Working title

Robot Automation: Intelligence through Feedback Control

Jorge Cortés and William B. Dunbar

August 31, 2007
To our families
# Contents

1 Introduction .............................................. 1  
  1.1 Introduction from CSM paper .......................... 1  
  1.2 Conclusions from CSM paper ......................... 4  

2 Basic MATLAB use ......................................... 7  
  2.1 Chapter objectives .................................. 8  
  2.2 Preliminaries ........................................ 8  
    2.2.1 Create your working directory .................. 8  
    2.2.2 Help! ........................................... 9  
  2.3 Mathematics with MATLAB ............................ 10  
    2.3.1 Numbers and operations ......................... 10  
    2.3.2 Output ......................................... 10  
    2.3.3 Variables ...................................... 11  
    2.3.4 Mathematical functions ......................... 12  
    2.3.5 Vectors ........................................ 12  
  2.4 Plots ................................................ 14  
  2.5 Programming ......................................... 16  
    2.5.1 Loops .......................................... 16  
    2.5.2 m-files ....................................... 18  
  2.6 Homework ............................................ 20  

3 Discrete-time dynamics .................................... 23  
  3.1 Chapter objectives ................................. 24  
  3.2 Discrete-time dynamic equations .................... 24  
  3.3 Fixed points of dynamic models ................. 28  
  3.4 Stability of fixed points ............................ 29  
    3.4.1 Linear models ................................ 30  
    3.4.2 Nonlinear models ............................. 31  
  3.5 The logistic map ................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>Wild behavior of the logistic map: period doubling</td>
<td>34</td>
</tr>
<tr>
<td>3.5.2</td>
<td>What happens for $r &gt; r_{\infty}$?</td>
<td>35</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Analysing the logistic map</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Homework</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Analysis and simulation of dynamic models</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Chapter objectives</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Predator-Prey Population Dynamic Model</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Robot Dynamic Model</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Homework</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Feedback control of dynamic models</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction: the magic of feedback</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Controlled dynamics</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Shaping the dynamics: forcing desired equilibrium points</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Shaping the dynamics: making the equilibrium stable</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>Homework</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>Feedback control of the Robobrain robot</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Chapter objectives</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Robot dynamic model</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Big picture</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>Wall-following control - simulation</td>
<td>68</td>
</tr>
<tr>
<td>6.5</td>
<td>Wall-following control - experimentation</td>
<td>72</td>
</tr>
<tr>
<td>6.6</td>
<td>Homework</td>
<td>76</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Motivation for the book. We feel it fills a hole. Audience we have in mind. Uses that can be given to this book.

Explain mechanics of the book for use as lecture notes for a course. Plenty of short explanations alternated with simple tasks to reinforce learning. Plenty of homework exercises, to be done individually and in small groups.

1.1 Introduction from CSM paper

The purpose of this paper is to describe a control education and outreach effort being undertaken at UC Santa Cruz (UCSC). The goal of our effort is to provide an introductory course on feedback control that is accessible to high-school students and first-year undergraduate students. In the early stages of our effort, we found ample motivation in the literature for creating such a course. In 1998, an article from the NSF/CSS Workshop on New Directions in Control Engineering Education [?] made the following recommendations regarding needed reform in undergraduate control education:

- “to provide practical experience in control systems engineering to first-year college students to stimulate future interest and introduce fundamental notions like feedback and the systems approach to engineering,” and

- “to encourage the development of new courses and course materials that would significantly broaden the standard first introductory control systems course at the undergraduate level.”
In 2003, a panel on future directions in control, dynamics, and systems provided a renewed vision of challenges and opportunities, along with recommendations to agencies and universities to ensure continued progress in areas of importance to the industrial and defense base [?]. One of the five primary recommendations is that the community and funding agencies invest in “new approaches to education and outreach for the dissemination of control concepts and tools to nontraditional audiences. As a first step toward implementing this recommendation, new courses and textbooks should be developed for both experts and nonexperts.” The panel also recommended the integration of software tools such as Matlab into these courses.

Outreach motivation has also been provided from outside the control field. Ten years ago, feedback control was identified by the National Science Education Standards as being fundamental to understanding systems, and systems in turn was identified as a unifying concept for K-12 science education [?]. In 2005, the need for curricular material that motivates middle school and high school students to pursue advanced work in science and mathematics in the United States was underscored by the Committee on Prospering in the Global Economy of the 21st Century, a subcommittee of the National Academy of Sciences, National Academy of Engineering, and the Institute of Medicine [?].

This article presents an overview of an introductory course on feedback control amenable to traditional and nontraditional audiences. The course material is based on fundamental concepts in dynamical systems, modeling, stability analysis, robustness to uncertainty, feedback as it occurs naturally, and the design of feedback control laws to engineer desirable static and dynamic response. The course also includes an introduction to Matlab, provides Matlab exercises to reinforce concepts, and concludes with design and application of a controller to achieve wall tracking with a kinematic robot experiment. The only prerequisite for the course is high school algebra, including basic trigonometry. During a four-week period in the summer of 2005, the course was taken by a group of 17 talented high school students, ranging from 9th grade to 11th grade. By the end of the course (~30 hours of lecture time), each student had successfully implemented a wall-tracking controller on a robot called Robobrain. The students acquired a basic understanding of the principles of feedback control and its importance in applications. The students also displayed a strong motivation to learn more advanced mathematics and engineering subjects.

This article focuses on the course content as well as our first-hand experiences teaching the class. Our hope is that this information is useful to instructors interested in pursuing new approaches to outreach. All lecture
material is freely available online for anyone to use, experiment with, and improve on if desired [?]. The target audience for the course includes high school students, first-year undergraduates in engineering and science, and graduate and postdoctoral researchers seeking an introduction to dynamics and control. The material can be used to initiate collaboration between control theoreticians and biologists, for example, by helping the biologists learn the language and principles behind feedback control. By design, the course material provides explicit motivation for learning more advanced mathematics (including linear algebra, calculus, and differential equations) and physics. Since the only mathematical prerequisite is high school algebra and basic trigonometry, modeling is done entirely in discrete time and kept to low dimensions.

References [??] describe precollege outreach efforts that focus on experimental design and control of robots, with the objectives of building robotic platforms and testing basic programming, engineering, and robotic concepts. The study by ? reports on a graphical-user-interface -(GUI)-based approach using LabView and LEGO Mindstorm robots to introduce engineering design, embedded computer control, sensors, and software platforms to incoming college freshman. In contrast to these references, robotics is not the focus in our course. Instead, the power of abstraction is stressed in the lectures, and the robotic platforms used in the final lectures are employed as a venue for the modeling and control design concepts.

Our success with the high school course has led to a new one-quarter undergraduate course at the Baskin School of Engineering UCSC entitled “Computer Engineering 8: Robot Automation: Intelligence through Feedback Control.” While most control courses are designed for third- and fourth-year undergraduates [??], our course is offered to first-year undergraduates. In a similar vein, a curriculum that focuses on systems and control for undergraduates at the Norwegian University of Science and Technology is described in ?. Their first-year course, “Introduction to Computerized Control,” covers more advanced topics than our crash course, beginning with continuous-time modeling using ordinary differential equations. A similar first-year course is described in ?, with calculus as the mathematical prerequisite.

At the other end of the prerequisite spectrum, the effort described in ? introduces feedback control to freshmen at the University of New Haven using little mathematical modeling. Instead, a GUI-based approach is taken using LabView software and four applied control experiments that students can choose from. An advantage of this application- and GUI-based approach is that the variety of experiments is likely to entice more students
CHAPTER 1. INTRODUCTION

to pursue control systems and engineering, while giving them broader in-
sight into what control can and cannot do. In contrast, an advantage of
including mathematical modeling from the beginning, as in our material
and in ??, is that control design and analysis are linked to math subjects the
students are taking or will soon be taking.

In the remainder of the article, we provide an overview of the course
material. Next, we describe the venue for the course given in the summer
of 2005. Finally, we summarize feedback from some of the students who
completed the course.

1.2 Conclusions from CSM paper

We have summarized the contents of an introductory course on feedback
control developed at UCSC. The course material, available at ?, is based on
fundamental concepts in dynamical systems, modeling, stability, robust-
ness, and the design of feedback control laws. We have reported our posi-
tive experience in using the material in a summer course for motivated high
school students during the summer of 2005. We believe that researchers
from systems and control theory will find the material useful for a vari-
ety of activities, ranging from education and outreach to high school and
undergraduate students, to interdisciplinary collaborations with scientists
from other disciplines.

An interesting study performed in ? examines how novices learn feed-
back control concepts, in particular how they gain a perspective on control
that includes abstraction, so that the concepts are not tied to a specific prob-
lem description. The study involves the use of a GUI-based software that
allows the students to connect and simulate a set of standard functional ob-
jects (for example, sensor, actuator, setpoint unit) that exist in many control
applications. As part of an introductory engineering course, the software
was made available to a group of high school students for about 45 min-
utes per day for 5 days. A key finding of the study is that “the idea of a
signal is a core concept in an expert’s conception of feedback systems, and
is intricately tied to the definition of the functional components that make
for a feedback control system.” The study further reported that, by the end
of the course, students still had trouble with the concept of signals.

In our course, we used Matlab and described vectors, which most stu-
dents had seen in a math or physics course, as a sequence of measurements
taken at snapshots of time. Most students needed a few days and several
exercises to connect the list of numbers (vector) in Matlab with the evolu-
tion of a model of a real physical system. Once they grasped this idea, we were able to describe the use of models for prediction and control design. We never formally introduced the word signal (suggestions for introducing this concept in primary and secondary education can be found in ?), and did not attempt to generalize the vector description to more general scenarios, for example, continuous time, since this step seemed unnecessary. As a future direction, our material, based largely on algebra and Matlab exercises, as well as a GUI-based approach to control such as the software described in ?, can further help students comprehend the concepts and gain intuition for control design.

Finally, a completely new approach to engineering education is being undertaken at the Olin College of Engineering, in Needham, Mass., see ?. The school opened to incoming freshman in 2001, and the emphasis of the curriculum is on entrepreneurship, interdisciplinary learning, and teamwork. Feedback control is considered an “important engineering concept” in the Olin curriculum [?, page 32]. The article also shows pictures of students and instructors working on autonomous vehicles, robots, and non-linear and chaos theory graphs. This overlap between the topics covered in our short course and the topics offered at Olin is reassuring, since Olin College is already demonstrating an impact on the need for engineering education reform.
Chapter 2

Basic MATLAB use

MATLAB is an interactive, numerical computation and graphics program. MATLAB will serve as a useful tool for illustrating some of the mathematical concepts and solutions that we will see throughout the course. Think of MATLAB as the ultimate calculator! As a complement to the material of this chapter, you can download the useful reference “Getting Started with MATLAB” from http://www.mathworks.com/access/helpdesk/help/techdoc/Matlab.html.

This chapter assumes you are familiar with trigonometric functions and exponents (covered in Algebra II / Precalculus in most US high schools). It helps if you have had some exposure to vectors. It also helps if you have experience with plotting data and functions (by using a graphing calculator, for example), and with programming for-loops, although these skills are not required.

<<<Add reference to Hongyun and Neil>>>
2.1 Chapter objectives

When we are finished covering this chapter, you should be able to:

1. **Define** what a vector is in your own words. Also, define five basic vector operations in terms a family member could understand, and without referring to MATLAB.

2. **Identify** those vector operations (of the five defined) that require the use of “.” in MATLAB, and to correctly use “.” when needed in such situations.

3. **Explain** (without using jargon), in the context of the discussion on sensors in the first class presentation, how a vector relates to a sensor. In your explanation, use an example of a real sensor and what a corresponding sensor vector might be. Also explain how a time vector would correlate to the sensor vector.

4. **Create** a time vector, given any start time and end time, and any constant sample period.

5. **Identify** when the use of “:” in MATLAB is and is not appropriate, for both for-loops and in defining vectors.

Once you can perform these tasks, you will be well prepared for the first quiz!

2.2 Preliminaries

This session begins once you have figured out how to start up MATLAB, and you are sitting in front of a computer screen on which the MATLAB command window is displayed. This has a prompt “≫” and a cursor awaiting instruction.

2.2.1 Create your working directory

The first thing you should do is to get to the working directory on the machine at your desk. The instructions for doing this will be provided on the white board.

---

1The importance of being able to explain complicated concepts to “laypeople” can not be overstated.
2.2. PRELIMINARIES

Once you are in the working directory, create a new folder with your last name as the label. Within your folder, create another folder with the name “matlab.” This directory is where you should keep all your work for this class.

Task 2.2.1 (Start MATLAB and go to your Working Directory) Double click the MATLAB icon on the desktop to start MATLAB. When the command window opens and you see the command prompt \( \gg \), type

\[
\gg \quad \text{cd C:} \backslash (\text{working directory}) \backslash (\text{your last name}) \backslash \text{matlab}
\]

From now on, at the beginning of every chapter, begin by following this procedure of getting into your working directory. All MATLAB related files that you save should be kept in this directory.

2.2.2 Help!

MATLAB has an extensive help facility and a number of demonstration programs. The command “help function-name” is the syntax. For example

\[
\gg \quad \text{help cos}
\]

provides a description of the cosine function “\( \cos(\cdot) \)”. The help facility is also divided into categories, and a list of categories can be found by typing “help” alone with no function-name argument. To access a list of functions under a category, simply type “help category-name”. For example,

\[
\gg \quad \text{help elfun}
\]

produces a list of the elementary functions that are available in MATLAB (and each is described by executing help again with the relevant function-name).

Task 2.2.2 (Help Yourself) Type “help” for the list of categories, choose a category (we suggest “elfun”), and type “help category-name” for the chosen category-name. Within the list of functions, pick one, and type “help function-name” for the chosen function-name. Given the help information, try to implement the chosen function.

