MediaMath: A Research Environment for Computer Vision

Minas E. Spetsakis

Dept. of Computer Science
York University
4700 Keele Street
North York, ONTARIO
CANADA, M3J 1P3
tel. (416) 736-2100 ext 77886, FAX: (416) 736-5872, e-mail minas@cs.yorku.ca

ABSTRACT

This paper describes MediaMath an object oriented environment for computer vision. MediaMath is an interactive, extendible interpreter for algorithm development and rapid prototyping. It contains a wide variety of vision algorithms and a comprehensive set of fundamental image operations. It provides a syntax idealy suited for translating mathematics to working programs that run efficiently. Examples of algorithms developed in MediaMath and samples of results are included.

The support of the NSERC (App. No. OGP0046645) is gratefully acknowledged.

1. Introduction

MediaMath is a software tool for research and teaching in a variety of areas like Computer Vision, Image Processing, Signal Processing, Robotics, Multimedia etc. It provides facilities for manipulation of most data structures that are generally used in these areas. Its target audience is the research and teaching community in the above areas. The user is assumed to be proficient in programming and comfortable with modern software tools. This user can use the system to implement algorithms in a small fraction of the time needed in plain C. The algorithms that have a mathematical form are particularly well suited for this purpose. This is not however a tool for interactive image editing or of any use to the non-programmer. MediaMath will be released for use free, sometime during the summer of 94.

The paper is organized in the following chapters: MediaMath Functionality, A brief History of MediaMath, MediaMath as Programming Language, Syntax of the Language, Linking to C, Examples of MediaMath algorithm development, Object System, Design Philosophy, Future of MediaMath, Conclusions.

2. MediaMath Functionality

MediaMath is a stand alone interpreter for the MediaMath language. It is capable of general programming but it was designed right from the beginning for image, audio and matrix manipulation. The most important features of the language include:

- C syntax. The syntax of the language is very similar to the syntax of C, and it is capable of general programming.

- Fully dynamic. Anything can change or be redefined at runtime. In fact everything *is* defined at runtime by reading a set of files from the default directory when starting up.
- Functional programming. MediaMath is a functional programming language like lisp, yet it fully supports imperative style.
- It has object oriented features like multimethods operator overloading etc.
- Rich set of image, vector and matrix operators. Several vision and robotics algorithms have already been implemented.
- Flexible computational model. The programmer can mix the highest level operations with the most primitive ones, or can tinker with the MediaMath engine internals. So the user not familiar with one programming style can use another.
- Dynamic linking. New functions written in MediaMath can just be typed in or loaded. New primitives written in C can be linked, unlinked, fixed, recompiled and relinked at run time without any negative side effects. In most cases the system can recover even after severe errors like segmentation violation.
- Memory management. The user does not have to worry what happens to large data structures after they cease being useful. The garbage collector will pick them up when they are no longer accessible. Unless the user needs many and big images or matrices on a small computer, freeing is not a concern.
- Small runtime size. MediaMath needs about 3MB to start and depending on the number of images used may need more memory.

- User interface. There are Emacs utilities that allow one to run MediaMath through emacs and send function definitions and statements across buffers. It also provides on-line help for any function.
- Software tools to write and link C programs to MediaMath. These take care of the tedious work like putting functions on the hash table, provide safety features for linking and unlinking etc.
- Connectivity. Provides facilities to communicate with other programs, like xv and gnuplot, and libraries, like pbmplus and numerical libraries, either by dynamic linking or by spawning a new process

All these make MediaMath an ideal tool to prototype computer vision and related algorithms of all kinds: flow estimation, structure from motion, image enhancement, robot navigation etc. Algorithms that are expressed in a clear mathematical way can be implemented debugged and optimized in a couple of hours (Arun *et al* 3-D point fitting, Horn and Schunck optic flow computation, Tsai and Huang structure from motion, Spetsakis and Aloimonos optimal structure from motion etc. None of them took more than an afternoon) [1, 8, 18, 20]. In addition MediaMath provides an excellent alternative to raw C programming because it has the same power but is interactive and incremental.

3. A brief History of MediaMath

The development of MediaMath started in July 1992. All the development is done in C and MediaMath's own language. Other languages were also used for various subprojects (awk for header manipulation, emacslisp for emacs interface, lex and yacc for

parsing etc). The core of the interpreter is very similar to a lisp interpreter but designed to be fast. In the initial stage it had also a lisp like syntax. This was soon abandoned because it made the translation of simple mathematical formulae to prefix difficult. A C like syntax was designed with some minor differences from C as explained below. The software tools for the development of MediaMath were implemented first. They are mostly written in awk. They started as simple 10 line awk scripts but have grown to about 500 lines.

Half way through the development there was some hesitation as to whether an object system was needed. But after counting the number of different versions of multiplications, additions, convolutions and concatenations, it was evident that an object system with multimethods would be useful. Otherwise the user would have to remember about 2000 function names (or at least remember the naming conventions) to use the above operations. So a simple object system was added. Other object oriented features are being planned and will be introduced as needed to avoid overdesign.

The on line help was also developed early on. Every function or variable can have a documentation string that acts as comment on the program listing and on line documentation at run time. There is also an apropos command that understands standard Unix regular expressions.

The first usable version of MediaMath was ready late August 1993. It was first used in the fourth year Computer Vision course at York. The students had an option to use MediaMath or plain C and none used C (the MediaMath sources were available, to scavenge library routines). No bug has been reported so far in the main functionality. It was

also used in the Intro to Robotics course to control a robot arm and a tethered moving robotic platform. Currently MediaMath is entering the Beta phase and will be released sometime during the summer of 1994.

4. MediaMath as Programming Language

MediaMath is designed to be easy to use and to look familiar to the user from the first encounter. So it uses the familiar syntax of C (with some omissions and extensions). It is an interpreted language, like Basic and Lisp. This means that you can give it an expression and get the answer right away. This is invaluable for incremental debugging. It is extensible in more than one ways: by defining functions that can be called seamlessly from the interpreter or by including new primitives in C.

