VISUAL TRACKING EMPLOYING MAPLE CODE GENERATION

CHRISTOPHER KUMAR ANAND, JACQUES CARETTE, AND ALEXANDRE KOROBKINE

Abstract

Closed-loop, model-based visual target recognition and tracking in extreme lighting conditions is expensive to develop and computationally resource-intensive. To reduce the development cycle, improve software reliability and reduce the computational requirements for such algorithms, we have adapted Maple code generation to the problem of automatically generating efficient implementations of families of Newton solvers, each of which estimates a set of related parameters in a target model. We describe the leading target model in detail, formulate target identification as an optimization problem, explain the challenges in solving this model and the resulting need for multiple solvers, and the main advantage provided by code generation in Maple. We also discuss the problem of partially-saturated images in this scheme and our approach to solving it, and desirable features for a future version of Maple which would improve the applicability to similar application domains and simplify the implementation of code generators.

1. Introduction

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, so we restrict our search to piecewise polynomial and rational polynomial patterns. The recognition of such patterns can be modeled simply as a constrained, nonlinear optimization problem. Recognition can be implemented as a solver, which in addition to estimating model parameters, can assign a likelihood to the estimates. Advanced model-based controllers make use of the likelihood information to improve the robustness of the controller to random and systematic noise sources.

Date: April 1, 2004.
2. Background on Code Generation

Code generation is now rather ubiquitous (over 500,000 hits for a search on “code generation” on Google as of the writing of this paper). It is present in various ways inside most serious development environments and frameworks. The only place where it is not omnipresent is in the standard undergraduate curriculum, which is perhaps why too many programmers still seem to not use such techniques routinely. It is also quite old, with code generators already showing up in the early 1960s ([6, 14, 12] to include just a few topical and relevant references), as the leap from compiler to code generators is quite small.

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

1. when complex program transformations are needed [11, 7]
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. [2, 1]

The first situation occurs most famously when automatic differentiation [8] is both required and applicable. There is now ample literature (from [13] onwards) 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 so fraught with error as to be deemed impossible. On the other hand, differentiation is a simple (symbolic) program.

The second situation is now emerging as rather common as well. Thus the growing popularity of GUI-builders, lexer and parser generators, java-from-DTD builders, and so on. This is also related to the emerging field of “software synthesis” and “automated software engineering”, with its own research journal, Automated Software Engineering; a particularly impressive real-world example from Boeing is [15]. This is also a trend which is engulfing the scientific computation community, from specific efforts like [5], to vast projects like [4, 3, 9].

The target recognition problem is particularly well-suited to being attacked 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.

This is doubly relevant as almost all of the downsides of using a domain specific languages, as listed in [2] are well mitigated by using Maple as the DSL.

2.1. Problem Definition. 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—which would give us partial or complete information on the relative position of the target. In this paper we will focus on a simple family radially-symmetric, essentially compact targets, which we will call spots. Transformed spots will have elliptical equiradiant contours. Given a target image,
we are required to estimate parameters for the position (translation), size (scaling), orientation and asymmetry (rotation and pitch).

2.2. Performance Requirements. 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 must detect and track a predefined pattern on a satellite. Sudden changes of lighting possibly saturating a significant part of the pattern, complete or partial “blindness” of the camera lens from the sunlight or from the lens moving into the shadow pose significant obstacles to any algorithm, and to overcome them we propose to use 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.

For the stability of the control, image processing must be performed quickly. We estimate the required frame rate to be 10-30 frames per second, which makes the control part of the application non-standard.

2.3. Need for Rapid Prototyping. 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 employment of Maple code generation reduces time for design, documentation, 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.

2.4. Advantage of Maple. Employing Maple code generation allows the production of efficient code for the Newton solvers. A function describing the efficacy of the model in relation to the actual light intensity function is passed into Maple. Maple then generates the C functions which calculate Jacobian and Hessian matrices for each of the needed sets of parameters (location of the spot – set of two parameters or the shape – set of three parameters). Known variables can be initialized within Maple routine or in the C function calling Maple-generated code. The use of Maple enables inexpensive change of underlying models.

