A machine vision extension for the Ruby programming language

WEDEKIND, J., AMAVASAI, B. P., DUTTON, K. and BOISSENIN, M.

Proceedings of the 2008 IEEE
International Conference on Information and Automation
June 20 -23, 2008, Zhangjiajie, China

A Machine Vision Extension for the Ruby Programming Language*

J. Wedekind, B. P. Amavasai, K. Dutton, M. Boissenin
Microsystem & Machine Vision Laboratory, Materials and Engineering Research Institute
Sheffield Hallam University
Pond Street, Sheffield S1 1WB, United Kingdom
J.Wedekind@shu.ac.uk, B.P.Amavasai@shu.ac.uk, K.Dutton@shu.ac.uk, Manuel.Boissenin@gmail.com

***Abstract*—Dynamically typed scripting languages have become popular in recent years. Although interpreted languages allow for substantial reduction of software development time, they are often rejected due to performance concerns.**

In this paper we present an extension for the programming language Ruby, called HornetsEye, which facilitates the development of real-time machine vision algorithms within Ruby. Apart from providing integration of crucial libraries for input and output, HornetsEye provides fast native implementations (compiled code) for a generic set of array operators. Different array operators were compared with equivalent implementations in C++. Not only was it possible to achieve comparable real-time performance, but also to exceed the efficiency of the C++ implementation in several cases.

Implementations of several algorithms were given to demonstrate how the array operators can be used to create concise implementations.

***Index Terms*—Computer Vision, Image Processing, Signal Processing**

I. Introduction

Machine vision is a broad field and in many cases there are several independent approaches solving a particular problem. Also, it is often difficult to preconceive which approach will yield the best results. The machine vision software can only be tested in a particular environment after the hardware platform to run it on is sufficiently developed and the software can be installed. Experience shows that - since hardware and software developers in a project often get to start and finish at the same time - it is important to preserve the agility of the software to be able to implement necessary changes in the final stages of a project.

This paper presents HornetsEye[1] which is an extension for Y. Matsumoto's programming language Ruby to facilitate rapid development of machine vision software. We have found that it is possible to provide a high amount of flexibility without sacrificing real-time capabilities. For example, in [1] the software library was used to implement the dual-tree complex wavelet transform.

The work presented in this paper was funded by the EPSRC Nanorobotics project. Furthermore, it benefits from the accumulated experience of developing machine vision software within the EU MINIMAN and EU MiCRoN project and from a continued effort to develop software which is useful beyond a single project. While development of this software was driven by the demands of projects in the micro- and nano-environment, we believe that the results presented in this paper apply to other environments as well.

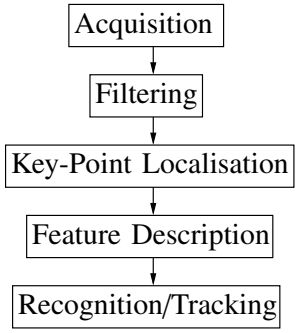

Figure 1. Overview of typical object recognition algorithm

Figure 1 shows an overview of a typical object recognition algorithm.

In chapter II the current state of the art is discussed and the approach of developing a Ruby-extension is justified. Chapter III presents optimized array operations. Chapter IV shows how different applications facilitated by these generic operations. Chapter V demonstrates the achieved performance. Chapter VI is the conclusion and VII is about future work.

II. State of the art

There are a number of active free and open source software projects in the area of machine vision. These include ITK, NASA Vision Workbench, OpenCV, OpenVidia, Camellia, PyGPU, and Gamera to name only a few. Machine vision systems require software for handling video and image files, accessing cameras, and visualizing results. To keep the size of the project manageable it is mandatory to make use of existing software projects.

Altough open source packages and libraries are available for free, integrating it requires significant time and effort[2].

*This work was supported by the Nanorobotics EPSRC Basic Technology grant GR/S85696/01

[1]Available at http://rubyforge.org/projects/hornetseye/

978-1-4244-2184-8/08/$25.00 © 2008 IEEE.

Furthermore with increasing size and adoption, software projects face the *evolution dilemma*[3]: As soon as there are multiple stakeholders, they require that it is a stable basis for development. But if making changes becomes too cumbersome, the project sooner or later will be superseded by a more progressive one.

The Ruby scripting language alleviates both problems. Ruby is a reflective, dynamically typed, object-oriented programming language. Furthermore Ruby uses code blocks as a unifying concept for loops, iterators, and function objects.