The fact that it is an interactive language means that one can execute statements interactively by just typing them in

```
a=3.14;
b=a*2;
c=sin(b);
b=to_integer(b);
```

The result of every statement is printed. Any executable statement can be executed interactively. And when we load a file, MediaMath pretends that it was typed in. If an error occurs then a helpful message is printed and we go back to the top level interpreter. We can see the history of function evaluations that led to the error by typing

```
BackTraceOld;
```

The variable *BackTraceOld* holds the backtrace of the previous error. It is a list of all the active function invocations.

5. Syntax of the Language

The syntax is the same as C with few exceptions, to accommodate the different features of the languages. These features are:

- *Functional programming*. MediaMath is a purely functional language so everything returns something, so we can have statements like *3 + while (a>0) a-=5*.
- *Soft typing*. Variables are not bound to types. So we can have statements like *a=3;a="three";*. And of course there are no type declarations for variables. But we can define new types with the *struct* keyword.
- *Objects*. MediaMath has some object oriented features like multi argument dispatch methods, so it has the syntax to deal with them.
- *Variable number of arguments*. We can define functions with variable number of arguments and keywords. One can use *make_complex(3.1,4.2);* or *make_complex(:im=3.1,:re=4.2);*.
- *Image and matrix operators*. There is a variety of operators to use for image and matrix manipulations like *a1 (*) s_templ;* or *m1<->m2;* to convolve an image with a template and concatenate two matrices.
- *No inherited mistakes*. There was no reason to repeat some of the mistakes that happened with C, like the use of caret for XOR, the lack of a power operator to do things like 3^2 and the funny way to access 2-d arrays with a[3][4].

Other than that the two languages have the same syntax.

The MediaMath specific operations are

`^` The power operator. The type of the operands can be any number: integer, unsigned integer, character, unsigned character or float. Depending on the type the interpreter will call the appropriate library function to do the job.

`(*)` The 2-D convolution operator. One operand must be an image or a scan line and the other a template or both templates.

`(|)` Vertical convolution. When the template is one dimensional it is treated as a column. This is useful for separable convolutions.

`(-)` Horizontal convolution. As above but the template is treated as a row.

`<|>` Vertical concatenation. Two objects like matrices images etc are stacked one on top of the other.

`<->` Horizontal concatenation. As above, but the concatenation is sideways.

`^T` Transpose. Matrix transpose operator.

`'` Quote. Protects its argument from evaluation. It returns the expression unevaluated. This is an advanced feature.

`..` Range. It returns a data type that indicates range. It is used for subarray access and some other functions that understand it.

6. Libraries and Modules

There are several libraries that can be loaded to MediaMath (some of them are loaded by default). These are the Image and Matrix Library, the Image Input Output, the List library, the Motion library, the Robot Arm, the Moving Platform, the Halftoning and Color Separation etc. We analyze the most important ones below.

6.1. Image and matrix library

This library introduces a set of types to operate upon, that can represent images and matrices. There 18 such types but a bit of taxonomy makes them look fewer and easy to understand.

These types are of two kinds. One dimensional and two dimensional. The 1-D ones are vectors *vec*, scan lines *scln* and 1-D templates *tmpl*. The 2-D ones are matrices *mat*, images *img* and 2-D templates *tmpl2*. So far six. Each one of them can be either float *f*, integer *i* or unsigned character *uc*. The names of all of them are composed by the initials of the underlying type and the short name of the type:

fvec	*fscln*	*ftmpl*
fmat	*fimg*	*ftmpl2*
ivec	*iscln*	*itmpl*
imat	*iimg*	*itmpl2*
ucvec	*ucscln*	*uctmpl*
ucmat	*ucimg*	*uctmpl2*

For every one of these types there is a routine to create an instance. We just prefix the name of the type with *mk_* as in *mk_fvec*.

The indices of matrix and vector objects of size *n* range from *1..n*. The range can be specified using the range operator *..* or just the size. Images and scan lines are pretty similar to matrices and vectors but start at *0*. The range of templates can be anything, so they are not constrained to start from zero or one. The central element of the template is the *0* or *0,0*. For example:

```
mk_fvec(1..4,[1,2,3.3,4]);
mk_ucmat(2,3,[[1,2],[3,4],[5,6]]);
mk_ucmat(1..2,1..3,[[1,2],[3,4],[-5,-6]]);
mk_fimg(128,128);
mk_fimg(0..1,0..1,[[1,2],[1,2]]);
mk_ftmpl2(-1..1,-2..2);
mk_ftmpl(-3,3,[1,1,1,0,2,2,2]);
```

One need not be concerned with deallocation of these data structures, although they are big, because there is an efficient garbage collector to do that. The garbage collector scans the memory every now and then and finds data that are inaccessible and deallocates them. There are functions available to deallocate them at will.

The pixels of the images or the elements of the matrices can be accessed with array reference syntax *a2[3,3]*. If we replace the integers with a range we get the corresponding part of the structure *a2[2..5,4];* or *a2[2..4,2..4];*.

Various attributes of an image or matrix can be accessed with the field access operator *->* like *a1->vmin;* or *a2->hsize;*. The initial *v* stands for the vertical dimension and *h* for the horizontal.

Any two structures can be concatenated side by side or one on top of the other with the `<->` and `<|>` operators respectively.

The package also provides generalized vectors and matrices *gvec* and *gmat*. Every element of these can be any data structure: image, matrix, float or even generalized vector or matrix. This is very useful in solving differential equations in vision where one needs to represent several derivatives of the images or several filtered versions of a number of images.

Most of the operators that apply to numbers apply to images like *a *= b + 3* where *b* can be either number or image. All image and matrix operations are done in floating point.

There are several standard numerical routines in MediaMath for matrix and image manipulation that range from SVD to histogram equalization to Fourier transform.

