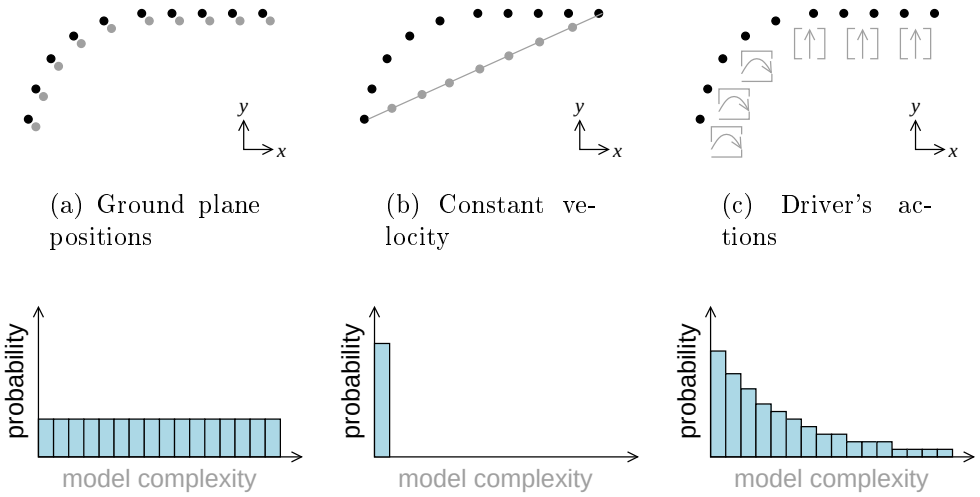
Distribution Only trajectories with constant velocity have nonzero probability.

Optimisation The number of parameters is very low, which should make optimisation easy.

Driver's actions model

A realistic model class for trajectories is one that gives a high probability to straight line trajectories, and lower, but nonzero, probabilities to trajectories with low curvature. This model class is somewhere between the generality of the *points on the ground plane* models and the *constant velocity* models. It will be based on the driver's preference for low changes in acceleration. We will refer to models of this class as a *driver's actions* model.

In order to define this model class, we will devise a compression algorithm that encodes in fewer bits the trajectories with lower curvature. The properties of these trajectory models that take into account vehicle dynamics are:



(a) Ground plane positions

(b) Constant velocity

(c) Driver's actions

Figure 5.2: Different trajectory model classes correspond to different prior probabilities. The top row is an iconic representation of an example in each model class, on the ground plane (data in black, models in grey). The bottom row is a sketch of the probability distribution associated to each model class.

Distribution Trajectories which involve low changes in acceleration have higher probability.

Optimisation The optimisation algorithm needs to consider a models with *different* complexity.

Comparing the three model classes

Figure 5.2 illustrates the three trajectory model classes.

- A *ground plane positions* model associates a uniform probability distribution for the trajectories of given length (Figure 5.2(a)).
- The *constant velocity* model class contains only trajectories in a straight line, any other trajectory has zero probability (Figure 5.2(b)).
- A *driver's actions* model encodes the driver's actions (accelerate, turn left, etc.). This class assigns a higher probability to actions that require smaller changes in acceleration. We will develop this model class in the current chapter (Figure 5.2(c)).

The problem is how to define the *driver's actions* model class, using the prior knowledge we have about vehicle motion. Later on we will define an optimisation algorithm for these *realistic* trajectory models.

5.3.3 Reducing the number of parameters

Each piece of prior knowledge can be used to reduce the number of parameters required to represent a trajectory. Let us derive the properties of vehicle trajectories that we will use to compress the data.

The physical model

Let us consider again the trajectories as the position of a vehicle on the ground plane over time. But, for a moment, assume that $p(t)$ is continuous. Our development will be based on the following properties [TL02]:

Properties 5.1 Let $p(t)$ be the position of the vehicle as a continuous function of time t.

1. The state of the vehicle at a given instant is entirely defined by the position $p(t)$ and its derivatives $\dot{p}(t)$ and $\ddot{p}(t)$.
2. Acceleration $\ddot{p}$ is piecewise constant.
3. Changes in the acceleration are small.

A linear model

The state of the vehicle at a given point is defined by the position p, velocity $v = \dot{p}$ and acceleration $a = \ddot{p}$,

$$\begin{pmatrix} p \\ v \\ a \end{pmatrix} \in \mathbb{R}^6$$

The model is linear,

$$\begin{aligned} \dot{p} &= v \\ \dot{v} &= a \end{aligned}$$

and acceleration a is left unspecified.

But in fact we are not working with continuous functions, but just with discrete quantities. Therefore, as an approximation, we need to use a system of finite difference equations to model the dynamics [LL92].

Given the state at time t, $(p(t), v(t), a(t))$, the position and the velocity at time $t+1$ are calculated as follows

$$\begin{aligned} p(t+1) &= p(t) + v(t) \\ v(t+1) &= v(t) + a(t) \end{aligned}$$

assuming that the acceleration $a(t)$ is given for all t. This is a linear transformation

$$\begin{pmatrix} p(t+1) \\ v(t+1) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} p(t) \\ v(t) \\ a(t) \end{pmatrix}$$

which is implemented in the function `MotionModel`. This function generates the list of states of the vehicle given the initial state of the vehicle and a list of accelerations.

```
MotionModel[ initial_, acceleration_ ]:=
 FoldList[
  (LinearModel.#1) ~Join~ #2 &,
  initial ~Join~ {First[acceleration]},
  Drop[acceleration,1]
 ]
```

where `LinearModel` is the matrix

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

The function `MotionModel` essentially iterates over each time step and multiplies `LinearModel` by the previous state. In a sense, this function *decompresses* a compressed description of the trajectory.

But we can still squeeze more bits out of the acceleration, by making use of Properties 2 and 3 in page 100.

Run-length encoding the acceleration

Property 2 in page 100 is that the acceleration a is piecewise constant. When the constant intervals are long enough, a run-length encoding will provide a *compressed* representation of the data.

Run length encoding of a string consists in replacing repeated consecutive elements $\alpha, \overset{n}{\alpha \cdots}, \alpha$ by the element itself and the number of repetitions, (α, n). For example, in the following sequence the symbol α is repeated three times consecutively, and the symbol β is repeated four times.

$$\alpha\alpha\alpha\beta\beta\beta\beta\alpha \mapsto (\alpha, 3)(\beta, 4)(\alpha, 1)$$

The function `MotionModel` that we have just defined, takes as second parameter the list of accelerations $a(0), a(1), \ldots$. The following function run-length *decodes* its arguments to a list of accelerations that can be given to `MotionModel`. It replaces each occurrence (α, n) by the symbol α repeated n times.

```
DecodeRunLength[code_] :=
 Map[
  Table[#[[2]], {#[[1]]}] &,
  code
 ]
```

The run-length *encoder* does not need to be implemented for this application. The search for the best model, if we were satisfied with this representation, would take place in the space of run-length encoded trajectories. But, again, the properties in page 100 allow us to compress more the parameter space.

Changes in acceleration are small: incremental encoding

Property 3 in page 100 is that changes in consecutive runs of the acceleration are small. That is, given the run-length encoded accelerations,

$$(\alpha_1, n_1), (\alpha_2, n_2), \ldots$$

the quantity $\mid \alpha_j - \alpha_{j+1} \mid$ is a small number, for $j \geq 1$. Therefore we can compress the parameters even further, by storing them as differences between two consecutive acceleration vectors. In this way shorter descriptions correspond to trajectories with low acceleration, which is what we want to favour.

When mapping parameters into binary strings we do not need the encoder, but the decoder. The following function takes as argument a list of pairs (α, n) similar to the argument of `DecodeRunLength`, but interprets the second element, n as an increment over the previous value.

```
DecodeIncremental[ code_ ] :=
 FoldList[ {#2[[1]], #1[[2]] + #2[[2]]} &,
  First[ code ],
  Drop[ code, 1 ]
 ]
```

The functions `DecodeIncremental`, `DecodeRunLength` and `MotionModel` can be combined to produce a trajectory as a list of points on the ground plane from a compact description of the accelerations and initial conditions $p(0)$ and $\dot{p}(0)$.

The final trajectory model

Given the initial condition and the incremental run length encoded trajectory, the function `MotionStates` will compute the state of the vehicle (position, velocity and acceleration) at each time step.

```
MotionStates[ {initial_, incrRunLengthEncoded_} ] :=
 MotionModel[
  initial,
  Flatten[
   DecodeRunLength[
    DecodeIncremental[ incrRunLengthEncoded ]
   ], 1
  ]
 ]
```

The trajectory of points on the ground plane is the sequence of positions over time, computed by `TrajectoryPositions`:

```
TrajectoryPositions[compressed_Trajectory] :=
  PositionsOnly[ TrajectoryStates[compressed] ]
```

where `PositionOnly` extracts the first component of each state vector.

Example

Let us consider the example of Figure 5.3. The *compressed* parameters of a trajectory are given in Figure 5.3 (left). The rows of the left hand side array

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

are the initial position and orientation. The rows of the right hand side array

$$\begin{pmatrix} 10 & (0,0) \\ 10 & (0,0.1) \\ 5 & (0.1,-0.1) \\ 10 & (0,0) \\ 20 & (0,-0.1) \\ 10 & (-0.4,0.2) \end{pmatrix}$$

are the run-length encoded changes in the acceleration, starting from the initial acceleration. For example, the row

$$(5(0.1,-0.1))$$

is interpreted by the function `TrajectoryPositions` as "add the vector $(0.1,-0.1)$ to the acceleration during 5 consecutive time steps".

The corresponding trajectory (Figure 5.3, right) consists of the list of points on the ground plane produced by the function `TrajectoryPositions`.

Bit size. Let us now use the `Mathematica` interpreter to calculate the size in bits of the compressed form of the trajectory (the *realistic* trajectory model), and the trajectory as *ground plane positions*

First of all, we need to load the generic packages defined in Chapter 4 and examine for which models or *types* the function `Compress` is defined.

```
In[43]:= Needs["Thesis`Cost`"]
In[44]:= Needs["Thesis`Compress`"]
In[45]:= AcceptedTypes
Out[45]= {Integer, IntegerModel, List, Natural, Rational,
   Real, Vector}
```

Now we need to include the package `Thesis`TrajectoryCompress``, which adds two new types to the list of accepted types: `GroundPlanePositions` to represent ground plane positions, and `Trajectory` to represent *compressed* trajectory models.

```
In[46]:= Needs["Thesis`TrajectoryCompress`"]
In[47]:= Needs["Thesis`Trajectory`"]
In[48]:= AcceptedTypes
Out[48]= {GroundPlanePositions, Integer, IntegerModel,
   List, Natural, Rational, Real, Trajectory,
   TrajectoryModel, Vector}
```

Let us assign to `c` the *compressed* trajectory of Figure 5.3

```
In[49]:= c = TrajectoryModel[{{0, 0}, {0, 1}}, {{10, {0,
   0}}, {10, {0, 0.1}}, {5, {0.1, -0.1}}, {10, {0, 0}},
   {20, {0, -0.1}}, {10, {-0.4, 0.2}}}];
In[50]:= g = TrajectoryPositions[Trajectory @@ c];
```

The expression `c` is the *compressed* model. The expression `g` is the list of points on the ground plane that correspond to `c`, calculated using the function `TrajectoryPositions` which we just defined.

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 10 & (0,0) \\ 10 & (0,0.1) \\ 5 & (0.1,-0.1) \\ 10 & (0,0) \\ 20 & (0,-0.1) \\ 10 & (-0.4,0.2) \end{pmatrix}$$

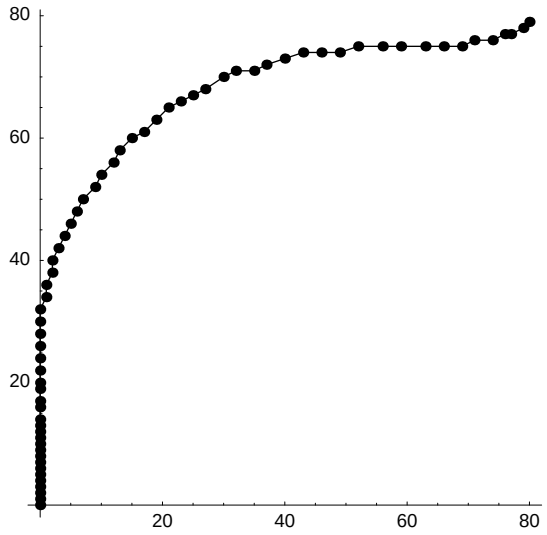


Figure 5.3: (Left) the parameters of a trajectory model. (Right) The trajectory on the ground plane.

Now we can compare the size in bits of both `c` and `g`. For simplicity, the function `Cost` as defined in the package `Thesis‘Cost‘` only accepts a restricted set of expressions that does not include `c`. But we can just use `Size` and `Compress` to calculate the cost of `c` in bits.

```
In[51]:= Size[Compress[c]]
Out[51]= 427
```

which is the number of bits required to represent the trajectory using the *compressed* model. Compare with the representation size of the trajectory as ground plane positions.

```
In[52]:= Size[Compress[g]]
Out[52]= 1668
```

5.4 A cost function and its optimisation

We have a function, `TrajectoryPositions`, that takes a *compressed* trajectory and returns a sequence of positions on the ground plane. In this sense, `TrajectoryPositions` plays the role of a decompressor.

So far we have studied the properties of the model, but no experiments in fitting the models to real data have been performed. For that purpose, we need to put together the function `TrajectoryPositions`, the cost function of Chapter 4, which is the size after compression, and an optimisation algorithm.

5.4.1 The cost function

The function `Cost` is defined in the package `Thesis‘Cost‘` of Chapter 4. In order to be of any practical use, the function `Compress` needs to be defined for trajectories.

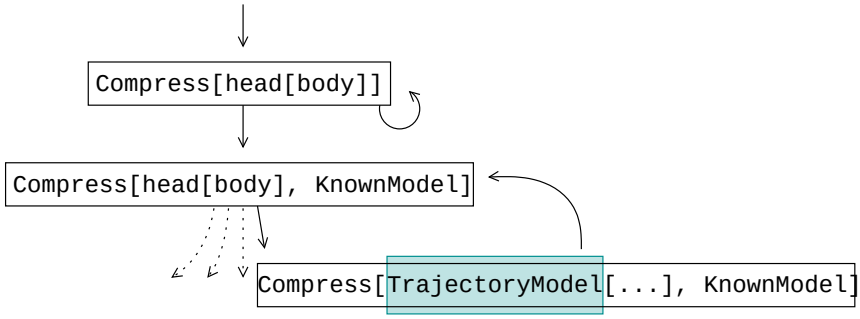


Figure 5.4: Additional function calls for the compression of trajectory models (compare with Figure 4.2 in Chapter 4).