Dynamically typed languages have become popular in recent years. For example Camellia, PyGPU, and Gamera are all making use of dynamically typed languages (the other projects listed above are implemented in C/C++). Dynamically typed languages are not new and many features of Ruby already can be found in Smalltalk-80[4]. Projects such as SageMath are successfully using dynamic scripting languages to integrate a wide range of other projects into a large solution.

Integrating software in Ruby is easy because

- interfacing with native code for writing extensions is simple
- classes can still be modified after declaration
- Ruby uses duck-typing, *i.e.* two objects are compatible if they support the same methods and properties

The last feature also alleviates the evolution dilemma since changes do not propagate as far through the class-hierarchy as they do in statically typed languages.

It would be desirable to port all required software to Ruby so as to take full advantage of the language properties. However for input and output (*e.g.* capturing camera images and displaying videos) it is necessary to interface with native code. Furthermore it is necessary to implement computationally expensive parts of the code in C/C++ as long as there is no sufficiently strong run-time optimizer for Ruby.

III. Optimization

The quickest way to integrate an existing C/C++ library into Ruby is to use the bindings-generator of the SWIG[5] project. However simply making the static data types of a C/C++ library visible in Ruby is insufficient for fully exploiting the features of Ruby.

An array data type to handle multi-dimensional arrays with elements of a single type was implemented. It is heavily inspired by M. Tanaka's NArray. NArray provides fast element-wise operations combined with methods to manipulate single elements or subarrays. However in contrast to NArray our data type is largely implemented in Ruby and thus allows definition of custom element-types.

Ruby offers methods to pack numerical data into a platform-dependent binary representation. *E.g.* integers can be converted to bytes and later on be retrieved as follows

```
[1,2].pack("cc")          => "\001\002"
"\001\002".unpack("cc") => [1, 2]
```

Using this methods an array data type was implemented in Ruby which operates on binary data[1]. Similar as in the NArray implementation, array elements are only temporarily represented as Ruby objects. Because the many run-time checks make string objects in Ruby too slow for our purpose, a class named `Malloc` for storing raw data was added to the extension. An object of type `Malloc` is used by a class named `Sequence` for storing sequences of elements with same element-type.

In Ruby the the existence of a method with a certain name can be checked during run-time using the method `Object::respond_to?`. This can be used to develop a method which tries to invoke an efficient native implementation before falling back to using a slower generic implementation[1]. Fast native implementations for the following operations on number sequences were added to the extension

- accumulative operations: `min`, `max`, `sum`
- element-wise unary operations: `minus`, `abs`, `sqrt`, `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`, `acos`, `asin`, `atan`, type conversions
- element-wise binary operations: bitwise and, bitwise or, `atan2`, `plus`, `minus`, `div`, `mul`, `pow`, `clip`, `binarise`
- selecting/redistributing a subset of elements using a mask: `mask`, `unmask`
- element-wise application of a lookup table: `map`

Note that each native implementation needs to be instantiated for some or all element types. The basic element-types are

- 6 integer types: 8-,16-, and 32-bit, signed/unsigned
- 2 floating-point types: single/double precision
- 2 complex number types: single/double precision
- 8 red-green-blue triples: integer/floating-point

Furthermore binary operations need to be instantiated for several combinations of element-types and they appear as array-array operations, array-scalar operations, and scalar-array operations.

The native implementations of binary operations were instantiated and registered using the three template classes

- `WrapArrayArray< T1, T2, F >`
- `WrapElementArray< T1, T2, F >`
- `WrapArrayElement< T1, T2, F >`

where `T1` and `T2` are element-types and `F` is a function object wrapping a binary operation such as `_plus` or `_pow`. The different combinations were instantiated recursively by using the following template classes

- `WrapBinaryFirst< T2, F, O >`
- `WrapBinarySecond< W, F, O >`
- `WrapBinaryAll< F, O >`

were `O` is one of `WrapArrayArray`, `WrapArrayElement`, or `WrapElementArray`. In this case `W` always is `WrapBinaryFirst`. Note that in practise the implementation