Currently, the algorithm efficiently recognizes 2D spots. However, since Maple is used to generate all core functions, the algorithm can be extended to higher dimensions.

When the model has been validated in the prototype, the code generator can be modified in Maple to produce code for an embedded target in parallel. This allows the evaluation of new models/algorithms in the prototype, as well as the application environment.

2.5. 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. Camera was chosen to be colour and of low cost, with capability of supporting up to 15 frames per second.
• 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 10.3.3 (7F44)
  – **Kernel Version**: Darwin 7.3.0

• Camera Overview
  – **Type**: Fire-i
  – **Driver**: IOXperts, Version: 1.1b22

3. Model

3.1. **Colour Space.** Images are streamed from the camera over a FireWire serial interface at 15 frames per second. The images are formatted as arrays of 8-bit RGB values, i.e. a pixel value of (255, 0, 0) is bright red and (0, 0, 0) is black. There are no negative pixel values, and if the gain adjustment is too high for 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 and converted to floating point numbers, so the effect of saturation may not be restricted to a cube in \( \mathbb{R}^3 \) aligned to the coordinate axes. 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.

3.2. **Advantages of Gradients.** 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 and suffer from 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. Gradient-based estimation can be efficiently pipelined, which will improve execution on pipelined architectures. For suitable gradient targets, even saturated images yield information about lighting which can be used to adjust camera gain. This is important when large lighting changes are expected.

3.3. **Advantages of Colour.** Simple coloured patterns can be used to identify orientation of the target. Having three spots of different 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. Fitting each spot is a non-convex problem, however 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 can be either a gray-scale image or a single component of a multispectral image, and 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 \( \alpha, \beta \) and \( \gamma \), therefore

\[
\begin{pmatrix}
    \alpha \\
    \beta \\
    \gamma
\end{pmatrix} = \left( \begin{pmatrix}
    r_0 & r_1 & r_2 \\
    g_0 & g_1 & g_2 \\
    b_0 & b_1 & b_2
\end{pmatrix} \right)^{-1} \begin{pmatrix}
    r_{\text{real}} \\
    g_{\text{real}} \\
    b_{\text{real}}
\end{pmatrix},
\]

where \( r_{\text{real}}, g_{\text{real}} \) and \( b_{\text{real}} \) are the colour values of each pixel from the original image. Now the array of \( \alpha, \beta \) and \( \gamma \) 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 grayscale solver for colour spot identification.

3.4. 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 paper 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, 3D \( \phi_{x,y,c} \) is used, where \( x \) and \( y \) define a pixel on the image and \( c \in \{ r, g, b \} \) defines a 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. The components of \( p \) are \( x, y \).

The basic model of a spot (Figure 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
Figure 1. 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 (3) determined by \( k \). The background illumination is determined by \( v_0 \) and the brightness of the centre of the spot is \( v_1 \).

\[
\begin{align*}
  k_0, k_1, k_2, k_3 & \quad \text{- determine the radial profile} \\
  a_1, a_2, a_3 & \quad \text{- determine the extent and eccentricity} \\
                  & \quad \text{(shape of the elliptical boundary)} \\
  b_x \text{ and } b_y & \quad \text{- } x \text{ and } y \text{ coordinates (in pixels) of the ellipse centre}
\end{align*}
\]

In addition, we add following constraints and conventions
\[
\begin{align*}
    s &\leq 1 \quad \text{- spot extent} \\
    f\big|_{s=1} & = 0 \quad \text{- background value for spot exterior} \\
    f\big|_{s=0} & = 1 \quad \text{- ideal brightness value at spot centre} \\
    a & \text{ positive definite} \quad \text{- so we get a spot}
\end{align*}
\]

We use these constraints are used 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,

\[
v_1 f(p) + v_0,
\]

by \(v_0\) and \(v_1\).