6.1.1. Convolutions

There are three convolution operators. The general convolution operator *(*)* which applies to images and templates when there is no ambiguity of the orientation of these data structures. The horizontal convolution *(-)* that means that the 1-D template is horizontal and the vertical *(|)* that means that the 1-D template is vertical. The reason to have one dimensional and two dimensional templates is to do separable convolutions faster. MediaMath also implements semi separable templates (e.g. templates that can be decomposed to a sum of separable templates, using SVD). The syntax is exactly the same.

6.2. List processing

The core of the MediaMath interpreter is very similar to a Lisp interpreter. Therefore the list library comes for free. There are over 50 such functions implemented to handle the usual Lisp data structures.

6.3. Image input output

The “preferred” format for image I/O is the PGM (Portable Gray Map). MediaMath links to the *pbmplus* library for this purpose. It also reads and writes raw format and can output postscript. A few more formats will be supported in the near future (TIFF, JPEG and GIF).

6.4. Motion library

There is a quite extensive set of routines to do structure from motion and flow estimation. Along with them comes a rich set of routines to generate synthetic images, display the results (e.g. needlemaps for flow estimation), compare the results of the algorithms with the ground truth to estimate the noise performance etc.

7. Linking to C

The MediaMath system provides a simple mechanism to extend its functionality by adding functions written in C. The mechanism is fundamental to the system and not an extra feature. It is used by the developers to write everything except the central part of the interpreter. As such it is extensively debugged and tested during the development. It does not incur any speed penalty and it is intended to be simple and flexible, and easy to

```
function H_S_flow(im1, im2,
              &optional lam &init 2.0,
              lam0 &init 0.01,
              sig &init 2.0,
              maxits &init 16,
              uv,
              solver &init jacobi,
              tol &init 1.0e-2,
              show)
     "Computes the optic flow..."
{
  local u, v, d1, d2, Ex, Ey, Et, ExEx,
        ExEy, EyEy, gt, A, b, x, imh, xv;

      /* Some templates that are useful */
  d1 = mk_der_tmpl(:order=3,:limit=3);
  d2 = mk_der2_tmpl(:order=3,:limit=3);
  gt = mk_gauss_tmpl(sig);
      /* If we are provided with a good guess */
  if (uv)
    {
      u=round_fimg(uv->u); v=round_fimg(uv->v);
      Ex = D_x(im1,d1)+intwarp_fimg(D_x(im2,d1),u,v);
      Ey = D_y(im1,d1)+intwarp_fimg(D_y(im2,d1),u,v);
      Ex *= 0.5; Ey *= 0.5;
      Et=im2-intwarp_fimg(im1,u,v)-Ex*u-Ey*v;
    }
  else                          /* No guess */
    {
      imh= (im1+im2)/2.0;
      Ex=D_x(imh,d1); Ey=D_y(imh,d1);
      Et=im2-im1;
    };
      /* Create the implicit matrix */
  A=make_H_S_mat(:d2  =lam*d2,
         :ExEx=ExEx=(Ex*Ex)(*)gt+lam0,
         :ExEy=ExEy=(Ex*Ey)(*)gt,
         :EyEy=EyEy=(Ey*Ey)(*)gt+lam0,
         :B1  =-2*lam*d2[0] + ExEx,
         :B2  =ExEy,
         :B4  =-2*lam*d2[0] + EyEy,
         :gt=mk_LP_tmpl(:sigma=2.0,:reduce=2));
  b=make_field_2d(:u=-(Ex*Et) (*)gt,
              :v=-(Ey*Et) (*)gt);
  solver(A,b,:maxits=maxits,:x=uv,:tol=tol,
             :show=show);
};
```

Figure 1a. The object oriented version of the Horn and Schunck algorithm with some improvements.

master for somebody familiar with the UNIX™ system.

Modules written in C can be linked and unlinked to the executable at run time by a simple command, e.g. if the module is named “hough.so” you can load it with the command

```
dlopen("hough.so");
```

If for some reason you want to unlink “hough.so” then

```
dlclose("hough.so");
```

is enough. You rarely need this however. If while using “hough.so” you discover a bug you can go back to the C code of “hough.so” correct the bug and link it again in the same manner. The old version is automaticly unlinked and you will notice no side effect. This comes in handy when you discover a bug and you already have a couple of hours useful work in the memory.

The C routines have to be written in a certain style in order to be linked. This is a set of simple conventions.

8. Examples of MediaMath Algorithm Development

Now we have a look at some MediaMath examples that demonstrate the language. We do not describe the syntax in detail but a reader with knowledge of C can follow the code.

8.1. Optic Flow using MediaMath

Optic flow algorithms usually need sophisticated image processing rather than matrix manipulation. Fig. 1a. shows the code for Horn and Schunck algorithm that uses

by default Jacobi iteration with 2×2 block diagonal preconditioner. When called with a key argument `:solver = ConGrad` uses Conjugate Gradient instead, which works much better. The Horn and Schunck algorithm is notoriously tricky to make it work, but MediaMath allows one to create test images that exactly fit the assumptions and makes it easy to modify and rerun. After very little experimentation the following modifications improved the performance considerably: combine it with the Lucas and Kanade algorithm [11, 5] (convolve the normal coefficients `ExEx` etc with a gaussian), use more accurate derivatives [2], use a guess if available and some other minor improvements.

