MODEL-BASED VISUAL TRACKING VIA MAPLE CODE GENERATION

Model-Based Visual Tracking via Maple Code Generation

By
Alexandre O. Korobkine, B.Eng

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.A.Sc
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2004) McMaster University
CAS Hamilton, Ontario

TITLE: Model-Based Visual Tracking via Maple Code Generation

AUTHOR: Alexandre O. Korobkine, B.Eng(McMaster University)

SUPERVISORS: Dr. Christopher Anand & Dr. Mark Lawford

NUMBER OF PAGES: x, 39

Abstract

Many algorithms, particularly in the area of image processing, are expensive to develop and computationally resource intensive. We illustrate the advantages of symbolic code generation using an example – closed-loop visual target recognition and tracking in extreme lighting conditions. We quantify the effect of symbolic code generation methods on code efficiency, and explain how these methods allowed us to reduce the development time as well as improve reliability. Working directly with symbolic models improves software quality by reducing transcription errors, and enabled us to rapidly prototype different models for the visual tracking application, where the need to evaluate trackers in their real-time context precludes the effective use of scripting languages. We describe the model in detail, including formulations as an optimization problem; explain the challenges in solving the model; present our method of building the solvers; and summarize the impact on the performance of our methods.

Acknowledgments

I would like to thank my supervisors, family, and friends for their support.

Contents

List of Figures

List of Tables

List of Symbols

$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$	Vector of "converted colours"	Ch. 4, page 20
$\begin{pmatrix} r_{real} \\ g_{real} \\ b_{real} \end{pmatrix}$	Vector of the colour values from the original image	Ch. 4, page 20
$\mathcal{S}$	Set of saturated pixels	Ch. 4, page 26
$\mathcal{V}$	Set $\{v_0, v_1, k_1, k_2, b_x, b_y, a_1, a_2, a_3\}$	Ch. 4, page 27
Ω	Region of pixels containing a light spot	Ch. 4, page 21
ϕ	Array storing the image	Ch. 4, page 21
θ_i	Angles of rotation of the spot around the i-th axis	Ch. 4, page 24
A	Matrix of "real colours"	Ch. 4, page 20
a	Matrix $\begin{pmatrix} a_1 & a_2 \\ a_2 & a_3 \end{pmatrix}$	Ch. 4, page 22
a_1,a_2,a_3	Extent and eccentricity coefficients of an ellipse	Ch. 4, page 21
b_x,b_y	x and y coordinates of the ellipse centre	Ch. 4, page 21
F	Fit of the model to the actual light intensity	Ch. 4, page 23
f	Function describing basic model of a spot	Ch. 4, page 21

$H_{\mathcal{U}}$	Hessian of a function with respect to variables $\mathcal{U}$	Ch. 2, page 15
$J_{\mathcal{U}}$	Jacobian of a function with respect to variables $\mathcal{U}$	Ch. 2, page 15
k_0,k_1,k_2,k_3	Radial profile coefficients of an ellipse	Ch. 4, page 21
l	Light spot	Ch. 4, page 21
p	Position and orientation of a spot in 3D	Ch. 4, page 24
P_i	Space coordinates of the centre of the spot; $i \in \{x, y, z\}$	Ch. 4, page 24
s	Function describing an ellipse of the general form	Ch. 4, page 21
u_n	Value of a vector $\mathcal{U}$ on an n^{th} step	Ch. 2, page 15
v_0	Background illumination of an image	Ch. 4, page 22
v_1	Brightness of the centre of the spot	Ch. 4, page 22

Chapter 1

Introduction

1.1 Motivation

Model fitting is a common subproblem in target recognition and can be described as follows: Given a set of data points, find a model that approximates the data best. Automatic target recognition is an application of computer vision to identify targets, such as tanks, airplanes or people in an image. The recognition procedure involves obtaining essential features from each local area in the image and comparing them to the templates of known targets. We present a target recognition application which is typical of model fitting:

- it can be formulated as nonlinear optimization problem;
- the objective function is sums over the indices of a large array;
- efficiency depends primarily on efficient memory access patterns.

1.1.1 Visual Tracking

In visual tracking applications, a series of images captured from CCD cameras must be processed in real-time to extract information about spatial positioning. This information can be used for target identification, object measurement, and closed-loop target acquisition. In some applications, the target has a high degree of regularity, and can be modeled mathematically or, as in the present case, can be designed to

optimize a predefined objective function. To compensate for harsh, dynamic lighting conditions, we consider the use of multi-colour, multi-brightness patterns, which would provide quantitative information about lighting even for saturated images. To work at different scales, we expect successful patterns to be smoothly varying, thus we restrict our search to piecewise polynomial and rational polynomial patterns. The recognition of such patterns can be modeled as a constrained, nonlinear optimization problem. Recognition can be implemented as a solver, which in addition to estimating model parameters, can assign a likelihood (the probability of a specified outcome) to the estimates. Advanced model-based controllers make use of the likelihood information to improve the robustness of the controller in the presence of random and systematic noise sources.

1.1.2 Satellite Acquisition

The intended application of the tracking software is remote, unassisted satellite acquisition. If one wishes to capture a satellite and perform maintenance on it, a camera mounted on a robotic arm could detect and track a predefined pattern on a satellite. Sudden changes of lighting, possibly saturating a significant part of the image, and complete or partial "blindness" caused by sunlight on the camera lens, pose significant obstacles to any algorithm. To overcome them, we propose the use of a model-predictive controller which incorporates confidence information derived in parallel with position estimates by our solver, as well as frame-by-frame illumination estimates to be used to control camera gain. In the constrained engineering environment of space, and the impossibility of direct human intervention, it is essential that the solver be able to extract position information from tens of images per second with resources significantly less than on a current desktop computer.

1.1.3 Rapid Prototyping and Code Generation

Prototyping is a process of creating pre-production models of a product to test various aspects of its design. The use of Rapid Prototyping (RP) technique allows us the quick production of prototypes. The quick production of prototypes was required for the purpose of determining which model will suit our application best. For example, RP would let us test the design for the tradeoffs between accuracy and efficiency. Some

models may lead to very accurate fitting, but require significant processing time. Other models may have the opposite properties.

The employment of Maple code generation reduces time for design, implementation and testing phases for each underlying model change. Modifying C functions by hand would be time-consuming, and using a mathematical scripting language would not allow testing of the controller.

1.2 Previous / Related Work

In this section we provide an overview of the two main research areas related to this thesis: (i) visual tracking and (ii) code generation.

1.2.1 Visual Tracking

This section provides background information on existing methods to perform visual tracking. The topic of visual tracking spans many different disciplines, therefore our goal is to provide only a basic conceptual framework.

Vision is a very useful robotic sensor, since it allows measurement of the environment without physical contact. Because a visual feedback loop can be used to correct the position of a robotic arm to increase the accuracy of a task, considerable effort has been devoted to the design of visual control of robot manipulators. In the past, visual sensing and manipulation were combined in an open-loop fashion,"looking" then "moving". The accuracy of the operations depends directly on the accuracy of the visual sensor and the robot end-effector. A visual feedback control loop will increase the overall accuracy of the system (Figure 1.1). In the perfect scenario, machine vision can provide closed-loop position control for a robot end-effector — this is referred to as "visual servoing" [9].

A taxonomy of visual servoing systems was introduced in 1980 by Sanderson and Weiss [14], into which all subsequent visual servo systems can be categorized. Their scheme can be described by asking the two following questions:

1. Does the control system have hierarchical structure, with the vision system providing set-points as input to the robot's joint-level controller, or does the visual controller directly compute the joint-level inputs?

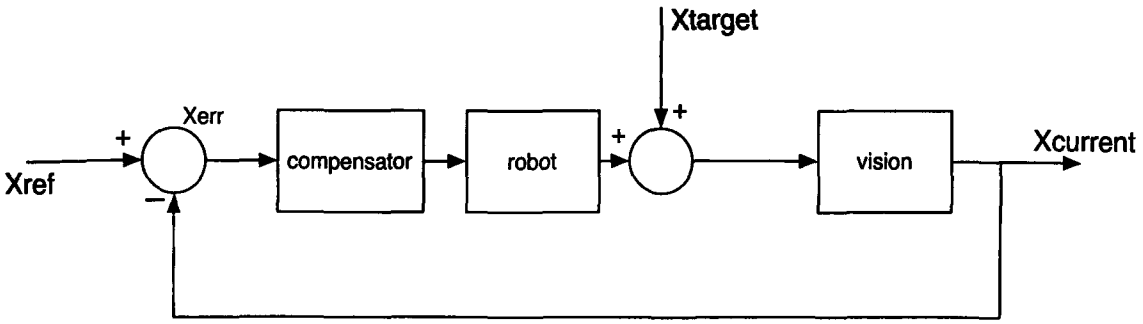

Figure 1.1: Visual feedback control loop.

2. Is the error signal defined in 3D coordinates or directly in terms of image features?

Therefore, the resulting taxonomy has four major categories, which we now describe.