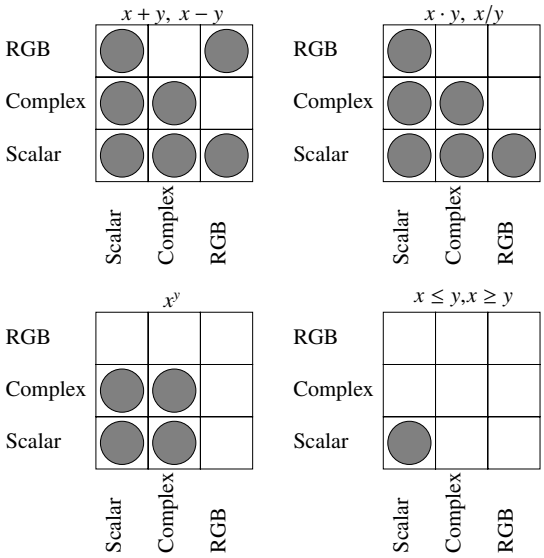

Figure 2. Binary operations for different element types

also needs to address the problem that binary operations usually only have a meaningful definition for some combinations of element-types as shown in fig. 2.

The remaining obstacle is that the native implementations needs to imitate the type coercions of Ruby at compile-time. These problems can be overcome by using template meta-programming techniques which were developed within the Boost project[6]. *E.g.* an entry of the default compile-time look-up table for return-types looks like this

```
template<>
struct _coercion< RGB< char >, double >
{
  typedef RGB< double > type;
};
```

The function object representing the binary plus operator for example makes use of the default compile-time look-up table as follows

```
template< typename T1, typename T2 >
struct _plus
{
  typedef typename _coercion< T1, T2 >::type
    result_type;
  result_type operator()( const T1 &x,
                          const T2 &y ) const
  { return (result_type)x + (result_type)y; }
};
```

Meta-programming techniques are also used to convert the C++ data type of the return value to a Ruby-identifier.

Using the class `Sequence`, a class named `MultiArray` was implemented which represents multidimensional arrays. In a similar fashion as for `Sequence`, native implementations for the following operations which are specific to multidimensional arrays were instantiated

- copying of sub-arrays
- convolutions with small filter kernels
- filling of sub-arrays

While within Ruby two objects supporting the same methods are compatible, this is not sufficient if a native library is expecting a certain data type. For this reason both NArray and RMagick provide methods for importing and exporting raw data in the form of string objects. Our implementation also offers raw data to be imported and exported using the methods `import` and `to_s`.

IV. Applications

Developing array operations as shown in the previous chapter is sophisticated and time consuming. We shall demonstrate however that due to its generic nature the current implementation already facilitates concise, real-time implementations of various algorithms.

A. Filtering

Both the Harris-Stephens combined corner- and edge-detector[7] as well as the Kanade-Lucas-Tomasi corner-detector[8] are based on the eigenvalues of the covariance matrix of gradient vectors taken from a local region of the image. Given a two-dimensional array of floating-point values `img`, that represents a grey-level image, the following code computes the Sobel gradient and then the values of the symmetric covariance matrix for each pixel. A Gaussian blur filter is used to accumulate the gradient values over a local region of the image. Finally the trace and determinant of the covariance matrix are computed for each pixel

```
sigma = 1
x = img.sobel_x
y = img.sobel_y
cxx = ( x ** 2 ).gauss_blur( sigma )
cyy = ( y ** 2 ).gauss_blur( sigma )
cxy = ( x * y  ).gauss_blur( sigma )
tr = cxx + cyy
det = cxx * cyy - cxy * cxy
```

The Harris-Stephens corner- and edge-detector uses a heuristic value based on the trace and determinant of the local covariance matrix[7] and can be computed for each pixel as follows

```
k = 0.1
result = det - tr * tr * k
```

The Kanade-Lucas-Tomasi corner-detector simply uses the smallest eigenvalue as a heuristic[8]. The smallest eigenvalue of a 2×2 matrix can be computed for each pixel using the trace and determinant as follows

```
dissqrt = ( tr ** 2 - det * 4 ).
          clip_lower( 0.0 ).sqrt
result = 0.5 * ( tr - dissqrt )
```

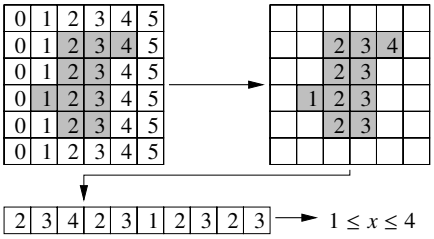

Figure 3. Masking an x-ramp

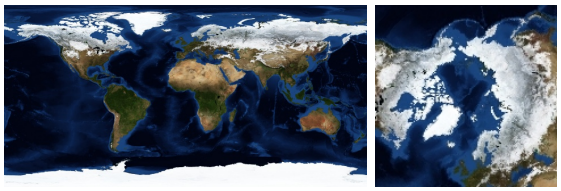

Figure 4. Warping images

The invocation of `clip_lower` is required to deal with minor numerical deviations leading to negative values.