The first part of the code in Fig. 1a. generates very accurate templates for the first and second derivatives and a Gaussian template. Convolution with these derivative templates gives the derivative of a sixth degree polynomial fitted to the image at every seven

```
generic gmult
{
  H_S_mat, field_2d : 'H_S_mat_mult;
};

function H_S_mat_mult(M,x)
     "Multiplies M (an implicit matrix...
{
  local u,v, d2, ExEx, ExEy, EyEy, rhou, rhov;

  d2=M->d2; ExEx=M->ExEx; ExEy=M->ExEy; EyEy=M->EyEy;
  u=x->u; v=x->v;
  rhou = -(u(|)d2 + u(-)d2) + (ExEx*u+ExEy*v);
  rhov = -(v(|)d2 + v(-)d2) + (ExEy*u+EyEy*v);
  make_field_2d(:u=rhou, :v=rhov);
};
```

Figure 1b. The multiplication operation for the implicit matrix and vector for the variant Horn and Schunck algorithm. The actual multiplication is done by `H_S_mat_mult`.

point region [10]. The next few lines compute the space and time derivatives and if there is a guess they take it into account to warp the second image. The warping is done with whole pixel displacements to save time and for this reason a correction is added. The next segment of the code constructs the implicit matrix `A` and the implicit vector of knowns `b` and gives them to the solver routine to find an x such that $Ax = b$. The multiplication operation is defined for the implicit matrices of type `H_S_mat` like `A` and vectors of type `field_2d` like `b` as shown in Fig. 1b. If the keyword `show` has the value of a function then at each iteration this function is called with the intermediate results as argument. By providing a suitable function `show` that displays on the screen, one can animate the iteration. This has proven invaluable in many respects.

The most important thing in the author's opinion, though, is the amount of time taken by the author to write this routine: less than two hours, including the time to generate the synthetic images. And then another couple of hours to write the object oriented version. Fig. 1 includes a few other minor improvements added since then.

8.2. Absolute orientation problem

The absolute orientation problem is: find the rotation and translation that brings one set of points as close as possible to another set of points. The solution using MediaMath is in Fig. 2 (the algorithm is in [1]). The input is two $3 \times$ `pointnum` matrices each column of which is a 3-D point. The points in the second matrix are the same as the points in the first but translated and rotated. The algorithm is adequately explained by the in line comments.

```
function absolute_orien(ps1, ps2)
     "absolute_orien return the...
{
  local psb1, psb2, ones, U, V, sv, pointnum;

  pointnum = ps1->hsize;
  ones = mk_fvec(pointnum)+pointnum^-0.5;
            /* Compute the averages psb1 and psb2 of the */
            /* two sets of points.                       */
  psb1 = ps1*ones;  psb2 = ps2*ones;
            /* Subtract the averages from every point    */
  q1 = ps1 - psb1*ones^T;
  q2 = ps2 - psb2*ones^T;
            /* Compute the sum of the outer products     */
  Q = q1*q2^T;
  sv = SVD(Q);
  U=sv[1]; V=sv[3];

  make_orien(:R=V*U^T,
             :T=(psb2-R*psb1)/sqrt(pointnum));
};
```

Figure 2. The solution to the absolute orientation problem.

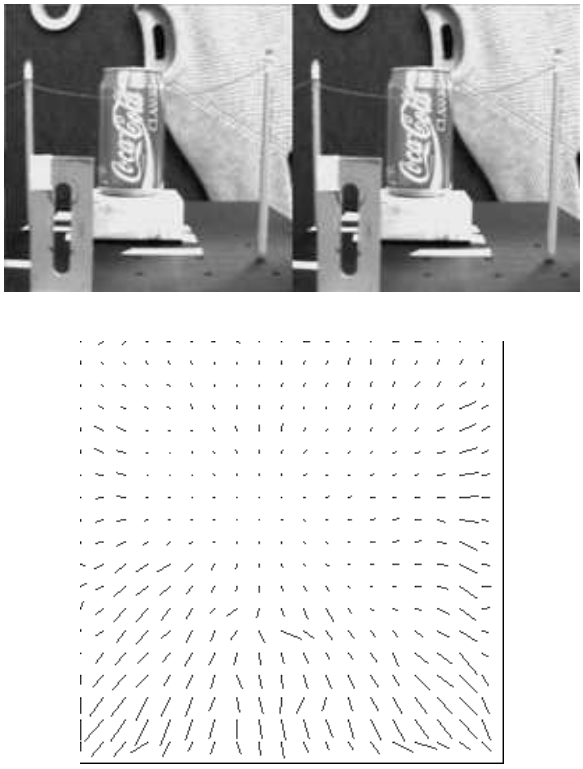


Figure 3. An image pair from the NASA sequence and the flow field as computed by the algorithm in Fig. 1. The camera moves towards the center of the can and the maximum flow is about 4 pixels (these are the first and fifth images of the sequence)

8.3. Numerical Analysis with MediaMath

Numerical analysis is integral part of computer vision, signal processing and robotics. MediaMath has two ways of doing numerical analysis. One is to call standard numerical routines to do diagonalizations, LU factorization, Fourier analysis etc. Most of these routines are already available to MediaMath. The other way is to provide the tools to write numerical routines easily and using the object system make them work on any kind of unusual input, like the implicit matrix in the flow algorithm above. To see how this is done, we write a routine to solve a system of linear equations using Jacobi.

Conjugate Gradient is as easy to write by modifying the syntax of routines in standard texts like [19, 6].

If the matrix is a regular matrix no definitions of methods are needed since they are provided by default. To be used for specialized matrices one has to define the matrix-vector multiplication, the norm of a vector etc (Fig. 5). The `show` optional variable if specified, is a function that outputs the intermediate solutions. If the solution is a vector field as in the flow algorithm above then `show` will display it on an *xv* window.

The above implementation of the Jacobi iteration is the simplest possible iterative way to solve a linear system. It does not always work and it is not the fastest. But the fact that the actual algorithm can be written in so few lines, helps it survive in textbooks. A practical algorithm for the same thing is the conjugate gradient which is also part of MediaMath.