According to the list of steps to extend the definition of `Compress`, given in page 83, we need to define:

- The function `Compress` for the model
- The function `Compress` for the data, given the model

In Section 5.3 the data type `TrajectoryModel` was used to represent *compressed* trajectory models. To compress a `TrajectoryModel` we just need to convert it into a binary string using the function `Compress` as defined for the data type `List`.

The compression of the data given the model consists in storing the difference, point per point, between the data and the trajectory on the ground plane defined by the model. The trajectory on the ground plane can be calculated from the model using the function `TrajectoryPositions` defined above in Section 5.3. The differences, as usual, can be compressed as small integers. In this way the function `Compress` is extended, in the package `Thesis‘TrajectoryCompress‘`, to compress trajectory data. Figure 5.4 illustrates the additional function call within `Compress` which is necessary to compress data whose model is `TrajectoryModel[...]`.

The trajectory cost function `Cost` that we have obtained is designed to be equivalent to the following definition:

Definition 5.2 Trajectory Cost Function The cost of a set of points on the plane $p_1, p_2, \ldots, p_n$, given the model $m(t)$ is defined as

$$c(\{p_1, p_2, \ldots, p_n\} \mid m) = \text{length}(\log^*(n)) + \sum_{t=1}^{n} \text{length}(\log^*(m(t) - p_t))$$

where the $\log^*$ function for encoding integers of Definition 4.4

Let us consider a dataset consisting of the position of an object at three time steps. Three points are too few for a realistic trajectory, but in this case it helps to see a dataset of size comparable to the model. In Section 5.5 we will see more complex examples. The data are

```
In[53]:= data = GroundPlanePositions[{0, 0}, {92, 115},
   {304, 193}];
```

which is displayed in Figure 5.5. The model family depends on a single parameter, `a`

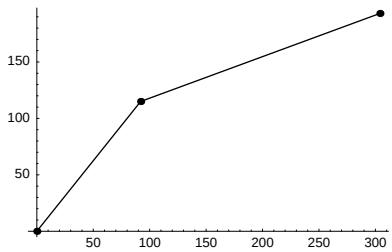


Figure 5.5: Example data.

$$\left(\begin{array}{cc} 0 & 0 \\ 100 & 100 \end{array} \right), \left(\begin{array}{cc} 3 & (\mathtt{a}, 0) \end{array} \right)$$

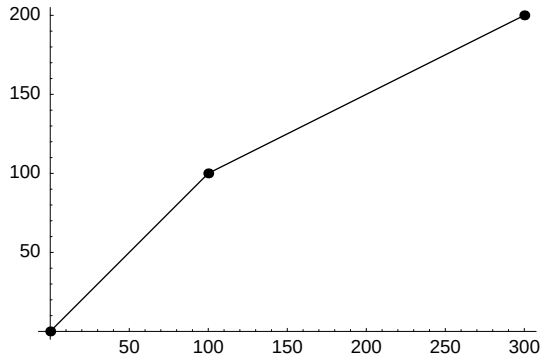


Figure 5.6: (Left) the parametric family of models that depends on the number a. (Right) example for parameter a=100.

```
In[54]:= model[a_] := TrajectoryModel[{{0, 0}, {100, 100}},
   {{3, {a, 0}}}]
```

An example trajectory is displayed in Figure 5.6. The model reads as:

- The trajectory starts at $(0, 0)$
- The initial velocity is $(100, 100)$
- The initial acceleration is $(\mathtt{a}, 0)$
- At each time step, the acceleration increases by $(\mathtt{a}, 0)$. There are three points in this trajectory.

The cost of the data for each parameter value is displayed in Figure 5.7. The two minima are at $a = 0$, which is the simplest model, and $a = 100$, which is the model that best fits the data. This is similar to the example of integers modelling integers of Chapter 4. The fact that the dataset is so small (just three points) makes the simplest model (parameter $a = 0$) feasible, and thus it has a low cost.

Let us compute the cost of the simplest model ($a = 0$), the model that best fits the data ($a = 100$), and a model at an intermediate value ($a = 50$).

```
In[55]:= Cost[model[0][data]]
Out[55]= 194
In[56]:= Cost[model[100][data]]
Out[56]= 200
In[57]:= Cost[model[50][data]]
Out[57]= 205
```

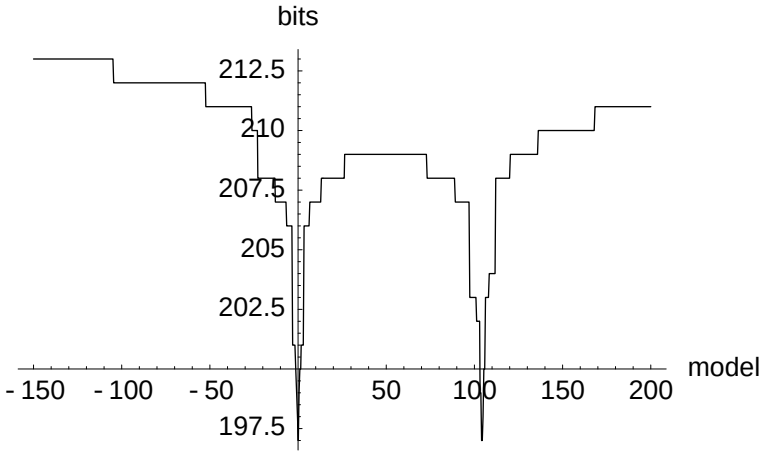


Figure 5.7: The cost of the models for $-150 < a \leq 200$.

Note that, although Figure 5.7 shows a clear double minima, the differences between the values are small. This also happened in the experiments of Chapter 3 and the reason might be the same, that the models provide similar explanations for most of the data, and only differ in a small "portion" of the data, which, nonetheless, is enough to discriminate between models.

5.4.2 Optimisation

Let us now estimate the minimum of the cost function. Given that the number of parameters is still high, we need an optimisation algorithm.

The idea is to fit trajectories to data incrementally in the number of points: a trajectory that approximates the first n points with low cost will be prolonged, in different ways, to fit the first $n+1$ points.

The trajectory is prolonged by adding an extra row to the "increments" part of the model (right hand matrix of Figures 5.3 and 5.6). Some prolongations are illustrated in Figure 5.8 and 5.9.

The optimisation function, named `CheapestTrajectory`, defined in the package `Thesis‘TrajectoryOptimization‘` is implemented as three recursive functions that call each other. One of them, `TrajectoryCandidate`, produces prolongations of a trajectory which will be used as candidates for the next step. The functions `BestTrajectoryModel` and `BestTrajectoryParameters` search exhaustively for the absolute minima among the set of candidates.

In the interactive examples of this chapter, the function used is `CheapestIteratorArray`, which is simply an exhaustive search for the minimum. The syntax is:

`CheapestIteratorArray[` f, $\{\{\alpha,\ \alpha_{\min},\ \alpha_{\max},\ \alpha_{\text{step}}\},\ \{\beta,\ \ldots\}\ \ldots\ \}$ `]`

which means that the function f is evaluated for parameters $\alpha, \beta, \ldots$ in the given intervals. The function `CheapestTrajectory` uses `CheapestIteratorArray`, and will be used in Section 5.5 to fit trajectories to synthetic and experimental data.

$$\begin{pmatrix} 0 & 0 \\ 100 & 100 \end{pmatrix}, \begin{pmatrix} 10 & (0,0) \\ 10 & (10\alpha, -10\alpha) \end{pmatrix}$$

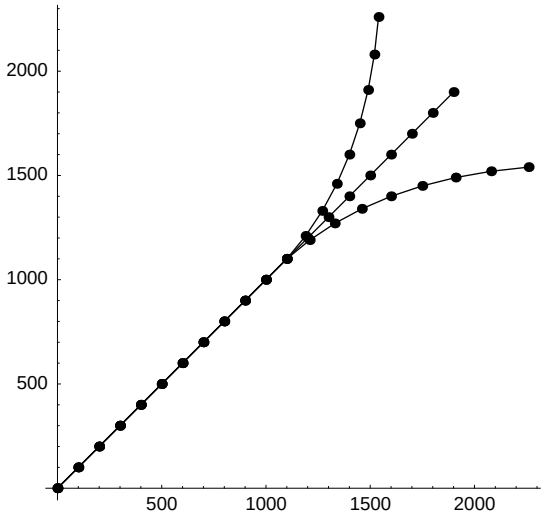


Figure 5.8: Trajectories defined in the interval $[1, 10]$ for $\alpha = -1, 0, 1$.

$$\begin{pmatrix} 0 & 0 \\ 100 & 100 \end{pmatrix}, \begin{pmatrix} 10 & (0,0) \\ 10 & (10, -10) \\ 10 & (10\alpha, -10\alpha) \end{pmatrix}$$

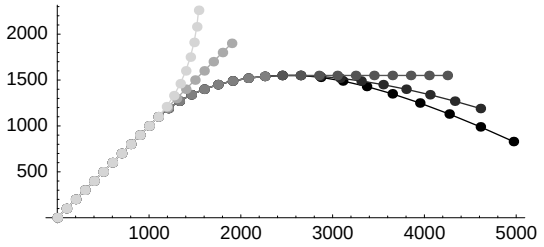


Figure 5.9: Prolongation of the trajectory of Figure 5.8 from the interval $[1, 10s]$ to the interval $[1, 20]$, by adding rows to the list of run-length encoded accelerations (right hand side matrix). The three longest trajectories in the plot are the displayed model with parameters $\alpha = -1, 0, 1$.

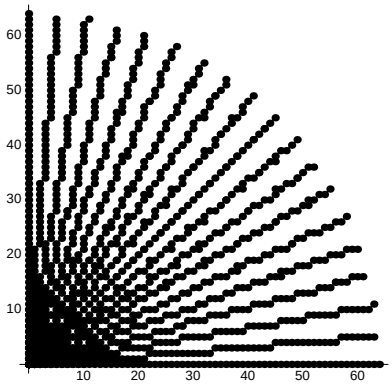


Figure 5.10: The range of models with variable initial angle, on the ground plane.

5.4.3 Example: a straight line

Let us see the algorithm working with a simple example, that will also allow us to compare the model and data size.

The models in the following family produce of 65 points with given initial speed and angle:

```
In[58]:= model[speed_, angle_] :=
   TrajectoryModel[speed*{{0, 0}, {Cos[angle],
   Sin[angle]}}, {{65, {0, 0}}}];
```

Each of those models is a trajectory in straight line. Figure 5.10 shows the family of trajectories when the speed is 1 and the angle ranges between 0° and 90° in steps of 5°.

The data are the points produced by the model when speed is 1 and angle is 45°.

```
In[59]:= data = TrajectoryPositions[Trajectory @@ model[1,
   45*Degree]];
```

The cost of `model[1][angle]` is given in Figure 5.11. For clarity, the scales are omitted. Note that the model used to produce the data is included in the family, and corresponds to the minimum.

The minimum is found at 45°.

```
In[60]:= Needs["Thesis`IteratorArray`"];
In[61]:= Needs["Thesis`TrajectoryOptimization`"];
In[62]:= CheapestIteratorArray[Hold[Cost[model[1.,
   angle][data]]], {{angle, 0*Degree, 90*Degree,
   5*Degree}}]
Out[62]= {angle -> 45*Degree}
```

The cost of `model[1.1][angle]` is given in Figure 5.12. The models are slightly faster than the trajectory, and there are two local minima, one at 40 degree, and another at 50 degree. The optimisation function only returns the last minimum.

```
In[63]:= CheapestIteratorArray[Hold[Cost[model[1.1,
   angle][data]]], {{angle, 0*Degree, 90*Degree,
   5*Degree}}]
Out[63]= {angle -> 50*Degree}
```

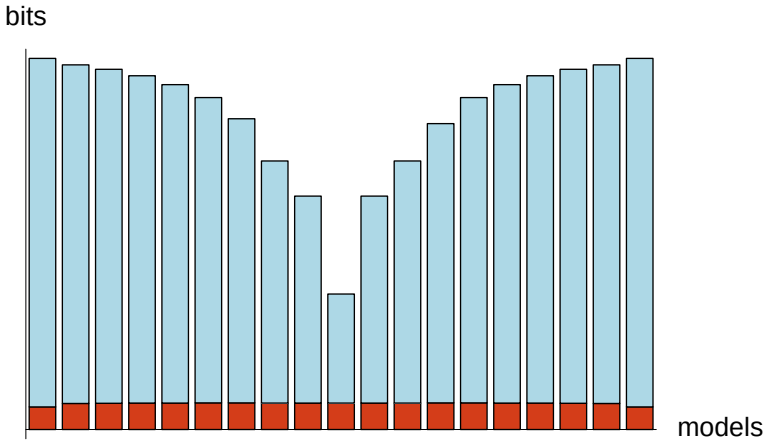


Figure 5.11: The cost in bits of the models with constant velocity and varying initial angle. The minimum is at the cost that generated the data.

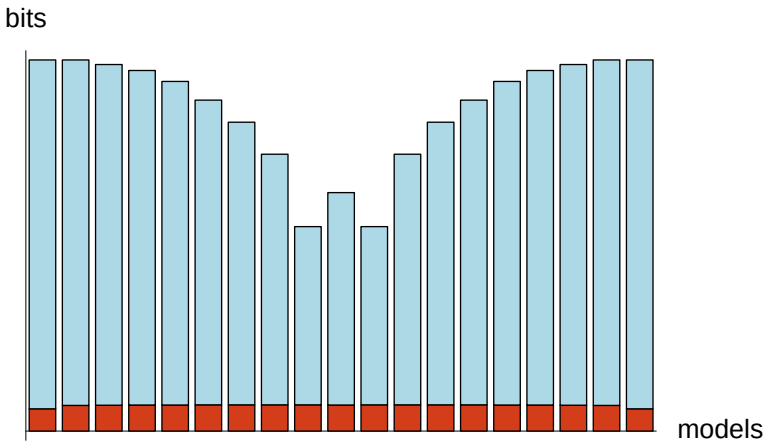


Figure 5.12: The cost in bits of the models with constant velocity and varying initial angle. The true speed is slightly higher than the estimated speed of the data.