Task 2.2.3 (Built-in Demonstrations) Type “demos” for the demonstrations. These are often more complicated than you bargained for, e.g. check out the cool “peaks” and “modes”, which give some idea of the graphics that MATLAB is capable of.
2.3 Mathematics with MATLAB

2.3.1 Numbers and operations

In MATLAB, numbers are what you expect: 12.4 entered after the prompt, followed by a carriage return (“Enter”), means exactly that to MATLAB:

\[
\gg 12.4 \text{ (“Enter”)}
\]
\[
\text{ans} = 12.4
\]

with answer, being the number 12.4 abbreviated as “ans” by MATLAB. MATLAB won’t do anything other than interpret the command as a number unless you tell it to do more. Note that “Enter” tells MATLAB to execute the command contained on that line.

Simple arithmetic operations are also straightforward:

\[
\gg 12.4 \times (48.5 + 342/39) \text{ (“Enter”)}
\]

means \(12.4(48.5 + \frac{342}{39})\), and produces the result:

\[
\text{ans} = 710.1385
\]

The arithmetic operations recognized by MATLAB are:

+ addition
- subtraction
* multiplication
/ division
\(\wedge\) raise to power

From now on, we will omit typing “Enter” in the notes when giving you exercises to do in MATLAB.

2.3.2 Output

If you get sick of MATLAB always printing the output “ans = …”, you can suppress printing the output of a particular command by ending the line with a semi-colon “;”, so

\[
\gg 6;
\]

outputs nothing, and

\[
\gg x = 6;
\]
assigns the number 6 to the variable \( x \) without printing the output. We’ll learn more about assigning values to variables in the next section.

MATLAB also uses customary scientific notation (is anyone unfamiliar with scientific notation?):

\[
\text{≫ cos}(1.57) \\
\text{ans} = 7.9366e-04
\]

which means \( 7.9366 \times 10^{-4} = 0.00079366 \).

MATLAB lets you know if you screwed up too. Suppose the variable \( y \) has no value assigned to it. Then, you can verify that the follow occurs:

\[
\text{≫ cos}(y) \\
\text{Error: Value of y unassigned.}
\]

(Earlier versions were actually programmed to respond to certain rude comments.)

### 2.3.3 Variables

You can create variables in MATLAB by assigning numerical values to variables names:

\[
\text{≫ x} = 3 \\
x = 3
\]

assigns the value of 3 to \( x \), and MATLAB remembers this until \( x \)'s value is re-assigned by another command. So, for example,

\[
\text{≫ 2.5} \times x \\
\text{ans} = 7.5
\]

Also, simply typing the variable name produces the last value assigned to that variable:

\[
\text{≫ x} \\
x = 3
\]

The value of a variable can also be defined as the result of a computation. For example,

\[
\text{≫ y} = (2 + x) \times 3 \\
y = 125
\]

assigns the value \((2 + x)^3 = 5^3 = 125\) to \( y \).
CHAPTER 2. BASIC MATLAB USE

Some variables are already assigned in MATLAB. For example, pi stands for $\pi = 3.14159...$. So

$$\gg z = 2 \times \pi + 6$$

$$z = 12.2832$$

Note that “ans” can be used like any other variable, although its value is reassigned each time you do a computation without assigning to a specific variable name.

2.3.4 Mathematical functions

MATLAB has a large number of built-in mathematical functions. Some of them are:

- abs(x) absolute value of $x$, $|x|$
- sqrt(x) square root of $x$, $\sqrt{x}$
- sign(x) sign of $x$
- exp(x) $e^x$
- log(x) $\ln(x) = \log_e(x)$
- log10(x) $\log_{10}(x)$
- sin(x) $\sin(x)$
- cos(x) $\cos(x)$
- tan(x) $\tan(x)$
- asin(x) $\sin^{-1}(x)$
- acos(x) $\cos^{-1}(x)$
- atan(x) $\tan^{-1}(x)$

MATLAB assumes that the argument is expressed in radians for the trigonometric functions. e.g.

$$\gg x = \pi$$

$$x = 3.14159$$

$$\gg z = \sin(x)$$

$$z = 0$$

2.3.5 Vectors

MATLAB handles vectors also. For us, a vector is a finite and ordered sequence of numbers, separated by commas. For example, consider the vector $x = (1, 3, 2, 4)$. In MATLAB, you enter this as

$$\gg x = [1, 3, 2, 4]$$

$$x = \begin{bmatrix} 1 & 3 & 2 & 4 \end{bmatrix}$$
The length of a vector is called its dimension. So, $x$ has dimension 4. One-dimensional vectors are referred to as scalars. Note that the variable discussion in Section 2.3.3 made use of scalars.

The numbers in a vector are referred to as elements of the vector, so 3 is the 2nd element of the vector $x$ defined above. You can access the $i$th element of a vector $y$ by typing “$y(i)$”. In the example, then,

\[
\gg x(3) \\
\text{ans} = 2
\]

When typing the vector, commas or spaces are used to separate elements, whereas MATLAB outputs the vector with spaces between elements rather than commas. Also, typing a vector means starting the sequence with “[” and ending with “]”, although MATLAB does not output brackets.

Amazingly, MATLAB can handle arithmetic defined on the elements of vectors. In fact, all five of the arithmetic operations defined in Section 2.3.1 can be used with vectors. For example, let’s take the vector $x = (1, 3, 2, 4)$ and multiply every element by 5:

\[
\gg 5 \times x \\
\text{ans} = 5 \quad 15 \quad 10 \quad 20
\]

The output is another vector with each element multiplied by 5! Now suppose we wanted to square each element of $x$:

\[
\gg x.\,\&\,2 \\
\text{ans} = 1 \quad 9 \quad 4 \quad 16
\]

For exponential operations on vectors, you need to add a period before the “$\&$” symbol, so that MATLAB knows that it must perform the operation element-by-element. There are other operations where this is also the case. Typically, if you try an operation without the period and it turns out that the period is required, an error will be printed indicating that you need the period.

**Task 2.3.1** With the artifice of the period, we can also multiply the elements of two vectors together if they have the same length. For instance, multiply

---

2If you are taking or will take AMS 27 Mathematical Methods for Engineers, you will learn a lot more about vector operations than what we’ll cover here. One of the purposes of this class is to provide you with explicit motivation for more advanced mathematics courses. As this course proceeds, you will see where more advanced math tools, such as those covered in AMS 27, would allow you to do more sophisticated engineering design and analysis.
the vectors \( w = (1, 2, 3, 4) \) and \( v = (17, 12, 23, 17) \). If you get an error from MATLAB, re-check that your are using the period correctly. Next, divide the elements of \( w \) by the corresponding elements of \( v \), using “/” and “.” □

The period is not necessary for sums and subtractions

\[
\begin{align*}
\gg x + z \\
\text{ans} & = 5 \ 5 \ 5 \ 5 \\
\gg x - z \\
\text{ans} & = -3 \ -1 \ 1 \ 3
\end{align*}
\]

We can sometimes apply functions on each element of a vector by simple operations on the vector as a whole: the exponentiation \( e^{(i)} \) of each element in \( x \) is obtained by

\[
\gg \text{exp}(x) \\
\text{ans} = 2.7183 \ 7.3891 \ 20.0855 \ 54.5982
\]

and the sine of each element is

\[
\gg \sin(x) \\
\text{ans} = 0.8415 \ 0.9093 \ 0.1411 \ -0.7568
\]

Now, a useful shorthand way to generate the vector \( x \) is

\[
\gg x = [1 : 1 : 4] \\
x = 1 \ 2 \ 3 \ 4
\]

The notation \( p : n : q \) instructs MATLAB to assign the ordered list of elements that starts with \( p \), ends with \( q \), and increments in steps of size \( n \).

**Task 2.3.2** Construct a vector \( z \) vector whose elements are the even integers from 2 to 1000 using the previous construction (try it with and without the semi-colon). Once you have constructed \( z \), construct the vector \( x = z/1000 \). Thinking of \( x \) as a time vector, specify the start time, the end time and the sample period.

**2.4 Plots**

Finally we are ready for some simple plotting. Let \( y = \sin(2\pi x) \), where \( x \) is the long vector constructed above in Task 2.3.2. We may plot \( x \) against \( y \) with the simple plot command:

\[
\gg y = \sin(2 \pi x); \\
\gg \text{plot}(x, y)
\]
2.4. PLOTS

We can plot more than one curve at the same time:

\[
\gg \quad \text{plot}(x, y, x, \cos(2 \times \pi \times x))
\]

Note how this plots a function directly, without first creating a new vector like \( y \).

We can even change the style used to plot the curves:

\[
\gg \quad \text{plot}(x, \sin(2 \times \pi \times x),'-', x, \cos(2 \times \pi \times x),':')
\]

which plots the sine curve with a dashed line and the cosine curve with a dotted one. The three plots are shown in Figure 2.1. More plotting styles can be found by executing “help plot”.

![Sample plots](image)

Figure 2.1: Sample plots

**Task 2.4.1 (Plotting the evolution of the US population)** Let us start using what we have learned so far by playing with the US Census data from 1900 to 2000, as depicted in the following table:

<table>
<thead>
<tr>
<th>Year</th>
<th>Population (mil)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900</td>
<td>75.995</td>
</tr>
<tr>
<td>1910</td>
<td>91.972</td>
</tr>
<tr>
<td>1920</td>
<td>105.711</td>
</tr>
<tr>
<td>1930</td>
<td>123.203</td>
</tr>
<tr>
<td>1940</td>
<td>131.669</td>
</tr>
<tr>
<td>1950</td>
<td>150.697</td>
</tr>
<tr>
<td>1960</td>
<td>179.323</td>
</tr>
<tr>
<td>1970</td>
<td>203.212</td>
</tr>
<tr>
<td>1980</td>
<td>226.505</td>
</tr>
<tr>
<td>1990</td>
<td>249.633</td>
</tr>
<tr>
<td>2000</td>
<td>281.422</td>
</tr>
</tbody>
</table>
 CHAPTER 2. BASIC MATLAB USE

Define two vectors, \( t \) and \( p \). The vector \( t \) should contain the years in the first column (remember the shorthand way to generate vectors in Task 2.3.2), and the vector \( p \) should contain the second column with the population for each year. Then, use the plot command to plot the evolution of the population throughout the century. Make the title “Population of the U.S. 1900-2000” and make the \( y \)-axis label “Millions.” You should get something that looks like Figure 2.2. Also, try to get a circle to show up at each data point as shown in the figure.

![Figure 2.2: US population 1900-2000.](image)

### 2.5 Programming

#### 2.5.1 Loops

Some calculations require repetition. For example, suppose you want to take \( \pi \), compute its square root, and then add one: \( \sqrt{\pi} + 1 \). Then, you want to square root the answer, and add one: \( \sqrt[4]{\pi} + 1 + 1 \). Suppose you want to do this 200 times(!). There is a way to do this will three lines of code, if you know how to program a for-loop. That is, you can repeat commands any number of times by placing them in a for-loop. The MATLAB construction of a for-loop is as follows. First, pick a loop counter variable \( (n, \text{ for example}) \) and then increment \( n \) from 1 to \( N \) using the following commands in MATLAB:

```matlab
for n = 1:N
    % commands
end
```
2.5. PROGRAMMING

≫ for n=1:N
     blah
end

where “blah” denotes the commands we want to repeat \( N \) times, and “end” terminates the loop. The “blah” commands can be just about anything you want. For example, to do the calculation above 200 times, we would set \( N \) to 199 (why?), and “blah” would incorporate the “square root and add one” commands. Specifically,

≫ x = sqrt(pi) + 1;
≫ for n=1:199
    x = sqrt(x) + 1;
end
≫ x
ans = 2.6180

At each iteration of the counter \( n \) in the loop, the value of \( x \) is reassigned as the latest calculated value that is square rooted and added to one. This is done 199 times in the loop since the calculation was done once before the loop started, and we wanted to do the calculation at total of 200 times. What happens if we do not start \( x \) with the value \( \sqrt{\pi} + 1 \)? Try it for yourself. It should make sense that when the loop starts (\( n = 1 \)), it can’t calculate \( \sqrt{x} + 1 \) if \( x \) has not been assigned a value yet. Assigning a start value for a variable that gets updated in a loop is called initializing a loop variable. If you forget to do this, rest assured - MATLAB will complain.

Consider now a more complicated for-loop that also makes use of the plot function. Suppose you want to plot the function \( f(x) = 1 - ax^2 \) nine times, each time with a different value for the parameter \( a \). Using ‘for’ we can do it as follows:

≫ x=[0:100]/100;
≫ a=[1.1,1.2,1.4,1.6,1.9,2,2.5,3,4]/4;
≫ plot(x,x)
≫ hold on
≫ for n=1:9
    fx=1-a(n)*x.^2;
    plot(x,fx)
end
≫ hold off

The initial plot command “plot(x,x)” gets everything going here (see Figure 2.3, left). The instruction “hold on” makes sure that previous plots are not erased by plotting the current one.
After typing these commands, you should get something similar to Figure 2.3, right.

![Figure 2.3: Plot of the function \( f(x) = x \) (left) and \( f(x) = 1 - ax^2 \) (right) for different values of the parameter \( a \).]

If you already have previous knowledge of a programming language, the idea of the loop should be familiar; what’s different is the MATLAB format and the fact that it can deal with plotting within the loop.

**Task 2.5.1** Consider the rule

\[ x_{n+1} = \cos x_n \]

If we start with \( x_1 \) and use the rule, we get \( x_2 = \cos x_1 \). Now, using the rule again, we can compute \( x_3 = \cos x_2 \). And so on. Use the ‘for’ command to compute the value of \( x_2, x_3, \ldots, x_{10} \) starting with \( x_1 = \pi/6 \).