9. Object System

One of the best ways to try one's ideas is to sit in front of an interactive system and test them. This is one of the reasons that Lisp became popular, that so many people are dependent upon Maple, that Emacs is the most popular editor in the Unix world. The advantages of incremental development are enormous. But the limits of complexity the user can handle are tighter than in a batch system. Consider for example the operation of multiplication. It makes intuitive sense to have all multiplications handled by the same operator, which can be done easily for the primitive types. But in the course of development of a system to do image manipulation, several new types of data structures are

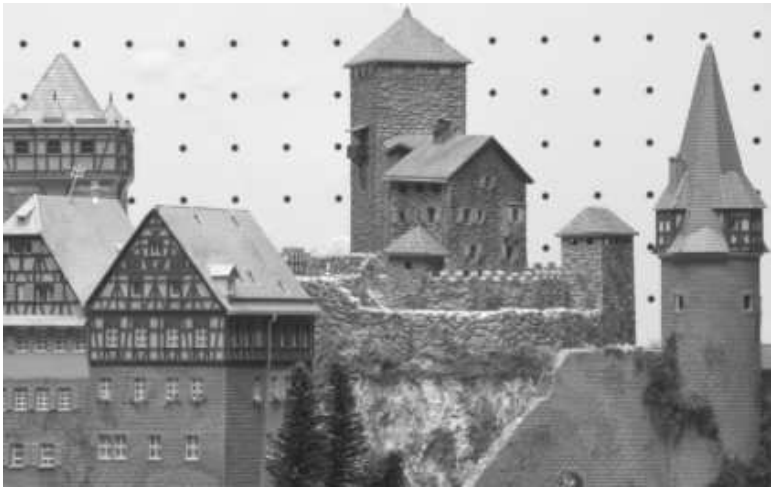


Figure 4. An image from the “castle sequence” from CIL at CMU and the depth map as computed by the structure from motion algorithm in [17] implemented in MediaMath. Dark areas are close to the camera. The flow varies between 20 and 30 pixels.

```
function jacobi(A,b,
            &optional x,
            tol &init 1.0e-2,
            maxits &init 64,
            show)
     "jacobi finds..."
{
  local i, dx, D;

  if (!x) x=b-b;
  D = ginverse(A->diagonal);
  if (show) show(nil);
  x -= dx = ( D * (A*x - b) );
  for (i=1; i<=maxits && gnorm(dx)>tol; i++)
    {
      x -= (dx = D * (A*x-b));
      if (show) show(x);
    };
  x;
};
```

Figure 5. The object oriented version of the Jacobi iteration.

introduced and quite a few of them should be multipliable among themselves or with other primitive types. Then a new operator should be introduced and pretty soon the user has to remember dozens of function names just for multiplication. Another case that the complexity is the limit is the solution of sparse linear systems. While the majority the linear systems in vision can be solved using Conjugate Gradient or ADI etc, the linear systems differ slightly in the structure of the sparse matrix. The result is that a function cannot be easily written that will solve a wide range of such problems.

The simple interactive scheme serves well the users that need sophisticated mathematical manipulations, but it is not adequate for research in vision that needs a great variety of data structures and an enormous array of operations. The answer to these problems is an object system. But the object oriented approach is not without dangers. Extremely

complicated programs can be written and are being written. Coding style can be much more sophisticated and sometimes takes priority over efficiency. The lines of real code go down, but declarations go up. The prime motivation for objects were windowing systems and as a result window applications fit the standard object systems better.

This does not mean that object oriented programming is a bad idea. But designing such a system is not straightforward. A too restricted system will be of little use. A too ambitious system will be an overkill. The design procedure we followed for MediaMath was inclined towards safety. We introduced the ability to extend of the type system by either defining a `struct` in MediaMath language or defining a new type in a module written in C (this type cannot be considered built-in because it is not defined in the main module). We defined generic functions and a mechanism for overloading operators. When enough experience and feedback will have been accumulated, we are going to introduce more object oriented features.

The object system has proven adequate for most purposes so far. One of the exceptions seems to be the suite of flow estimation algorithms where each algorithm uses a slightly different data structure. An object hierarchy would help to put them in a top down classification (leaving only very few out). On the other hand, after so many years of analysis and testing of all of these algorithms, common patterns have emerged and it would be possible to design a small set of algorithms that each one is a superset of several well known flow algorithms, in much the same way that the Horn and Schunck method was combined with the Lucas and Kanade above. So it is not clear if this particular problem needs a more sophisticated programming environment or a better algorithmic

design.

9.1. Type system

There are two kinds of types in MediaMath. The first kind are the *lisp types* which are divided in the *primitive* (like `cons`, `symbol` etc) and the `special` from which the `struct`s are built. The primitive ones are defined at compile time and the special at run time. The other kind are the `C` types, the *primitive* ones (`int`, `unint`, `float` etc) and the *large* ones that are any `C` data structure that MediaMath understands (file pointers, images, vectors, dynamic library handles etc). The primitive ones are defined at compile time and the large ones at link time (the linking can occur in the middle of a session though).

The user needs to know only the names of the types. Internally all the types are given a unique number and this is used to index the tables of the generic functions.

Most users will only need to define a `struct` type. The syntax is similar to `C`.

```
struct complex
      "Structure for complex entities."
{
  re = 0;
  im = 0;
};
```

defines a structure with the name `complex`, with two fields that are initialized (if we don't want to initialize then we omit the `= 0`) and a documentation string. After the evaluation of the structure definition the interpreter defines the function `make_complex` automatically. It also defines the function `complex_p` and knows how to access the fields of the structure with the `->` operator.

9.2. Generic functions and overloading

MediaMath provides multimethod generic functions to invoke any of about two thousand functions using about a dozen operators (like multiplication, addition, convolutions etc). A generic function is an array (of arrays of arrays ... if the generic function specializes on many methods) of functions, with a mechanism to save space for empty spots. When such a generic function is invoked the table is accessed and the proper function is retrieved and invoked. In order to define, for example, the multiplication for the complex structure above, we add the corresponding combinations of data types to the *gmult* generic function

```
generic gmult
{
  complex, complex: 'complex_by_complex;
  { int, unint, float, char, unchar }, complex :
      'number_by_complex;
  complex, { int, unint, float, char, unchar } :
      'complex_by_number;
};
```

When we invoke the multiplication operator *gmult* with *c1 * c2* the proper function (*complex_by_complex* if both are complex) will be invoked. If there was a previous definition for the same combinations of data types it is replaced without affecting the rest.