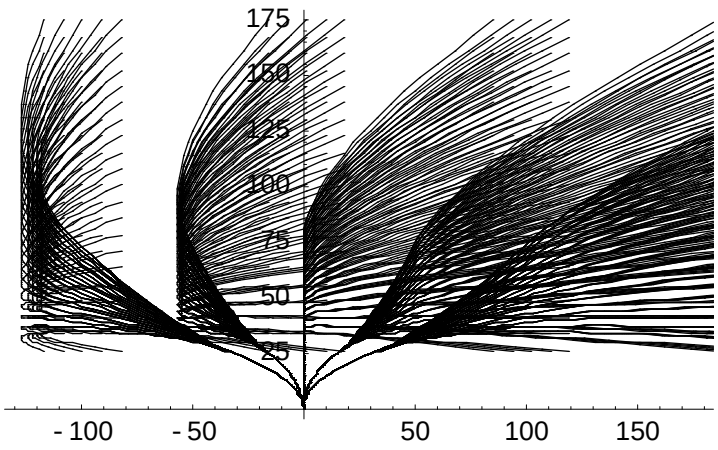


Figure 5.13: A family of trajectories with three parameters, on the ground plane.

5.4.4 Example: complex model and optimisation

Let us consider the trajectory of Figure 5.3. A family of three parameters is defined as follows for parameters `a`, `b` and `c`:

```
In[64]:= model[a_, b_, c_] := TrajectoryModel[{{0, 0}, {0,
   1}}, {{25, {a, 0}}, {20, {0, b}}, {20, {c, 0}}}];
In[65]:= Needs["Thesis`TrajectoryExperiments`"];
In[66]:= data = TrajectoryPositions[TrajectoryExampleData];
```

The range of trajectories included in this family with three parameters is displayed in Figure 5.13.

Let us compute which model has minimum cost.

```
In[67]:= CheapestIteratorArray[Hold[Cost[model[aa, bb,
   cc][data]]], {{aa, -0.1, 0.1, 0.05}, {bb, -0.05, 0.15,
   0.01}, {cc, 0.5, 0.7, 0.05}}]
Out[67]= {aa -> 0., bb -> 0.03, cc -> 0.65}
```

The model with minimum cost is displayed with the original data in Figure 5.14. Note that, although the fit is not exact, the "best model" is significantly simpler than the "model" used to generate the data, in Figure 5.3.

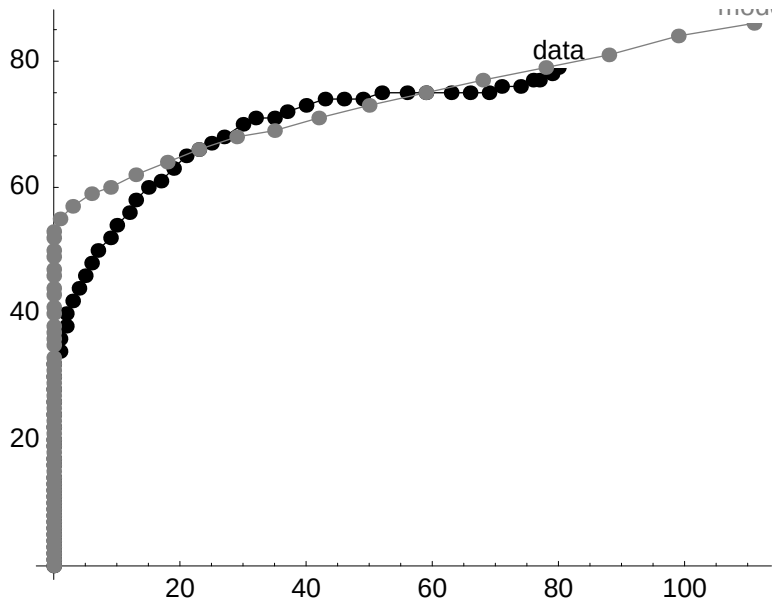


Figure 5.14: The data (black) and the best model (grey) on the ground plane.

5.5 Experiments

In this section the cost function and the optimisation method of Section 5.4 are used to select the models with minimum cost for synthetic and real trajectories.

The MDL method can be used to choose between models of different complexity, and one of its applications is in the discrimination of the best fit even in the presence of noise (which might have an effect in the model complexity). In these experiments we will fit models to data with and without noise and compare the optimal model complexity in each case.

Further, we would like to know that, if the compression ratio is useful in discriminating between models for the same data, could we use it to compare between fits for different data?

- Which is the relation between compression and quality of the fit. In particular,
  - Can we use compression to compare between different models $m_1, m_2, \ldots$ that describe the same data d?
  - Given datasets $d_1, d_2, \ldots$, let m_i be the *cheapest* model for dataset d_i, for $i = 1, 2, \ldots$. Can we use set a threshold r so that if the compression ratio [Say00] $\text{size}(d_i)/\text{size}(m_i) > r$ then the fit of model m_i to data d_i is considered *good* in any sense?
- The robustness of the fitting method in the presence of occlusions and noise in the data.
- The ability of the method to strike a balance between model complexity and goodness of the fit of the model to the data.

The first experiments use synthetic data, in which a simple but irregular dataset is modelled, and occlusions and noise are added.

These tests are followed by experiments with 21 real trajectories of vehicles on a ground plane, as produced by a prototype car tracker [Sul94] from real image sequences. Models of different complexities are fitted to the data.

5.5.1 Setup of the experiments

The software used for these experiments includes the functions `Compress`, `Cost`, `Cheapest` and `CheapestTrajectory` and additional functions for display, verification, data generation and debugging. The functions are written in `Mathematica` version 4.0, and run on a Sun workstation at 450 MHz. Each evaluation of the `Cost` function takes a few seconds. Each optimisation takes between ten minutes and half an hour depending on the model complexity.

The data consists of points on the ground plane given by pairs of coordinates. When the data are experimental measurements on the ground plane, the spatial units are centimetres, and therefore integers are used for the coordinates on the ground plane.

5.5.2 Synthetic data

The main question to ask is under which circumstances the method described in the previous sections strikes a balance between model and data complexity.

In particular, we are interested in seeing if the cost function assigns higher, values to trajectories under the effect of noise on the data.

The family of models is divided into classes. The models within each class have similar complexity. The complexity of a model is given by the number s of rows in the run-length encoded acceleration matrix

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \begin{pmatrix} n & (\ldots,\ldots) \\ \vdots & (\ldots,\ldots) \\ n & (\ldots,\ldots) \end{pmatrix}$$

The parameter n is the step size. The total number of points in the trajectory is sn. Since all trajectories have the same length, 32, the step size n is $32/s$. Higher step size n corresponds to lower complexity s.

The simplest models have one row $n = 1, s = 32$, and the model is a vehicle with constant velocity. The most complex models have eight rows, $n = 8, s = 4$. The experiments consist in finding the model m_n with complexity n that yields lowest cost, for $n = 1, 2, \ldots$. Then the cost of the models m_1, $m_2, \ldots$ are compared to see which is the optimal complexity for the data.

A typical trajectory is shown in Figure 5.15. Each point is an element of the trajectory, and the lines join segments in sequence. In the case of synthetic data, the trajectory begins at (0,0), the bottom-left corner of the plot.

In the following experiments only the step size n (and therefore, the number of iterations s needed to cover the entire trajectory) varies from one experiment to the next.

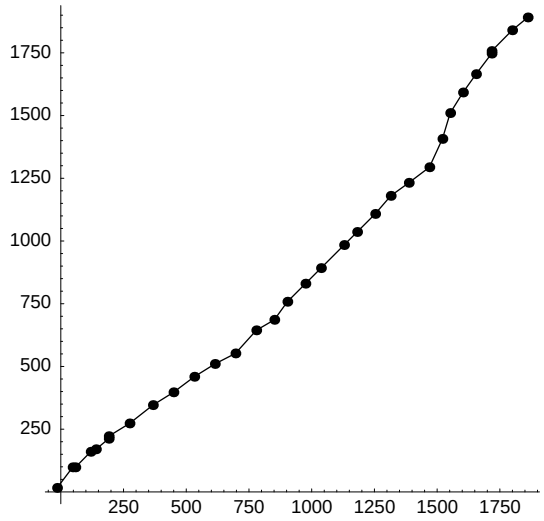


Figure 5.15: Dataset `Hand`: A simple trajectory obtained by clicking on the screen. The units are arbitrary.

Simple trajectory

Synthetic data can be created by clicking on the screen and converting each point to a pair of coordinates on the ground plane. The dataset `Hand`, displayed in Figure 5.15, was created in this way. It is sufficiently simple to illustrate the basic properties of the method. In the next section we will use trajectories obtained experimentally.

First of all, we run the algorithm that finds the cheapest trajectory on the dataset `Hand`, using a different step size. The step size that yields the best compression is considered the best one.

Then we create the datasets `Occlusions` and `Noisy` from `Hand` and run the optimisation algorithm again. The comparison between minima at different step sizes, allows us to estimate the extent to which the distorted datasets affect the performance of the algorithm.

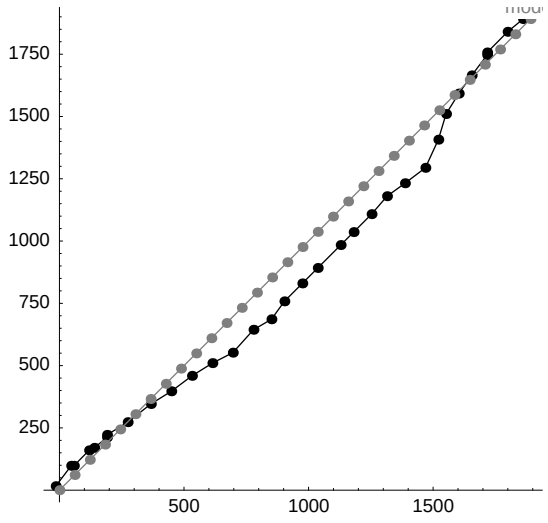
The optimisation algorithm has the following parameters:

- The initial speed is in the range [50, 100] which is searched in steps of one unit.
- The initial angle is in the range (0°, 360°), which is searched in steps of 15°.
- The increment intervals, both for the x and y variables, are in the range [-40, 40] which is searched in steps of one unit.

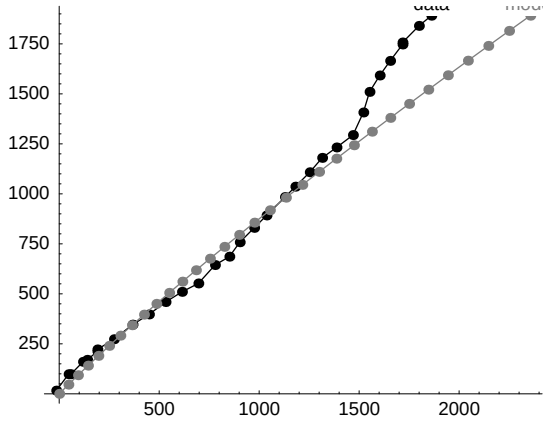
Note that the spatial units are the same as in the plots.

The results for different step sizes are displayed in Figure 5.16, which shows the model with lowest cost when the step size n is a constant, for $n = 4, 8, 16, 32$. The data is displayed in black, and the model is displayed in grey.

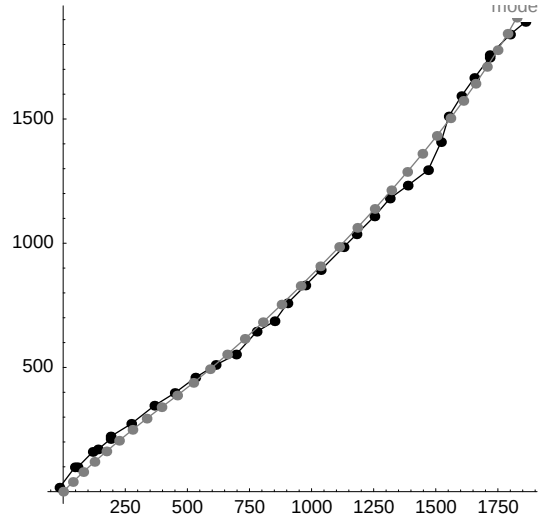
Comparing the minima for each step size n, the minimum is attained at $n = 16$. Therefore, a model of medium complexity produces the shortest description of the data. The model with higher complexity $s = 4$ gives a close fit to the data, whereas the model with lower complexity $s = 1$ is the simplest. The lowest cost is reached in between, avoiding the use of an overly complex model.



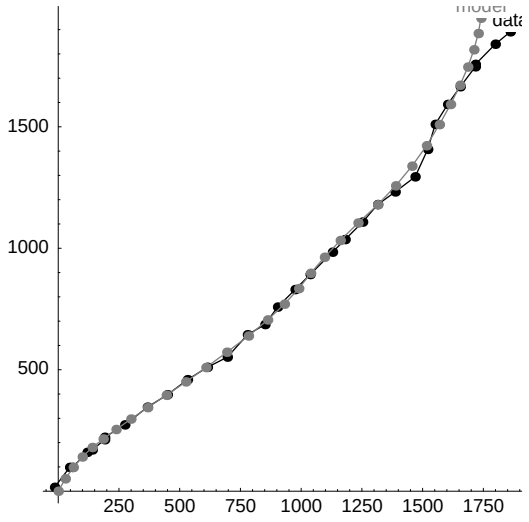
(a) $s = 1$, $n = 32$, $c = 1134$



(b) $s = 2$, $n = 16$, $c = 1133$

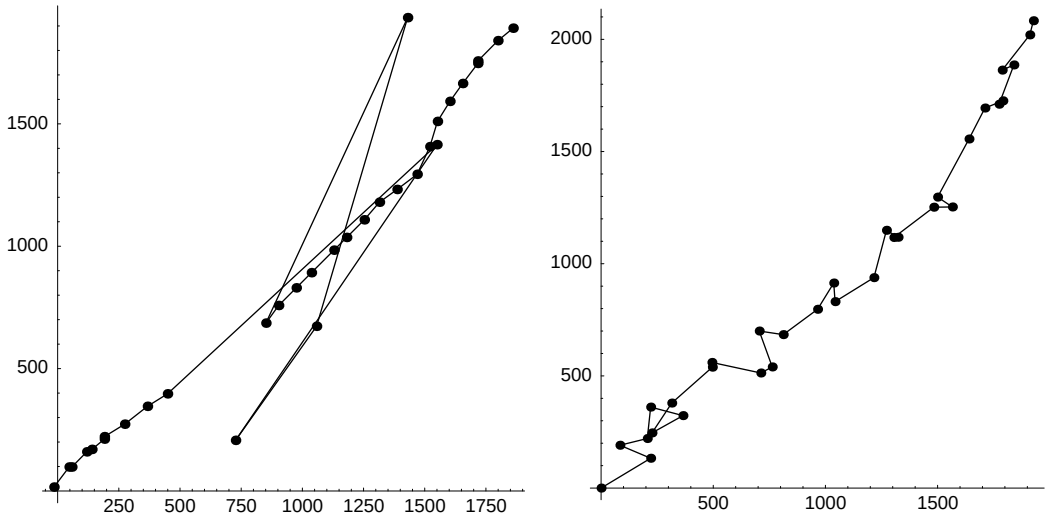


(c) $s = 4$, $n = 8$, $c = 1121$



(d) $s = 8$, $n = 4$, $c = 1178$

Figure 5.16: Dataset `Hand`. Best fits for varying model complexity s.