If the control architecture has underlying hierarchical structure and uses vision to provide set-points to the joint-level controller, therefore making use of feedback to internally stabilize the robot, it is referred to as a "dynamic look-and-move" system. In contrast, "direct visual servoing" entirely eliminates the robot controller by replacing it with a visual servo controller that directly computes joint inputs, therefore using vision alone to stabilize the mechanism.

Almost all implemented systems follow the dynamic look-and-move approach for the following reasons [9]:

1. Relatively low sampling rates available from vision make direct control of a robot end-effector with complex, non-linear dynamics an extremely challenging control problem.

2. Many robots already have an interface for accepting incremental position commands, which simplifies the construction of the visual servo system, and makes the methods more portable.

3. Look-and move approach separates the kinematic singularities of the mechanism from the visual controller, allowing the robot to be considered as an ideal motion device.

The second major classification of systems distinguishes "position-based" control from "image-based" control. In "position-based" control, extraction of features from the image is performed and features are used together with a geometric model of the target and the known camera model to estimate the position of the target with respect to the camera. In "image-based" servoing, control values are computed on the basis of image features directly. The image-based approach may lead to reduction of computational delay, elimination of the necessity for image interpretation, and elimination of errors due to sensor modeling and camera calibration. However, it presents a significant challenge to controller design, due to the plant being non-linear and highly coupled.

As it was mentioned earlier, in position-based visual servoing, features are extracted from the image and used to estimate the position of the target with respect to the camera. Using these values, an error between the current and the desired position of the robot is defined. A positioning task is fulfilled if the error is brought to zero. Position-based control efficiently separates the control issues – the computation of the feedback signal – from the estimation problems involved in computing the position from visual data.

In image-based visual servo control the error is defined directly in terms of image feature parameters. As with position-based control, a task is achieved, when the error is equal to zero. This is usually resolved by the approach in which the robot is moved to a goal position and the corresponding image is used to compute a vector of desired image feature parameters. Although the error is defined in the image parameter space, the control input is defined either in joint coordinates or in task space coordinates. Therefore, there is a necessity to relate changes in the image feature parameters to changes in the position of the robotic arm. The disadvantage of image-based methods comes from the presence of singularities in this mapping. A singularity is a point at which an equation, surface, or mapping in this case, "blows up" or becomes degenerate.

Independent of the control approach used, a vision system has to extract the information needed to perform the servoing task. Hence, visual servoing infers the solution to a set of difficult static and dynamic vision problems. Many implementations simplify the vision problem by: painting objects white or using artificial targets. Others use motion detection for locating a moving object to be grasped.

In less methodical situations, vision has typically relied on the extraction of sharp contrast changes, referred as "corners" or "edges", to indicate the presence of object boundaries or surface marks in an image. Processing of the entire image to extract these features would require the use of extremely high-speed hardware in order to work with a sequence of images at camera rate. However, not all pixels in the image are contributing, therefore a significant reduction in computation time can be achieved if only a small region around each image feature is processed. A promising technique for making vision less computationally intensive is to use "window-based" tracking methods. These methods have several advantages among them: little requirement for special hardware and computational simplicity; however initial positioning for each window typically pre-supposes a solution to a potentially complex vision problem. Window-based tracking algorithms are usually carried out in two stages. First, one or more windows are acquired using a nominal set of window parameters. In the second stage, the windows are processed to locate image features and from their parameters a new set of windows parameters is computed. A disadvantage of this technique is that this scheme is susceptible to mistracking caused by background and/or foreground obstructing edges. Also, large windows increase the range of motions that can be tracked, but greatly reduce the tracking speed and increase the likelihood that a distracting edge will cause the disruption of tracking.

In many realistic cases, neither of the above mentioned approaches by themselves yield the robustness and desired performance. To illustrate the tradeoffs between these methods, we will use an example from [9]. Suppose a visual servoing task relies on tracking the image of a circular opening over time. In general, the opening will project as an ellipse in the camera. There are several algorithms for detecting this ellipse and recovering its parameters:

1. If there is a high contrast between the interior of the "opening" and the area around it, binary thresholding, which is an imaging technique that labels all gray pixels as either black or white before processing begins, followed by a calculation of the first and second central moments can be used to localize the feature.

2. If the ambient illumination changes greatly over time, but the brightness of the "opening" and of the surrounding region are roughly constant, a circular

template could be localized using sum of square differences (SSD) methods augmented with brightness and contrast parameters.

3. The "opening" could be selected in an initial image, and located using SSD methods. This differs from (2) in that only correlation of the center of the "opening" with the starting image is found, and not the actual location of the center. This is useful for servoing a camera to maintain the opening within the view; this approach is not useful for a manipulation tasks that need to calculate a position relative to the centre of the "opening".

4. If there are constant changes in contrast and background, the "opening" could be tracked by performing edge detection and fitting an ellipse to the edge location. Short edge segments could be located using the feature-based methods. Once the segments have been fit to an ellipse, the orientation and location of the segments would be adjusted for the tracking using geometry of the ellipse.

One of the most important issues in visual-based control is the choice between using an image-based or position-based system. Most systems dealing with moving objects are built on position-based methods. In contrast, the accuracy of image-based methods for static positioning is less sensitive to calibration than comparable position-based methods.

In order for a visual-servo system to provide good tracking performance for moving targets, attention must be paid to the modeling of the dynamics of the robot, the target, and the vision system. Other issues for consideration include whether or not vision system should "close the loop" around robot axes which can be controlled by position, velocity or torque.

The remaining part of this subsection will provide a brief introduction to two of the widely used visual processing tools.

MATLAB's Image Processing Toolbox provides a comprehensive set of reference-standard algorithms and graphical tools for image processing, analysis, visualization, and algorithm development. The toolbox consists of statistical functions, edge-detection algorithms, image segmentation algorithms, and morphological operators. User can restore noisy or degraded images, enhance images for improved intelligibility, extract features, analyze shapes and textures, and register two images. Most toolbox

functions are written in the open MATLAB language, which gives user the freedom to inspect the algorithms, modify the source code, and create custom functions.

National Instruments vision development software includes IMAQ Vision, a library of vision functions, and IMAQ Vision Builder, an interactive environment for prototyping vision applications. IMAQ Vision provides the use of gauging, caliper, measurement, and edge-detection functions to automatically locate edges and to measure distance and angles between edges, points, and parts. Other functions include blob analysis, morphology, image enhancement and quantitative analysis. IMAQ Vision Builder and IMAQ Vision work closely together to simplify vision software development so that the developer can apply vision in measurement and automation applications.

It should be noted that the goal of the thesis was not to focus on the existing visual processing algorithms such as edge-detection, but rather to illustrate the advantages of the symbolic code generation by solving the model-based visual tracking problem.

1.2.2 Code Generation

The text in this subsection is excerpted from the paper "Visual Tracking Employing Maple Code Generation" [1]. The paper was coauthored by Christopher Anand, Jacques Carette, and myself and presented at the Maple Summer Workshop 2004.

Code generation is now ubiquitous (over 500,000 hits for a search on "code generation" on Google as of the writing of this thesis). It is present in various ways inside most serious development environments and frameworks. It is also quite old, with code generators already showing up in the early 1960s (e.g., [7, 13, 16]), as the leap from writing a compiler to writing a code generator is quite small.

There are at least two circumstances in which code generation has proven to be effective:

1. when complex program transformations are needed [8, 12],

2. when a program can be expressed very succinctly in a domain-specific language, but requires lengthy and sometimes very complex code in mainstream languages [4, 5].

The first situation occurs most famously when automatic differentiation [10] is both

required and applicable. There is now ample literature ([15, 16]) that shows that smooth optimization problems are incomparably easier to solve when Jacobians and Hessians are available; on large problems of real interest, however, the functions to differentiate are usually given by very large programs with a multitude of inputs. Computing derivatives numerically is well-known to be a futile task, and computing them by hand (symbolically) is also fraught with error. On the other hand, symbolic differentiation is a relatively simple program.

The second situation is now becoming common as well, though the growing popularity of GUI-builders, lexer and parser generators, Java-from-DTD builders, and so on. These code generation techniques are also related to the emerging fields of "software synthesis" and "automated software engineering".

The chosen method to solve target recognition problem examined in this thesis is particularly well-suited to being tackled by code generation techniques:

1. it requires automatic differentiation,

2. it can be succinctly described using mathematics as the "domain language", and

3. it requires experimentation at the "model" level.

1.3 Methodology and Contributions of the Thesis

1.3.1 Methodology

The following subsection summarizes the methodology introduced in the thesis.

At a minimum, whatever target we choose, we must be able to recognize its translates under affine and perspective transformations. Ideally we would like to be able to robustly identify the transformation between the identified pattern and a base family member, giving us information on the relative position of the target. In this thesis we will focus on a simple family of radially-symmetric, essentially compact targets, which we we will call *spots*. Transformed spots will have elliptical equiradiant contours. Given a target image, we estimate parameters for the position (translation), size (scaling), orientation and asymmetry (rotation and pitch).