Almost all the operators can be overloaded. The `->` for instance can be overloaded and it is overloaded when one defines two different data structures that share one or more field name. Another example of overloading is the *A->diagonal* in the Jacobi method above. Matrix *A* is can be a regular matrix represented internally as a standard 2-D `C` array. It has no field that holds the diagonal. *A->diagonal* is implemented by overloading the *get_diagonal* function that is called in response.

A large number of implicit matrices for the flow algorithms and a large number of filters are implemented using such techniques.

9.3. Example: Gabor filters

As an example of using the object system to simplify programming we show how the optimized Gabor convolution templates (kernels) work on MediaMath. The 2-D Gabor functions are defined as

$$G_s(x, y) = \sin(k_x x + k_y y) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$G_c(x, y) = \cos(k_x x + k_y y) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

One can construct a 2-D template and do the Gabor convolution, but even if the $\sigma = 2$ the template must be at least 11×11 to avoid truncation effects. To avoid the high cost of 2-D convolutions we can rewrite them as

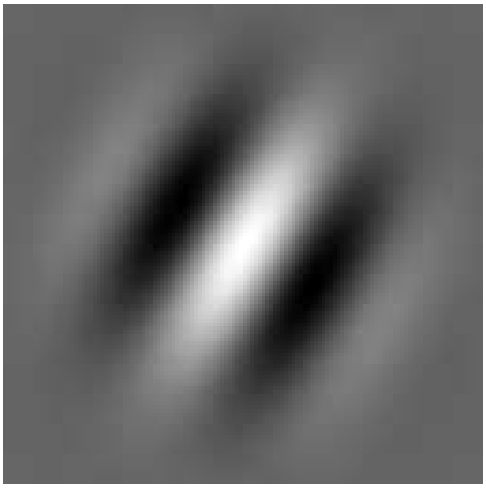
$$G_s(x, y) = \left[\sin(k_x x) e^{-\frac{x^2}{2\sigma^2}} \right] \left[\cos(k_y y) e^{-\frac{y^2}{2\sigma^2}} \right] + \left[\cos(k_x x) e^{-\frac{x^2}{2\sigma^2}} \right] \left[\sin(k_y y) e^{-\frac{y^2}{2\sigma^2}} \right]$$

and similarly for G_c. Then the 2-D convolution becomes a sum of two separable convolutions. A separable convolution can be computed as two 1-D convolutions one in the horizontal and the other in the vertical direction. Any template that can be written as a sum of very few separable templates is a semi separable template.

One could define a function that given an image and a semi separable template computes the convolution. But then if one needed a set of semi separable templates that comprise of a different kind of 1-D templates (e.g. the convolution with a uniform function can be computed much faster as a running sum rather than ordinary convolution), then a

new function has to be defined. The same is true if new types of images are introduced (color images of various standards (RGB, CMYK, HSV etc), complex images etc). The different function names that the user has to remember can grow geometrically out of control.

Using the object system these problems can be eliminated. We overload the generic convolution operator *gconvolve* to include convolutions between images and semi separable templates and we define these convolutions in terms of the generic vertical and horizontal convolution operators. Thus we can convolve an RGB image with a semi



```
gtc = gtrim(mk_cgabor_tmpl2(6.0,:k=0.55,:theta=3.14/6));
im  = mk_fimg(32,32);
im[16,16] = 100;
im = Mag2_y(Mag2_x(im(*)gtc));
write_eps_img(im,"gabor_response.eps", :rescale=t);
```

Figure 6. The impluse response of a Gabor filter and the code that generated it. The function *gtrim* reduces the size of the filter by cuting off the parts of the tail with negligible contribution. The functions *Mag2_x* and *Mag2_y* magnify by two using smooth interpolation.

separable template as long as the 1-D templates inside the semi separable can be convolved with an RGB image. As the number of new templates increases the resulting savings in code size and complexity become enormous.

10. Design Philosophy

As stated in the introduction there are several vision and image processing software tools under development or already deployed. Unfortunately not all groups of users are covered and MediaMath targets one of these groups of people that have some skill in software development, need to implement complicated mathematical algorithms in vision and other areas, cannot afford the delays of a less efficient system and want quick ways to do testing and prototyping.

The MediaMath project focused mainly on three concepts: efficiency, ease of programming, and mathematical style.

10.1. Efficiency

There are several conflicting opinions on the issue of efficiency versus code readability. The one extreme, usually advocated by numerical analysts, is that efficiency should be the main criterion and style should be secondary. The other extreme is represented by computer scientists and states the opposite. These two conflicting approaches are due to historical practices, availability and cost of hardware, availability of optimising software (in this sense computer scientists are spoiled), inherent complexity of the programs (numerical algorithms are complicated but their (unoptimized) implementation is relatively simple, while the opposite is in general true for computer science domains like

windowing systems), the time scales involved (numerical routines can run for hours, while most in most domains of computer sciencce routines run for fraction of a second: open window switch context, search a typical file for a string) etc. This in general is a very controversial issue and it is hard to find two authorities that agree. But computer vision cannot follow either of these extremes because it shares elements with both. The time scales involved (with toaday's general purpose hardware) are in the order of minutes and the programming complexity approaches the level of a small operating system. Thus the approach we followed is that of a highly optimized core with the higher levels succesively more structured.

All the core image manipulation routines are written in C and are highly optimized. The interpreter is also highly optimized using `gprof` and other tools, with a non interfering garbage collector that is only called from a few very specific places in the code so that very few routines have to protect their arguments and local variables from it. The garbage collector is simple mark and sweep but it is very fast because all the persistant data structures are kept outside the memory that is swept.

Several design decisions were made so that the system performs best in a broad class of algorithms or applications (flow algorithms, structure from motion, color manipulation, compression etc). This influenced the set of primitives and the mixture of utilities included in the system.

10.2. Programming style

All such systems have to do a trade off among programmer's freedom, safety and efficiency. MediaMath followed a middle path. The user is encouraged to use a safe style but is free use any style. Particular care has been taken to avoid crashing by all means, so that the user can continue debugging but the knowledgeable user is provided with the means to interfere with the internals of the system which can lead to irrecoverable errors. Common errors cannot do that because these possibly dangerous commands have names that cannot be typed easily by mistake. The error checking is done only for cases that an error can lead to memory corruption or incomprehensible results, to increase the effieciency and the flexibility.