**2.5.2 m-files**

Frequently, it becomes tedious to keep on repeating the same sets of commands, or re-entering a bunch from a previous MATLAB session. A far more efficient approach is to place the MATLAB commands into a text file that we call something like `rubarb.m` (rubarb is, of course, not the only word one can use here, but the .m is essential). The format should be identical to how the commands would appear on the MATLAB command line. Whenever we want to execute the commands contained in the “m-file”, all we need to do is enter “rubarb” on the command line and MATLAB will read and execute them.
To create, edit and modify an m-file, you need to be able to open the text file with a suitable editor. There is an editor in MATLAB that you can use, but any other editor is also acceptable. To use the MATLAB editor, simply type

\[ \gg \text{edit} \]

This provides you with another window containing the editor; in this window you can happily type commands. You must save the file before you can execute the commands in it.

**Task 2.5.2** Write an m-file that when invoked from the MATLAB command window automatically generates the plot in Figure 2.2.

Using m-files, you can define your own functions. These functions can be used much like the existing functions, such as \( \sin( ) \) and \( \sqrt{x}( ) \). For example, typeset the following lines in your editor and name the file `myfunction.m`,

```matlab
function output = myfunction(x)
    output = exp(x.^2).*x;
end
```

The first word ‘function’ tells MATLAB that we are defining a function. The name of the new function is ‘myfunction’. It takes as argument the variable \( x \), and spits out ‘output’ - which is defined in the second line (what would have happened if we do not use the period?). Now, with the function saved as `myfunction.m`, one can use the function as follows:

\[ \gg \text{myfunction}(2) \]

\[ \text{ans} = 109.1963 \]

As before, adding “;” at the end would have naturally suppressed the output. Much more complicated function definitions are possible, and we will make extensive use of m-file functions in this class.

**Task 2.5.3** Plot `myfunction` from -1 to 1 using an increment step size of 0.1.

**Task 2.5.4** Let us combine our knowledge of the instruction ‘for’ with the definition of new functions. Create a new function (call it `until`) that takes as an argument an integer \( n \) and spits out the vector \( x = (1, 2, 3, \ldots, n) \).

---

3 Make it a habit: Always keep the name of the function in the m-file the same as the saved m-file name. So, an m-file that has “function out = treeline(in)” as the first line should be saved as `treeline.m`.
2.6 Homework

Make sure that you can perform each of the chapter objectives defined in Section 2.1. In addition, turn in the problems below. Problems C1p1-C1p4 are to be done by everyone individually. Problem C1g1 is to be done by each group. Only the group secretary needs to turn in group problems. Please write your name clearly on the top of each page, and for group problems include the group name at the top of each page.

1. Print your commands and output for Tasks 2.3.1 and 2.3.2. You may write your answers to the “time vector” questions in Task 2.3.2 by hand. Even better, you can answer these questions by showing the commands (and output) that access the appropriate elements of the vector \( x \).

2. Print your commands for Task 2.4.1 and include the plot that you created (circles on data points required).

3. Print your commands and output for Task 2.5.1. Your output needs to only include \( x_{10} \).

4. Turn in your m-file from Task 2.5.2.

5. **Group Problem: Artificial Sensor.** Using some of the functions listed in Section 2.3.4, and any other MATLAB functions you like, create a new function called GroupSensorFun1.m. The purpose of this function is to emulate a real sensor. This requires your group to think about a real sensor, what the data coming from it might look like, and to emulate this data with MATLAB functions. If there is “noise” on the sensor data, for example, you might consider using the \( \text{rand.m} \) function. Your group must justify the functions you use in order to emulate the real sensor. The function should take a time vector input and return a vector of the same length that contains the “sensor data.”

Once you have your GroupSensorFun1.m defined, create an m-file that applies your function to the until.m function created in Task 2.5.4. For \( n = 200 \), plot the output vector of your function versus the output of the until.m function\(^4\). Label and title the figure as you

\(^4\)You may choose a different value for \( n \) if it seems appropriate. The chosen value should be such that a correlation can be made between the real sensor you are emulating and the artificial sensor data that you plot. How does \( n \) correlate to time?
see appropriate, keeping in mind that it is sensor data that you are plotting.

Your group secretary must turn in the two functions `GroupSensorFun1.m` and `until.m`, the m-file that creates the plot, and the plot itself. Also, include a paragraph that first defines the real sensor that your `GroupSensorFun1.m` function is supposed to be emulating, and then justifies the functions you use in your `GroupSensorFun1.m` function so that it emulates the real sensor. Extra points will be given for creativity and conciseness of code!
Chapter 3

Discrete-time dynamics

In this chapter, we introduce discrete-time dynamical systems, and discuss why they are important for engineering and scientific purposes. We will also learn about equilibrium points for these systems, and what stability of equilibrium points means and how to determine it.


In Chapter 2, we introduced the concept of a vector, and how to manipulate and plot vectors in MATLAB. This chapter will build on these tools so that we can examine vectors that have some scientific or engineering meaning. Specifically, we will use discrete-time dynamic equations, described below, to create vectors. These vectors, in turn, define the evolution of the dynamic process that the equations are describing.
3.1 Chapter objectives

This chapter assumes you have studied the material and completed the homework for Chapter 2. When we are finished covering this Chapter, you should be able to:

1. **Define a dynamic model** in a way that could be understood by one of your non-engineering friends. Give an example of a dynamic system, from engineering or science, and describe why someone would be interested in a model of that system.

2. **Identify** a qualitative example of a system that has a stable and attractive equilibrium point. Also, identify a qualitative example of a system that has an unstable equilibrium point.

3. **Identify** the fixed points for a given model, and, when possible, determine the stability of the fixed points using mathematical analysis. Be able to use Matlab to plot different orbits that demonstrate the stability properties of each fixed point.

4. **Create** a model that has no fixed points. Also, create a model that has infinitely many fixed points. Be able to identify a system in real life that has no equilibrium states, and another real-life system that has infinitely many equilibrium states.

3.2 Discrete-time dynamic equations

Let us begin with an example of a discrete-time dynamic equation that is very simple and produces an interesting result.

**Group Task 3.2.1** All you need to do this task is a calculator and a notepad. Pick an arbitrary number and call it $x_1$. Enter $x_1$ in your calculator, set your calculator to radian mode and press the cosine button. Call the resulting number $x_2$. Continue to press the cosine button, labeling the successive readouts as $x_3$, $x_4$, and so on. Continue this until $x_N = x_{N+1}$ for some number $N$. What is the value of $N$? What is the value of $x_N$? Notice that you have created a vector $(x_1, x_2, ..., x_N)$, where $N$ is the number of times you press the cosine button.

The discrete-time dynamic equation that generates the vector you’ve created is

$$x_{n+1} = \cos x_n.$$  \hfill (3.1)
3.2. DISCRETE-TIME DYNAMIC EQUATIONS

The equation tells you how to calculate the next element in the vector, given the previous element. Notice that all elements are uniquely determined, given the start value \( x_1 \), and that a for-loop could be written to calculate all the elements, as done in Task 2.5.1 of Chapter 2. In this chapter, you will learn how to use mathematical analysis to show why the cosine function converges to the same number (called a fixed point), no matter what value \( x_1 \) you start with.

If you have taken an algebra course, you are used to equations. In this course, we focus on equations that have specific properties, and the two adjectives discrete-time and dynamics describe these properties. Let us now separate and define these words, starting with dynamics.

If you have taken physics, you may have been introduced to the word “dynamics.” Here is an online encyclopedia entry for dynamics, courtesy of Answers.com:

**dynamics** A branch of mechanics\(^1\) that deals with the motion of objects, which may be further divided into kinematics, the study of motion without regard to the forces producing it, and kinetics, the study of the forces that produce or change motion. Motion is caused by an unbalanced force acting on a body. Such a force will produce either a change in the body’s speed or a change in the direction of its motion (acceleration). The motion may be either translational (straight-line) or rotational, or both.

**Group Task 3.2.2** Think of an autonomous robot, and an environment in which it operates, such that the robot exhibits translational but not rotational motion. Now, think of another (possibly different) robot that exhibits both translational and rotational motion. Identify the types of motions that our robot is capable of exhibiting. Lastly, try to identify sensors that provide translational or rotational motion data.

In this course, “dynamic equations” more generally refers to equations that describe a system's evolution in time. If the system is mechanical, as referenced in the description above, it is the motion of an object that is evolving in time. If the system is electrical, it is the motion of electrons that are evolving in time. If the system is a population, it is the interaction, birth and death rates of people that are evolving in time. Dynamic equations are also referred to as the dynamic model (or just model) of a system. There are

\(^1\)http://www.answers.com/topic/dynamics

\(^2\)Mechanics is, in turn, a branch of physics, and applied mechanics is considered a branch of engineering.
models for inverted pendulums, robots, cars, satellites, and even certain biological organisms.

**Group Task 3.2.3** Recall the *time vectors* we studied in Chapter 2. Generically, we can denote a time vector as \((t_1, ..., t_N)\), with \(t_1\) the start time and \(t_N\) the end time. Each element \(t_i\) is a specific instant in time. Assume a constant *sample period*, denoted as \(\Delta\). A constant sample period means that the time between \(t_n\) and the previous element \(t_{n-1}\) is the same, for any \(n = 2, ..., N\). Determine the **model of time**, that is, the equation that determines each element \(t_{n+1}\), given the previous element \(t_n\). How “good” is this model?

One can define a varying sample period, instead of a constant sample period, but we won’t have to worry about that in this class. Also, in AMS 27, you will learn about *continuous time*, which you can think of as what you’d get by monitoring a clock continuously; instead of discrete time instants, you’d have all times in the interval \([t_{\text{start}}, t_{\text{stop}}]\).

Now, a dynamic equation is called “discrete-time” if the equation is evaluated at snapshots in time, at a chosen sample period. Consider the equation (3.1) as an example, and recall the discrete time vector \((t_1, t_2, ...)\). Each calculated value of the variable \(x_n\) can be associated with the discrete-time \(t_n\). Just as the elements of a time vector are *instants in time*, elements in the vector created by a discrete-time dynamic equation are the *instants in the evolution* of the system that is modeled. So, instead of a dynamic equation that models the continuous evolution of, say, the position of a robot roaming around the room, we deal with discrete-time dynamic equations that predict the position of the robot at discrete times as it roams around the room. For example, our model might predict the robot position every 0.02 seconds.

A generic dynamic model is written as

\[ x_{n+1} = f(x_n), \]

where \(f\) can be any function, for example, \(f(x) = x^2\) or \(f(x) = e^{\text{arctan}(x)}\). The vector \((x_1, x_2, ...)\) generated from a model is referred to as the **orbit starting from** \(x_1\).

**Task 3.2.4** Let’s use MATLAB to plot some orbits of the model \(x_{n+1} = \cos x_n\). First, retrieve the programs `orbit.m` and `iterates.m` from the course web page\(^3\). Type `help orbit` in your command window to learn how to

\(^3\)http://www.soe.ucsc.edu/classes/cmpe008/Fall06/
use it. For instance, \texttt{orbit(’cos’,0,25)} should generate the left plot in Figure 3.1. Now, join the data points to make your plot look the right plot in Figure 3.1.

![Figure 3.1: Orbit of the equation \(x_{n+1} = \cos x_n\).](image)

Now, why are scientists and engineers interested in models for different types of systems? And how do you know you have a “good” model?

**Group Task 3.2.5** Think of any system in engineering or science that would be of interest to model using dynamic equations. Think about why you would want to model that system, and how you might determine if the model is any good.

Here are some of the reasons for developing dynamic models of systems. By having a mathematical description of the system, one can use mathematical analysis to reason about why the system does what it does, for example, why the cosine function converges to the same number no matter what number you start with, as seen in task 3.2.1. We will develop the analysis tools later in this chapter that will allow you to show precisely why the cosine function does this.

Another reason to develop a dynamic model is that, with the model, one can use simulations to predict how the system is likely to respond under various conditions. A simulation is simply a calculation of an orbit for a given model. So, with a model, one can plot orbits to examine the effects of different environmental conditions on how the system will respond. For example, Air Traffic Management experts at NASA Ames run tens of thousands of simulations with models to examine the effects of runway density and various weather conditions on new management strategies. In
The primary reason we care about models in this class is that models enable us to design feedback control equations that result in desirable autonomous behavior. A cruise controller, for example, is designed by using the model. It will help you to keep this reason in your mind as we proceed with the course. The instructors will also help by reminding you of this reason, as it is one of the most important messages of the whole course!

A good model of a system is one that allows you to reliably predict how the system will evolve in the near future. An air traffic model that is unaffected by changes in weather patterns is NOT likely to be a good model. In almost all cases, no model is perfect. If models were perfect, the weather man would always be 100% correct.

### 3.3 Fixed points of dynamic models

Consider again the generic model

\[ x_{n+1} = f(x_n). \]  

(3.2)

A point \( x_* \) is a fixed point of (3.2) if \( f(x_*) = x_* \). A fixed point is also referred to as an equilibrium point, or equilibrium state. The word “equilibrium” intuitively refers to the fact that the system being modeled prefers to remain at this point for all time, if it starts there. For example, a model of a falling object would naturally have the ground as an equilibrium state.

**Task 3.3.1** For a model with the equilibrium point \( x_* \), what is the orbit starting from \( x_* \)?

**Task 3.3.2** What are the fixed points of \( f(x) = x + x^2 - 1 \)?

**Group Task 3.3.3** Identify a model that has no equilibrium points. Then, identify a model that has infinitely many equilibrium points. Lastly, what are the equilibrium states of a pendulum.