The advantages obtained by careful mathematical modeling and optimization-based parameter extraction can easily be overwhelmed by development time, and sometimes by computational costs as well. The expense comes from two sources: freedom in choosing the target pattern/model and the need to develop multi-stage solvers to achieve convergence requirements for the highly nonlinear models. The present method of generating a family of efficient Newton solvers from any target model efficiently solves this problem. By generation of Newton solvers, we mean the generation of optimized Jacobian and Hessian matrices for different sets of parameters, which are used in a basic Newton method iteration. The code generator described in this thesis currently generates solvers for 1D and 2D models, but could easily be generalized to higher dimensions.

In model fitting against images, data is stored in large arrays. Calculation of the sum over elements in arrays is an expensive procedure. Efficient use of the cache is required to minimize the execution time, which will be bounded by memory accesses. The easiest way to ensure this is to group all accesses to one pixel of data (within a solver iteration) together. Jacobian and Hessian matrices will contain many common subexpressions, therefore optimization on "the inner sum" is crucial. Since Hessian matrices are symmetric, we only need to calculate the upper triangular portion of them.

Given a family of Newton solvers (indexed by the power set of the set of model parameters), we can use heuristics or benchmarking to assemble them into a non-linear solver with good convergence properties.

1.3.2 Contributions of the Thesis

In addition to the clearly-demonstrated efficiency of the final code, symbolic code generation also allowed us to reduce development time, and improve software quality. These were really the main motivators for using symbolic code generation in the first place. Software quality for such code is largely determined by the transparency of the translation between the mathematical model and the implementation in a programming language. Symbolic code generation makes this as transparent as possible. Rapid development again depends on the time it takes to translate a model into a subroutine, which is essentially free once the framework for code translation is in

place. This was especially important for the visual tracking problem, because there is a large family of models we wanted to explore, and it would not have been practical to evaluate them without efficient implementations capable of doing model fitting in real time with the image acquisition.

To summarize, using the visual tracking example, we have demonstrated how symbolic code generation can improve software quality, code efficiency and reduce development time.

1.4 Development History

The work presented in this thesis consisted of three major development stages: design, implementation, and testing.

During the design stage, with guidance from Christopher Anand and Mark Lawford, methodology was decided. Proposed spot and light models were throughly investigated theoretically, together with 3D positioning ideas and mathematical details.

Upon initial implementation of the proposed model in Maple, Christopher Anand and myself encountered problems attempting to generate code due to Maple's pitfalls as described in Section 6. At this point Jacques Carette was consulted due to his extensive experience with Maple as both a user and former developer of the product. He was able to identify the Maple limitations causing the problems and proposed the solution, described in Section 6, to overcoming these Maple pitfalls. Maple's shortcomings, and the need for thorough testing of the generated code, demanded significant effort to achieve efficient generation of the solvers.

After the generated solvers satisfied our requirements, the QuickTime API was used to develop acquirement of images from the cameras. Sample code from "Apple Developers Connection" (`SGDataProcSample`) was used to attain initial ideas on interfacing with firewire devices. Implementation of a procedure to acquire arrays of RGB pixel values of images from the camera was carried out. Colour conversion, convergence, and saturation were the major issues that needed attention in the implementation stage. Lastly, design and implementation of the main control module, which linked the acquisition of the data from the cameras with the generated solvers, was concluded.

In the testing stage, thorough testing of the application as a whole was carried out. Various spot models were tested. Efficiency and optimization issues, as well as accuracy versus speed tradeoffs for various spot models were confirmed.

1.5 Organization of the Thesis

In Chapter 2, we give a short context for symbolic computing and code generation, and provide a short description of Newton's method (as it is used in this application area). In Chapter 3, the experimental setup is described. Chapter 4 constitutes the core of the thesis, where the models and the structure of the solvers are described in detail. The Maple code generation procedure is outlined in Chapter 5. Detailed results are presented in Chapter 6, followed by the conclusions and ideas concerning future work.

Chapter 2

Preliminaries

This chapter provides the background necessary to understand the thesis and provides pointers to more detailed sources for the interested reader.

2.1 Maple and Symbolic Computing Basics

Symbolic computation is computation with variables and constants according to the rules of analysis, algebra and other branches of mathematics – formula manipulation using symbols, unknowns, and formal operations, as opposed to conventional computer calculation using numeric data and character strings.

Consider the problem: Determine the value of $a^4 + b^4 + c^4$, when

$$\begin{aligned} a + b + c &= 6 \\ a^2 + b^2 + c^2 &= 10 \\ a^3 + b^3 + c^3 &= 25 \end{aligned}$$

Maple solves this problem via the following command, producing the result of 106:

```
> simplify (a^4+b^4+c^4, {a+b+c=6, a^2+b^2+c^2=10, a^3+b^3+c^3=25});
```

106

Furthermore, Maple solves a quadratic equation of the general form symbolically as follows:

```
> solve (a*x^2+b*x+c, x);
```

$$\frac{-b+\sqrt{b^2-4ac}}{2a}, \frac{-b-\sqrt{b^2-4ac}}{2a}$$

For a more complete introduction to Maple's use of symbolic computation refer to "Maple Help - [introduction]", which comes as a part of Maple package.

2.2 Maple and Code Generation Basics

We will demonstrate Maple code generation by a simple example:

Define a function $f = 1 - \frac{x}{2} + 3x^2 - x^3 + x^4$:

```
> with (CodeGeneration):
> res := [f := 1-x/2+3*x^2-x^3+x^4];
```

$$res := [f := 1 - \frac{x}{2} + 3x^2 - x^3 + x^4]$$

Create an optimized list of instructions to compute f:

```
> genres := codegen[optimize] (res);
```

$$genres := \quad t_2 = x^2, t_5 = t_2^2, f = 1 - \frac{x}{2} + 3t_2 - t_2 x + t_5$$

Generate C code:

```
> C ([genres], output=string, declare=[x::float]);
```

$$\begin{aligned} "t_2 &= x * x; \\ t_5 &= t_2 * t_2; \\ f &= 0.1e1 - x/0.2e1 + 0.3e1 * t_2 - t_2 * x + t_5;" \end{aligned}$$

Calling codegen with tryhard leads to generation of more efficient code:

```
> genres_hard := codegen[optimize] (res, tryhard);
```

$$genres\_hard := \quad t_1 = x^2, f = 1 - \frac{x}{2} + (3 - x + t_1)t_1$$

```
> C ([genres_hard], output=string, declare=[x::float]);
```

$$\begin{aligned} "t_1 &= x * x; \\ f &= 0.1e1 - x/0.2e1 + (0.3e1 - x + t_1) * t_1;" \end{aligned}$$

The resultant generated code to compute f with `tryhard` option has less floating point instructions and consumes less memory (less local variables) than without it.

For a more complete introduction to Maple's use of code generation refer to "Maple Help - [codegen]".

2.3 Maple Functions

In this section we briefly describe the list Maple functions used to define the models and generate the solvers.

`simplify`	"Apply simplification rules to an expression"
`with`	"Interactive package management utilities"
`solve`	"Solve equations"
`Matrix`	"Construct a Matrix"
`Transpose`	"Transpose a Matrix, Vector, or scalar"
`add`	"Add up a sequence of values"
`sum`	"Definite and indefinite summation"
`seq`	"Create a sequence"
`diff`	"Differentiation or Partial Differentiation"
`C`	"Translate Maple code to C code"
`codegen[optimize]`	"Common subexpression optimization"
`codegen[gradient]`	"Compute the Gradient of a Maple procedure"
`codegen[prep2trans]`	"Prepare a Maple procedure for translation"
`codegen[makeproc]`	"Make a Maple procedure from formulae"
`codegen[joinprocs]`	"Join the body of two Maple procedures together"

For a more information on Maple commands refer to "Maple Help".

2.4 Newton's Method

The solvers employed in the thesis are based on the multivariate Newton's method.

Let $J_{\mathcal{U}}$ be the Jacobian and $H_{\mathcal{U}}$ be the Hessian of a function with respect to the

variables $\mathcal{U}$. The Newton iteration is defined by the recursion:

$$u_{n+1} = u_n - H_{\mathcal{U}}(u_n)^{-1} J_{\mathcal{U}}(u_n),$$

which in practice is implemented by solving

$$H_{\mathcal{U}}(u_n)(u_n - u_{n+1}) = J_{\mathcal{U}}(u_n).$$

Chapter 3

Experimental Set Up

3.1 Prototype Hardware and Software

Hardware was chosen so that the clock speed of the CPU is relatively close to the upper limit of the clock speed of the CPU of a modern embedded system. Also, chosen platform was thought to simplify the development.