MediaMath facilitates incremental development, so it relieves the user from memory management duties, and permits dynamic linking of modules. MediaMath processes can run for weeks without crashing or even any visible memory leaking. Severe errors like bus error can happen only by linking to untested C code. Even then, the system recovers gracefully and unless the `malloc` or the garbage collector data structures get corrupted, the system can continue.

Since most users are familiar with C and Unix, a large part of the standard libraries has been linked or ported to MediaMath. No effort has been made to improve any of them because this would lead to too many standards and there was no guarantee that the result would be better.

The object system is simple and efficient. And after several months of use there has been almost no need for something more complicated. But this is a rather subjective

judgement, so we decided to introduce classes, hierarchy and inheritence sometime in the near future for users that prefer this style.

10.3. Mathematical structure

The effort, when designing the language interface was to keep the syntax as close to C as possible but at the same time allow the program to be as readable and descriptive as a mathematical formula. The set of mathematical primitives were chosen to cover a large class of operations and for most of them a binary operator symbol was provided. We tried to keep the completeness of Standard Image Algebra [16] and the simple formalism needed by an interactive system. We avoided making a wish list first and then try to implement the primitives as this might interfere with the efficiency.

The default behaviour of complicated operations such as convolution is to preserve the associativity and commutativity of the operations. This is the main reason we chose the periodic boundary condition as the default behaviour of all the convolution operators. The multiplication routines do not try to optimize by swaping arguments so the multiplication is working correctly for matrices too.

There is a wide range of filters that are commonly used in vision. A large collection of them is implemented in MediaMath. The common characteristic of all the routines that create such filters is that besides the usual parameters (like the σ for a Gaussian template) the accuracy can be specified as well. The derivative, for example, can be specified by the degree of the polynomial used and the number of elements in the template generated (or how many of the elements with very small values in the tails we cut) [12, 10]. This way

the user can check the sensitivity of an algorithm under development to the quality of the components used and trade speed for accuracy.

The convolution operation is quite expensive and MediaMath has built in one dimensional templates and semiseparable templates to reduce the cost. Most of the two dimensional filters if treated as matrices have a very small number of non zero singular values. For this reason they can be treated as a sum of equal number of separable templates to reduce the cost of computation [14].

10.4. Influence from other systems

There are several tools for vision and image processing, both public and proprietary, experimental and production grade. All of them have advantages and disadvantages and almost without exception all have very specific audiences. A partial list follows.

The popular program *xv* was developed at University of Pennsylvania by J. Bradley. It is an excellent X11 based program for image display, conversion, color editing and simple image editing. It has an excellent user interface and is very portable. Xv has no facilities for image processing, matrix manipulation etc. It can be easily connected to other programs without modifying its source, and MediaMath takes advantage of that to display images.

Pbmplus is a commonly used package mainly for conversions between formats. It provides a library and a set of programs that work as Unix filters that other programs can link to. MediaMath relies on that for conversions.

Khoros is another popular program that has a sophisticated (but not easy to learn) user interface. Its main innovation is the very versatile visual programming language, *cantata*. Khoros is a huge X11 based tool that does not require programming expertise, unless one wants to introduce new primitives. The image manipulation is done by connecting programs through unix sockets, so despite its size it is not a heavy memory user. Of course sockets introduce computation overhead (copying and converting to and from the external *viff* format).

Obvius is a Common Lisp based system to do interactive image manipulation. It has several advantages like object oriented design, incremental compilation and built in display facilities. It was designed at Media Lab, MIT mainly with the computer vision researcher in mind. It relies on lisp for the general programming which is good and bad. Lisp provides anything one might ask, but it is also a heavy memory consumer. It needs at least 20 MB of swap space just to start and to do anything reasonable the swap space should be more than 50 MB. It also uses the familiar prefix lisp syntax. Although some aspects of its design were revolutionary, the choice of language, the non portable built in display facilities and may be overdesigning, somewhat limited its wide acceptance.

MATLAB is the most developed of the family of matlab tools. It is not an image processing software, but such a functionality is added as a separate toolbox which contains a collection of image processing routines.

Vista is a software environment from University of British Columbia. It has display facilities and a set of standalone programs that can be piped together. It is mainly targeted for general image manipulation like edge detection, filtering etc.

SDSC Image Tools is a set of libraries and command line programs that do image manipulation and conversion. They can be used alone or combined with shell scripts.

APPLY is an application specific software for the WARP machine developed at Carnegie Mellon.

IAL (Image Algebra Language) is a language that attempts to cast all the image operations in a few well defined forms. It presents some very intersting theoretical concepts, but its flexibility has limits (e.g. does not provide notation to access individual pixels). Some of these problems were addressed by I-BOL, and overall it has been a very influential [16, 15, 4].

IUE (Image Understanding Environment) is a big 5-year DARPA (now called ARPA) project that attempts to provide a common base for all Image Understanding research. The IUE specification is ready and the implementation should be ready in the near future [13, 9].

Pacco is a Tcl-Tk based system for image processing. It is implemented by adding functionality to tcl-tk.

The list can go on for quite a while. The MediaMath project builds on the experience gained from all of these.

There are several ideas blended together in MediaMath. Some of these ideas come from other vision systems and some from tools in other areas. The particular kind of interactive style is based on Maple and Macsyma. The idea to build the interpreter in raw C from scratch was very successful for Maple. And the idea to fashion it after Lisp

worked very well for Emacs, because the lisp interpreters are simple, there is accumulated experience on how things should be done and Lisp was the most successful interpreted language.

The choice of syntax was influenced by Awk a simple language that has a C-like syntax. One can become an expert programmer in Awk in a few hours by reading the tiny manual. So we decided to stay as close to `C` as possible. *IAL* has influenced us in the choice the simplest possible set of operations.