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

\[
F = \sum_{p \in \Omega} (\phi_p - (v_1 f(p) + v_0))^2
\]

where

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

The above allows the following:

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

3.4.1. Initialization. Since Newton’s method is an iterative procedure, that is sensitive to initial guesses, it is important to choose them well. We use heuristics to determine reasonable initial guesses, and the tuning of these heuristics can make a big difference to the performance of the initial target acquisition. We are currently using simple 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 in this area. From the image we can extract initial guesses for

- the average radius of the light spot on the image
- values for \(v_0\) and \(v_1\)
- 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. \(b_x\) and \(b_y\) can be chosen as a centre of the image fragment. We have found it sufficient to use as a first guess for the parameters \(a_1, a_2\) and \(a_3\), which determine the shape of the spot is a circle, the following values:

\[
a_1 = 6.25 \times 10^{-6}, \quad a_2 = 0.00, \quad a_3 = 6.25 \times 10^{-6}.
\]

These dimensions of the circle above ensure that the initial spot covers the entire image fragment (in this case, a 640x480 image).
3.4.2. 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 $\mathcal{U}$ 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 C functions that calculate Jacobian and Hessian matrices. These are used in Newton’s method to optimize $F$ with respect to any subset of system’s parameters.

3.5. Convexity. In a neighbourhood of every local minimum, the objective function is necessarily convex, and Newton’s method will converge. 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 as a sine wave to itself has multiple (periodic) minima:

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

To determine convexity for a single spot, we sampled 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, otherwise Newton’s method may fail to converge.) We found that, for the model described in this paper, when fitting $b_x$ and $b_y$, eigenvalues were always real and positive with magnitude of $1.0 \times 10^8$. As for $a_1$, $a_2$ and $a_3$, eigenvalues were also always real and positive with magnitudes $1.0 \times 10^{17}$. 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 $1.0 \times 10^{12}$, and the other two–negative with the magnitude of $1 \times 10^5$).

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

3.6. Saturation. Before 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 $\Omega$ now becomes:

$$\Omega_{\text{without saturation}} = \Omega - \{(x, y) | (x, y) \in \mathcal{S}\}.$$ 

To perform the computation of $F$ excluding saturated pixels, which contain little useful information about illumination, we must conditionalize the processing pixel by pixel. 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—one with and one 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.

3.7. Need for Multiple Solver Stages. 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. Assuming that the size of the spot is larger than the whole captured image, during the first stage the location of spot is found ($b_x$ and $b_y$). Having an exact position of the centre of the spot makes the fitting of $a_1$, $a_2$ and $a_3$ a convex problem. So in a second stage, in just a few iterations, the shape of the spot is found.

4. Solver

4.1. Newton’s Method. Newton’s method is an iterative procedure for finding the root of a system of nonlinear equations. It is often used in optimization to find an extremum of an objective function. An initial guess is used to find a better solution. This procedure is repeated until satisfactory results are found. If the objective function being optimized is convex, then Newton’s method will converge in theory, although in practice it may be very slow and numerical errors may prevent it from converging. If the objective function is not convex, it will not converge in general, but under mild assumptions, every local minima is contained in a neighbourhood on which the function is convex, which is contained in the basin of convergence of the local minima under the Newton iteration. Depending on the structure of the problem, it may be possible to find a series of functions which approximate the objective function in some way 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. Perhaps after a judicious change of variables, the subspaces may be taken to be affine subspaces of the original variable space.

Let $J_{\mathcal{U}}$ be the Jacobian of $F$ and $H_{\mathcal{U}}$ be the Hessian of $F$ with respect to the variables $\mathcal{U}$. $u_n$ is a solution on the $n$ step (vector-column of the $\mathcal{U}$ parameters). The Newton iteration is defined by the recursion:

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

4.2. Efficient Implementation. Advantages to mathematical modeling and optimization-based parameter extraction can easily be overwhelmed by computational cost and development time. 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, which involves rational functions, efficiently solves this problem. The code generator described in this paper currently generates solvers for 2D models, but could easily be generalized to higher dimensions. It will be tested on 3D problems in medical imaging in the future.

Large arrays are needed to store of information of captured images. Calculation of the sum over elements in arrays is an expensive procedure. Efficient use of the cache is required to minimize the
Table 1. Number of flops per pixel in the generated solvers with and without joint optimization of the code list, and with and without the optional tryhard. Jointly optimizing the Jacobian and Hessian produced a $20 - 25\%$ reduction in flops. Using codegen[optimize] with 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.