- System Hardware Overview
  - *Machine Model* : PowerBook G4 12"
  - *CPU Type* : PowerPC (1.1)
  - *Number Of CPUs* : 1
  - *CPU Speed* : 1 GHz
  - *L2 Cache (per CPU)* : 512 KB
  - *Memory* : 768 MB
  - *Bus Speed* : 133 MHz
- System Software Overview
  - *System Version* : Mac OS X 10.3.4 (7H63)
  - *Kernel Version* : Darwin 7.4.0

3.2 Cameras

Cameras were chosen to be colour and of low cost, with capability of supporting up to 15 frames per second.

- Cameras Overview
  - *Camera type* : Fire-i
  - *Interface* : FireWire 400 Mbps
  - *Driver* : IOXperts, Version: 1.1b22
  - *Camera type* : iSight
  - *Interface* : FireWire 400 Mbps
  - *Driver* : IOXperts, Version: 1.1b22

3.3 Quicktime

The QuickTime API was chosen to provide acquisition of images from the cameras. QuickTime is Apple's multiplatform multimedia technology for handling video, sound, animation, graphics, text, interactivity, and music. As a cross-platform technology, QuickTime can deliver content on Mac OS X, as well as all major versions of Microsoft Windows. Augmenting its cross-platform capabilities, QuickTime supports every major file format for images, and every significant professional file format for video. QuickTime 6 is used to author professional-quality, ISO-compliant MPEG-4 audio and video files.

The QuickTime API comprises more than 2500 functions that provide services to applications. These services include audio and video capture and playback, movie editing, composition, and streaming, still image display, audio-visual interactivity, and so on. The API also supports a wide range of standards-based formats. It is dedicated to extending the reach of application developers by letting them invoke the full range of QuickTime capabilities.

A further motivation for using the QuickTime API was that we were interested in experimenting with the code generators, and not low level video handling. The high level API allowed us to concentrate on the main topic of the thesis.

Chapter 4

Models and Solvers

In this chapter we describe the models and the structure of the solvers in detail, including formulations as optimization problems, and explanations of the challenges faced in solving the models.

4.1 Model

4.1.1 Colour Space

The images are formatted as arrays of unsigned 8-bit RGB values. If the gain adjustment is too high for the given lighting conditions, some of the pixels with one or more 255 component values may in reality be clipped to that value from a higher value. In the target environment, this may be quite common. Before processing, these values are usually linearly transformed to another colour space (described in Section 4.1.2) and converted to floating point numbers, so that the effect of saturation may not be restriction to a cube in $\mathbb{R}^3$ aligned to the coordinate axes. We note that in this type of application, it is common to use binary (black and white, without intermediate gray values) patterns, and simpler image processing involving statistical rounding of byte values to binary values [9].

4.1.2 Advantages of Coloured Gradients

In current practice, binary (i.e., black and white) targets are used. Gradient (i.e., grayscale or coloured) images have many advantages over binary ones. In a noisy image, every point on the gradient contributes to the determination of the position. However only the points at the transition for the binary pattern are contributing; accordingly, the usage of the gradients should be more robust with reduced quantification error. Calculation of the likelihood of estimated parameters is easier for the gradient model and can be used in robust model-based control. For suitable gradient targets, even saturated images yield information about lighting which can be used to adjust camera gain. This property is important when large lighting changes are expected.

Simple coloured patterns can be used to identify orientation of the target. Having three spots of well differentiated colours, e.g. red, blue, and green, the position and the size of these three spots yield the position and the orientation of the target. Since differences in hues are orthogonal to brightness, there is no interference between the spots. While fitting each spot is a non-convex problem, the problem of locating the centre of each spot is convex.

To simplify the exposition and the development of the algorithm, the problem of fitting the spots is decomposed into two parts: conversion to a colour space in which the different spot colours are pairwise orthogonal [including identification of the different colour values], followed by the extraction of spot parameters from a real-valued spot. Identification of the different colours must take lighting and camera calibration into account. The real-valued spot received from the camera can be either a gray-scale image or a single component of a multispectral image. We will not make a distinction.

Colour conversion is achieved as follows: given the matrix of "real colours" as

$$A = \begin{pmatrix} r_0 & r_1 & r_2 \\ g_0 & g_1 & g_2 \\ b_0 & b_1 & b_2 \end{pmatrix}$$

and the spot colours as α, β and γ, we have that

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = A^{-1} \begin{pmatrix} r_{real} \\ g_{real} \\ b_{real} \end{pmatrix},$$

where r_{real}, g_{real} and b_{real} are the colour values of each pixel from the original image. Now the vector of α, β and γ values of each pixel is processed by the algorithm.

Note that coloured images are produced by the camera with red, blue and green values interleaved, so it may not be efficient to use a solver for a grayscale model for colour spot identification.

4.1.3 Model Equations

Although part of the motivation for performing code generation in Maple was to be able to inexpensively compare multiple models and families of models, this thesis focuses on the code generation aspects; therefore only the most successful family of models is presented here, with sufficient detail to expose the issues, common to all models, with non-convexity and the structure in the space of variables we can use to get around them.

Let the two-dimensional array $\phi_{x,y} \in \mathbb{R}$ represent the stored image. If the colour information is introduced, $\phi_{x,y,c} \in \mathbb{R}$ is used, where x and y define a pixel on the image and $c \in \{r, g, b\}$ defines its colour. A model of a spot is fitted over a small region of pixels $\Omega \subset \mathbb{Z}^2$ which is believed to contain a light spot l with components x, y.

The basic model of a spot (Figure 4.1) is

$$f = k_0 + k_1 s + k_2 s^2 + k_3 s^3,$$

$$s = \begin{pmatrix} x - b_x \\ y - b_y \end{pmatrix}^t \begin{pmatrix} a_1 & a_2 \\ a_2 & a_3 \end{pmatrix} \begin{pmatrix} x - b_x \\ y - b_y \end{pmatrix}$$

where

k_0, k_1, k_2, k_3	-	determine the radial profile
a_1, a_2 and a_3	-	determine the extent and eccentricity
b_x and b_y	-	x and y coordinates (in pixels) of the ellipse centre

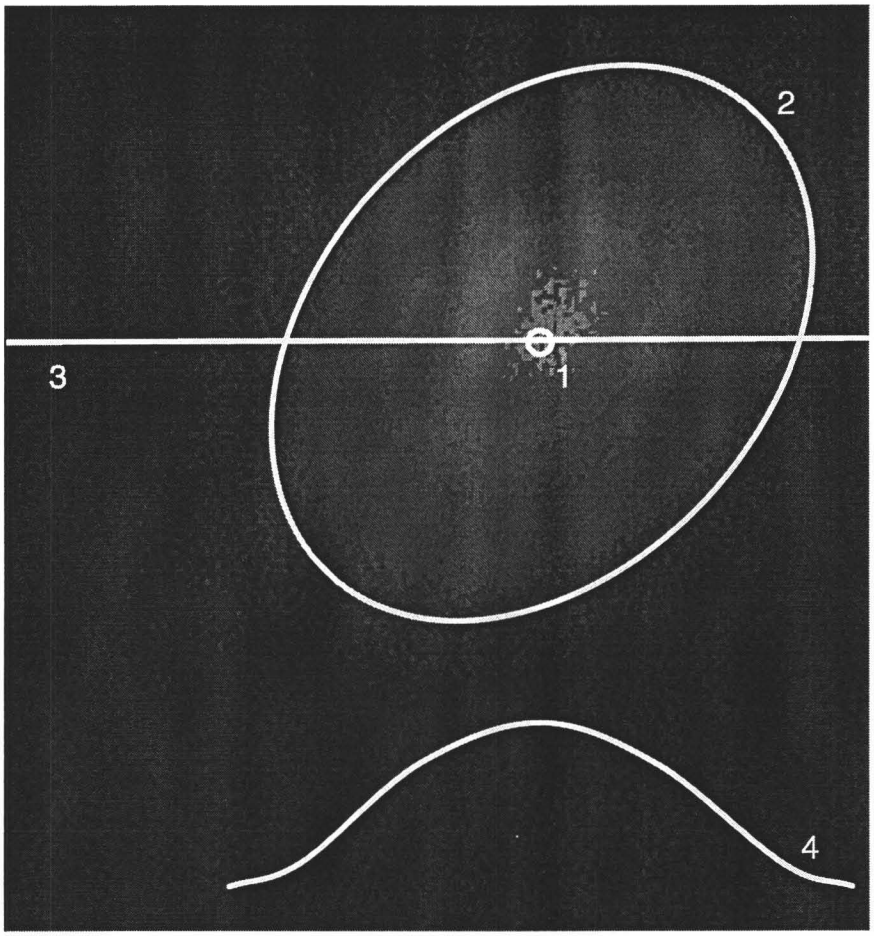

Figure 4.1: Actual captured image of one gray-scale target image.