(a) Dataset `Occlusions`: modified measurements simulating occlusion in a range of points.

(b) Dataset `Noisy`: uniform random noise added to both coordinates of each point

Figure 5.17: Datasets derived from `Hand`

Simulating occlusions

Occlusions consist of the total or partial disappearance of the object being tracked, because other objects come between it and the camera. This situation is typical of busy traffic scenes. For this reason, 3D models for tracking are often chosen on the basis that they can help to overcome occlusion.

Occlusions are simulated in the dataset `Hand` by modifying the position of points on the plane for a range of points in time. Instead of replacing points by a "null" symbol that would represent the existence of no measurement, the points in the occluded range are displaced to random locations. The underlying assumption is that a measurement is always given for the position of the vehicle at those images, even if it is wrong. The result is the dataset `Occlusions` in Figure 5.17(a).

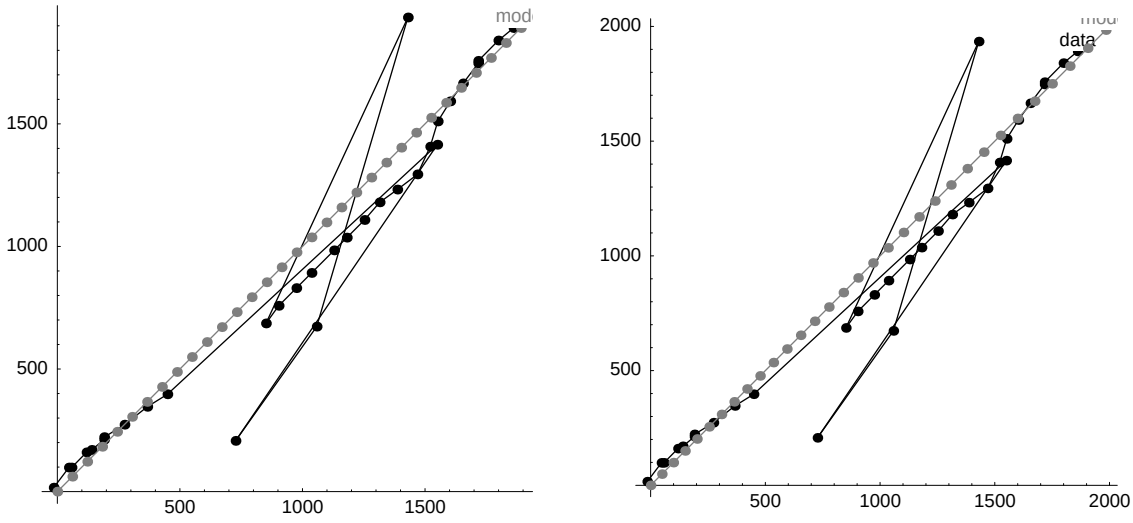
Running the same optimisation algorithm as above, with the same parameters, yields the cost values of Figure 5.18. The optimal number of segments is 2 (step 16). Similar to the case for the dataset `Hand`, a balance is reached between model complexity and closeness to the data.

The large errors in the data do not prevent finding a good model for the trajectory, which illustrates that the fitting method is not easily influenced by outliers.

Simulating noise

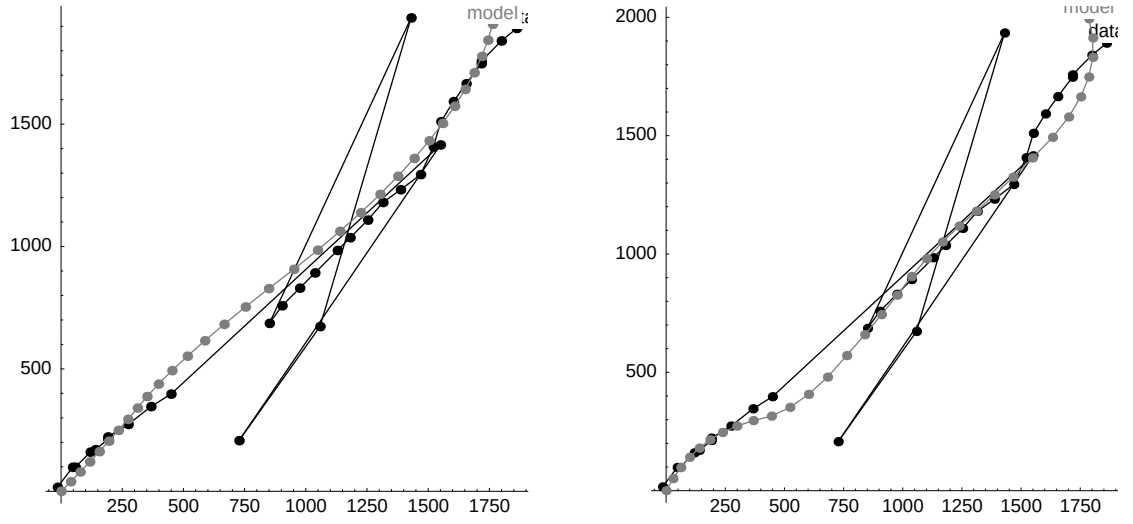
Let us now consider the case when the data just has additive noise. Noise in the data can be simulated by simply adding small random values to the coordinates of the ground plane positions.

In the case of these experiments, uniformly random displacements (in the interval $[0, 200]$ for each coordinate) are applied to each of the ground plane positions that form



(a) $s = 1$, $n = 32$, $c = 1152$

(b) $s = 2$, $n = 16$, $c = 1203$



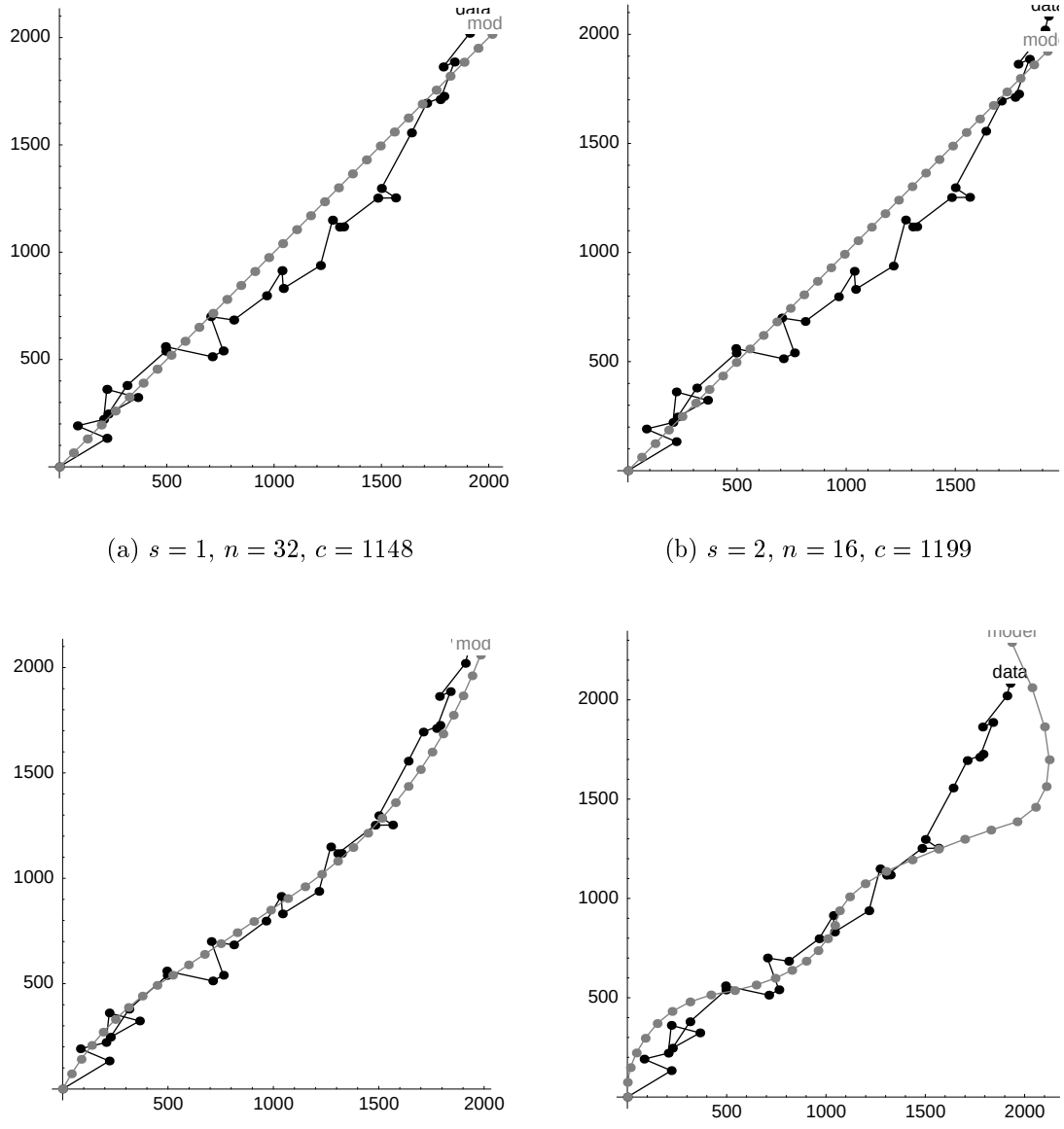
(c) $s = 4$, $n = 8$, $c = 1204$

(d) $s = 8$, $n = 4$, $c = 1281$

Figure 5.18: Dataset `Occlusions`. Best fits for varying model complexity s.

the data, to create a noisy version of the trajectory. The result is the dataset `Noisy` as displayed in Figure 5.17(b).

The optimisation algorithm, with the same parameters as for the datasets `Hand` and `Occlusions`, gives the cost values of Figure 5.19. The cheapest model is the one with lowest complexity (step size $n = 32$). Note that this complexity is lower than the complexity of the cheapest models for the datasets `Hand` and `Occlusions`.



(a) $s = 1$, $n = 32$, $c = 1148$

(b) $s = 2$, $n = 16$, $c = 1199$

(c) $s = 4$, $n = 8$, $c = 1238$

(d) $s = 8$, $n = 4$, $c = 1385$

Figure 5.19: Dataset `Noisy`. Best fits for varying model complexity s.

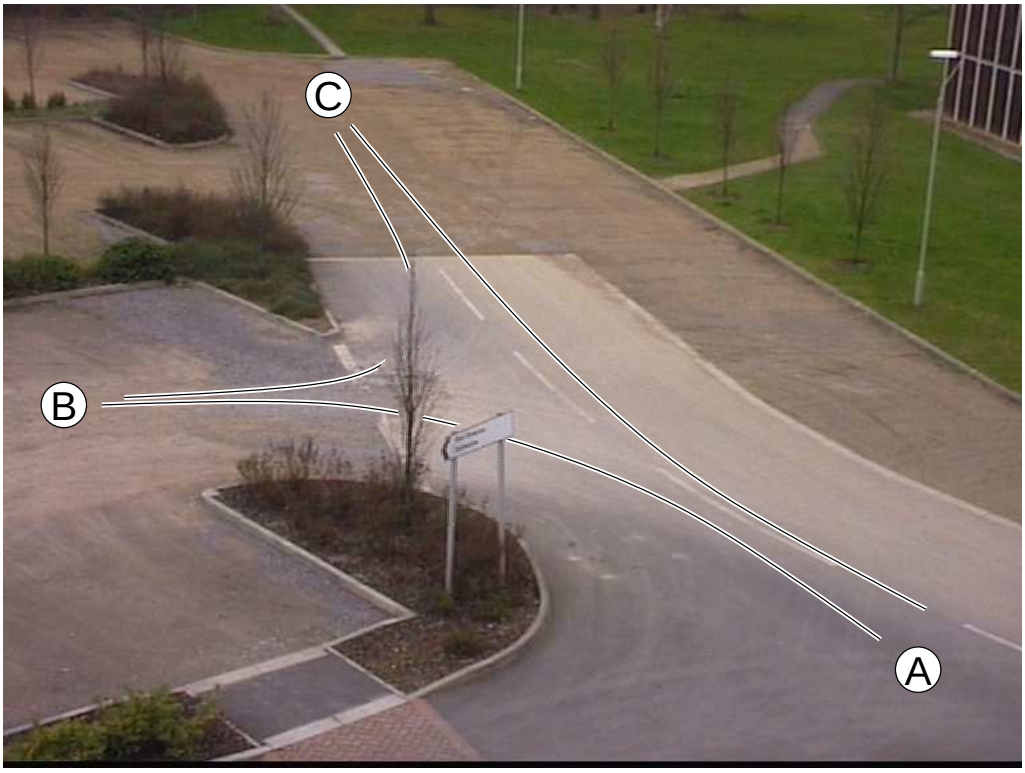


Figure 5.20: The car park scene. The trajectories begin and end at the points A, B or C.

5.5.3 Experimental data

Let us now examine the performance of the cost function with data obtained from measured vehicle trajectories.

The dataset consists of 21 trajectories produced by a car tracker [Sul94]. The data was produced by a vehicle moving on a car park between the points A, B and C of Figure 5.20. The trajectories are labelled with the format $\alpha\beta n$ where α is the starting point, β is the end point, and n is an index to distinguish between different trajectories.

In order to evaluate how the method handles different model complexities, the experiments consist in optimising using different step size n. This means that the data is divided into contiguous segments of n points. At each iteration in the optimisation, a segment of n points is added to the data being modelled. This corresponds to the leftmost index in the run-length encoded description of the trajectory. For example, the step size in Figure 5.8 is 10.

This is similar to what we have seen with synthetic trajectories. A difference between these experiments and the experiments with synthetic data, described in page 113, is that now the datasets have different lengths, and therefore there is a variation in the number of steps required to cover each dataset.

The optimisation algorithm is applied to find the cheapest model that fits each of the trajectories. There are 21 trajectories, and 9 different step sizes $n = 4, 5, 6, 7, 8, 9, 10, 11, 12$.