**Group Task 3.3.4** A simple model of a car’s velocity is \( v_{k+1} = v_k + (\Delta/m)[-bv_k + h_k] \), where \( v_k \) is velocity, \( h_k \) is the slope of the road, \( b \) is the drag coefficient, \( m \) is the mass of the car, and \( \Delta \) is the sample period. If the road is flat,
3.4 Stability of fixed points

Given a dynamic model, the first task is typically to calculate the fixed points of the model. The second task is to determine how the system reacts if it starts nearby a fixed point. A falling object that starts near the Earth clearly ends up there quickly. The cosine model (3.1) also demonstrated the property of approaching a fixed point (your calculator told you what the fixed point was in that case). In other cases, the system evolves away from its fixed point. An example is the upright position of an inverted pendulum. The calculation of a fixed point, first, and determining how the system behaves near that fixed point, second, are very important to quantify for engineering purposes. For example, Ford engineers would want to know that the equilibrium point of a cruise-controlled car is always within 1 mph of the set speed, and the car will always converge to the equilibrium speed from any start speed.

The question of how a system behaves near a fixed point is analytically addressed by looking at the stability of the fixed point \( x^* \). A fixed point \( x^* \) is stable if, for any \( x_1 \) sufficiently close to \( x^* \), every future element in the orbit remains as close to \( x^* \) as desired. A fixed point that is not stable is said to be unstable. These definitions admittedly take a little getting use to, so let’s think more about them using some familiar examples.

Consider the inverted pendulum again. The upright fixed point \( x^* \) of an inverted pendulum is unstable. The reason is that the only \( x_1 \) that results in an orbit that remains near \( x^* \) is \( x_1 = x^* \). Any other \( x_1 \) near \( x^* \) results in divergence of the orbit away from \( x^* \). The downward fixed point \( x_s \), on the other hand, is stable since any \( x_1 \) near \( x_s \) results in an orbit that remains near \( x_s \). Furthermore, the closer \( x_1 \) is to \( x_s \), the closer the orbit will remain to the downward position \( x_s \).

In engineering, “remaining close” is often not good enough. Ford would
not be happy if their control engineering told them “I can get ‘er within 10 mph, boss.” A stronger condition is if the fixed point is stable and attractive. A stable fixed point \( x^* \) is attractive if the orbit converges to \( x^* \), either in finite time or as time approaches infinity. The fixed point of the cosine model (3.1) is an example of a stable and attractive fixed point. If the inverted pendulum model includes a term for friction, the downward position also becomes a stable and attractive fixed point. (What if there is no friction?)

**Group Task 3.4.1** Given examples of (possibly different) systems, not previously mentioned, that have a stable but not attractive fixed point, a stable and attractive fixed point, and an unstable fixed point.

Given a model \( f \), we can calculate fixed points. In the coming sections, you will see how to use mathematical analysis and plotting orbits to determine the stability of a fixed point, rather than leaving it to a qualitative description based on intuition. This makes the Ford management happy, and saves the control engineer his job. While analysis is sometimes possible, for any given model, we can always use for-loops and plots (which you already know!) to examine the stability of any fixed point. In other words, there is a lot you can do already!

### 3.4.1 Linear models

The simplest model, for which to determine fixed points and their stability, is a linear model. The function that corresponds to a linear model is \( f(x) = ax \), where \( a \) is any real number.

**Task 3.4.2** What are the fixed points of \( x_{n+1} = f(x_n) = ax_n \)?

You should have discovered that \( x^* = 0 \) if \( a \neq 1 \), and \( x^* \) is any real number if \( a = 1 \). If \( a = -1 \), there are no fixed points. Now, we want to determine the stability of the fixed points. Assume now that \( a \neq 1 \), so that we only need to determine the stability of \( x^* = 0 \) since this is the only fixed point. Before doing some analysis, let’s do what we know we can: generate plots.

**Task 3.4.3** Consider the model \( f(x) = x/2 \) (in this case, \( a = 1/2 \)). Using the orbit.m function, decide if the fixed point \( x^* = 0 \) is stable, stable and attractive, or unstable. You should try different values for \( x_1 \) to convince yourself. Next, do the same with \( f(x) = 2x \) (in this case, \( a = 2 \)).
The plotting should have helped you determine the stability of the fixed point $x^* = 0$ for two cases: in one case, $a = 1/2$; in the other case, $a = 2$. You should have found that the $a = 1/2$ case is stable and attractive, while the $a = 2$ case is unstable. Now, let’s proceed with a more formal analysis using algebra.

Let $a$ be any value. After one iteration of the model, we have $x_2 = ax_1$. After two iterations, $x_3 = ax_2 = a^2x_1$. After $n$ iterations, we have $x_n = a^{n-1}x_1$. What happens as $n$ gets large? If $-1 < a < 1$, then $a^n \to 0$ and $x_n \to 0$. Instead, if $a > 1$ or $a < -1$, then $x_n \to \infty$. So the fixed point $0$ is stable and attractive if $-1 < a < 1$, and $0$ is unstable if $|a| > 1$.

What about $a = 1$? Any chosen $x_1$ is a fixed point, and starting as close as you want to this $x_1$ implies the orbit will remain that close. For this value of $a = 1$, any point is a stable (but not attractive) fixed point.

### 3.4.2 Nonlinear models

Many models of real systems are not linear, or nonlinear. Our robot model is an example of a nonlinear model, since it has sine and cosine functions. We will study this model in detail in later Chapters. If the model is nonlinear, what can we say about the stability of a fixed point? In this section, we explore this question.

Suppose that $x_*$ is a fixed point for the nonlinear function $f (f(x_*) = x_*)$. To determine the stability of $x_*$, let’s define the variables $\varepsilon_n = x_n - x_*$, which represent the deviation of the updated variable $x_n$ from the fixed point $x_*$. Looking at the deviation variable is a nice way of simplifying what we are looking for: to determine stability and attractivity of the fixed point $x_*$, we need only to determine if the orbits of $\varepsilon_n$ remain close to, and even converge to, the origin, i.e., the fixed point $\varepsilon_* = 0$.

Given a starting $\varepsilon_1 \neq 0$, let’s see whether the deviation $\varepsilon_n$ grows or decays as $n$ increases. To help our analysis along, we have to introduce and approximation, so that the nonlinear function becomes a linear function. Once we have a linear function, we can use our results from the previous section. Substitution for the first iteration yields

$$x_2 = f(x_1) = f(x_* + \varepsilon_1) \approx f(x_*) + f'(x_*)\varepsilon_1 + \text{error},$$

where $f'(x) = df(x)/dx$ is the derivative of $f$ with respect to $x$, and $f'(x_*)$ is this derivative evaluated at the fixed point $x = x_*$. In this equation, the approximation is called a “Taylor approximation”. Don’t worry if you don’t know about what this is. It simply says that, up to small error, one
can approximate the value of the function $f$ at $x_\ast + \varepsilon$ by the value of $f$ at $x_\ast$ and a small correction involving the derivative of $f$ at $x_\ast$. (Don’t worry also if you don’t know what a derivative is - we’ll provide it for you when necessary).

Now, using $x_2 = x_\ast + \varepsilon_2$, the above equation can be rewritten as

$$x_\ast + \varepsilon_2 = x_\ast + f'(x_\ast)\varepsilon_1 + \text{error}$$

or, equivalently,

$$\varepsilon_2 = f'(x_\ast)\varepsilon_1 + \text{error}$$

Let us ignore the small error term for now. Then, we obtain the linear model

$$\varepsilon_{n+1} = f'(x_\ast)\varepsilon_n, \quad n = 1, 2, ... \quad (3.3)$$

Task 3.4.4 Based on our knowledge of linear models, can you find conditions on $f'(x_\ast)$ that guarantee that the deviation $\varepsilon_n$ will asymptotically go to 0? Hint: the derivative of $f(x) = ax$ is $f'(x) = a$ for any $x$.

As for linear models, we deduce that for $|f'(x_\ast)| < 1$, the equilibrium point is stable and attractive, and for $|f'(x_\ast)| > 1$, the equilibrium point is unstable.

Task 3.4.5 Consider the model $x_{n+1} = x_n^2$. Find the fixed points and determine their stability. Hint: the derivative of $f(x) = x^2$ is $f'(x) = 2x$, for any $x$.

It turns out that nothing can be said about the stability of a fixed point $x_\ast$ of any nonlinear system when the linearization gives $|f'(x_\ast)| = 1$.

We could develop similar stability tests for models $f$ which are not 1-dimensional. Think for instance of the model $f(x_1, x_2) = (\sin x_1, x_1 \cos x_2)$. The model for our robot is in fact 3-dimensional. In order to do stability tests for fixed points of higher dimensional models, calculus and matrix algebra are required. You will take these courses within the school of engineering curriculum. In this course, for higher-dimensional models, we will test stability of fixed points only by simulation, that is, by making use of for-loops and plotting in MATLAB to determine the behavior of orbits near the fixed point.

Group Task 3.4.6 Now that you have seen how to determine stability quantitatively (that is, by using mathematical analysis), let’s redo the group task
3.4.1. Given examples of (possibly different) systems, not previously mentioned, that have a stable but not attractive fixed point, a stable and attractive fixed point, and an unstable fixed point. Instead of being qualitative, you examples should be quantitative. That is, for each type of stability, provide the model, show the fixed point and show the stability property for that fixed point.

3.5 The logistic map

In this section, we play with the logistic equation and see some very wild behavior. If you are interested in more related material, read the cool book [?]. The logistic map is given by

\[ x_{n+1} = rx_n(1 - x_n) \]

The logistic map is used to model the population growth of some species. Here, \( x_n \geq 0 \) is a measure of the population in the \( n \)th generation and \( 0 \leq r \leq 4 \) is the growth rate.

<<< At the beginning, tell the kids to define an m-file with the logistic map >>>

Task 3.5.1 Plot the function \( f(x) = rx(1 - x) \) in MATLAB for \( r = 1/2, 1, 2 \), and with 100 values of \( x \) between 0 and 1. You should get something similar to Figure 3.2.

![Figure 3.2: Plot of the function \( rx(1 - x) \) for different values of \( r \).](image)
Task 3.5.2 Find the equilibrium points of the logistic map and determine its stability properties. Hint: the derivative of \( f(x) = rx(1 - x) \) is \( f'(x) = r(1 - 2x) \) for any \( x \).

3.5.1 Wild behavior of the logistic map: period doubling

Let us play a bit with the logistic map and MATLAB to see some interesting behavior. Recall the handy MATLAB files `orbit.m` and `cobweb.m` that we have been using so far. Once you have started MATLAB, do the following:

1. check that for small growth rate \( r < 1 \), the population goes extinct: \( x_n \to 0 \). Try with \( r = 1/2 \) and \( x_0 = 0.1 \).

2. check that for \( 1 < r < 3 \), the population grows, and eventually reaches a nonzero steady state. Try with \( r = 2.8 \). Plot the first 50 points of the orbit starting from \( x_0 = 0.1 \).

3. check that for bigger \( r \), say \( r = 3.3 \), the population builds up again but now oscillates about the former steady state, alternating between a large population in one generation and a smaller population in the next. This type of oscillation, in which \( x_n \) repeats every two iterations, is called a period-2 cycle.

4. things get even more interesting when we continue making \( r \) bigger. Try \( r = 3.5 \). The population approaches a cycle that now repeats every four generations, i.e., a period-4 cycle.

Actually, further period doublings occur as \( r \) keeps increasing. Computer experiments show the next cool result. Let \( r_n \) denote the value of \( r \) where a \( 2^n \)-cycle first appears. Then we have the following sequence of bifurcations,

\[
egin{align*}
    r_1 & = 3 & \text{period 2 is born} \\
    r_2 & = 3.449\ldots & \text{period 4 is born} \\
    r_3 & = 3.54409\ldots & \text{period 8 is born} \\
    r_4 & = 3.5644\ldots & \text{period 16 is born} \\
    \vdots \\
    r_{\infty} & = 3.569946\ldots & \infty
\end{align*}
\]

Successive bifurcations come faster and faster. Ultimately, the \( r_n \) converge to the limiting value \( r_{\infty} \) above.
3.5. THE LOGISTIC MAP

3.5.2 What happens for $r > r_{\infty}$?

According to and , Robert May, who was the professor who studied the logistic map, posed a similar question to his students on a corridor blackboard. The answer is complicated. For many values of $r$, the orbits never settle down to a fixed point or a periodic orbit – instead, the long-term behavior is aperiodic.

Task 3.5.3 Try $r = 3.9$, and plot using orbit.m the first 51 points of the orbit starting at $x_0 = .12$. Try to plot also the corresponding cobweb with cobweb.m. You’ll see what we mean! (you should get something similar to Figure 3.3. Try also changing the initial condition).

![Figure 3.3: Orbit and corresponding cobweb of the logistic equation.](image)

One good guess is that the dynamics becomes more and more complicated as $r$ increases. Actually, the situation is even more subtle than that. To see the long-term behavior of the system for all values of $r$ at once, we are going to plot the orbit diagram. To do that, we need to create a computer program. The idea is described next.

Task 3.5.4 The orbit diagram is a plot of the orbits of the logistic equation against the values of $r$. We have to do a computer program with two loops, something like the following.
CHAPTER 3. DISCRETE-TIME DYNAMICS

<table>
<thead>
<tr>
<th>Name:</th>
<th>Orbit diagram algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal:</td>
<td>Plot orbit diagram of logistic equation</td>
</tr>
</tbody>
</table>

The idea is to have a figure where many orbits of the logistic map are plotted (one for each value of $r$ between 3.4 and 4). So the $x$-axis of the figure will correspond to values of $r$, and the $y$-axis will correspond to values of the orbits.