Figure 4.1 represents an actual captured image of one gray-scale target image, with overlays to demonstrate how the model variables decompose naturally into subspaces according to their geometric meaning. The center of the spot (1) is determined by b; the shape of the spot, given by scaling/rotation is determined by a, and represented by the $s = 1$ contour (2); in the presented model, any cross-section through the centre of the spot (3) will be a stretching of the basic cross-section (4) determined by k. The background illumination is determined by v_0 and the brightness of the centre of the spot is v_1.

In addition, we add the following constraints and conventions

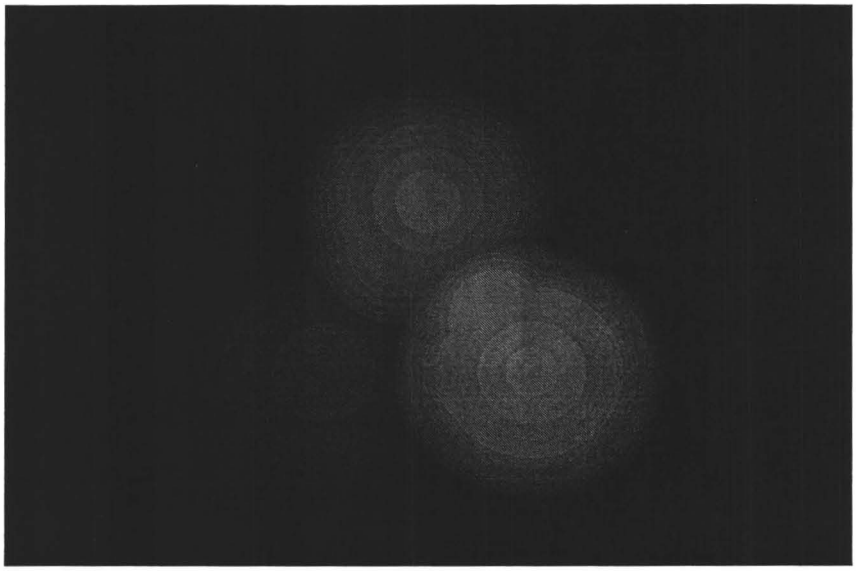

Figure 4.2: Three coloured spots.

$s \leq 1$	-	spot extent
$f\|_{s=1} = 0$	-	background value for spot exterior
$f\|_{s=0} = 1$	-	ideal brightness value at spot centre
a positive definite	-	so we get a spot

We use these constraints to eliminate the parameters $k_0 = 1$ and $k_3 = -(k_0 + k_1 + k_2)$. Variations of spot and background illumination are represented in the complete model,

$$v_1 f(l) + v_0.$$

Using the least squares method, the best fit of this model to the actual light intensity function ϕ of that chosen area can be found.

$$F = \sum_{l \in \Omega} (\phi_l - (v_1 f(l) + v_0))^2$$

where

$$\Omega = \{(x, y) | s \leq 1\}.$$

We now minimize the error between the chosen model and the actual light intensity

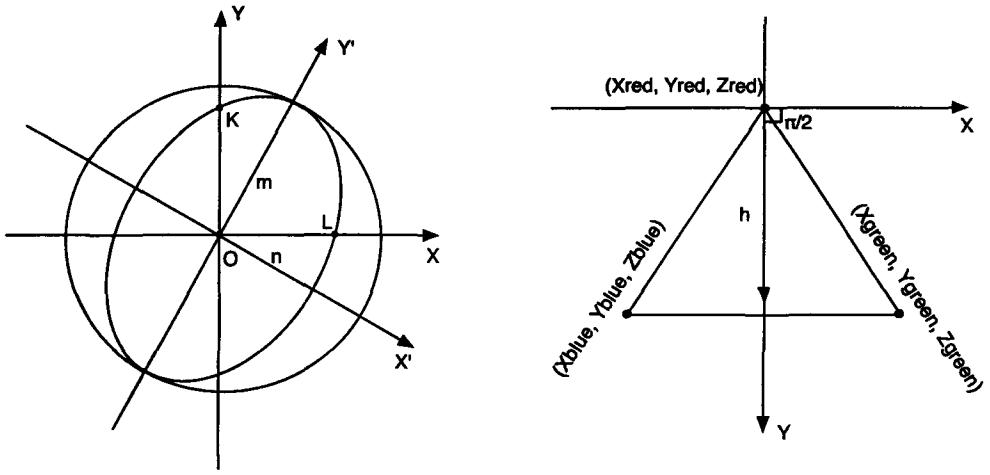

Figure 4.3: (a) Single spot, and (b) calculation of angle of rotation around z-axis

function as follows:

$$\min_{\text{subset of } \{a\text{'s, } b\text{'s, } k\text{'s, } v\text{'s}\}} F.$$

4.1.4 3D Position

Using three colored spots as presented in (Figure 4.2), we can determine all six parameters needed for completely describing the position of the surface with the pattern on it. Let p describe the position and the orientation of one spot in 3D:

$$p = (P_x, P_y, P_z, \theta_x, \theta_y, \theta_z)',$$

where P_i are the space coordinates of the centre of the spot, and θ_i are the angles of rotation of the spot around the i-th axis.

Given the parameters a_1, a_2, a_3, b_x and b_y of a spot, the following quantities can be computed: P_x, P_y, P_z, θ_x and θ_y. Having three different spots also allows the computation of θ_z. Calculation of P_x and P_y is trivial from b_x, b_y and a reference point.

An ellipse of the general form $a_1x^2 + 2a_2xy + a_3y^2 = 1$ in xy coordinates (Figure 4.3a), could be transformed to standard form $\frac{x'^2}{m^2} + \frac{y'^2}{n^2} = 1$ in the $x'y'$-plane.

Solving for the eigenvalues λ_1 and λ_2 of the matrix

$$\begin{pmatrix} a_1 & a_2 \\ a_2 & a_3 \end{pmatrix},$$

yields

$$m = \sqrt{\frac{1}{max\{\lambda_1, \lambda_2\}}}, n = \sqrt{\frac{1}{min\{\lambda_1, \lambda_2\}}}.$$

P_z is calculated as $P_z = D_f m_f / m$, where D_f and m_f are obtained from camera calibration – D_f is the distance from the camera, and m_f is the length of major semi-axis of the captured spot.

To find θ_x and θ_y we need to determine $|OL|$ and $|OK|$ (Figure 4.3a). L and K are simply x- and y- intercepts of an ellipse, and $|OL|^2 = \frac{1}{a_1}$, $|OK|^2 = \frac{1}{a_3}$. Now,

$$\cos\theta_y = \frac{|OL|}{m}, \cos\theta_x = \sqrt{\frac{\tan^2\theta_y + 1}{\tan^2\theta_y + \frac{m^2}{|OK|^2}}}$$

Finally, by employing three spots, we calculate θ_z. Let (x_r, y_r, z_r), (x_g, y_g, z_g), and (x_b, y_b, z_b) be coordinates of the centers of the red, green, and blue spots respectively (Figure 4.3b). In the desired orientation, vector $\mathbf{h} = (0, y_g, 0)$ makes an angle $\frac{\pi}{2}$ with the positive x-axis. In an arbitrary position of the three spots, vector $\mathbf{h}$ becomes $(\frac{x_g+x_b}{2} - x_r, \frac{y_g+y_b}{2} - y_r, \frac{z_g+z_b}{2} - z_r)$. The projection of the vector $\mathbf{h}$ onto the xy-plane becomes $\mathbf{h}' = (\frac{x_g+x_b}{2} - x_r, \frac{y_g+y_b}{2} - y_r, 0)$. From $\mathbf{h}'$ we can obtain θ_z:

$$\tan(\theta_z + \frac{\pi}{2}) = \frac{\frac{y_g+y_b}{2} - y_r}{\frac{x_g+x_b}{2} - x_r}.$$

Using three coloured spots also solves the problem of deciding if the rotation was clock- or counter-clockwise around the x- and y-axis. Since z_r, z_g, and z_b are known, we can tell which of the spots is now closer to the camera, providing us with the direction of rotation around x and y axes. Another benefit of using three spots is that the position is computed three times, and the results can be weighed with respect to how well each of the spots was captured. We are still determining the most robust and efficient method of using this redundant information.

4.2 Solver

4.2.1 Saturation

Before the fitting of the parameters defining the location and the shape of the spot can be performed, detection of saturation of the captured image must be carried out. If the image is found to be saturated, saturated pixels are removed when F is computed. Furthermore, if $\mathcal{S}$ is the set of saturated pixels, set Ω now becomes:

$$\Omega_{without\ saturation} = \Omega \setminus \mathcal{S}$$

To perform the computation of F while excluding saturated pixels [which contain little useful information about illumination], the pixel by pixel processing must be conditional. This will increase execution time of the algorithm. Since saturation of the image will not always occur, for each set of parameters, two versions of the solvers will be produced–with and without saturation control. The likelihood of saturation is calculated each frame based on the estimate for v, and this information is used to decide which solver to use on the next frame.