In a first approximation, a visual comparison *by human perception* has been done between the model and the data, and a rank between 0 and 4 was assigned. The

	$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s = 8$	$s = 9$	$s = 10$	$s = 11$	$s = 12$
ab1									
ab2									
ab3									
ab4									
ac1									
ac2									
ba1									
ba2									
ba3									
ba4									
ba5									
bc1									
bc2									
bc3									
bc4									
ca1									
ca2									
cb1									
cb2									
cb3									
cb4									

Table 5.1: Rough estimate of the quality of the fit for each trajectory and each step size.

purpose is to have an overview of the performance of the fitting algorithm, so that we know which model complexity (as step size n) and trajectory are worth looking at. The results for varying dataset and model complexity are displayed on Table 5.1. This table is used as a guide for the properties we want to look at.

Understanding Table 5.1. The scores are produced by a *subjective* classification according to the following rank:

 model fits most of the data and ends at the same location.

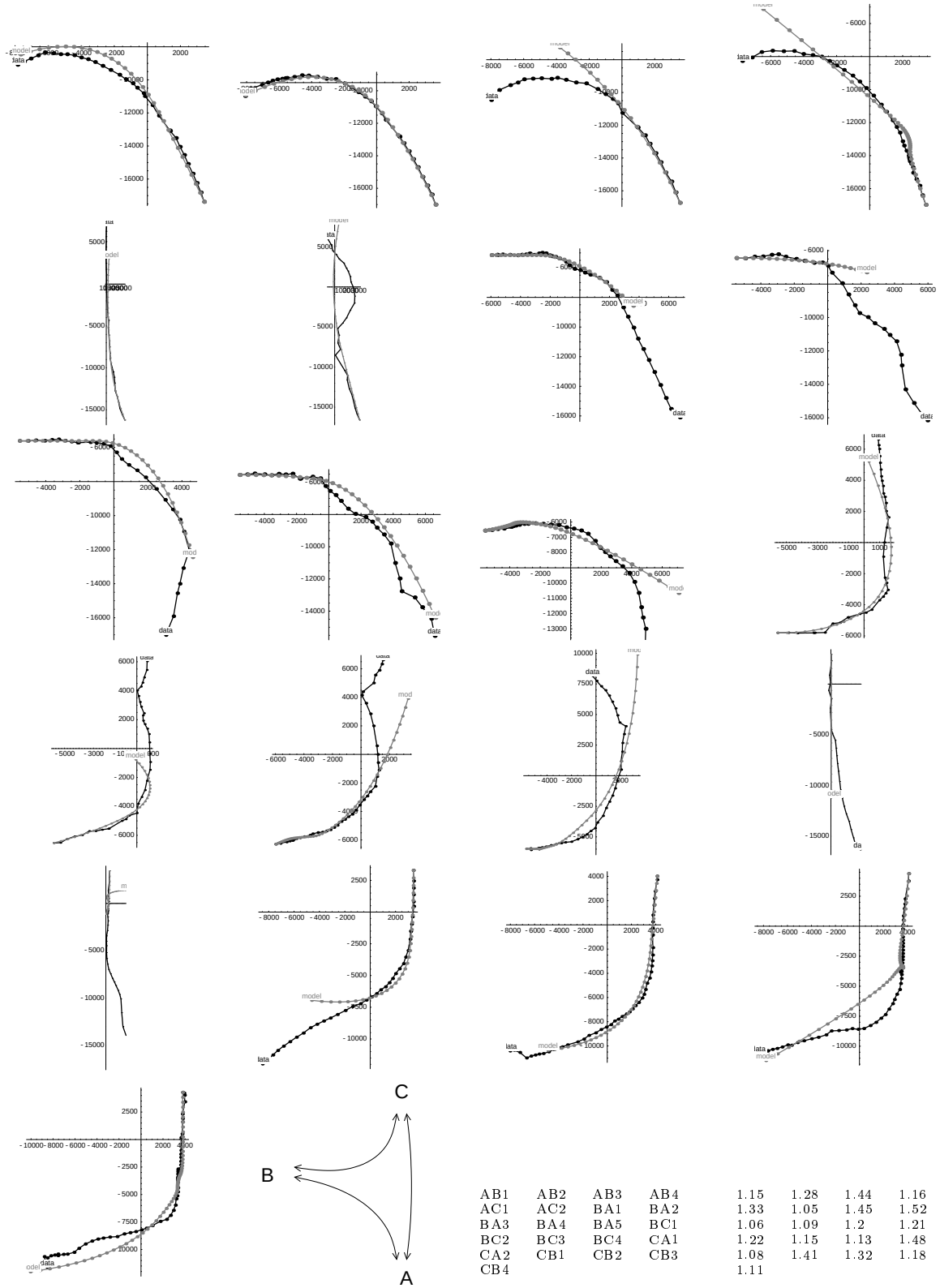
 model fits most of the data, but ends at a different location.

 model fits about half of the data.

 the first few points of model and data coincide. This corresponds to a successful search for the initial angle.

 the trajectory given by the model is completely different to the data

Best model complexity. We can see in Table 5.1 that the best results are obtained for step size of around 10. Let us look more closely at the results for step size 10.



AB1	AB2	AB3	AB4	1.15	1.28	1.44	1.16
AC1	AC2	BA1	BA2	1.33	1.05	1.45	1.52
BA3	BA4	BA5	BC1	1.06	1.09	1.2	1.21
BC2	BC3	BC4	CA1	1.22	1.15	1.13	1.48
CA2	CB1	CB2	CB3	1.08	1.41	1.32	1.18
CB4				1.11			

Figure 5.21: The fitting method applied to 21 experimental trajectories, with $n = 10$. The compression ratio of each best fit is given at the bottom-right table.

Figure 5.21 shows the trajectories, their models, and the compression ratio achieved when the model is used to compress the data. As usual, the gray lines correspond to the model, the black lines are the data. Each model is the best fit found by the optimisation algorithm.

Optimisation parameters. The parameters for the optimisation are:

- The data were taken in non-overlapping sequences of 10 images. That is, the best trajectory for the first ten points was found. Then it was prolonged in different ways to include the next 10 points. The best one was chosen, and so on.
- The initialisation intervals were
  - Speed between 0 and 1000 centimetres per 4 images in steps of 50, which is between 0 and 250 cm. per image in steps of 50. Thus the maximum speed is 62.5 metres per second or equivalently 225 kilometres per hour, well beyond the speed limit.
  - Angle between 0° and 360° in steps of 5 degrees.
- The acceleration increment intervals were the same along each axis, between -40 and 40 in steps of 5 units. The acceleration units are cm/s^2.

Compression ratio as a measure of quality of the fit. When examining how the method works with different datasets we will not be able to compare the value of the cost function, which is the size after compression. This is a consequence of the trajectories having different lengths. For example, a model fit to a short trajectory will almost always have a low cost value, independently of whether the fit is good or bad.

A better metric might be the compression ratio [Say00]

$$\frac{\text{size before compression}}{\text{size after compression}}$$

as an attempt to make the comparison independent of the size of the data. It is expected that a high compression ratio will correspond to a good fit, but this is not true across different datasets. as will be discussed in Section 5.6. The compression ratio of each best fit is included in Figure 5.21.

5.6 Discussion

The experiments of Section 5.5 have been produced by combining

- A computable approximation to the model selection method.
- A realistic model for vehicle trajectories based on the driver's sequence of actions over time.
- An optimisation method.

The following four issues are discussed below:

Tradeoff. The main property of the method is that it strikes a balance between model complexity and accuracy of the fit to the data.

Optimisation. The experiments illustrate the fact that, by choosing a model based on the physical properties of objects in motion, we can reduce the number of parameters of the model class. This makes optimisation easier.

Compression. Using a model class based on the physical properties of the motion of the vehicle, we have achieved compression of the trajectories. At the same time, the compression ratio can be used to choose between models for a particular dataset d. But, the question now is, can the compression ratio be used to compare the match of models across different datasets $d_1, d_2, \ldots$?

5.6.1 Tradeoff using different model complexities

The main motivation behind the experiments with the synthetic datasets was to compare models with different complexity.

The question is whether one can expect the model fitting method to choose a model with intermediate complexity, that adjusts to the properties of the data. For example, given datasets d and d', where d' is d plus noise, one would expect a simpler model to be best fit for d' than d, because there is less useful information in d' than in d.

As it can be seen from Figure 5.16, the preferred model for the dataset `Hand` was made of segments of eight points, $n = 8$. Under the same conditions, the preferred models for the same dataset but affected by occlusions, `Occlusions`, were both the models that divided the trajectory in segments of 8 and 16 points (Figure 5.18). When random noise was added to each point, dataset `TrajectoryNoise`, the preferred model was the simplest one that considered the trajectory as a straight line with constant speed, Figure 5.19.

We can interpret the choice of a simpler model when the data has noise added, to the loss of *information* that the data has suffered. The model is not describing the same data in `Hand` and `Noisy`. The model describes a smaller portion of the information conveyed by the data in the later case.

As a consequence, we can expect this model fitting method to fail gracefully as the amount of noise increases.

5.6.2 Optimisation and additional knowledge

A model class and a cost function that assigns a cost according to the data, is of no use without a way to find the model with minimum cost.

The problem encountered when deciding about which trajectories to consider in Chapter 3, was to avoid a high number of parameters. The solution at that point was to restrict the family of trajectories to those that describe trajectories with constant velocity. In this way the optimisation became a search on two parameters.

The solution proposed in this chapter has a higher number of parameters, and therefore an optimisation algorithm benefits from additional knowledge. This additional knowledge is fact that the trajectory model is somehow local: the state of the vehicle at time t only depends on the previous four time steps. As a consequence, the

prolongation of the best fit for kn points, for integers k and n, is a good estimation of the best fit for $(k+1)n$ points.

Failure with high model complexity

In Table 5.1 we can see an almost total failure to fit models of low step size n. The bad performance of the fitting method for high model complexity (step size $n = 4$) might be for either of the following reasons:

- the parameters of the search were set for trajectories with longer step size n. Thus, a trajectory with step size 4 tends to swing around the data, always with higher acceleration (as in Figures 5.18 and 5.19, both bottom right), or
- a small step size n needs a higher number of steps s, each of them corresponding to three more parameters. Thus the size of the model increases more than the benefits of using a more accurate model.

The plots for the fits with step size $n = 4$ show that the "best fit" tends to be a straight line. Therefore we can conclude that we do not need to refine the parameter set, but simply that the model complexity increases too much.

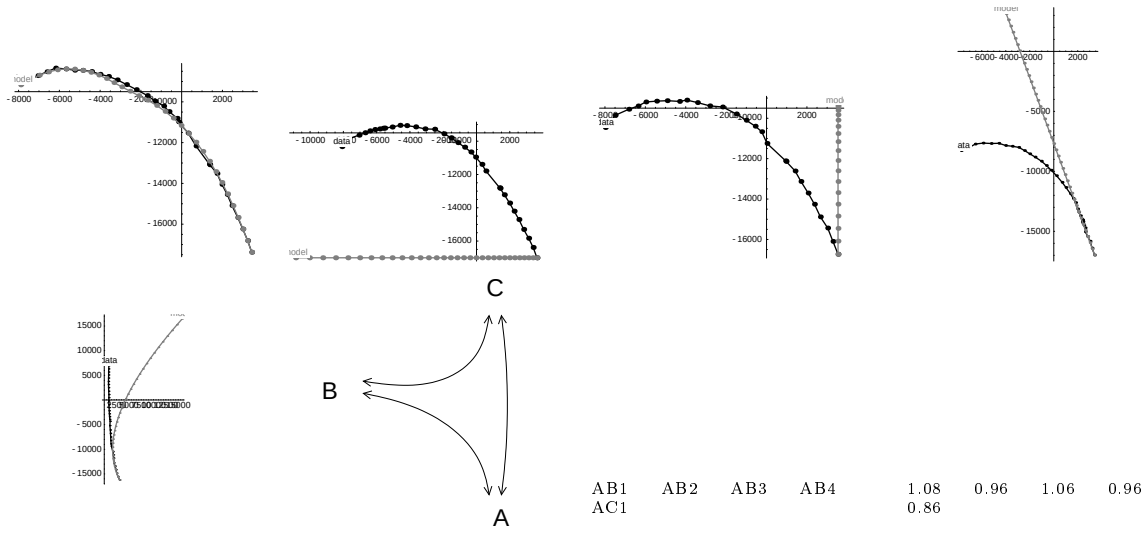


Figure 5.22: The fitting method applied to some of the 21 real trajectories, when each data iteration processes $n = 4$ points. It fails in most cases. The compression ratio of each best fit is given at the bottom-right table. Note that in most cases the compression ratio is lower than one (no compression achieved).

5.6.3 Compression ratio as measure of understanding

The cost function of models has been used to compare the way different models $m_1, m_2, \ldots$ describe the data d. The best model is the shortest one m_i. In terms of data compression, this is also the model with highest compression ratio

$$r_j = \frac{\text{length}(d)}{\text{length}(m_j)}$$

It seems reasonable to take the MDL-based method one step further and question whether the compression ratio conveys any additional information. The answer can only be positive when datasets of different sizes are considered. For example, we might want to compare different model classes $\{m_{ij}\}_{i,j=1}^{n,m}$ with different data sets $\{d_i\}_{i=1}^{n}$. Can a family of models be considered better just by achieving a higher compression ratio?

One idea would be to choose a threshold, and only take as a good fit the trajectories that achieve a compression ratio above the threshold.

But the experiments show this would not work. Trajectories AB3, BA1, and BA2 of Figure 5.21 have the highest compression ratios of the whole dataset while their fit is particularly bad. And, otherwise, there are particularly good fits, such as the one for trajectory AC2, with a low compression ratio.

The reason behind these differences between compression ratios might in that the best fits to the trajectories AB3, BA1 and BA2 can still model a large portion of the data, while the best fit to the trajectory AC2, although correct from the point of view of an observer, is only explaining a smaller part of it, and considering the rest of it noise. The conclusion is that different datasets, albeit produced by the same source, contain different amounts of noise.

5.7 Conclusion

This chapter illustrates two points:

- The MDL principle at work when choosing between models of different complexity
- A class of trajectory models that can be used with the cost function of Chapter 3 for vehicle tracking.

5.7.1 MDL at work

We have used the MDL method of Model Selection to choose a trajectory model among a family of trajectory models with different complexities. Vehicle trajectories were defined as models which represent the trajectory in a number s of segments. Model complexity is approximately the number of segments. In fitting trajectory models to data, a tradeoff needed to be established between the complexity of the model, and the quality of the approximation it provides to the data. The MDL-based method presented in this chapter provides a solution for those cases.