1: Set $i = 0$.
2: Set $r = 3.4 + i$.
3: Iterate the logistic map for 200 cycles (after 200 cycles, the system should settle down to its eventual behavior – the settling portion of the response is called the transient) starting from $x_0 = .6$.
4: Once the transients have decayed, plot many points, say $x_{201}, \ldots, x_{400}$, versus the current value of $r$ in the figure.
5: Set $i = i + 0.005$. If $r = 4.0$, exit the algorithm. Otherwise, return to step 2.

Once you have the algorithm setup, you should get something similar to Figure 3.4.

Looking at Figure 3.4, you can see that at $r = 3.4$, the attractor (the set of points to which the orbits eventually tend) is a period 2-cycle, as indicated by the two branches. As $r$ increases, both branches split simultaneously, yielding a period-4 cycle. This splitting process continues as $r$ increases, until at $r = r_\infty \approx 3.57$ the map becomes chaotic, and the attractor changes from a finite to an infinite set of points.

For $r > r_\infty$, the orbit diagram reveals an unexpected mixture of order and chaos, with periodic windows interspersed between chaotic clouds of dots. The large window beginning near $r \approx 3.83$ contains a stable period-3 cycle. A blow-up of part of the period-3 window is shown in the right plot of Figure 3.4. A copy of the orbit diagram reappears in miniature!

3.5.3 Analysing the logistic map

Let us try to understand the dynamics of the logistic map. Recall that we already know that the equilibrium points are 0 and $1 - 1/r$. The origin is stable for $r < 1$ and unstable for $r > 1$. The other equilibrium point is stable for $1 < r < 3$, and unstable for $r > 3$. Figure 3.2 gives a graphical clarification of these results.
To find the fixed points, you have to look for the intersection between the plot of $f$ and the line $y = x$. For $r < 1$, there is only one intersection at $x = 0$ in the interval $[0, 1]$. Right at $r = 1$, a new intersection point forms at $x = 0$. For $r > 1$, there are two intersection points.

Now, we are going to prove the following neat result: “the logistic map has a 2-cycle for any $r > 3$”. How do we do that? Let us first look at what a 2-cycle really means. It simply means that there exists $p$ and $q$ such that

$$f(p) = q, \quad f(q) = p.$$  

Now, the cute thing is to realize that this equation is equivalent to

$$f^2(p) = f(f(p)) = p$$

i.e., all we are looking for is for fixed points of $f^2$!

**Task 3.5.5** Find an explicit expression for $f^2$, where $f(x) = rx(1 - x)$. Because $f$ is a quadratic polynomial, then $f^2$ is a quartic polynomial. Its graph is given in Figure 3.5.

**Task 3.5.6** Find the 4 fixed points of $f^2$. As a hint, realize that we already know two – the fixed points of $f$ are necessarily fixed points of $f^2$.

How do we determine the stability of a 2-cycle? That’s a very good question. Actually, the solution is simple: we reduce the question to study the stability of a fixed point, which we know how to do. Both $p$ and $q$ are fixed points of $f^2$, right? Therefore, the original 2-cycle is stable precisely if $p$ and $q$ are stable fixed points of $f^2$. 

---

Figure 3.4: Orbit diagram (left, zoom on the right) of the logistic equation.
Task 3.5.7 Building on the previous discussion, determine the stability of the 2-cycle of the logistic map for $r > 3$. 
3.6 Homework

Make sure that you can perform each of the chapter objectives defined in Section 3.1. In addition, turn in the problems below. Problems C2p1–C2p3 are to be done by everyone individually. Problem C2g1 is to be done by each group. As before, only the group secretary needs to turn in group problems. Please write your name clearly on the top of each page, and for group problems include the group name at the top of each page.

1. To support the results of group task 3.2.1, use quantitative analysis to show why the equation (3.1) converges to the fixed point $x^* = 0.7391$. That is, show that this fixed point is stable and attractive. Hint: the model is nonlinear, and for $f(x) = \cos(x)$, the derivative is $f'(x) = -\sin(x)$. Also, your analysis should make use of equation (3.3).

2. Write your own m-file that creates the right plot in Figure 3.1. In other words, redo task 3.2.4, without using the orbit.m and iterates.m functions. Instead, use a for-loop and plot functions within your new function to get the same effect. Turn in your m-file and the resulting plot.

3. Turn in your answers to Tasks 3.3.1 and 3.3.2.

   The purpose of this problem is for you to use simulations (that is, to plot orbits) to determine stability properties for the fixed points calculated in group task 3.3.4. Where possible you will also use mathematical analysis to justify your stability conclusions. In addition, you will examine the settling time for the orbits computed, which is a commonly used measure of performance in the dynamics and control community.

   The model of the car’s speed is
   \[ v_{k+1} = v_k + \frac{\Delta}{m} [-bv_k + h_k]. \]
   Let $\Delta = 0.01$, $m = 20$, and $b = 0.5$ (don’t worry about dimensions). The road incline effect is captured by $h_k$. If $h_k = 0$, the road is flat. If $h_k > 0$, the road is inclined downhill. If $h_k < 0$, the road is inclined uphill.
Note that for this model, the engine term $u_{\text{eng}}$ discussed in the introductory slides, is not present. You can think of this model as describing the dynamics of the car when the engine (and therefore the cruise control) is turned off, so that $u_{\text{eng}} = 0$. The result is a car that “coasts.”

In the problems below, set $v_1 = 25$ and define a time vector $t = (t_1, t_2, \ldots, t_N)$ with start time $t_1 = 0$, end time $t_N = 200$, and constant sample period $\Delta$. For all plots required below, be sure to turn grid on.

This problem is broken down into the following parts:

(a) Identify the equilibrium speed $v_*$ for any constant $h_k$ (so $h_k$ is constant for all time, that is, for any $k = 1, 2, \ldots$).

(b) (Zero incline case). Write an m-file that plots the speed orbit versus time $t$. On the plot, show the speed orbit in black. Also, show the equilibrium speed on the plot, using a red dashed line. Discuss whether the orbit converges to your fixed point calculation from part (a). Does the fixed point appear to be stable and attractive, or unstable?

Since $h_k = 0$ for all time, the model is linear and in the form discussed in section 3.4.1. Use mathematical analysis to show whether the fixed point $v_*$ is stable and attractive or unstable. Are both stability conclusions (through plotting orbits, and through mathematical analysis) consistent?

(c) (Downhill incline case). Set $h_k = 10$ for all time, and using the same m-file, plot the speed orbit versus time $t$ ON THE SAME PLOT that was generated for the zero incline case. This requires the use of the hold on command. As before, show the speed orbit in black, and include the equilibrium speed on the plot using a red dashed line. Discuss whether the orbit converges to your fixed point calculated in part (a). Does the fixed point appear to be stable and attractive, or unstable?

(d) (Uphill incline case). Set $h_k = -20$ for all time, and using the same m-file, plot the speed orbit versus time $t$ ON THE SAME PLOT that was generated for the zero and downhill incline cases. As before, show the speed orbit in black, and include the equilibrium speed on the plot using a red dashed line. Discuss whether the orbit converges to your fixed point calculated in part (a). Does the fixed point appear to be stable and attractive, or unstable?
(e) **Settling time calculation**. For each of the three orbits, calculate the *settling time value*, which is the time value at which the orbit has come within ±1 of the equilibrium speed. This is an important measure of performance for control systems. Engineers like to know, for example, that the disk drive speed controller can quickly change rates to within a small % of a commanded rate. This means good performance translates to small settling times. In that sense, determine which orbit shows the best “performance,” meaning the orbit is the fastest to get within ±1 of equilibrium. Can you explain physically why this orbit has the fastest settling time?

Your group secretary should turn in the single m-file, and the single plot with all three orbits and corresponding equilibrium speeds. Title and label the plot as you see appropriate. Lastly, include a separate sheet(s) of paper that presents your answers to the questions posed within parts (a), (b), (c), (d) and (e).
Chapter 4

Analysis and simulation of dynamic models

In the previous chapter we introduced models of discrete-time dynamical systems. For these models, we learned about equilibrium points and the stability of equilibrium points. In this Chapter, the equilibrium and stability of two example dynamic models are examined. One model is for a predator-prey system, and the other model is for the robots that we’ll be experimenting with later in the course. In addition, you will simulate these models, to predict how they will respond and to aid in analyzing the stability of the equilibrium points.

Recall from Chapter 3 that a dynamic model refers to equations that describe a system’s evolution in time. In this chapter, we’ll explore two models in detail: one model of a predator-prey population system, and another model of the robot that we will be experimenting with in the coming weeks. Given a model, we want to know: 1) what are the equilibrium points, and 2) what is the stability of these points. We will use algebra to determine 1), and simulations to determine 2) in this chapter.
4.1 Chapter objectives

This chapter assumes you have studied the material and completed the homework for Chapter 3. When we are finished covering this Chapter, you should be able to:

1. Identify all of the variables (positions, orientation, control inputs) and parameters (update time) for the model of our robot. Derive the equilibrium position, orientation, and control inputs, and the stability of these fixed points. Also, be able to describe the equilibrium state of the robot to a layperson, that is, to a non-expert, such as a non-engineer family member.

2. Identify, for a given dynamic model of a system, some of the assumptions made about the system when the model was constructed. For the robot model, in particular, be able to state at least two assumptions made when this model was created.

3. Create a Matlab function, with parameter values as inputs, that plots different orbits for a given model. Use the orbits to decide on the stability properties of fixed points calculated for the model.

4.2 Predator-Prey Population Dynamic Model

The predator-prey problem refers to an ecological system in which we have two species, lynxes and hares, where lynxes eat hares. This type of system has been studied for decades and it is known to exhibit very interesting and, in certain cases, realistic dynamics. Figure 4.1 (right) shows a historical record taken over 90 years in the population of lynxes versus hares. As can been seen from the graph, the annual records of the populations of each species are oscillatory in nature.

Let us construct a simple model for the evolution of the populations. To do so, we will keep track of the rate of births and deaths of each species.

Let \( H \) represent the population of hares and \( L \) represent the population of lynxes. Let \( k = 1, 2, \ldots \) denote the days at which we look at the populations, starting on January 1, 1845. For instance, \( H_k \) is the hare population at day \( k \). Let \( b \) be the percentage of the current hare population that is expected to be born over a years time, which is assumed to be constant and

\[ \frac{dH}{dt} = bH - cHL, \]

\[ \frac{dL}{dt} = -aL + dHL. \]

\[ \text{Figures taken from “The Connected Curriculum Project” at http://www.math.duke.edu/education/ccp.} \]
4.2. PREDATOR-PREY POPULATION DYNAMIC MODEL

Figure 4.1: Canadian lynx versus hare

the same for every year. By this notation, from day $k$ to day $k + 1$, $b_r H_k/365$ new hares are born (we divide between 365 because $b_r$ is the annual rate).

On the other hand, let $d_f$ be the percentage of the current lynx population that is expected to perish over a year's time, which is also assumed to be constant and the same for every year. Therefore, from day $k$ to day $k + 1$, $d_f L_k/365$ lynxes are expected to perish. Finally, let us model the fact that the lynxes are eating the hares with a term which is proportional to the hares and lynxes populations. Let’s say that at day $k$, $aL_k H_k/365$ hares are killed by the lynxes, and that $aL_k H_k/365$ new lynxes are “produced” by eating hares. Here, $a$ is a real number.

Written in the form of a dynamic model, the system is

$$H_{k+1} = H_k + \frac{b_r}{D} H_k - \frac{a}{D} L_k H_k,$$

$$L_{k+1} = L_k - \frac{d_f}{D} L_k + \frac{a}{D} L_k H_k,$$

where $D = 365$. This simple model makes many simplifying assumptions – such as the fact that hares never die of old age or causes other than being eaten – but it often is sufficient to answer simple questions about the system. To illustrate the usage of this system, we can compute the number of lynxes and hares from some initial population. This is done by starting with $(H_1, L_1)$ and then using equations (4.1) to compute iteratively the populations in the following day, i.e., by calculating the orbit of the difference equation (4.1). A sample orbit for a specific choice of parameters and initial conditions is shown in Figure 4.2. Notice the similarities with the plot in Figure 4.1!

**Task 4.2.1** Compute the equilibrium population levels $(H_*, L_*)$ from the model (4.1).
Figure 4.2: A simulation of the predator-prey model with $a = .007$, $b_r = .7$ and $d_f = .5$. The initial populations are 20 hares and 35 lynxes.

**Group Task 4.2.2** Write a function in MATLAB whose output is a plot of an orbit for the predator-prey model, like the orbit shown in Figure 4.2. Define the function to have the parameters $a$, $b_r$, $d_f$, the initial populations $H_1$, $L_1$, and the number of iterations $N$, all as the INPUT arguments, so that you can vary the values of these parameters. Name the file PredPrey.m. **Hint:** The first line of the file should be `function out = PredPrey(H1,L1,a,b,d,N)`, and a for-loop should calculate the orbit using the start point $(H_1,L_1)$, and using the parameter values $(a,b,d)$ with the model (4.1). The parameter $N$ will tell the for-loop when to stop. Recreate the orbit in Figure 4.2.

**Group Task 4.2.3** Use the function that you created in Group Task 4.2.2 to determine the stability character of the equilibrium points that you computed in Task 4.2.1. Are the equilibrium points stable, stable and attractive, or unstable? You may use the values for $(a, b_r, d_f)$ given for Figure 4.2.

We can associate to a single system many different models. The model you use depends on the questions you want to answer. In the previous example, we know that there are various important things that we have not considered, like the fact that hares never die. An important question, which we briefly discussed in the last Chapter, is ...
What do we want models for?