4.2.2 Initialization

The solvers are based on the multivariate Newton's method. In theory, if the objective function being optimized is convex, and if the starting point for the Newton iteration is sufficiently close to the actual solution, Newton's method will converge at a second order rate [6]. In practice, convergence of the Newton's method may be very slow and numerical errors may prevent it from converging at all [2, 3].

However, for our practical purposes, convexity of the subproblems and good first approximations (Sections 4.2.4 and 4.2.5) provided to the Newton's method allow it to compute reasonable estimates of the model's parameters with quadratic convergence. Seven to ten iterations were required to efficiently locate the center of the spot, and only three to five interations to determine the spot's shape. These results closely correspond to the theoretical quadratic convergence results described in [2] for the pure multivariate Newton's method with a good initial approximation.

Since Newton's method is extremely sensitive to initial starting points, we need to choose them well. We are currently using heuristics based on weighted averages and

pixel-value histograms. Since we have vendor-supplied, optimized libraries for these functions, we have not used code generation for this. From the image we can extract initial guesses for

- the average radius of the light spot on the image,
- the values for v_0 and v_1,
- the position of centre

The profile of every spot should be the same, thus the same values of k are used for all the spots. v_0 describes a background light level. b_x and b_y can be chosen as a centre of the image fragment. We have found it sufficient to use as a first guess a circular spot:

$$a_1 = 6.25 * 10^{-6},\ a_2 = 0.00,\ a_3 = 6.25 * 10^{-6}.$$

These dimensions ensure that the initial spot covers the entire (640 × 480) image fragment.

4.2.3 Fitting

To perform the fitting, let $\mathcal{V} = \{v_0, v_1, k_1, k_2, b_x, b_y, a_1, a_2, a_3\}$. It is necessary to minimize F with respect to a subset of the variables $\mathcal{U} \subseteq \mathcal{V}$.

$$\min_{\mathcal{U}} F.$$

To find the minimum of this function, partial derivatives with respect to all parameters from U are taken. As a result a system of equations is obtained, each of which equals to zero. The multivariable Newton's method can now be applied.

With the given model, we employ Maple to generate optimized C functions that calculate the Jacobian and Hessian matrices. These are used in a multivariate Newton's method to optimize F with respect to any subset of the system's parameters.

4.2.4 Convexity

We know that target recognition is not convex in general by looking at the multiple-spot case, because, for example in one dimension, matching a periodic pattern such

	Eigenvalues of the Hessian matrices
b	all – real, positive, magnitude of 1.0×10^8
a	all – real, positive, magnitude of 1.0×10^{17}
a,b	two – real, negative, magnitude of 1.0×10^5 three – real, positive, magnitude of 1.0×10^{12}

Table 4.1: Eigenvalues of the Hessian matrices.

as a sine wave to itself has multiple (periodic) minima:

$$\text{error} = \int_{-n\pi}^{n\pi} (sin(x) - sin(x+\delta))^2 dx = 2n\pi(1 - cos(\delta)).$$

To determine if our problem is convex for a single spot, we sampled the eigenvalues of the Hessian matrix during experiments with different solvers and various spots and lighting conditions. The problem is convex iff the Hessian is positive definite iff all the eigenvalues are positive. In practice, we require positive eigenvalues of similar magnitude. We found that, for the model described in this thesis, when fitting b_x and b_y, eigenvalues were always real and positive with magnitude approximately 1.0×10^8. As for a_1, a_2 and a_3, eigenvalues were also always real and positive with magnitudes 1.0×10^{17}. However, when trying to solve for both, the position of the centre and the shape of the spot within the same procedure, it was found that the Hessian matrices always had at least one negative eigenvalue (three of the five eigenvalues were positive with the magnitude of 1.0×10^{12}, and the other two–negative with the magnitude of 1.0×10^5) (Table 4.1).

With this profiling information, we know that a naive implementation where we simultaneously optimize all variables is likely to fail, except when we are tracking an essentially stationary spot from frame to frame, uninterrupted by lighting changes.

4.2.5 Need for Multiple Solver Stages

Depending on the structure of the problem, it may be possible to find a series of functions which approximate the objective function and which are sufficiently convex to make a staged solver – one which solves a series of increasingly difficult problems – converge efficiently in practical cases. The most straightforward method of finding

such a series of objective functions is to restrict the original function to subspaces of the original domain.

Due to lack of convexity when solving for the position of the centre and the shape of the spot in a single stage, it is necessary to have multiple solver stages. During the first stage, in seven to ten iterations, the location of spot is found (b_x and b_y) under the assumption that the size of the spot is larger than the whole captured image. Having an exact position of the centre of the spot makes the fitting of a_1, a_2 and a_3 a convex problem. In the second stage, in just a few iterations, the shape of the spot is then quickly found.

Chapter 5

Implementation

Outline of Maple code generation procedure:

1. Function F is defined. Summations over (x, y) are omitted.

```
> f := add (k||i * s^i, i = 0..3):
> X := <x - b_x, y - b_y>:
> A := Matrix (2, 2, symmetric, (x,y) -> a|| (x + y - 1)):
> sTmp := (Transpose (X).A).(X):
> fTmp := (phi[xInt, yInt] - (v||1 * function(x,y) + v||0))^2:
> F := eval (fTmp, function = unapply (eval (f, s = sTmp), x, y)):
```

2. Set $\mathcal{U}$ is defined (i.e., if we are fitting the center, $\mathcal{U}$ is defined as $\{b_x, b_y\}$, etc).

3. Jacobian and Hessian matrices are computed (*note*: only half of the entries of the Hessian matrix are computed due to symmetry). `args` $= \mathcal{U}$ and `numArgs` $= \#\mathcal{U}$.

```
> allElements := [seq (
>   seq(hess||i||j = hess||i||j + diff (diff (F,args[i]), args[j]),
> i = 1..j), j = 1..numArgs),
>   seq(jac||i     = jac||i + diff (F,args[i]), i = 1..numArgs)]:
```

4. Code generation of Jacobian and Hessian matrices is performed.

```
> opted := codegen[optimize] (allElements, tryhard):
```

5. Local variables are retrieved. On average, Maple introduces 20 temporary variables, which become local variables for the C functions.

6. The loop body is generated into a string.

```
> codeString := C ([opted], output=string, precision=single,
>   declare=[x::float, y::float]):
```

7. - A C function is generated by wrapping the `codeString` in `for` loops over (x, y),
   - elements from the top right half of the Hessian matrix are copied to its bottom left one, and
   - local variables are declared.

Fitting is then performed as follows. Variables which will be held constant for this particular optimization are initialized, and variables to be optimized are marked. The C function from above is used to generate H and J. The LAPACK package is used for implementing Newton's method, via `sgetrf_` and `sgetrs_`. `sgetrf_` is used to factor the H matrix and `sgetrs_` is used to solve the $Hx = J$ system. A better approximation to the initial guess is calculated, and is repeatedly used in the Maple-generated C function to obtain successively "better" values for the optimized variables. Depending on the application and the timing requirements, either the difference between two consecutive approximations or the number of iterations can be used as termination criteria for the algorithm.

Examples of Maple generated functions could be found in Appendix A.

Chapter 6

Results

Measurements of floating point operations in code generated using optimization strategies described in Section 1.3 confirmed our expectations. Table 6.1 and Figure 6.1 show flops counts, with and without joint optimization of the generated code for the Jacobian and Hessian matrices, and with and without the Maple `tryhard` option that attempts to optimize the generated code for speed. Jointly optimizing the Jacobian and Hessian produced a $20 - 25\%$ reduction in flops, while the use of the `tryhard` option produced a $30 - 40\%$ reduction in flops. This does not reflect the equally important reduction in memory traffic and reduction in local variables by jointly calculating the Jacobian and Hessian in one loop. Therefore, by jointly calculating and optimizing the Jacobian and Hessian matrices, the resultant code is faster and requires less memory.

The technology, at least as currently present in Maple 9, is finicky, but perseverance pays off: there are workarounds for most of the quirks, so that armed with some patience, it seems to always be feasible to coax Maple into producing usable code. And at least Maple helps: there are no such integrated packages in Mathematica, and the available third-party packages do not seem to be well-supported or up-to-date. To our knowledge, it is not possible to do model-driven symbolic development in Matlab.

As an example of the interoperability problems we encountered, the Maple commands `diff` and `codegen[GRADIENT]` both know how to differentiate `sum`, but neither `codegen[C]` nor `CodeGeneration[C]` know how to deal with `sum`; more annoyingly, there is code available that can deal with this automatically

	Separately Optimized Jacobian and Hessian		Jointly Optimized Jacobian and Hessian	
		+tryhard		+tryhard
b	152	97	112	78
a	176	117	135	88
a,b	396	220	325	205
a,b,v	461	284	394	230

Table 6.1: Number of flops per pixel in the generated solvers.