B. Bounding box

This section introduces x-ramps, y-ramps, and masking operations and gives an example on how they can be used to compute a bounding box.

A two-dimensional array `img` of integer values representing a grey-level image is given. The array is then binarized by thresholding (*i.e.* applying a step function to each element)

```
mask = 1 - img.binarise( 128 )
```

To find the bounding box with C/C++, one would usually use a nested loop which increments a pointer to the data and the body of the loop would update the parameters of the bounding box when a pixel of the mask is encountered. Implementing this in Ruby is not an option if there are real-time constraints. However using x-ramps and y-ramps which were inspired by the visualizations in [9] one can solve this problem using array operations. An x-ramp for example can be created using the following method

```
def xramp( *shape )
  retval = MultiArray.new( MultiArray::LINT,
           *shape )
  for x in 0...shape[0]
    retval[ x, 0...shape[1] ] = x
  end
  retval
end
```

By selecting only the pixel of the x-ramp which are indicated by the mask one obtains an array of which the minimum and maximum are the lower and upper x-coordinates of the bounding box (see fig. 3).

The Ruby code is as follows

```
x = xramp( *mask.shape )
y = yramp( *mask.shape )
box = [ x.mask( mask ).range, y.mask( mask ).range ]
```

C. Warping images

This section introduces warps. The function `map` together with the x- and y-ramps can be used to implement a function for warping images. The input image is warped using a three-dimensional array `v` of size $w \times h \times 2$ containing the warp vectors. The warp vectors are computed by applying array operations to x- and y-ramps. The following code warps an equirectangular projection on an azimuthal projection

```
w, h = img.shape[0], img.shape[1] / 2
v = MultiArray.new( MultiArray::LINT, h, h, 2 )
x = xramp( h, h )
y = yramp( h, h )
c = 0.5 * h
v[ 0...h, 0...h, 0 ] =
  ( ( ( x - c ).atan2( y - c ) / PI + 1 ) *
    w / 2 - 0.5 )
v[ 0...h, 0...h, 1 ] =
  ( ( x - c ) ** 2 + ( y - c ) ** 2 ).sqrt
result = img.warp_clipped( v )
```

The input image[2] and result are shown in fig. 4

D. Lucas-Kanade Tracker

In this final example we want to use all techniques introduced in the previous sections to make a concise implementation of the inverse compositional Lucas-Kanade tracker[9]. The Lucas-Kanade algorithm tracks an object by warping the input image on a template image and iteratively updating the transformation parameters to reduce the difference between the template and the warped image. In this example we are modelling two-dimensional translations and rotations as shown in the following equation.

$$W(\vec{x};\vec{p}) = \begin{pmatrix} x\cos(p_3) - y\sin(p_3) + p_1 \\ x\sin(p_3) + y\cos(p_3) + p_2 \end{pmatrix} \quad (1)$$

The input image and the template are given as two-dimensional floating-point arrays `img` and `tpl`. Furthermore a vector `p` with the initial parameters of the model is required. Then the gradient of the template (`gx` and `gy`), the product of the Jacobian and the gradients (`c`), and the Hessian (`hs`) are initialized as follows

```
# p = Vector[ ?, ?, ? ]
w, h = *tpl.shape
x, y = xramp( w, h ), yramp( w, h )
sigma = 5.0
gx = tpl.gauss_gradient_x( sigma )
gy = tpl.gauss_gradient_y( sigma )
c = Matrix[ [ 1, 0 ], [ 0, 1 ], [ -y, x ] ] *
    Vector[ gx, gy ]
hs = ( c * c.covector ).collect { |e| e.sum }
```