As we said earlier, we generally want models to predict how the system will behave under different situations. Models will help us to find out what is going on with the system. For instance, in the predator-prey example, questions we might want to answer include

- Given the current population of hares and lynxes, what will it be next year?
- If we hunt down lots of lynxes in a given year, what will the effect on the rabbit and lynx population be?
- How do long term changes in the amount of rabbit food available affect the populations?

We can answer these questions once we have constructed a model for the evolution of the populations. Of course, the answers we get will depend on the model we use. So it is very important to be conscious about the assumptions made when constructing the model. For instance, in our case we have assumed that

- The predator species is totally dependent on the prey species as its only food supply
- The prey species has an external food supply and no threat to its growth other than the specific predator

You can imagine that the answers we get will be very different if a sudden frost freezes the food supply in a given year!

4.3 Robot Dynamic Model

Now, we’re ready to introduce you to a model of the robot that we will be programming and experimenting with. First we need to define the variables and the parameters that will be in the model. To begin, examine the schematic image of an enlarged Robobrain robot sitting the middle of a room, given in Figure 4.3.

The variables that define the position of the robot in the room are $x$ and $y$, using the coordinate frame indicated in the figure. The orientation of the robot, that is, the direction that the robot is heading, is denoted by the variable $\theta$, and we define this angle to be measured from the horizontal $x$-axis to the dashed line that runs through the middle of the robot.
Figure 4.3: Schematic drawing of an enlarged Robobrain robot operating in a room. The border in the drawing represents the room walls, and the coordinate system shows that the lower left corner is denoted \((x, y) = (0, 0)\) and the upper right corner is denoted \((x, y) = (X, Y)\), with \(X\) and \(Y\) given by the dimensions of the room.

Also indicated in the figure are the **translational velocity** \(u\) (the rate of moving forward or backward) and the **rotational velocity** \(v\) (the turning rate). These velocities can be computed as a function of the angular velocities of the left and right wheels, denoted \(\omega_L\) and \(\omega_R\), respectively, measured in radians per second (rad/sec). We now proceed with generating these functions.

The velocities of the right and left wheels, \(v_R\) and \(v_L\), respectively, measured in meters per second (m/sec), are given by

\[
\begin{align*}
v_R &= \omega_R r, \\
v_L &= \omega_L r,
\end{align*}
\]

where \(r\) is the radius of each wheel in meters (m). Since we can command the independent angular velocities \((\omega_L, \omega_R)\) with each motor, we can directly control the independent velocities \((v_L, v_R)\).

Now, the translational and rotational velocities \((u, v)\) are calculated from the velocities \((v_L, v_R)\) as

\[
\begin{align*}
u &= \frac{1}{2} (v_R + v_L), \\
v &= \frac{2\pi}{l} (v_R - v_L),
\end{align*}
\]

\[\text{A nice web site that helps clarify this derivation is http://www.phys.unsw.edu.au/~jw/rolling.html} \]
where \( l \) is the width of Robobrain (the distance between the wheels). These variables are also shown in Figure 4.3. The units are (m/sec) for \( u \) and (rad/sec) for \( v \).

Substitution of the equations for \((v_L, v_R)\) in terms of the commanded angular velocities \((\omega_L, \omega_R)\) gives the following equations for \((u, v)\)

\[
\begin{align*}
u &= \frac{r}{2}(\omega_R + \omega_L), \\
v &= \frac{2\pi r}{l}(\omega_R - \omega_L).
\end{align*}
\]

Thus, we can directly control the independent translational and rotational velocities \((u, v)\). Therefore, we refer to \( u \) and \( v \) as the controls, or the control inputs.

We are now in a position to present the model of the dynamics of Robobrain. In other words, as with the predator-prey example, we will define an update rule for the three variables that describe a new location of the robot \((x_{k+1}, y_{k+1}, \theta_{k+1})\), in terms of the current values \((x_k, y_k, \theta_k)\), the sample period \(\Delta\), and the current values for the controls \((u_k, v_k)\). Note that since the forward and turning speeds can change with time, depending on the commands we send to the motors, we use the \( k \) subscript on \( u \) and \( v \).

For previous discussions, recall that \(\Delta\) is the sample period and denote the sample times as \(t_k = k \cdot \Delta, k = 1, 2, \ldots\). Also, \(x_k\) is the \( x \) position of the robot at time \( t_k \), using the same subscript notation for the other variables. The discrete-time dynamic model of the robot dynamics is given by

\[
\begin{align*}
x_{k+1} &= x_k + \Delta u_k \cos(\theta_k) \\
y_{k+1} &= y_k + \Delta u_k \sin(\theta_k) \\
\theta_{k+1} &= \theta_k + \Delta v_k
\end{align*}
\]

**Group Task 4.3.1** Explain the model \(\theta_{k+1} = \theta_k + \Delta v_k\) physically. Identify operating conditions under which this is a good model. Then, identify operating conditions under which this is NOT a good model.

**Task 4.3.2** Compute the equilibrium position(s) and orientation(s) \((x_*, y_*, \theta_*)\) from the model (4.2). Determine the commanded speeds \((u_*, v_*)\) that are required for the robot to be at equilibrium. Lastly, with commanded speeds at \((u_*, v_*)\) for all time, determine the stability of the equilibrium \((x_*, y_*, \theta_*)\).

**Group Task 4.3.3** Write a function in MATLAB whose output is a plot of an orbit for the robot model, with \(u_k = 4\), for all \(k = 1, 2, \ldots\), and with \(\Delta = 0.01\). Define the function to have the following inputs: initial positions \(x_1, y_1\), and orientation \(\theta_1\), the number of iterations \(N\), and \(v_c\), defined as the constant
rotational speed (that is, $v_k = v_c$ for all $k = 1, 2, ...$, with $v_c$ as an input parameter). Name the file Robot.m. Plot three different orbits, using the parameter values $(x_1, y_1, \theta_1, N) = (2, 0, \pi/4, 500)$, with a different value for $v_c$ for each orbit, namely $v_c = 1$, $v_c = 4$ and $v_c = 10$. Hint: Follow your approach for creating FredPrey.m, now using the model (4.2).

The previous tasks brought to light an intriguing property of systems for which we have a control input. By modifying the control $u_k$, for example, by making it a feedback function of the position $x_k$, we can alter the equilibrium and even the stability of the modeled system through feedback control! This is precisely how feedback control is used to stabilize pendulums in the upright position. We will examine this capability more in the coming Chapter.
4.4 Homework

Make sure that you can perform each of the chapter objectives defined in Section 4.1. In addition, turn in the problems below. Problems C3p1–C3p3 are to be done by everyone individually. Problem C3g1 is to be done by each group. As before, only the group secretary needs to turn in group problems. Please write your name clearly on the top of each page, and for group problems include the group name at the top of each page.

1. For \((a, b_r, d_f) = (0.01, 0.5, 0.9)\), calculate the equilibrium populations \((H_*, L_*)\). Plot the orbits for \(H_k\) and \(L_k\) starting from \((H_1, L_1) = (40, 15)\), labeling the plot with title, xlabel, ylabel and legend commands.

2. Turn in your solution to Task 4.3.2.

3. Reuse the function created in Group Task 4.3.3, modifying the function so that the roles of \(u_k\) and \(v_k\) are reversed. Rename and re-save the function as \texttt{RobotMod.m}. The constant value for \(u_k\), denoted \(u_c\), is an input parameter, and \(v_k\) is set equal to 4, for all \(k = 1, 2, \ldots\). Create three plots, using the same parameter values as before for \((x_1, y_1, \theta_1, N)\), and using \(u_c = 0.1, 1, 12\). Each plot will show the orbit for a specific value for \(u_c\). Label the plots as you see fit.

4. Group Problem: Equilibrium and Stability Properties for a Cruise-Controlled Car. The purpose of this problem is for you analyze a cruise-controlled car model, by determining equilibrium points and using simulations (that is, to plot orbits) to determine the stability of the equilibrium points. Where possible you will also use mathematical analysis to justify your stability conclusions.

The model of the car’s speed is

\[
v_{k+1} = v_k + \frac{\Delta}{m}[-bv_k + u_k + h_k].
\]

Let \(\Delta = 0.01\), \(m = 20\), and \(b = 0.5\) (again, don’t worry about dimensions). The road incline effect is captured by \(h_k\). If \(h_k = 0\), the road is flat. If \(h_k > 0\), the road is inclined downhill. If \(h_k < 0\), the road is inclined uphill. The engine accelerator/brake pedal are the control input, captured by \(u_k\). If \(u_k > 0\), the car is accelerating; if \(u_k < 0\), the car is decelerating (by braking)\(^3\).

\(^3\)Most cruise controllers don’t brake to slow down; the controller usually stops accelerating and lets drag slow the car (corresponding to setting \(u_k = 0\)). Here, we’re considering a cruise controller that will actively brake to decelerate.
The cruise control model is

\[ u_k = c_1 (v_{\text{des}} - v_k) + c_2, \]

where \( c_1 \) and \( c_2 \) are constants chosen by the control designer, and \( v_{\text{des}} \) is the constant desired driving speed chosen by the driver. Let \( v_{\text{des}} = 75 \).

The control objective is for you to choose \( c_1 \) and \( c_2 \) so that \( v_{\text{des}} \) is a stable and attractive equilibrium point for the car speed dynamic model. As in many real-life control applications, you will see that this objective becomes complicated by the presence of \( h_k \), which you can think of as a disturbance.

For orbit calculations in the problems below, use \( v_1 = 40 \) and define a time vector \( t = (t_1, ..., t_N) \) with start time \( t_1 = 0 \), end time \( t_N = 200 \), and constant sample period \( \Delta \). For all plots required below, also turn grid on.

This problem is broken down into the following parts:

(a) \( (c_2 = 0, h_k \neq 0 \text{ and constant}) \).
   i. Identify the equilibrium speed \( v_* \).
   ii. Choose a value for \( c_1 \), using the stated control objective as a guide. Since no control can have infinite magnitude, and the engine acceleration and braking deceleration are no exception, assume that the maximum value \( u_k \) can be is 1000, with a minimum of -1000. You can also assume the maximum speed of the car to be 100. These bounds will help you make an engineering decision to decide on a good value for \( c_1 \).
   iii. Create a MATLAB function CarOrbit.m with parameter inputs \( (c_1, c_2, h_k) \) that plots the \( v_k \) orbit. Plot the orbit for parameter values \( (c_1, 0, -5) \), using the value for \( c_1 \) determined in part (ii), and decide on the stability of the equilibrium point calculated in part (i) based on this orbit.
   iv. State whether you met the control objective. If not, state how close you got to meeting the control objective, that is, state how close your \( v_* \) got to \( v_{\text{des}} \).

(b) \( (c_2 \neq 0, h_k = 0) \). (Ignoring the disturbance in control design)
   i. Identify the equilibrium speed \( v_* \).
ii. Choose a value for $c_2$, based on the stated control objective. Next, revisit your choice for $c_1$, based on the stated control objective and the bounds on the control $u_k$.

iii. Reuse your CarOrbit.m function. Plot the orbit for parameter values $(c_1, c_2, 0)$, using the values for $c_1$ and $c_2$ determined in part (ii), and decide on the stability of the equilibrium point calculated in part (i).

iv. State whether you met the control objective. Compare this controller with the controller from part (a), and decide which is better in terms of meeting the control objective.

(c) ($h_k$ time varying). (Affect of the disturbance on control design) Modify your CarOrbit.m function, renaming and re-saving it as CarOrbitDist.m. In the new function CarOrbitDist.m, remove $h_k$ as a constant parameter input. Instead, make $h_k$ the following function of time:

$$h_k = 10 \sin(2\pi t_k/50).$$

The parameter inputs for the new function should be just $(c_1, c_2)$.

i. Plot the orbit for parameters $(c_1, 0)$, using the value of $c_1$ calculated in part (a)(ii).

ii. Plot the orbit for parameters $(c_1, c_2)$, using the values calculated in part (b)(ii).

iii. State which controller does better at meeting the control objective, based on the two orbits calculated. State why you feel the chosen controller is better.

Your group secretary should turn in the two function m-files, in addition to the responses and plots that go along with answering all of the problems. As usual, title and label the plot as appropriate.
Chapter 5

Feedback control of dynamic models

In this chapter we give our first steps into the realm of feedback control. First, we will get a rough idea of what feedback is. Then, we will see how to turn unstable discrete-time dynamical systems into stable ones using feedback. What we have learned in previous lectures will be very helpful to do this. We will even be able to assign fixed points of the system wherever we want.
CHAPTER 5. FEEDBACK CONTROL OF DYNAMIC MODELS

5.1 Introduction: the magic of feedback


The term feedback is used to refer to a situation in which two (or more) dynamical systems are connected together such that each system influences the other and their dynamics are thus strongly coupled. By dynamical system, we refer to a system whose behavior changes over time, often in response to external stimulation or forcing. Simple causal reasoning about a feedback system is difficult because the first system influences the second and the second system influences the first, leading to a circular argument. This makes reasoning based on cause and effect tricky and it is necessary to analyze the system as a whole. A consequence of this is that the behavior of feedback systems is often counterintuitive and it is therefore necessary to resort to formal methods to understand them.

So, various important things to notice: (i) feedback is “interconnection” between systems; (ii) just when we think we understand how a feedback system works, the interconnection between different systems might have surprises in store for us. As said above, reasoning based on cause and effect gets tricky (it’s like asking: who came first, the chicken or the egg? It gets crazy!); and (iii) you’ll be exposed in the coming lectures to powerful formal methods that will get you through the tricky business and allow you to analyze the behavior of feedback systems.

Feedback allows us to make a system behave as desired. It keeps variables constant and stabilizes unstable systems. Feedback also reduces the effects of disturbances and component variations, allowing a new freedom to designers.