Generalisation to image sequences. The method presented in this chapter uses the information provided by the data points only to evaluate the cost function. The data, which currently is a sequence of points on the plane, could be replaced by a sequence of images. The cost function, instead of taking points on the plane as data, would take the image sequence as data, and such a cost function has been presented in Chapter 3. The MDL principle is applied iteratively in order to obtain the model trajectory which yields the shortest description of the data. This would work the same way if the data consisted of points on the plane or a sequence of images.

5.7.2 The trajectory models

The trajectory models used in this chapter correspond to data (de)compressors optimised for the trajectories of vehicles, according to the physical properties of their motion.

The state of the vehicle at each time step is a function of the initial conditions (position, velocity, acceleration), together with the increments in acceleration. The acceleration is a piecewise constant function of time, corresponding to the drivers' pressing of the pedals and turning of the steering wheel.

This representation of the trajectory in terms of initial conditions and the accelerations, can be used to produce compact descriptions of the trajectories with higher probability.

Better compression is achieved when the model is a more accurate description of the data. Experiments are used to find good fits to synthetic and experimental trajectories.

The method is also robust against noise and occlusions. The presence of noise and occlusions makes the method prefer simpler models.

Chapter 6

Trajectory analysis

This chapter presents a method for the analysis of vehicle trajectories produced by a vehicle tracker. The input is a list of points on the ground plane. The output is a labelling of the trajectory according to four possible actions of the driver: accelerate, break, turn left and turn right.

$$\text{Vehicle tracker} \xrightarrow{\text{points on the ground plane}} \text{Trajectory analysis}$$

The contents of this chapter have been published in [FM98]. The techniques used in this chapter are a departure from the MDL method used in the rest of the thesis.

6.1 Introduction

We are given a vehicle trajectory as points on the ground plane, typically the output of a vehicle tracker. This trajectory is a list of *ground plane positions*, $p(1), \ldots, p(n)$, taken at regular time steps. Each position at time step $1 \leq t \leq n$ is a pair of coordinates $p(t) = (x(t), y(t))$ on the ground plane. Our purpose is to label the trajectory of the vehicle with a sequence of labels, one per time step, that roughly describes the purpose of the driver at each time step.

One way to approach this problem is to approximate the ground plane positions with a continuous function of time, $f(t)$, and classify the vehicle's motion according to the values of the derivatives of f. For example, a vehicle is accelerating at instant t when the norm of $\dot{f}(t)$ is above a threshold. This approach has two problems.

- The continuous approximation f should take into account that the data corresponds to the motion of a vehicle. In particular, approximating the data with a generic family of polynomials will require taking into account that many polynomials do not correspond to a realistic vehicle trajectory. Thus, a polynomial f could approximate $p(t)$ for continuous t, but the values of the derivatives $\dot{f}(t)$ and $\ddot{f}(t)$ might be very different.

- Even when the position, velocity and acceleration of f are similar to the original source of the data, and therefore $\dot{f}, \ddot{f}$ are good estimates of the velocity and acceleration of the vehicle, the vehicle's states are not necessarily a good measure of the driver's actions. For example, the vehicle might be moving momentarily to the left even though the driver is turning to the right, or the tracked trajectories might contain errors that need to be filtered out.

Those two problems will be addressed in this chapter. In particular,

- A trajectory model is fit to data produced by a vehicle tracker. This trajectory model is a low degree and low curvature polynomial.
- A method for the analysis of trajectories and their labelling is proposed. It is based on a Hidden Markov Model with four internal and observed states.

The low curvature approximation of trajectories uses the least squares error model, together with a problem statement that allows a solution in closed form. Hidden Markov Models are used to model the driver's actions over time with four simple internal states and four observed states. The observed states are obtained using thresholds in the derivatives of the low curvature approximation to the trajectory. Viterbi's algorithm is used for finding the most probable sequence of internal states. A Hidden Markov Model is trained with frequencies from a dataset of 21 trajectories. The performance of the model for labelling of trajectories is illustrated with experiments.

6.2 Low curvature trajectory approximation

In this section a continuous approximation of a trajectory is defined. The approximating function is designed to be both close to the original data and to have low curvature.

The approximation is defined as the parameter for a particular cost function that minimises at the same time

- the distance between the approximation and the given measurements and,
- the curvature of the approximation.

We will divide the problem in two steps:

- Polynomial approximation: the data is first approximated using a polynomial, which might have high degree. This polynomial serves as continuous approximation to the data.
- Variational problem: the variational equation is solved under the constraint that the solution is a low degree polynomial that might not fit the data, but approximates it.

Let us first define the cost function as a functional. Later on we will study how to minimise the functional.

6.2.1 Definition of the cost function

Input. We are given a list of n points on the ground plane, $p(1), \ldots, p(n)$, which are produced by a vehicle tracker at equal time intervals. These are the measurements that will be used in the low curvature approximation.

Output. The purpose is to find a smooth approximation f to the measurements, with the property that the curvature of f should be reduced, as it corresponds to the motion of a moving vehicle. In [MWS96b, MW97] vehicle motion on the ground plane is modelled by stochastic differential equations involving the speed and the steering angle. The constraints in the equations are realistic in not allowing vehicles to slip sideways: the velocity vector is always directed along the axis of the car.

The cost function. The motion model assigns to each possible trajectory f of the car a cost $\texttt{CurvatureCost}(f)$, which is high if f has a high acceleration or rapidly changing curvature. An additional cost $\texttt{ApproximationCost}(f)$ depends on the compatibility between the trajectory f and the measurements. The total cost is a weighted sum of `CurvatureCost` and `ApproximationCost`. The most likely trajectory is estimated by minimising the weighted sum over a suitable space of trajectories. An expression for that cost appears in [MW97]. The alternative proposed here consists in splitting the problem in two steps:

1. Approximate the data, which are discrete points on the ground plane, using a low degree polynomial $\hat{f}$ that fits the data closely, even though $\hat{f}$ might have high curvature. The polynomial $\hat{f}$ is a continuous approximation to the data.

2. Approximate $\hat{f}$ by a function f with low curvature.

In order to choose an approximation f to $\hat{f}$, let us define a cost for f as a measure of how close is f to $\hat{f}$, and the curvature of f:

$$\texttt{TotalCost}(f) = \mu \cdot \texttt{CurvatureCost}(f) + \texttt{ApproximationCost}(f, \hat{f})$$

where μ will be chosen experimentally. Note that `TotalCost` is a functional of the function f [Bol61]. The cost functions are defined as follows:

`ApproximationCost` measures the way a function f approximates the polynomial $\hat{f}$.

$$\texttt{ApproximationCost}(f, \hat{f}) = \int_1^n (f(t) - \hat{f}(t))^2 dt$$

`CurvatureCost` measures the curvature of the approximating function f The `CurvatureCost` is lower when f has small second order derivative, which is the acceleration.

$$\texttt{CurvatureCost}(f) = \int_1^n (\frac{d^2 f}{dt^2})^2 dt$$

In practice, these are the steps to follow, given the data as points on the ground plane, in order to produce the low curvature approximation.

1. Take a short segment of the data (in our case, 10 points)

2. For each component j, calculate $\hat{f}_j$, a low degree polynomial that approximates the data

3. For each component j, use the closed form solution of the minimisation problem for the functional, in which the approximating polynomial $\hat{f}_j$ has been replaced, to obtain the low curvature approximation.

The data are points on the plane, with two components, x and y. Let us work independently on each component, in order to find approximations f_x and f_y with low cost. The final low curvature function will be (f_x, f_y). In the following formulas we will assume that the components we are using are the numbers $p(1), \ldots, p(n)$.

6.2.2 Analytic optimisation of the cost function

The direct estimation of the minimum cost trajectory would be difficult because $\texttt{TotalCost}(f)$ is a highly non-linear function of f. But it is possible to obtain a closed form solution for the functional `TotalCost` [Bol61].

Polynomial approximation

Let us fit a polynomial $\hat{f}$ to the data. Let $\hat{f}(s) = \hat{f}_0 + \hat{f}_1 s + \hat{f}_2 s^2$ be the degree two polynomial that minimises the least squares error with the samples,

$$\sum_{t=1}^{n} (\hat{f}(t) - p(t))^2$$

Note that a higher degree polynomial may be needed in applications where n is large and the underlying trajectory is complicated. In the current application degree two is sufficient. The function $\hat{f}$ fits closely to the measurements, but in general $\hat{f}$ has high second order derivatives, which result in high curvatures for the ground plane trajectory.

Variational equation

The polynomial $\hat{f}$ must now be replaced by a function which fits the measurements closely and which has a low second order derivative.

The low curvature approximation is the global minimum of the functional `TotalCost`,

$$\begin{aligned} \texttt{TotalCost}(f) &= \mu \cdot \texttt{ApproximationCost}(f) + \texttt{CurvatureCost}(f) \\ &= \lambda^4 \cdot \underbrace{\int_{t_1}^{t_n} (f(t) - \hat{f}(t))^2 dt}_{\text{error}} + \underbrace{\int_{t_1}^{t_n} (\frac{d^2 f}{dt^2})^2 \, dt}_{\text{curvature}} \end{aligned} \tag{6.1}$$

where $\hat{f}$ is the low degree polynomial used to approximate the data as a continuous function, and the constant λ is defined as $\mu^{-4} = \lambda$ to simplify the formula of the solution.

Analytic solution of the variational equation

The Euler-Lagrange equation [Bol61] for `TotalCost` is linear,

$$\frac{d^4 f}{dt^4} + \lambda^4 f = \lambda^4 \hat{f} \tag{6.2}$$

The polynomial $\hat{f}$ has degree two, thus $f = \hat{f}$ is one solution of Equation 6.2. The general solution f_m to 6.2 is [Bur68]

$$\begin{aligned} f_m(t) &= \exp\left(\frac{\lambda t}{\sqrt{2}}\right)\left(a_1 \cos\left(\frac{\lambda t}{\sqrt{2}}\right) + a_2 \sin\left(\frac{\lambda t}{\sqrt{2}}\right)\right) \\ &\quad + \exp\left(-\frac{\lambda t}{\sqrt{2}}\right)\left(a_3 \cos\left(\frac{\lambda t}{\sqrt{2}}\right) + a_4 \sin\left(\frac{\lambda t}{\sqrt{2}}\right)\right) + \hat{f}(t) \end{aligned}$$

where $a_1, \ldots, a_4$ are arbitrary real numbers. The expression $\texttt{TotalCost}(f_m)$ is a quadratic polynomial in $a_1, \ldots, a_4$. The values $a_1, \ldots, a_4$ at which the minimum of $\texttt{TotalCost}(f_m)$ is attained, can be replaced into f_m to obtain f.

The process described above yields each of the two components (f_x, f_y) of the low curvature approximation.

6.3 Hidden Markov models for trajectory labelling

In this section we will use a method for labelling measurements from vehicle trajectories, based on the low curvature approximation produced in the previous section.

6.3.1 Motivation

Given the measurements as a sequence of positions of a vehicle on the ground plane at regular time steps, it is possible to find a piecewise continuous approximation that both approximates those measurements and keeps a low curvature, as explained in Section 6.2. Now each of the segments can be labelled as an observed action, according to a function related to the curvature of the vehicle. Let us consider the following low level four *states* for the driver,

Ahead ↑ the vehicle is moving with approximate constant velocity

Left ↶ the vehicle is turning left

Right ↷ the vehicle is turning right

Stop × the vehicle stops suddenly, the motion is characterised by second derivatives above a threshold

Each of these actions is understood to be chosen by the driver at each point. The list of actions is kept simple and general. Later on they can be grouped up to obtain a simplified description of the trajectory.

By assigning a label to each segment depending on the acceleration and curvature we may still not obtain the sequence of actions taken by the driver. Errors of the vehicle tracker as well as spurious movements, which are not relevant for our application, would produce segments that would be labelled wrongly. In order to estimate the actions decided by the driver, let us use a Hidden Markov Model [RLS83, RJ86].

A Hidden Markov Model is a model used to describe a sequence of states such that

- the transition between one state and the next is decided by a random variable (Markov Chain of first order). This is the *state transition probability*.

- at each time step, an observation is a random variable that depends on the state. This is the *observation probability*.

The result is a Markov Chain in which we might not determine the internal states from the observations at each element of the chain, but in which we can assign probabilities to each sequence of states.

Once we have defined the Hidden Markov Model, Viterbi's algorithm can be used to calculate the most likely sequence of actions taken by the driver. But before that we need the parameters of the Hidden Markov Model: the transition probabilities between actions, and the probabilities that relate the actual actions with the observed ones.

6.3.2 Initial labelling of segments

The sequence of labels that will be input in the HMM has yet to be computed from the continuous approximation to the data. We will see now how to label each segment with the observations $\{\uparrow, \curvearrowright, \curvearrowright, \times\}$ to each segment. Later on the HMM will be used to compute the most probable sequence of driver's actions, or internal states, that correspond to those observations.

The label for each segment depends on functions of the continuous approximation to the segment. Let $f(t) = (x_t, y_t)$ be the smooth approximation obtained in Section 6.2, where t belongs to the time interval $[1, n]$ of data that f approximates.

The following estimates refer to the interval of data that f approximates. The least speed u of the car is estimated by

$$u = \min\{\sqrt{\dot{x}_t^2 + \dot{y}_t^2} \mid 1 \leq t \leq n\}$$

Let s be the time at which the curvature of the trajectory attains its maximum absolute value and let w be the wheelbase of the car. The greatest steering angle $\theta_{\max}$ of the car is estimated by

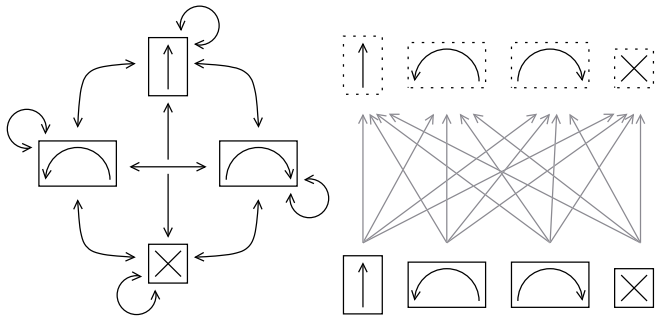
$$\theta_{\max} = w \frac{\ddot{y}_s \dot{x}_s - \ddot{x}_s \dot{y}_s}{(\dot{x}_s^2 - \dot{y}_s^2)^{3/2}}$$

The quantity $\Phi = u\ \theta_{\max}/w$ is an estimate of the greatest rate of change of orientation. The segment is classified as one of $\{\uparrow, \curvearrowright, \curvearrowright, \times\}$ according to table 6.1.

class	condition
$\uparrow$	$-1/2 \leq \theta \leq 1/2$
$\curvearrowright$	$1/2 < \theta$
$\curvearrowright$	$\theta < -1/2$
$\times$	$u < 100$

Table 6.1: Classification of a measurement segment depending on the greatest rate of change of orientation θ, and acceleration u. The unit of θ is radian/s, and the unit of u is cm/s.