[2]Image source: NASA Visible Earth project

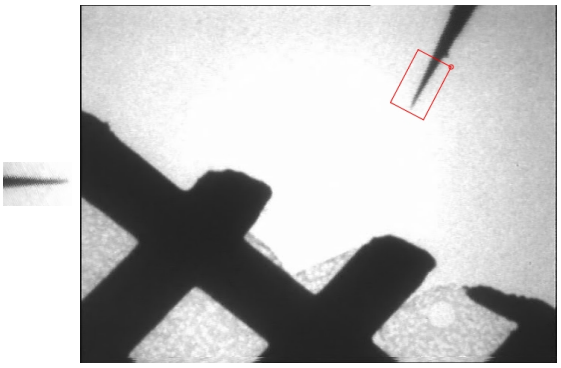

Figure 5. Tracking template (left) and input image (right)

Table I
PERFOMANCE OF LUCAS-KANADE IMPLEMENTATION

load image	track (5 iterations)	display	**total**
26 ms	128 ms	137 ms	290 ms

After that a new image `img` is acquired and the parameters of the model are updated a number of times until the error is sufficiently small. This requires computing the difference between the warped image and the template. The difference is used to estimate the change of the parameters using the Hessian and the product of Jacobian and gradients[9]. The code for performing a single update follows here

```
field = MultiArray.new( MultiArray::LINT, w, h, 2 )
field[ 0...w, 0...h, 0 ] =
  x * cos( p[2] ) - y * sin( p[2] ) + p[0]
field[ 0...w, 0...h, 1 ] =
  x * sin( p[2] ) + y * cos( p[2] ) + p[1]
diff = img.warp_clipped( field ).
  to_type( MultiArray::SFLOAT ) - tpl
s = c.collect { |e| ( e * diff ).sum }
d = hs.inverse * s
p += Matrix[ [  cos(p[2]), -sin(p[2]), 0 ],
             [  sin(p[2]),  cos(p[2]), 0 ],
             [          0,          0, 1 ] ] * d
```

Fig. 5 shows a video frame of a nano-indenter operating in a transmission electron microscope[3]. The tracking algorithm is able to track the object if the nano-indenter moves with a limited speed. The video was processed for demonstration purposes only with no specific application in mind. Table I shows the time the implementation requires to process a frame (AMD Athlon, 64-bit, 2.2 GHz, 1 GByte RAM, GCC version 4.1.3, Ruby version 1.8.6). The images have a size of 640×480 pixel and the size of the tracking template is 80×50 pixel.

V. RESULTS

Table (II) shows the required processing time in milliseconds for performing equivalent operations with Mimas (the C++ library we developed in the MINIMAN and MiCRoN project), NArray, and HornetsEye. Different operations on a 1000×1000 single-precision floating-point array were executed 1000 times and the average time was taken.

[3]Images courtesy of Sheffield University Nanorobotics Group

Table II
SPEED COMPARISON OF EQUIVALENT OPERATIONS

	Mimas/Boost	**NArray**	**HornetsEye**
constructor	2.7 ms	16.2 ms	17.8 ms
m.fill(1)	2.7 ms	17.8 ms	17.9 ms
m*m	6.8 ms	19.0 ms	19.3 ms
m*2	6.7 ms	19.0 ms	19.4 ms
subarray	3.0 ms	16.2 ms	18.1 ms

Table III
SPEED COMPARISON WITH WEAKLY CONSTRAINED GARBAGE COLLECTOR

	Mimas/Boost	**NArray**	**HornetsEye**
constructor	2.7 ms	8.4 ms	7.8 ms
m.fill(1)	2.7 ms	2.7 ms	2.8 ms
m*m	6.8 ms	10.0 ms	8.1 ms
m*2	6.7 ms	8.9 ms	7.2 ms
subarray	3.0 ms	2.2 ms	3.7 ms

The C++ library seems to be much faster when copying arrays or when filling them with a value is required. This is probably due to the fact that neither NArray nor HornetsEye are currently making use of the highly optimized routines of the C++ standard template library. However to ensure that equivalent operations are measured in Ruby and C++, the mark-and-sweep garbage collector of Ruby was invoked after every command to force destruction of the result as it happens in C++. This puts Ruby at a disadvantage.

Table (III) shows how the results change if the garbage collector of Ruby is only invoked once at the end of a measurement loop.

In this case NArray outperforms the C++ implementation in some cases. As expected HornetsEye requires slightly more processing time than NArray since it is not implemented solely in C++. We have not yet found the reason why our implementation for copying subarrays is significantly slower than the one of NArray.