Feedback is ubiquitous in both natural and engineered systems. Everyday life applications of feedback include:

1. flight control

2. feedback amplifier for long-distance telephone networks and television networks

1 Available online at http://www.cds.caltech.edu/~murray/amwiki.
5.2. CONTROLLED DYNAMICS

3. CD player
4. And so many more...

5.2 Controlled dynamics

From the standpoint of control it is interesting to explore the possibilities of shaping the dynamic behavior of a system by using feedback control. Let us first introduce the class of systems we are talking about. We consider controlled systems of the form

$$x_{k+1} = f(x_k, u_k)$$ (5.1)

In general, external inputs (forces, electrical currents, chemical concentrations) are bounded (i.e., we cannot produce a force as infinitely large!). Therefore, \(u\) is restricted to belong to some bounded set, let’s call it \(U\). We only know about 1-dimensional systems, so for us \(U \subset \mathbb{R}\), which in English, means that the set \(U\) is a subset of the real numbers. For example, \(U = [0, 1]\), which is all numbers \(x\) satisfying \(0 \leq x \leq 1\). Later in the course we will also consider higher-dimensional systems.

If we do not exert any control over the system (i.e., the system is unforced), then \(u_k = 0\), and we simply have

$$x_{k+1} = f(x_k, 0)$$

This is a discrete-time dynamical system like the ones we saw in Lecture 3.

Example 5.2.1 (Cruise control example) Recall the cruise control model

$$v_{k+1} = v_k + \frac{\Delta m}{m}[-bv_k + u_{\text{eng},k} + u_{\text{hill}}].$$ (5.2)

Recall also that the speed of the car \(v_k\) is the state (so \(x = v\) here) and the gas pedal \(u_{\text{eng},k}\) is the control input. We’ve included the subscript \(k\) now to emphasize that the control signal can change with time. The map \(f\) is \(f(v, u) = v + \frac{\Delta m}{m}[-bv + u_{\text{eng},k} + u_{\text{hill}}]\). The unforced system (i.e., when the engine is off and the road is flat) is

$$v_{k+1} = v_k + \frac{\Delta m}{m}[-bv_k]$$ (5.3)

Task 5.2.2 What are the equilibrium points of (5.3)?
5.3 Shaping the dynamics: forcing desired equilibrium points

The first thing we may be interested in is to find controls that give desired equilibrium points. For that purpose we will consider a controlled system of the form (5.1). The equilibrium of this system are given by \( f(x_{eq}, u_{eq}) = x_{eq} \). Therefore, the equilibria that can be achieved the points \( x \) such that there exist \( u \in U \) with \( f(x, u) = x \). (Note: The notation \( u \in U \) simply means that \( u \) is in the set \( U \)).

**Task 5.3.1 (Cruise control example)** Let \( v_{ss} \) a desired velocity for our car. Assume for simplicity that we are driving on a flat road, so that \( u_{hill} = 0 \) in (5.2). Find the control \( u_{eng,ss} \) that makes \( v_{ss} \) an equilibrium.

Based on this discussion, here it is a simple algorithm to make your desired point an equilibrium of the system:

1. Select you desired “wanna-be” equilibrium point \( x_{ss} \)
2. Find the corresponding control \( u_{ss} \) such that \( f(x_{ss}, u_{ss}) = x_{ss} \)
3. Choose the control input to be \( u_k = u_{ss} \) for the system (5.1).

After performing these two steps, we have the new system

\[
 x_{k+1} = f(x_k, u_{ss})
\]

We have managed to make the “wanna-be” equilibrium point \( x_{ss} \) into an actual fixed point of the system! But in general, this equilibrium can be stable or unstable. Let’s see it in the cruise-controller example.

**Task 5.3.2 (Cruise control example)** Let’s plot the orbits of cruise controller example on a flat road once we have chosen the input \( u_{eng,ss} \) that makes our desired speed \( v_{ss} \) an equilibrium. Therefore, consider the equation

\[
 v_{k+1} = v_k + \frac{\Delta}{m}[-bv_k + u_{eng,ss}],
\]  

(5.4)

where you should substitute \( u_{eng,ss} \) by the one you obtained in Task 5.3.1. Write a program in MATLAB that asks for the values of the parameters \( \Delta \), \( b \) and \( m \), a desired velocity \( v_{ss} \), an initial velocity \( v_0 \) and the number of iterations \( N \). The program should plot the first \( N \) points of the orbit of (5.4) starting from \( v_0 \). Use the following set of values: (i) \( \Delta = 1, m = 100 \) and \( b = 10 \), and (ii) \( \Delta = 30, m = 100 \) and \( b = 10 \). Is the equilibrium stable in both cases? Why?
5.4. **SHAPING THE DYNAMICS: MAKING THE EQUILIBRIUM STABLE**

In Task 5.3.2, we have seen that our strategy has been only partially successful so far. We got \( v_{ss} \) to be an equilibrium of the system, but depending on the values of the parameters, this equilibrium might be stable or unstable. We certainly don’t want this for our car! We want it to be stable, independently of the values that the parameters take (within certain limits). We can do this by means of control. That’s what we explain in the next section.

**Task 5.3.3** Consider the following one-dimensional controlled system

\[
x_{k+1} = x_k^2 + u_k
\]

Let \( x_{ss} = 2 \). Find the control \( u_{ss} \) that makes \( x_{eq} \) an equilibrium. Is \( x_{ss} \) a stable equilibrium point of the resulting system \( x_{k+1} = f(x_k, u_{ss}) \)? Plot the first 7 points of the orbit starting from \( x_0 = 2.001 \). You should get something like Figure 5.1.

![Figure 5.1: Orbit starting from \( x_0 = 2.001 \) of the system \( x_{k+1} = x_k^2 + u_{ss} \).](image)

5.4. **Shaping the dynamics: making the equilibrium stable**

Let’s go over our former steps. Remember the simple algorithm we used to make our desired point an equilibrium. Let’s slightly modify the last one to keep our options open. We rewrite it as follows:

1. Select your desired “wanna-be” equilibrium point \( x_{ss} \)
2. Find the corresponding control \( u_{ss} \) such that \( f(x_{ss}, u_{ss}) = x_{ss} \)
3. Choose the control input to be \( u_k = u_{ss} + \tilde{u}_k \) for the system (5.1). You can think of \( \tilde{u}_k \) as a new control that we have not chosen yet. This is the way in which we keep our options open.

After performing these three steps, we have the new system

\[
x_{k+1} = \tilde{f}(x_k, u_{ss} + \tilde{u}_k) = \tilde{f}(x_k, \tilde{u}_k)
\]

with state \( x \) and input \( \tilde{u} \). Now, the unforced new system looks like

\[
x_{k+1} = \tilde{f}(x_k, 0)
\]

which has \( x_{ss} \) as an equilibrium point.

Our idea now is to find the feedback \( \tilde{u}_k \) that makes the equilibrium stable. Let us illustrate the basic idea of making this equilibrium stable. The controller

\[
\tilde{u}_k = K(x_{ss} - x_k)
\]

is called a P (P for proportional) controller, because the control action is proportional (with constant \( K \)) to the deviation of the state \( x_k \) from the desired equilibrium \( x_{ss} \).

The idea is that we can tweak \( K \) to make the desired equilibrium stable! Let’s see how this works.

**Task 5.4.1** Substitute (5.7) into equation (5.6). What is the expression that you get?

After having done Task 5.4.1, you will have a system that looks like

\[
x_{k+1} = \tilde{f}(x_k, K(x_{ss} - x_k))
\]

For simplicity, let us denote \( g(x_k) = \tilde{f}(x_k, K(x_{ss} - x_k)) \). Then, our system simply looks like \( x_{k+1} = g(x_k) \). We know that \( g(x_{ss}) = x_{ss} \). Now, what do we know about the stability of this equilibrium point? Well, we can use what we learned in Lecture 3 to figure it out. Basically, we have to resort to computing the derivative of \( g \) at \( x_{ss} \) and see if it is larger than 1 (unstable) or smaller than 1 (stable). Let’s do it. To compute the derivative of \( g \), we have to use the chain rule. Don’t worry if you do not know about these things. You can just assume that the result obtained in the next equation is correct (but do not forget to come back to these notes in the future once you
5.4. **SHAPING THE DYNAMICS: MAKING THE EQUILIBRIUM STABLE**

learn how to compute partial derivatives!). We have that the derivative of $g$ is

$$g'(x_{ss}) = \frac{\partial f}{\partial x}(x_{ss}, 0) - \frac{\partial f}{\partial u}(x_{ss}, 0) \cdot K$$  \hspace{1cm} (5.8)

Let’s use this in an example.

**Example 5.4.2** Consider the system of Task 5.3.3. We have obtained the control $u_{ss} = -2$ that makes $x_{ss} = 2$ an equilibrium of the system $x_{k+1} = x_k^2 + u_k$. Choosing $u_k = u_{ss} + \tilde{u}_k = -2 + \tilde{u}_k$, we obtain the system

$$x_{k+1} = \tilde{f}(x_k, \tilde{u}_k) = x_k^2 - 2 + \tilde{u}_k$$

Note that the unforced system, $x_{k+1} = x_k^2 - 2$ has an unstable equilibrium at $x_{ss} = 2$. Following (5.7), we choose $\tilde{u}_k = K(x_{ss} - x_k)$ to get

$$x_{k+1} = g(x_k) = x_k^2 - 2 + K(2 - x_k)$$

From equation (5.8), we compute $g'(x_{ss}) = 2x_{ss} - K = 4 - K$. Therefore, if we choose $K$ such that $3 < K < 5$, we get $|g'(x_{ss})| < 1$ and $x_{ss}$ is stable.

**Task 5.4.3** Pick $K$ between 3 and 5, and plot various orbits starting close to the equilibrium point 2 of the system

$$x_{k+1} = x_k^2 - 2 + K(2 - x_k)$$

You should get orbits that look like the ones in Figure 5.2. What difference do you observe with respect to the plot in Task 5.3.3?

Example 5.4.2 reveals the magic of feedback. By appropriately choosing the constant $K$, we can make stable an otherwise unstable equilibrium! How to choose $K$ is determined by equation (5.8): you should pick $K$ such that $|g'(x_k)| < 1$. 
Figure 5.2: Orbits starting from $x_0 = 2.5$ (left) and $x_0 = 1.6$ (right) of the system $x_{k+1} = x_k^2 - 2 + 4(2 - x_k)$. 
5.5 Homework
Chapter 6

Feedback control of the Robobrain robot

In Chapter 3, we introduced the Robobrain model, for which you calculated equilibrium points, and plotted orbits in Matlab, assuming constant translational and rotational speeds. In this chapter, we implement a wall-following control, so that the robot will follow a wall around the room while staying at a desired separation distance. This means that the speeds are not constant, but a function of the actual position and orientation of the robot (remember the cruise controller equation). This requires feedback, and while this is easy in simulations with the model, it requires the use of sensors on the real robot.

For feedback purposes, the measured sensor distances must be converted into position and orientation values using geometry.

To see the connection between the simulations and real-life, you will implement the wall-following control, in Matlab to compute predicted orbits, and on the real robot to compute actual orbits. For a properly designed controller, you will see that these orbits are in fact reasonably similar. Thus, feedback control makes the mathematics (discrete-time dynamic equations) consistent with real-life, which is a beautiful thing!
6.1 Chapter objectives

This chapter assumes you have studied the material and completed the homework for Chapter 3. When we are finished covering this lecture, you should be able to:

1. **Identify** all of the variables (positions, orientation, control inputs) and parameters (update time) for the model of our robot, and be able to explain each of the variables to a non-expert, such as a non-engineer family member.

2. **Describe** the wall-following control objective to a non-expert. Relate the heading of the robot, relative to the wall, and the distance from the wall, to the control objective. Explain in a general way how the sensors give you this information.

3. **Define** the wall-following control objective mathematically.

4. **Show** that for the \(x\) and \(\theta\) equations in the robot model, the feedback wall-following control results in only ONE fixed point, which is precisely the desired fixed point to achieve the wall-following control objective.

5. **Explain** how the dynamic equations of the robot are used to design and test a wall-following control, which eventually gets run on the real robot.

6.2 Robot dynamic model

Examine again the schematic image of an enlarged Robobrain robot sitting the middle of a room, given in Figure 6.1. From previous discussions, recall that \(\Delta\) is the sample period and denote the sample times as \(t_k = k \times \Delta\), \(k = 1, 2, \ldots\). So, the time instants at which we COMPUTE the position and orientation of the robot are \((t_1, t_2, \ldots) = (0, \Delta, 2\Delta, \ldots)\). Also, \(x_k\) is the \(x\) position of the robot at time \(t_k\), using the same subscript notation for the other variables. In other words, \((x_1, x_2, \ldots)\) are the positions of the robot in the \(x\) direction at times \((t_1, t_2, \ldots)\), \((y_1, y_2, \ldots)\) are the positions of the robot in the \(y\) direction at times \((t_1, t_2, \ldots)\), and \((\theta_1, \theta_2, \ldots)\) are the orientations of the robot in the \(\theta\) direction at times \((t_1, t_2, \ldots)\).
6.2. ROBOT DYNAMIC MODEL

Figure 6.1: Schematic of an enlarged Robobrain robot operating in a room. The border in the drawing represents the room walls, and the coordinate system shows that the lower left corner is denoted \((x, y) = (0, 0)\) and the upper right corner is denoted \((x, y) = (X, Y)\), with \(X\) and \(Y\) given by the dimensions of the room.