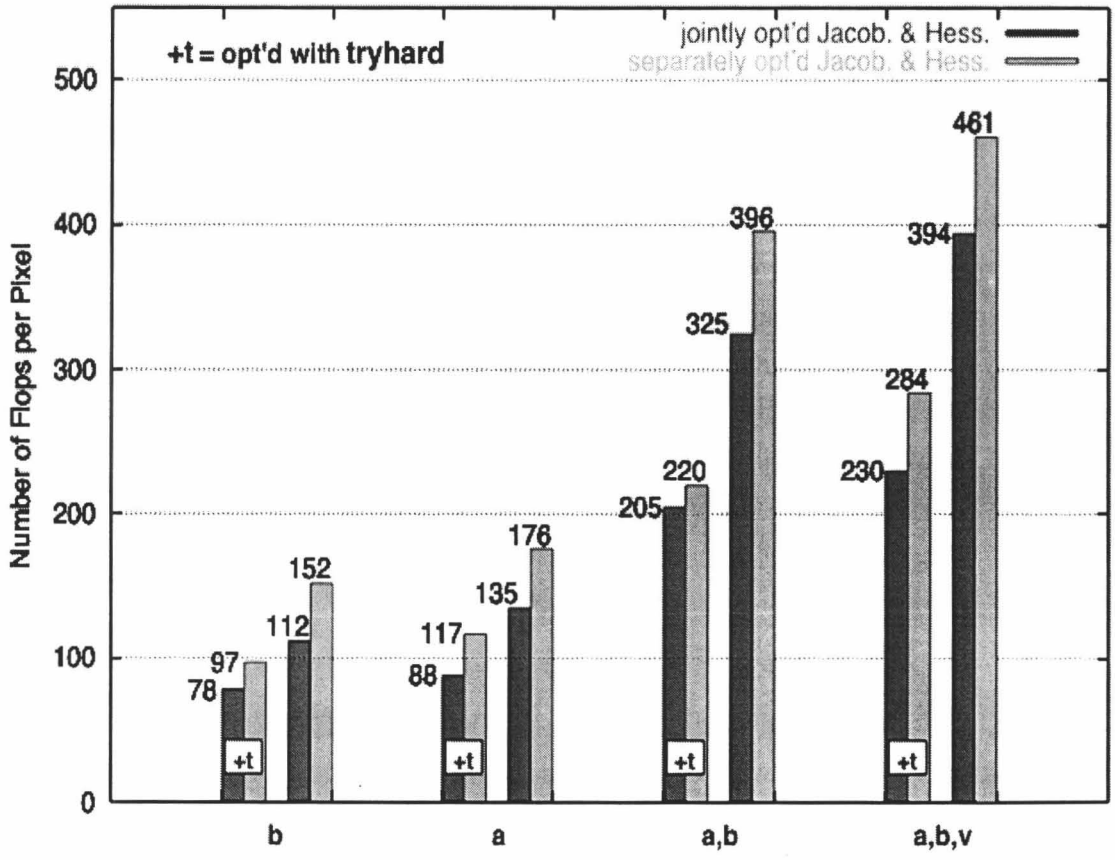

Figure 6.1: Number of flops per pixel in the generated solvers.

(`codegen[prep2trans]`), but it needs to be hand-applied. Another annoyance is that optimization is very important–but the most powerful optimizations are only available from `codegen[optimize]` through its `tryhard` option, and not through `CodeGeneration[C]`, even though it has an `optimize` option. Of course, `tryhard` is not a panacea: how is one to interpret

```
codegen[optimize](Sum(sin(i^2)+i^3*cos(i^2+1),i=a..b),'tryhard');
```

returning

$$t29 = i^2, t1 = \sum_{i=a}^{b} \sin(t29) + t29\, i \cos(t29 + 1).$$

However if an expression like the above is converted to an appropriate procedure (via `prep2trans`), then no optimizations at all are performed on the inner-loop expression. This forces a user to perform optimizations on very small code chunks, and then hand-assemble the results into a larger more coherent whole. This seems rather counter to the idea of using code generation in the first place. Being able to correctly optimize models which use higher-order constructs like `sum` is crucial for the more complex models we want to investigate.

Furthermore, other difficulties arise, in particular with the use of the C function `pow` in the C output. We have not been able to predict the actual pattern, but polynomials such as x^3 frequently get translated to `pow`$(x,$ `(double)`$3)$ instead of $x*x*x$, even if x is an integer variable. This appears to be related to `CodeGeneration` (correctly) guessing that the eventual output is supposed to be a floating point number, but this is not clear.

Nevertheless, it is possible to generate code that is usable. It could be achieved as follows:

- using the Maple symbolic differentiation operator `diff` on expressions, and `codegen[optimize]` with option `tryhard` on collections of `Sum`-free terms, then
- partly hand-building the resulting code into a large procedure (with help from `makeproc`, `prep2trans` and `joinprocs` from the `codegen` package), followed by
- post-processing pass to remove calls to `pow`.

Given the importance that code generation from models is likely to take in the future, we have been careful to submit all our observations on these matters to the makers of Maple, and we hope that these will help them to produce an improved version of the tools for code generation.

Chapter 7

Conclusion and Future Work

Model-based visual tracking in extreme lighting conditions is currently expensive both in terms of development and computation. Adaptation of Maple code generation to the problem of automatically generating efficient implementations of families of Newton solvers reduced the development cycle, improved reliability and reduced the computational requirements. Use of symbolic computation allowed the design of a code generator to be independent of the model and number of independent variables. The designed algorithm solved to the problem of imaging in extreme lighting conditions in which the camera is saturated for a significant portion of the target image. We investigated the problems caused by limitations in the current implementation of code generation in Maple, as well as the other problems to which designed method is applied, and also carried out related approaches to code generation.

Overlapping with the current work, Olesya Peshko (also at McMaster) is applying generated code to a visual contour extraction problem arising in radiation therapy planning. As part of that project, it will be necessary to generalize the code generators presented in this thesis to three and four dimensions.

The principal task remaining in the current project is the integration of the solvers developed so far into a Model Predictive Controller (MPC) [11]. Originally, MPC was developed to meet specific control needs of power plants and petroleum refineries. MPC technology can now be found in a wide variety of application areas which include automotive, chemicals, aerospace, and metallurgy. By adapting these techniques to target recognition, and incorporating uncertainty and brightness informa-

tion, our MPC will simultaneously control the acquisition mechanics and the camera parameters, which we expect to yield a significantly more robust closed-loop control system.

Bibliography

[1] C. Anand, J. Carette, and A. Korobkine. Target recognition algorithm employing maple code generation. Maple Summer Workshop, 2004.

[2] Stephen Boyd and Lieven Vandenberghe. Convex Optimization. Cambridge University Press, New York, NY, 2004.

[3] Morton M. Denn. Optimization by Variational Methods. McGraw-Hill Book Company, Delaware, 1969.

[4] A. van Deursen and P. Klint. Little languages: Little maintenance? Journal of Software Maintenance, 10:75–92, 1998.

[5] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, 35(6):26–36, June 2000.

[6] Roger Fletcher. Practical Methods of Optimization. A Wiley-Interscience Publication, Dundee, 1980.

[7] A. Gibbons. A program for the automatic integration of differential equations using the method of Taylor series. Comp. J., 3:108–111, 1960.

[8] Andreas Griewank. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[9] S. A. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. IEEE Trans. Robotics and Automation, 12(5):651–670, October 1996.

[10] H. G. Kahrimanian. Analytical differentiation by a digital computer. Master's thesis, Temple University, May 1953.

[11] A.W Pike, M.J Brimble, M.A. Johnson, A.W. Ordys, and S. Shakoor. Predictive control. In W. S. Levine, editor, The Control Handbook, chapter 51. CRC Press, Florida, 1996.

[12] Louis B. Rall. Automatic Differentiation: Techniques and Applications, volume 120 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1981.

[13] Alan Reiter. Automatic generation of Taylor coefficients (TAYLOR). Prog. 3, Mathematics Research Center, University of Wisconsin - Madison, 1965.

[14] A. C. Sanderson and L. E. Weiss. Image-based visual servo control using relational graph error signals. IEEE Transactions on Robotics and Automation, pages 1074–1077, 1980.

[15] Joseph M. Thames. SLANG, a problem-solving language for continuous-model simulation and optimization. In Proceedings of the ACM 24th National Conf. ACM, New York, 1969.

[16] R. E. Wengert. A simple automatic derivative evaluation program. Comm. ACM, 7(8):463–464, 1964.