The list of segments for the entire trajectory is thus reduced to a string of labels drawn from the set $\uparrow, \curvearrowright, \curvearrowright, \curvearrowright$ The next step is computing the sequence of internal states from the sequence of observations.



(a) Markov Chain

(b) Observations from internal states

Figure 6.1: Hidden Markov Model of the four internal states and four possible observations.

	internal	observed
accelerate	↑	↑
left	↶	↶
right	↷	↷
stop	×	×

Table 6.2: Internal states and observed labels.

6.3.3 Hidden Markov Models

A Hidden Markov Model [RJ86], is used to model a sequence of observations with four internal states. The internal states form a Markov Chain. Each state of the chain produces an observation according to some probability distribution. The model has two main components:

State transition The probability that an internal state i is followed by another internal state j in the sequence is given by the entry at (i, j) in a *transition matrix* A with rows and columns indexed by states.

Observation of state The probability that a state i is observed as label j is the entry (i, j) of an *observation* matrix B, with rows indexed by the states and columns indexed by the observation labels.

Initialisation The probability that the initial state of the sequence is i.

If the system is in state i, then the probability of a transition to state j is A_{ij}. If the system makes a transition to state j then the probability of the observation k is B_{jk}. The distribution of the initial state is given by a vector π indexed by the states.

The proposed model is illustrated in Figures 6.1 and 6.2, and contains four *internal states* which are labelled {↑, ↶, ↷, ×}. These are the true states of the car, corresponding to the actions ahead, turning left, turning right, stopped. The are also to be inferred from the measurements. The HMM has four output labels, {↑, ↶, ↷, ×}. These are the labels obtained from the measurement segments as described above.

Estimating the model parameters. The model has a large number of parameters. Fortunately, it is possible to estimate these parameters given a training set. This training set consists of a sequence of states together with the sequence of corresponding observations.

- From the frequency in which state i is followed by state j, it is possible to estimate entry (i, j) of the transition matrix.
- From the frequency in which state i lead to observation j, it is possible to infer the entry (i, j) of the observation matrix.

Therefore the matrices A and B are learnt from a training set of trajectories, segmented by hand.

Estimating the states with highest probability. Given data in the form of a sequence of observations, it is possible to calculate the probability of each sequence of internal states that could have produced those observations. This can be done by a linear combination of the probability that each sequence of internal states produced the data (which can be calculated with the help of the observations matrix). The weights of this linear combination are the probability of each sequence of internal states (which can be calculated with the help of the transition matrix and the initialisation vector).

But it is not necessary to perform those calculations. Viterbi's algorithm [RJ86] produces the sequence of states that has higher probability, and yields its probability as a result. And the algorithm does so in linear time with respect to the number of internal states and the length of the sequence.

Viterbi's algorithm will be used in the following experiments to derive a sequence of internal states from the observed labels, therefore providing the last step to obtain a sequence of driver's actions (which are the internal states) from a trajectory given by a vehicle tracker (which is a sequence of points on the plane).

6.4 Experiments

So far we have studied a method to obtain continuous measurements from a discrete trajectory, in the form of a low curvature approximation, and a method to model those measurements in terms of internal states of the vehicle. Now we proceed to use this method with trajectories obtained from a car tracker.

6.4.1 Setup

The trajectory labelling algorithm has been tested on a set of 21 image sequences displaying a vehicle moving across a car park. The images were taken by a digital video camera looking through a second floor window. A typical image is shown in Figure 6.2.

The car had a known geometric model, and the same car was used for all the sequences. The wheelbase was $w = 250$ cm. The camera was calibrated using the method described in page 54, and measurements were obtained using the model-based vehicle tracker described in [Sul94]. The tracker measures the position (x, y) of the

Figure 6.2: A typical image

centre of the rear axle and the orientation θ of the vehicle. In these experiments only the (x, y) measurement was used. The angle measurements were discarded. The calibration included the determination of the ground plane position relative to the camera. The tracker was initialised by hand in the first image of each sequence.

6.4.2 Parameters

The sequence of measurements for each trajectory was divided into overlapping segments, each containing 10 measurements. Each pair of adjacent segments overlapped by 9 measurements. The time between the first measurement in a segment and the last measurement was 1.6 seconds, which corresponds to using an image of each four. Experiments showed that the performance of the algorithm was degraded if the overlap between adjacent segments was reduced.

The matrices A, B and the initial state vector π are

$$A = \begin{pmatrix} 111/121 & 5/121 & 3/121 & 2/121 \\ 1/32 & 31/32 & 0 & 0 \\ 5/69 & 0 & 64/69 & 0 \\ 2/63 & 2/63 & 3/63 & 56/63 \end{pmatrix}$$

$$B = \begin{pmatrix} 114/132 & 6/132 & 10/132 & 2/132 \\ 9/34 & 24/34 & 1/34 & 0 \\ 28/73 & 1/73 & 42/73 & 2/73 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\pi = \left(\frac{12}{21}, \frac{1}{21}, \frac{3}{21}, \frac{5}{21} \right)^\top$$

Although the diagonal entries of B are large (for example, if the state of the car is $\boxed{\uparrow}$, then there is a high probability, $114/132 \geq 0.86$, of observing the label $\boxed{\uparrow}$) they are not equal to 1, because there is a small probability that the observed label will not correspond to the true state of the car.

The probabilities in the matrices A, B, and the vector π were obtained by the hand segmentation of 21 training sequences showing the motions of a car in a car park.

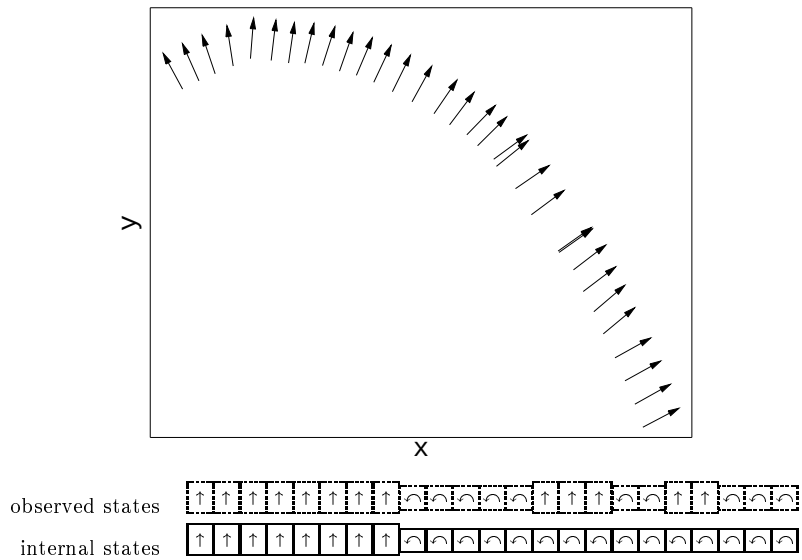


Figure 6.3: A left turn

For each trajectory, the list of segments was reduced to a sequence of labels using the method described in Section 6.3. Viterbi's algorithm was applied to this sequence to produce the sequence with highest probability according to the Hidden Markov Model.

Example 6.1 A LEFT TURN A typical trajectory of the car is shown in 6.3. The car moves from bottom right to top left. The arrow represents the normal to the trajectory, drawn as if the driver were extending the right arm out of the window. The base points of the arrows are the coordinates of the vehicle on each sample. The sequence of extracted labels and the most likely sequence of states, as identified by Viterbi's algorithm, are shown in Figure 6.3.

Now how the trajectory is effectively divided in two regions: the first one, in which the vehicle is moving forward, and the second one, when the vehicle turns to the left. The small portions of trajectory, during the turn, in which the vehicle seems not turning are filtered out. □

Example 6.2 A STOP AND A RIGHT TURN A more complex trajectory is shown in Figure 6.4. The trajectory begins at the top left and then turns right. First, the vehicle stops (at the crossroad), then it moves ahead and turns right. The sequence of estimated internal states is divided in two segments. The first half are a stop, as it corresponds to the first half of the sequence in time which the car arrives to a halt and waits. In the second half the short measurements that indicate that the vehicle might not be turning are filtered out.

which means that the driver stops, and then turns to the right. □

Example 6.3 A COMPLEX TRAJECTORY Another trajectory is shown in Figure 6.5. The car moves from bottom left to top right. In this case the final turn in the trajectory of the vehicle is not filtered out, but the spurious observations in the middle of the trajectory disappear. The trajectory is effectively divided in four parts.

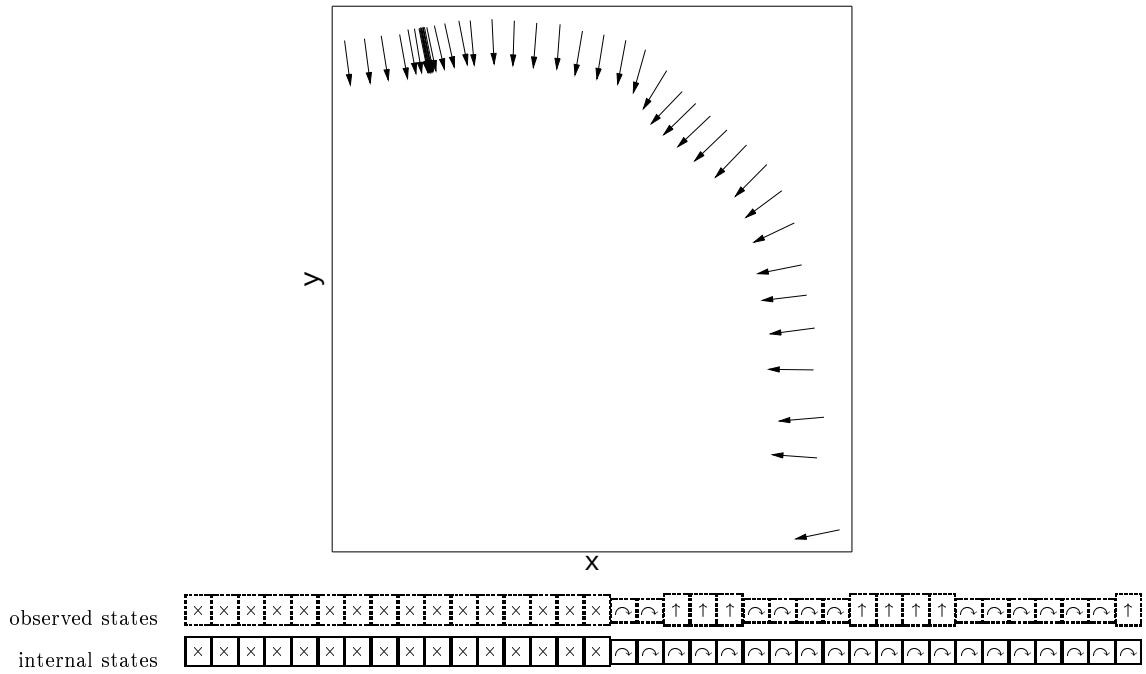


Figure 6.4: A stop and a right turn

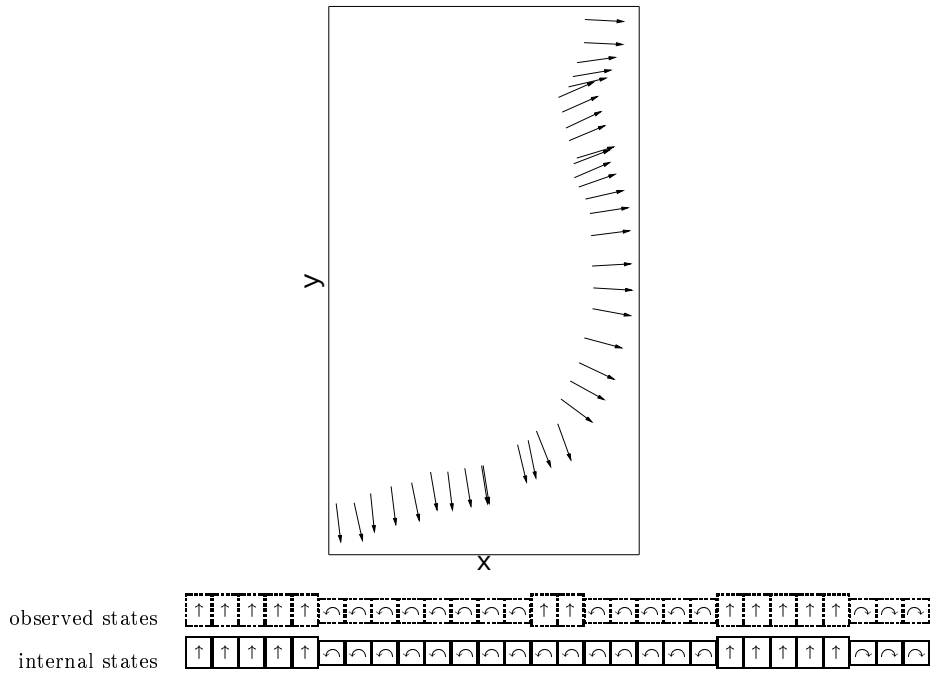


Figure 6.5: A more complex trajectory

□

6.5 Conclusion

A method to label tracked vehicle trajectories has been presented. This method approximates discrete measurements with a low curvature continuous function, and processes it in order to obtain a labelling of the trajectory. A label has been assigned to each original measurement. This label is intended to represent the driver's action rather than the instant physical motion of the vehicle at that point.

In order to label the trajectory, which was given in discrete form, we have first approximated the trajectory with a low curvature polynomial. This polynomial was obtained as a minimisation of a functional which combined a cost in terms of goodness-of-fit, and a cost in terms of curvature. Therefore, the polynomial is expected to have both an appropriate fit to the data, and a low curvature, which would not be the case with standard polynomial approximation.

We have then suggested a way to label those segments into observed actions, using simple thresholds on the derivatives of the approximation.

Then, a HMM estimated the actual actions taken by the driver of the car, in terms of moving ahead, turning, or stopping. The low curvature approximations have been used only for short segments, but the approximation method can be extended to longer and more complex trajectories.

Overall, the method is characterised by the following properties:

Closed-form solution The solution is given in closed-form: solving an equation symbolically produces the low curvature approximation. This contrasts with the optimisation methods used in the rest of this thesis that require the computation of the cost function for particular parameters.