Figure 6 shows the average processing time for multiplying one-dimensional arrays of different sizes with a scalar. The C++ implementation was compared with HornetsEye's and NArray's. For different array sizes the multiplication was performed 100 times with a weakly constrained garbage collector. For small sizes the C++ implementation is much faster than both Ruby implementations. This is due to the fact that the array manipulations in Ruby and the garbage collector have a larger overhead. For larger arrays the benefits of the garbage collector become dominant. The sudden change in slope of all curves at a certain array size can be attributed to the memory cache of the CPU. For bigger arrays HornetsEye is the most efficient implementation.

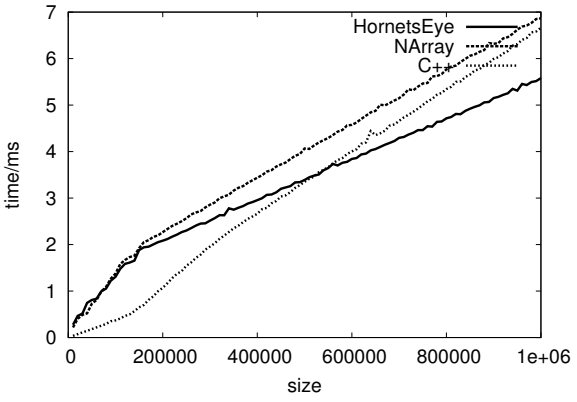

Figure 6. Speed comparison for array-scalar multiplication

VI. Conclusion

A Ruby-extension providing native optimized implementations of generic array manipulation methods for machine vision was presented. The Ruby-extension already facilitates implementation of various machine vision algorithms. We have shown that no major compromise in speed is required for adopting a dynamically typed language for machine vision algorithms, where the major part of the processing time is used for basic operations on arrays. We believe that this is the case for most machine vision algorithms. We are not aware of any other free and open source software package for machine vision which combines speed and flexibility in an equal way.

A drawback is increased memory requirements. The implementations require intermediate results to be stored when a sequence of element-wise operations is performed. This could pose a problem for embedded platforms. The large number of instantiated array operations for different element-types also leads to a large shared library (currently 9.1 MByte).

Contrary to common belief, an interpreted language can be faster than a static implementation. Table III shows that the garbage collector of Ruby can be faster than the static memory management of a naive C++ implementation.

In a similar way, as manually optimizing assembler programs largely became redundant after optimizing compilers reached maturity, the labor of supporting interpreted languages with native extensions for the mere purpose of increasing performance could become unnecessary if interpreters with sufficiently strong run-time optimization would become available.

VII. Future Work

Possible future work is the integration of GPU (graphical processing unit) operations and the use of parallel processing. This could be done in the same way as the current native implementations are integrated. Further opportunities for optimization are loop-ordering and -merging for operations on subarrays. Also the software is currently only tested by running a set of examples after compilation. An improvement would be to use the unit-testing library of Ruby. Also a number of I/O facilities still need to be integrated under GNU/Linux as well as Microsoft Windows Finally implementations of a feature-based object-recognition and -tracking algorithm are planned.

References

[1] J. Wedekind, B. Amavasai, and K. Dutton, "Steerable filters generated with the hypercomplex dual-tree wavelet transform," in *IEEE International Conference on Signal Processing and Communications*, 2007, pp. 1291–1294.

[2] Zaljko Obrenovic and Dragan Gasevic, "Open source software: All you do is put it together," *IEEE Software*, vol. 24, no. 5, pp. 86–95, 2007.

[3] Marcus Denker and Stephane Ducasse, "Software evolution from the field. an experience report from the Squeak maintainers," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 81–91, 2007.

[4] Laurence Tratt and Roel Wuyts, "Guest editors' introduction: Dynamically typed languages," *IEEE Software*, vol. 24, no. 5, pp. 28–30, 2007.

[5] D. M. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599–609, July 2003.

[6] A. Gurtovoy and D. Abrahams, "The Boost C++ metaprogramming library," Tech. Rep., Mar. 2002, http://www.boost.org/libs/mpl/doc/paper/mpl_paper.pdf.

[7] C. G. Harris and M. Stephens, "A combined corner and edge detector," *Proceedings 4th Alvey Vision Conference*, pp. 147–151, 1988.

[8] Jianbo Shi and Carlo Tomasi, "Good features to track," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.

[9] S. Baker and I. Matthew, "Lucas-kanade 20 years on: a unifying framework," *International Journal of Computer Vision*, vol. 56, no. 3, pp. 221–55, Feb. 2004.