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.

The complexity of the Maple-generated Newton solvers is found to be directly proportional to the complexity of the model. Length of the typical Maple-generated Newton solver (for 2 or 3 variables) is approximately 180 lines of code and 200 flops per pixel. Solvers are generated by executing codegen[optimize] function with the tryhard option. Without the tryhard option, generated solvers slightly increase in length and number of floating point operations per pixel (230 lines of code, 250 flops).

There is an advantage to the joint optimization of the Jacobian and the Hessian matrices. Maple’s codegen[optimize] function eliminates common subexpressions very effectively when these matrices are generated together. If the optimization of code is performed separately for Jacobian and Hessian matrices and the results are concatenated, both the length of the solver and the number of flops per pixel are roughly doubled.

4.3. Maple Implementation. There are limitations of Maple code generation for imaging applications. Maple generated efficient loop bodies for the Newton solvers, which we custom package to get one-trip-through-cache-per-solver-iteration behaviour.

Outline of Maple code generation procedure:
(1) Function $F$ is defined. Summations over $(x, y)$ are omitted.

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

(2) Set $\mathcal{U}$ is defined.

(3) Jacobian and Hessian matrices are computed (note: only half of the entries of the Hessian matrix are computed). \(\text{args} = \mathcal{U}\) and \(\text{numArgs} = \#\mathcal{U}\).

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

(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.

(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 of the spots is performed as follows. An image is read pixel by pixel into 2D array. Constant variables, if any, are initialized. Optimized variables are marked. Maple function, generating C function that calculates Jacobian and Hessian matrices for given optimized variables, is called. LAPACK package is used for Newton’s method. Two functions, \text{sgetrf} and \text{sgetrs}, are called. The first one is used to factorize the $H$ matrix and the second one is to solve the $H = Jx$ system. The better approximation to initial guess is calculated and is used in Maple-generated C function to obtain the ”better” values for optimized variables and the procedure is repeated. Depending on an application and the timing requirements, either the difference between two consecutive approximations of the Newton’s method or the number of iterations can be used as termination criteria for the algorithm.

4.4. Problems encountered. The technology, at least as currently present in Maple 9, is extremely finicky. One can not boldly claim that “it does not work”, tempting as that may be in the heat of repeatedly being frustrated by inexplicable quirks. But perseverance pays off: there are workarounds for most of the quirks, so that armed with ample patience, it seems to always be feasible to coax Maple into producing usable code.
For example, there are silly interoperability problems. In particular, `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 (`codegen[prep2trans]`), but it needs to be hand-applied instead of being automated. 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

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

returning

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

On the flip side, no optimizations are performed at all using an expression like the above, but inside a procedure that has been passed through `codegen[prep2trans]`. 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` (higher-order because they bind a variable) is crucial for the more complex models we hope to be able to investigate.

Furthermore, other difficulties arise, in particular with the use of the `pow` function 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, if instead one falls back to using just `diff` on expressions, `codegen[optimize]` with option `tryhard` on collections of `Sum`-free terms (given as a list as there are quirks with Matrix that need to be avoided), then partly hand-building the resulting code into a large procedure (with some help from `makeproc`, `prep2trans` and `joinprocs` from the `codegen` package), it is possible to generate code that is usable, once a post-processing pass to remove calls to `pow` has been done.

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 Maplesoft, and we hope that they will help to produce the next improved version of the tools for code generation.

5. Conclusion/Future Work

It was shown that model-based visual tracking in extreme lighting conditions is expensive both in development and in 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 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. Investigation of the problems caused by limitations in the current implementation of code generation in Maple, other problems to which designed method is applied, and related approaches to code generation was carried out.

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 paper 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) [10]. 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 including automotive, chemicals, aerospace, and metallurgy. By adapting these techniques to target recognition, and incorporating uncertainty and brightness information, 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.

REFERENCES


McMaster University, Department of Computing and Software

E-mail address: {anandc,carette,korobkao}@mcmaster.ca