Appendix A

Maple Generated Functions

A.1 Function to calculate Hessian and Jacobian matrices for a_1, a_2, and a_3.

```
#include <math.h>

void function__opt__a1a2a3 ( float **phi, float a1, float a2, float a3, float b_x, float b_y,
float k0, float k1, float k2, float k3, float v0, float v1, int N, int M, float *jac, float *hess )
{
 float t64;
 float t2;
 float t39;
 float t3;
 float t12;
 float t5;
 float t66;
 float t49;
 float t63;
 float t20;
 float t48;
 float t47;
 float t61;
 float t58;
 float t57;
 float t27;
 float t29;
 float t65;
 float t30;
 float t31;
 float t19;
 float t9;
 float t4;
 float t10;
 float t7;
 float t46;
 float t50;
 float t43;
 float t44;
 float t62;
 float t59,
 float t60;
```

```
 float t11;
 float t51;
 float t54;
 float t55;
 float t45;
 float t53;


 float hess11;
 float hess12;
 float hess22;
 float hess13;
 float hess23;
 float hess33;
 float jac1;
 float jac2;
 float jac3;


int x_int, y_int;
float x, y;

int x_int_lower,x_int_upper;
float x_lower, x_upper, D4;

hess11 = 0;
hess12 = 0;
hess22 = 0;
hess13 = 0;
hess23 = 0;
hess33 = 0;
jac1 = 0;
jac2 = 0;
jac3 = 0;


for (y = 1.f, y_int = 0; y_int < M; y_int++, y++)
{
D4 = a2*a2*(y-b_y)*(y-b_y) - a1*(a3*(y-b_y)*(y-b_y) - 1.0);
if (D4 < 0)
{
x_int_lower = 0;
x_int_upper = -1;
}
else
{
D4 = sqrt (D4);
x_lower = (- a2*(y-b_y) - D4) / a1 + b_x + 1.0;
x_int_lower = floor (x_lower) - 1;

if (x_int_lower < 0)
{
x_lower = 1.f;
x_int_lower = 0;
}

x_upper = (- a2*(y-b_y) + D4) / a1 + b_x + 1.0;
x_int_upper = ceil (x_upper) - 1;

if (x_int_upper > N)
{
x_int_upper = N;
}
}

for (x = x_lower, x_int = x_int_lower; x_int < x_int_upper; x_int++, x++)
{
if (phi[x_int][y_int] < 255)
```

```
{

t31 = x - b_x;
t29 = t31 * t31;
t30 = y - b_y;
t20 = t29 * a1 + (0.2e1f * t31 * a2 + t30 * a3) * t30;
t65 = 0.1e1f + t20;
t19 = t20 * t20;
t9 = (0.5e0f * t19 + 0.1e1f) * t65;
t58 = 0.2e1f / t9;
t54 = phi[x_int][y_int] - v0;
t2 = -v1 + t54 * t9;
t7 = pow(t9, -0.2e1f);
t66 = t2 * t2 * t7;
t53 = t31 * t30;
t10 = (0.2e1f + 0.2e1f * t20 + 0.3e1f * t19) * t53;
t3 = t54 * t10;
t50 = t3 * t58;
t39 = 0.15e1f * t19 + t65;
t12 = t39 * t29;
t5 = t54 * t12;
t49 = t5 * t58;
t55 = t2 * t7;
t47 = -0.20e1f * t55;
t64 = t49 + t12 * t47;
t48 = t66 * t58;
t63 = t48 * t12;
t51 = t2 * t58;
t62 = -0.10e1f * t66 + t54 * t51;
t57 = 0.1e1f + 0.3e1f * t20;
t61 = t57 * t62;
t27 = t30 * t30;
t60 = t62 * t29 * t27;
t59 = (0.2e1f + 0.6e1f * t20) * t53 * t62;
t46 = -0.40e1f * t55;
t11 = t39 * t27;
t45 = t11 * t48;
t44 = t11 * t47;
t43 = t10 * t47;
t4 = t54 * t11;
hess11 = hess11 + t5 * t5 * t58 + t29 * t29 * t61 + (t5 * t46 + t63) * t12;
hess12 = hess12 + t10 * t63 + t5 * t43 + t29 * t59 + t64 * t3;
hess22 = hess22 + t10 * t10 * t48 + (0.4e1f + 0.12e2f * t20) * t60 + (t10 * t46 + t50) * t3;
hess13 = hess13 + t12 * t45 + t5 * t44 + t57 * t60 + t64 * t4;
hess23 = hess23 + t10 * t45 + t3 * t44 + t27 * t59 + (t50 + t43) * t4;
hess33 = hess33 + t11 * t11 * t48 + (t11 * t46 + t58 * t4) * t4 + t27 * t27 * t61;
jac1 = jac1 - 0.10e1f * t12 * t66 + t2 * t49;
jac2 = jac2 - 0.10e1f * t10 * t66 + t2 * t50;
jac3 = jac3 - 0.10e1f * t11 * t66 + t4 * t51;
}
}

}

jac[0] = jac1;
jac[1] = jac2;
jac[2] = jac3;

hess[0] = hess11;
hess[1] = hess12;
hess[2] = hess13;
hess[3] = hess12;
hess[4] = hess22;
hess[5] = hess23;
hess[6] = hess13;
hess[7] = hess23;
hess[8] = hess33;
}
```

A.2 Function to calculate Hessian and Jacobian matrices for b_x and b_y.

```
#include <math.h>

void function__opt__b_xb_y ( float **ph1, float a1, float a2, float a3, float b_x, float b_y,
float k0, float k1, float k2, float k3, float v0, float v1, int N, int M, float *jac, float *hess )
{
 float t1;
 float t2;
 float t33;
 float t34;
 float t22;
 float t23;
 float t4;
 float t24;
 float t42;
 float t27;
 float t8;
 float t47;
 float t7;
 float t43;
 float t15;
 float t16;
 float t12;
 float t10;
 float t38;
 float t45;
 float t46;
 float t44;
 float t39;
 float t36;
 float t37;
 float t40;


 float hess11;
 float hess12;
 float hess22;
 float jac1;
 float jac2;

int x_int, y_int;
float x, y;

int x_int_lower,x_int_upper;
float x_lower, x_upper, D4;

hess11 = 0;
hess12 = 0;
hess22 = 0;
jac1 = 0;
jac2 = 0;


for (y = 1.f, y_int = 0; y_int < M; y_int++, y++)
{
D4 = a2*a2*(y-b_y)*(y-b_y) - a1*(a3*(y-b_y)*(y-b_y) - 1.0);
if (D4 < 0)
{
x_int_lower = 0;
x_int_upper = -1;
}
else
{
```

```
D4 = sqrt (D4);
x_lower = (- a2*(y-b_y) - D4) / a1 + b_x + 1.0;
x_int_lower = floor (x_lower) - 1;

if (x_int_lower < 0)
{
x_lower = 1.f;
x_int_lower = 0;
}

x_upper = (- a2*(y-b_y) + D4) / a1 + b_x + 1.0;
x_int_upper = ceil (x_upper) - 1;

if (x_int_upper > N)
{
x_int_upper = N;
}
}

for (x = x_lower, x_int = x_int_lower; x_int < x_int_upper; x_int++, x++)
{
if (phi[x_int][y_int] < 255)
{

t16 = (x - b_x) * (0.1e-3f * x - 0.1e-3f * b_x) + (y - b_y) * (0.1e-3f * y - 0.1e-3f * b_y);
t15 = t16 * t16;
t12 = (0.5e0f * t15 + 0.1e1f) * (0.1e1f + t16);
t43 = 0.2e1f / t12;
t40 = phi[x_int][y_int] - v0;
t4 = -v1 + t40 * t12;
t10 = pow(t12, -0.2e1f);
t47 = t4 * t4 * t10;
t34 = t16 + 0.15e1f * t15;
t27 = -0.2e-3f * x + 0.2e-3f * b_x;
t8 = t34 * t27 - 0.20e-3f * x + 0.20e-3f * b_x;
t2 = t40 * t8;
t46 = t2 * t43;
t45 = t4 * t10;
t39 = t4 * t43;
t44 = -0.10e1f * t47 + t40 * t39;
t42 = 0.3e1f * t16;
t38 = t47 * t43;
t37 = -0.40e1f * t45;
t36 = -0.20e1f * t45;
t33 = 0.30e-3f * t15 + 0.20e-3f + 0.20e-3f * t16;
t24 = -0.2e-3f * y + 0.2e-3f * b_y;
t23 = t27 * t27;
t22 = t24 * t24;
t7 = t34 * t24 - 0.20e-3f * y + 0.20e-3f * b_y;
t1 = t40 * t7;
hess11 = hess11 + t8 * t8 * t38 + t44 * (t23 + t23 * t42 + t33) + (t8 * t37 + t46) * t2;
hess12 = hess12 + t44 * (0.1e1f + t42) * t27 * t24 + (t2 * t36 + t8 * t38) * t7 + (t46 + t8 * t36) * t1;
hess22 = hess22 + t7 * t7 * t38 + t44 * (t22 + t22 * t42 + t33) + (t7 * t37 + t1 * t43) * t1;
jac1 = jac1 - 0.10e1f * t8 * t47 + t2 * t39;
jac2 = jac2 - 0.10e1f * t7 * t47 + t1 * t39;
}
}

}

jac[0] = jac1;
jac[1] = jac2;

hess[0] = hess11;
hess[1] = hess12;
hess[2] = hess12;
hess[3] = hess22;
}
```