Recall from Chapter 4 that the discrete-time dynamic model of the robot dynamics is given by

\[
\begin{align*}
    x_{k+1} &= x_k + \Delta u_k \cos(\theta_k) \\
    y_{k+1} &= y_k + \Delta u_k \sin(\theta_k) \\
    \theta_{k+1} &= \theta_k + \Delta v_k
\end{align*}
\]  

(6.1)

The translational velocity \(u_k\) (the rate of moving forward or backward) and the rotational velocity \(v_k\), at time instant \(t_k\), are defined from Chapter 4 in terms of the commanded angular velocities \((\omega_L(t_k), \omega_R(t_k))\) at time \(t_k\). The equation is given by

\[
u_k = \frac{r}{2}(\omega_R(t_k) + \omega_L(t_k)), \quad v_k = \frac{2\pi r}{w}(\omega_R(t_k) - \omega_L(t_k))\]

(6.2)

Since we can control \((\omega_L(t_k), \omega_R(t_k))\) independently at each time \(t_k\), we can directly control the independent translational and rotational velocities \((u_k, v_k)\) at any time \(t_k\). Therefore, we refer to \(u_k\) and \(v_k\) as the controls, or the control inputs.

Now, we will be designing our control using \(u_k\) and \(v_k\). However, in the code that runs on the robot, you are required to specify the commanded motor angular velocities \((\omega_L(t_k), \omega_R(t_k))\)!! Therefore, we need a way to convert \(u_k\) and \(v_k\) into \((\omega_L(t_k), \omega_R(t_k))\). As with many other exercises in this class, we can do this conversion using algebra.
Task 6.2.1 From equation (6.2), suppose that \( u_k \) and \( v_k \) are known and that you want to calculate \( (\omega_L(t_k), \omega_R(t_k)) \). Using algebra, show that

\[
\omega_R(t_k) = \frac{u_k}{r} + \frac{w v_k}{4\pi r}, \quad \omega_L(t_k) = \frac{u_k}{r} - \frac{w v_k}{4\pi r}.
\] (6.3)

So, once you have a control equation that computes \( u_k \) and \( v_k \) at any time \( t_k \), you can use the equations above to calculate \( \omega_L(t_k) \) and \( \omega_R(t_k) \), which are included in the code running on the robot as the commanded angular speed instructions to each motor.

### 6.3 Big picture

Remember that the robot model is a way to predict how the robot evolves in time, given our chosen commanded speeds \( u_k \) and \( v_k \). Next, we’ll propose a wall-following control objective, and feedback control equations for \( u_k \) and \( v_k \). We’ll then use the model to again predict how the feedback controlled robot evolves in time. Remember this is what you did for the cruise controlled car in the last group homework assignment. While you used some analysis to pick control parameters for the cruise controller, the wall-following control parameters will be chosen by: plotting orbits for different parameter values, and deciding on the best values based on the orbits that display the best wall-following response.

The real usefulness of calculating predicted responses with a model is that we can tweek our control parameters until the predicted orbits indicate good wall-following performance by the robot. We can do this without consuming our robots batteries!

After we’ve chosen the control parameters, then comes the acid test - if we apply the same control on the real robot, do the predicted orbits look like the real orbits? If the model was perfect, and you calculated the orbit starting from where the real robot is starting, then the two orbits would be identical. However, NO model is perfect. At best, the two orbits will be close.

So, let’s get on with calculating predicted orbits, followed by real experiments to see if our controller actually works!

### 6.4 Wall-following control - simulation

The control objective is for \((x_a, \theta_a) = (d_{sep}, \pi/2)\) to be a stable and attractive equilibrium point, while the robot moves at a constant speed \( u_{nom} \).
in the $y$ direction.

From Figure 6.1, remember that the actual robot is MUCH smaller than the room, and the objective means we’ll want the robot to be moving vertically, at a constant speed, along the dashed vertical line shown in the figure. The schematic in Figure 6.2 shows this configuration. Now, let’s begin to design a controller to meet the objective.

The wall following control is given by

$$u_k = u_{\text{nom}}, \quad v_k = k_p (x_k - d_{\text{sep}}) + k_d \frac{x_k - x_{k-1}}{\Delta}, \quad (6.4)$$

where $k_p \geq 0$ and $k_d \geq 0$ are the control parameters to be chosen. As with the cruise control objective from the group homework problem of Chapter 4, where the goal was to make the desired speed a stable and attractive equilibrium point, we must FIRST determine if $(x_*, \theta_*) = (d_{\text{sep}}, \pi/2)$ is indeed an equilibrium point. THEN, we can calculate orbits in Matlab to determine if this is a stable and attractive equilibrium point.

**Group Task 6.4.1** Consider the model (6.1) with the feedback controller (6.4) implemented for the commanded speeds. For the $x$ and $\theta$ equations,
show that \((x_*, \theta_*) = (d_{sep}, \pi/2)\) is the ONLY equilibrium point. Also, when \(x\) and \(\theta\) are at equilibrium, determine if the resulting dynamic equation for the position \(y\) is consistent with the control objective for \(y\).

OK - you’ve determined that \((x_*, \theta_*) = (d_{sep}, \pi/2)\) is indeed an equilibrium point. Now, we can calculate orbits in Matlab to determine if this is a stable and attractive equilibrium point. Of course, whether or not we get stability and attractivity will depend on our choices for \(k_p\) and \(k_d\).

**Group Task 6.4.2** Modify the file `Robot.m` that you created in Chapter 4. Rename the file and function to be `robobrainControl.m`. The purpose of this function is to create orbits for the model (6.1) with the feedback controller (6.4) implemented for the commanded speeds. This new function should have the following input parameters: the initial values \((x_1, y_1, \theta_1)\), the time step \(\Delta\), the number of iterations \(N\), the desired separation distance \(d_{sep}\) and the control gains \(k_p, k_d\). The function should output the three orbits \(x, y, \theta\), and the elapsed time \(t_N = N\Delta\). **Hint:** The first line of the file should be something like

```matlab
function [x,y,theta,time] = robobrainControl(x1,y1,theta1,delta,N,kp,kd,dsep)
```

**Note:** Unlike the other recent function files you’ve created, the output is NOT a plot; the output is the three orbit vectors and an elapsed time value. The reason for this output format will become clear in Group Task 6.4.4.

The output of your `robobrainControl.m` function should be orbits for \(x, y\) and \(\theta\), which you can then plot in the Matlab workspace.

**Group Task 6.4.3** Using your `robobrainControl.m` file, generate a plot of \(y\) vs. \(x\) using the same parameters defined in the caption of Figure 6.3.

**Group Task 6.4.4** Go to the course web page `www.soe.ucsc.edu/classes/cmpe008/Fall06` download the file `robot_movie.m` and save it in your working directory. In order for this program to work, you need to have `robobrainControl.m` working correctly. The function `robot_movie.m` outputs a visualization of the orbits. Type `help robot_movie.m` to understand the syntax and use the program to get cool movies of Robobrain working. Modify the control parameters \(k_p\) and \(k_d\) to see how they affect stability and attractivity of the equilibrium point \((x_*, \theta_*) = (d_{sep}, \pi/2)\). Through that process, identify what you think are the best choices for \(k_p\) and \(k_d\).
Figure 6.3: Simulation of feedback control for keeping a desired separation from the wall at a constant speed. Initial condition is \((x_1, y_1, \theta_1) = (3, -2, 0)\), nominal wall following velocity is \(u_{\text{nom}} = 1\) and desired separation is \(d_{\text{sep}} = 1\). Control gains are \(k_p = 1/3\) and \(k_d = 1\), with sample period \(\Delta = 0.05\) and \(N = 200\) iterations. The black line is the \(y\) vs. \(x\) orbit, the red line shows \(d_{\text{sep}}\) and the black rectangle is the wall. The blue graphic depicts the initial location of the robot.
An example simulation from `robot_movie.m` is shown in Figure 6.3. Now that we have a controller that works in simulation, let’s get on with the experiments!

### 6.5 Wall-following control - experimentation

To meet the control objective, the robot must FIRST find a wall, then turn parallel to it, before the wall tracking controller can be of any use. This means we need an *initialization* phase of our controller, and this is precisely the use for the front sensor.

By now you have calibrated your infrared (IR) sensors, which means you have coefficients that can be used to *convert the RAW DATA from each sensor into DISTANCE* (measured in centimeters). From IR sensors, the measured distances to the wall at any time $t_k$ are denoted as follows:

- $d_f(t_k)$ – measured from the front,
- $d_{l,f}(t_k)$ – measured from the left front,
- $d_{l,m}(t_k)$ – measured from the left middle, and
- $d_{l,b}(t_k)$ – measured from the left back.

The distances are shown pictorially in Figure 6.4.

![Figure 6.4: Schematic of robobrain sensing a nearby wall, where the triangles depict the IR sensors used to determine distances to the wall.](image)

The initialization phase of the algorithm, in which the robot first finds a wall, and then turns parallel to it, is now defined.
Name: Robobrain sensor initialization algorithm
Goal: Initialize position of Robobrain to follow a wall

1: Go forward with control $u = u_{\text{nom}}$ and $v = 0$ until $d_f \approx 2d_{\text{sep}}$.
2: Turn clockwise slowly in place with control $u = 0$ and $v = -0.05u_{\text{nom}}$ until all three left sensors register wall measurements. Continue to turn slowly, keeping track of the values of the left sensors. As soon as $d_{l,b} = d_{l,m} = d_{l,f}$, stop.
3: Reset time to be $t_1 = 0$ and define $\theta = \pi/2$ and $x_1 = d_l(t_1) + w/2$. Note that $d_l(t_1) = d_{l,b}(t_1) = d_{l,m}(t_1) = d_{l,f}(t_1)$.

Practically, $d_{l,b} = d_{l,m} = d_{l,f}$ is impossible, since the sensors are not perfect, and neither is the sensor calibration procedure. Still, it is enough to have a condition that says “if the three sensors are equal within 0.5 cm, then proceed with step 3 of the sensor initialization algorithm.”

Now, at $t_1$, the robot is in the position $(x_1, \theta_1)$, with $\theta_1 = \pi/2$ and $x_1 = d_l(t_1) + w/2 > d_{\text{sep}}$, as shown in Figure 6.5. Note that $x_1 > d_{\text{sep}}$ because the robot stops and turns when $d_f = 2d_{\text{sep}}$. From this initial position, the goal of the wall-following control is to drive the robot to meet the stated control objective.

For feedback purposes, equation (6.4) requires the measured $x_k$. Therefore, the sensor distance measurements must be converted into position $x_k$.

Figure 6.5: Schematic of robobrain after the sensor initialization algorithm is complete, with $\theta_1 = \pi/2$ and $x_1 = d_l(t_1) + w/2$. 
CHAPTER 6. FEEDBACK CONTROL OF THE ROBObRAIN ROBOT

The following sensor update algorithm shows how the IR distance measurements are used to compute \( x_k \) at any time \( t_k \).

<table>
<thead>
<tr>
<th>Name:</th>
<th>Robobrain sensor update algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal:</td>
<td>Calculate the position ( x_k ) from the left IR sensors</td>
</tr>
</tbody>
</table>

Starting with \( k = 1 \), execute the following steps for all update times \( t_k \), \( k \geq 1 \):

1: Set \( \theta_k = \pi/2 + \alpha_k \), where

\[
\alpha_k = \frac{1}{3} \left\{ \arctan \left[ \frac{d_{l,b}(t_k) - d_{l,f}(t_k)}{l} \right] + \arctan \left[ \frac{d_{l,m}(t_k) - d_{l,f}(t_k)}{l/2} \right] + \arctan \left[ \frac{d_{l,b}(t_k) - d_{l,m}(t_k)}{l/2} \right] \right\},
\]

and \( l \) is the length of Robobrain.

2: Set

\[
x_k = \frac{1}{3} \left( d_{l,b}(t_k) + d_{l,m}(t_k) + d_{l,f}(t_k) \right) + \frac{w}{2} \cos(\alpha_k).
\]

3: Set \( k = k + 1 \), and return to step 1.

**Group Task 6.5.1** Write a program that converts the three left scalar distance measurements \( d_{l,f}, d_{l,m} \) and \( d_{l,b} \) into the state values \( x \) and \( \theta \) using equations (6.5) and (6.6). Call the program `sensorupdate.m`.

Note that \( \alpha_k \) is the deviation of \( \theta_k \) from \( \pi/2 \), so when you’ve met the control objective, \( \alpha_k = 0 \). Also, observe that \( \alpha_k \) is needed to compute \( x_k \). To summarize, from the algorithm above, we can convert IR sensor distances into the position \( x_k \), which can in turn be used for feedback in equation (6.4). Lastly, the feedback in equation (6.4) is converted into commanded angular speeds for the motors using equation (6.3). A block diagram schematic showing the relationship between all of these calculations is show in Figure 6.6.

Why is it called closed-loop feedback control? By driving the motors, which moves the robot to a new position, we are in turn affecting the IR sensor distance readings, which in turn affect \( x \) and \( \alpha \), which defines our commanded \( u \) and \( v \), and therefore our commanded motor speeds. The
new speeds then move the robot to a new position, and the cycle is repeated at each sample time $t_k$. The arrows in the schematic show this affect forming a closed-loop, which means no matter where you start in the logic loop, you always end up returning to that point.
6.6 Homework

Make sure that you can perform each of the chapter objectives defined in Section 6.1. In addition, turn in the problems below. Problems C4p1–C4p3 are to be done by everyone individually. There is no group problem, since teams will be working toward the final project, which involves implementing the logic in Figure 6.6 on the real robots. Details regarding the final project requirements will be handed out in class.

1. Turn in your own derivations for Group Task 6.4.1.

2. Turn in your plot from Group Task 6.4.3, and a copy of your robobrainControl.m file.

3. Turn in your plot from Group Task 6.4.4, and include your choices for $k_p$ and $k_d$. State whether or not you’ve met the control objective.