Low number of labels For the trajectories considered in these experiments, four states are enough to represent the action of the driver, and subject to filtering by the Hidden Markov Model. The result is a simple segmentation of the trajectories into repeated labels.

It can be suggested with these results that HMMs can be the basis of fast algorithms which analyse trajectories. The sequence of labels produced by the method presented in this chapter can be used to identify trajectories with low probability, which correspond to abnormal driver behaviour.

Chapter 7

Conclusion

This thesis presents contributions to research in Model Selection for Computer Vision. In particular, this thesis studies the MDL method of Model Selection and its application to vehicle trajectory detection from images.

The MDL method. The MDL method of Model Selection is usually studied as a probabilistic method. In this thesis the MDL method has been presented from a combinatorial point of view, as an optimal method of Model Selection. The originality of this approach is in the lack of assumptions about prior probability distributions, and the intuitive appeal of translating the properties of the MDL method into properties of binary trees whose leaves are the computer programs that halt. Furthermore, while the standard approach to MDL starts from the axiom that description length should be minimised, in this thesis the MDL method is obtained as a minimal element for an order relation among Model Selection methods.

This interpretation of the MDL model selection method provides a more straightforward correspondence between the definition of the method and its implementation as a computer program.

The theory of MDL can potentially be extended from computers that handle binary strings to rule-based systems that manipulate generic symbolic expressions. This use of human-readable expressions together with the expressive power of rule-based systems will probably increase the role of automatic simplifications in the implementations of the MDL method, and therefore increase its execution speed.

Computer Vision. A computable approximation of the MDL method is used to detect the motion of vehicles in traffic sequences. The aim is a vehicle tracking and event detection system. The main consequence is that the MDL method is able to choose an appropriate model complexity in the presence of noise and occlusions. Besides, its implementation is conceptually simple.

We have illustrated how Computer Vision algorithms can be developed as data compression algorithms. Data compression is used to match geometric models against images, and also to match trajectory models against trajectory data. In particular, the following ideas have been developed:

- An ad-hoc implementation of MDL applied to vehicle tracking (Chapter 3). This served to guide further development towards a proper derivation of the MDL from first principles.

  - 3D shapes are used to model the vehicles, using a purpose-built 3D rendered that features projective mapping of image bitmaps.
  - It uses an off-the-shelf data compressor.
  - Experiments assessing the validity of the approach.

- A software package `Thesis‘Compress‘` that facilitates the definition of computable approximations to MDL (Chapter 4), whose main attributes are:
  - Abstract functions that include non-computable MDL and computable approximations.
  - Separation of the definition of MDL and its computable approximations.
  - Reusable code.
  - Provides a framework for the comparison of different computable approximations to MDL, which includes model selection methods such as Maximum Likelihood.

- An application of the software package `Thesis‘Compress‘` to trajectory modelling and analysis (Chapter 5). Experiments provided an interpretation of motion modelling close to information theory, in which the error model was not Gaussian, as in the standard approach with Kalman filters, but the Universal Measure. Our work included
  - translation of vehicle dynamics into a compression algorithm, and
  - experiments assessing the robustness of MDL against noise and occlusions with synthetic and experimental data.

Trajectory analysis. Besides, an alternative method for trajectory analysis is presented (Chapter 6). It combines a low-curvature polynomial approximation to the trajectory with a Hidden Markov Model to model the driver's actions. Experiments show that this method can be used to segment noisy vehicle trajectories, corresponding to simple driver actions.

Choice of computer language. The Model Selection methods used in this thesis can be generalised to any other model classes so long as it is possible to implement the corresponding compression algorithms. The choice of computer language, or Universal Turing machine, is crucial because the language should facilitate minimising the description length, and at the same time must be comfortable for humans. Improvements in the choice of Universal Turing machine will provide better measures of complexity.

Bibliography

[Aka74] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19:716–723, June 1974.

[Bol61] O. Bolza. *Lectures on the Calculus of Variations*. Dover, 1961.

[Bri90] Kay Brisdon. *Hypothesis Verification using Iconic Matching*. PhD thesis, The University of Reading, 1990.

[Bur68] J. C. Burkhill. *The Theory of Ordinary Differential Equations*, volume 21. University Mathematical Texts, 1968.

[Can86] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[CLK+00] Robert Collins, Alan Lipton, Takeo Kanade, Hironobu Fujiyoshi, David Duggins, Yanghai Tsin, David Tolliver, Nobuyoshi Enomoto, and Osamu Hasegawa. A system for video surveillance and monitoring. Technical Report CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2000.

[CR99] Peter Chi and C. T. Russell. Statistical methods for data analysis in space physics. http://www-ssc.igpp.ucla.edu/personnel/russell/ESS265/-Ch9/autoreg/, March 1999.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley and Sons, 1991.

[Die95] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Computing Surveys*, 27(3), 1995.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, chapter 6. Rewrite Systems, pages 243–320. 1990.

[Edw92] A. W. F. Edwards. *Likelihood*. The Johns Hopkins University Press, 1992.

[Fat92] Richard J. Fateman. A review of *Mathematica*. *Journal of Symbolic Computation*, 13:545–579, 1992.

[Fau93] Olivier Faugeras. *Three-Dimensional Computer Vision*. Artificial Intelligence. MIT Press, 1993.

[Fer01] James Ferryman. Pets2001 dataset. http://pets2001.visualsurveillance.org/, 2001. Datasets for the IEEE workshop on Performance Evaluation of Tracking Systems.

[Fer02] James Ferryman. People and vehicle tracking: Big brother looking after you. *Measurement + Control*, 35(8):197–203, September 2002.

[FM98] R. Fraile and S. J. Maybank. Vehicle trajectory approximation and classification. In Paul H. Lewis and Mark S. Nixon, editors, *British Machine Vision Conference*, 1998.

[FM99a] Roberto Fraile and Stephen J. Maybank. Building 3D models of vehicles for computer vision. In Dionysius P. Huijsmans and Arnold W. M. Smeulders, editors, *VISUAL'99: Visual Information and Information Systems Conference Proceedings*, number 1614 in Lecture Notes in Computer Science, pages 697–702. Springer, 1999.

[FM99b] Roberto Fraile and Stephen J. Maybank. Model matching using all grey levels. Technical report, Department of Computer Science, The University of Reading, 1999.

[FM00a] Roberto Fraile and Steve Maybank. Image compression for trajectory refinement. In James Ferryman, editor, *Proceedings of PETS2000*, pages 46–49, 2000.

[FM00b] Roberto Fraile and Steve Maybank. Selection of rigid models for visual surveillance. In James Ferryman and Anthony Worrall, editors, *Proceedings SIRS2000*, pages 289–294, 2000.

[FM01] Roberto Fraile and Steve Maybank. Comparing probabilistic and geometric models on lidar data. In Michelle A. Hofton, editor, *Workshop on Land Surface Mapping and Characterization Using Laser Altimetry*, pages 67–70. ISPRS, October 2001.

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 2nd edition, 1990.

[FWSB97] James M. Ferryman, Anthony D. Worrall, Geoff D. Sullivan, and Keith D. Baker. Visual surveillance using deformable models of vehicles. *Robotics and Autonomous Systems*, 19:315–335, 1997.

[Gar90] Martin Gardner. *Mathematical Circus.* Penguin, 1990. Reprint — original published by Knopf in 1979.

[Ger99] Neil Gershenfeld. *The Nature of Mathematical Modeling.* CUP, 1999.

[GL96] Warren F. Gardner and Daryl T. Lawton. Interactive model-based vehicle tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(11):1115–1121, November 1996.

[Grü98] Peter D. Grünwald. *The Minimum Description Length Principle and Reasoning Under Uncertainty.* PhD thesis, CWI, 1998.

[How] Denis Howe. Free on-line dictionary of computing. http://foldoc.doc.ic.ac.uk/. Version 2002-03-17 02:30.

[HY01] Mark H. Hansen and Bin Yu. Model selection and the principle of minimum description length. *Journal of the American Statistical Association*, 96(454):746–774, 2001.

[IEE99] IEE. *Motion Analysis and Tracking*, number 1999/103 in Electronics and Communications. IEE, May 1999.

[Jay96] E. T. Jaynes. Probability theory: The logic of science. (A final version of this document is in preparation and will be published by CUP), March 1996.

[Jon87] Simon L. Peyton Jones. *The Elements of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[KN95] H. Kollnig and H.-H. Nagel. 3d pose estimation by fitting image gradients directly to polyhedral models. In *ICCV'95*, pages 569–574, Cambride, MA, US, 1995.

[Knu92] Don E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison Wesley Longman, 3rd edition, 1997.

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley Longman, 3rd edition, 1998.

[KWM94] Dieter Koller, Joseph Weber, and Jitendra Malik. Robust multiple car tracking with occlusion reasoning. In J.-O. Eklundh, editor, *ECCV'94*, number 800 in LNCS, pages 189–196. Springer-Verlag, 1994.

[lGA02] Jean loup Gailly and Mark Adler. The `gzip` homepage. http://www.gzip.org/, June 2002.

[LL92] H. Levy and F. Lessman. *Finite Difference Equations*. Dover, 1992.

[LV92] Ming Li and Paul Vitányi. Inductive reasoning and Kolmogorov complexity. *Journal of Computer and System Sciences*, 44:343–384, 1992.

[LV97] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate Texts in Computer Science. Springer, 2nd edition, 1997.

[LZ76] Abrahan Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(1):75–81, January 1976.

[Mae89] Roman E. Maeder. *Programming in Mathematica*. Addison-Wesley, 1989.

[Mae96] Roman E. Maeder. *The Mathematica programmer II*. Academic Press, 1996.

[Mae00] Roman E. Maeder. *Computer Science with Mathematica*. CUP, 2000.

[Mar82] David Marr. *Vision —A Computational Investigation into the Human Representation and Processing of Visual Information*. Freeman, 1982.

[Mel76] Z. A. Melzak. *Mathematical Ideas, Modeling & Applications. Volume II of Companion to Concrete Mathematics*. John Wiley & Sons, 1976.

[MF01] S. J. Maybank and R. Fraile. Minimum description length method for facet matching. In Jun Shen, P. S. P. Wang, and Tianxu Zhang, editors, *Multispectral Image Processing and Pattern Recognition*, number 44 in Machine Perception and Artificial Intelligence, pages 61–70. World Scientific Publishing, 2001. Also published as special issue of the International Journal of Pattern Recognition and Artificial Intelligence.

[MRW+94] J. Malik, S. Russell, J. Weber, T Huang, and D. Koller. A machine vision based surveillance system for california roads. Technical report, Computer Science Division, University of California, 1994.

[MW97] S. J. Maybank and A. D. Worrall. *Algebraic Frames for the Perception-Action Cycle*, chapter Path Prediction and Classification Based on Nonlinear Filtering, pages 323–343. Number 1315 in LNCS. Springer-Verlag, 1997.

[MWS96a] Stephen J. Maybank, Anthony Worrall, and Geoff D. Sullivan. Filter for car tracking based on acceleration and steering angle. In R. B. Fisher and E. Trucco, editors, *British Machine Vision Conference*, pages 615–624. BMVA, 1996.

[MWS96b] Stephen J. Maybank, Anthony D. Worrall, and Geoff D. Sullivan. A filter for visual tracking based on a stochastic model for driver behaviour. In B. Buxton and R. Cipolla, editors, *Computer Vision, ECCV96*, volume 1065 of *Lecture Notes in Computer Science*, pages 540–549. Springer, 1996.

[NT97] N.A.Thacker and T.F.Cootes. Vision through optimisation. CVOnline, 1997. http://www.dai.ed.ac.uk/CVonline/.

[Pos02] Jef Poskanzer. PBMPlus. http://www.acme.com/, 2002. A library for converting and manipulating various image file formats.

[Pra76] Ronald E. Prather. *Discrete Mathematical Structures for Computer Science*. Houghton Mifflin, 1976.

[Ris83] Jorma Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, June 1983.

[RJ86] L. R. Rabiner and B. H. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–17, January 1986.

[RLS83] L. R. Rabiner, S. E. Levinson, and M. M. Sondhi. On the application of vector quantization and hidden Markov methods to speaker-independent isolated word recognition. *Bell Systems*, (62):1075–1105, 1983.

[RMFB98] Paolo Remagnino, Stephen J. Maybank, Roberto Fraile, and Keith Baker. *Advanced Video-based Surveillance Systems*, chapter Automatic Visual Surveillance of Vehicles and People, pages 95–105. Kluwer Academic, 1998.

[Rou84] Peter J. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, 79(388):871–880, December 1984.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2nd edition, 2000.

[Sch78] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, March 1978.

[Sha48] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[SHB99] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis and Machine Vision*. PWS Publishing, 2nd edition, 1999.

[SK52] J.G. Semple and G. T. Kneebone. *Algebraic Projective Geometry*. Oxford University Press, 1952.

[Sul92] G. D. Sullivan. Visual interpretation of known objects in constrained scenes. *Phil. Trans. R. Soc. London*, (337):361–370, 1992.

[Sul94] G. D. Sullivan. *Real-Time Computer Vision*, chapter Model-based vision for traffic scenes using the ground plane constraint. CUP, 1994.

[TL02] Paul Allen Tipler and Ralph A. Llewellyn. *Modern Physics*. W. H. Freeman and Co., 4th edition, 2002.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of The London Mathematical Society*, 42(3, 4, 7):230–265, 544–546, November, December 1936. Corrections in part 7.

[vR99] C. J. van Rijsbergen, editor. *Computer Journal, Special Issue in Kolmogorov Complexity*, volume 42. OUP, 1999.

[WB68] C. S. Wallace and D. M. Boulton. An information measure for classification. *Computer Journal*, 11(2):185–194, August 1968.

[WD99] C. S. Wallace and D. L. Dowe. Minimum message length and kolmogorov complexity. *Computer Journal*, 42(4):270–283, 1999.

[Wel84] Terry A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6), June 1984.

[Wol99] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 4th edition, 1999.

[WSB94] A. D. Worrall, G. D. Sullivan, and K. D. Baker. A simple, intuitive camera calibration tool for natural images. In *Proc. 5th British Machine Vision Conference*, pages 781–790. BMVA, 1994.

[ZB95] Heyun Zheng and Steven D. Blostein. Motion-based object segmentation and estimation using the mdl principle. *IEEE Transactions on Image Processing*, 4(9):1223–1235, September 1995.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[ZY96] Song Chun Zhu and Alan Yuille. Region competition: Unifying snakes, region growing, and Bayes/MDL for multiband image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):884–900, 1996.