The object system has so far been kept simple. It includes only the most needed elements. While this has proven adequate, the object system will be extended to serve the programming style of a larger pool of users.

11. Future of MediaMath

There are several projects around MediaMath going on. Among them is the halftoning and color separation application and library which is almost complete, a camera calibration library whose development started in March 1994 and it is expected to finish by the same time in 1995 [3]. A range image analysis system that will finish during the summer 1994 for advanced active perception [7]. Other plans include the suite for signal processing which is currently planned for completion by September 1994 and a mathematical morphology package.

11.1. Parallelism

The most interesting plan for the future though is the ParallelMedia. This is a toolkit that works with MediaMath and starts remote MediaMath processes on other machines

on the network. The remote processes then act as compute and storage servers for the original process.

The way this works is as follows. A data structure like a matrix or an image can be moved on a different machine. All that is left on the local MediaMath process is a data structure with a pointer to the remote process's address space. Then, when the execution of a program calls for the multiplication of two images or two matrices then if the images are on the same machine the multiplication is done on this machine and the result stays there. Otherwise the images are moved to the same machine and the execution is done there. The original MediaMath process does not wait for the servers to finish, it just accepts a pointer for the position of the result and moves on to the next calculation. When a process tries to access a remote data structure that is not yet ready it blocks waiting for a responce on a socket. When a server is done with a computation then it sends a message through the socket.

This form of parallelism is simple and effective. The user has enough control of it to use it efficiently. The only drawbacks are that in order to avoid deadlocks, certain sequences of operations have to be disallowed which leads to small reduction in efficiency.

There is another form of parallelism that involves special hardware. Since special purpose hardware is becoming more and more common it is expected modern image processing systems to be able to use that. There are a few well known (and expensive) machines that can speed up computation on images but the main thrust will come from things like TIM-40 (a module containing a Texas Instruments C40, memory and glue

hardware that can be interconnected in various ways), the Analog Device's new 21060 DSP and the soon to be released Transputer T-9000. Products like these will eventually put special purpose hardware within budget for many applications. Consider for example the 21060. It can have 6 links to other 21060's, two serial ports, 0.5MB on chip SRAM, can execute a theoretical maximum of 120MFlop and can be interconnected in many configurations common in DSP without much glue logic. And this performance is matched by several other future and current DSPs.

For these reasons the MediaMath model of computation was designed compatible with the SIMD model. In such a system images reside in the array of processing elements and distributed in stripes or blocks. Matrices, vectors and templates have one copy per process. The type system is augmented by the distributed equivalents of the image type, the matrix type etc, which contain information about the remote address of the data structure on the grid. To perform an operation, a message that contains the opcode and the addresses or values of the arguments is broadcast to all processing elements. Then the processing elements execute the opcode. Using the object system this operation can be made transparent to the user.

The above approach is more or less the classic approach to SIMD. A new approach that became feasible with the introduction of transputers and sophisticated DSPs is to replace the low level PEs with of the shelf DSPs or other chips [4], that offer better price efficiency ratios.

Using more intelligent nodes in a grid has several advantages. One is that the overhead is kept low by issuing mostly high level commands. Another is that programming is

simplified by using the libraries that are often supplied by the manufacturer or third party. And finaly the application running on the host has the luxury of more time to do the local house keeping.

12. Conclusions

MediaMath is a versatile package for computer vision and signal processing and related areas. Although it has less than two years of development behind it, it can be used in courses and in research. The code right now is well over 70,000 lines including C, awk and MediaMath files. It runs on SunOs 4.1 or later but if the dynamic loading utilities are excluded, then it runs on many other Unix systems like Dec Ultrix and SGI. Recently other workstations incorporated dynamic libraries and flexible application programmer interfaces similar to SunOs' `dlopen` in their operating systems (IBM, Hewlet-Packard) and we plan to port MediaMath to them as soon as we get access to such a workstation.

Among the features of MediaMath is its ability to connect to other programs seamlessly like xv and gnuplot. Early in the design generality and versatility were given higher priority than features like ploting or displaying. The reason for this is that if the software can connect to several other server programs for display and plotting then it can take advantage of the best software in the market. After all an enormous effort would be required create the utilities that would outperform xv or gnuplot. So this functionality is left for the future.

MediaMath has been used in two courses so far with excellent results. It allowed the students to work on several projects that in pre-MediaMath times each one was

considered a term project. Two graduate students do their thesis on it and several new algorithms have been tested using MediaMath. The software is entering the Beta phase now and a public release is expected during the summer of 1994.

References

1. K. S. Arun, T. S. Huang, and S. Blostein, “Least squares fitting of two 3-D point sets,” *IEEE Trans. PAMI* **9(5)** pp. 698-700 (1987).

2. J. L. Barron, D. J. Fleet, and S. S. Beauchemin, *Performance of Optical Flow Techniques,* RPL-TR-9107, Robotics and Perception Lab, Queen’s University (Jul2 1993).

3. A. Basu, “Active calibration: Alternative strategy and analysis,” *CVPR,* pp. 495-500 (1993).

4. J. Brown and D. Crookes, “A High Level Language for Parallel Image Processing,” *Image and Vision Computing* **12**(2) pp. 67-79 (1994).

5. D. J. Fleet and K. Langley, “Toward Real Time Optical Flow,” *Vision Interface,* pp. 116-124 (1993).

6. G. Golub and C. F. Van Loan, *Matrix Computations,* Johns Hopkins (1990).

7. J.Y. Herve and Y. Aloimonos, “Exploratory active vision: Theory,” *CVPR,* pp. 10-15 (1992).

8. B. K. P. Horn and B. G. Schunck, “Determining Optical Flow,” *Artificial Intelligence* **17** pp. 185-204 (1981).

9. P. Kahn, “Building Blocks for Computer Vision Systems,” *IEEE Expert*, pp. 40-50 (December 1993).

10. E. Karabassis and M. E. Spetsakis, *Families of templates for fundamental image operations,* York TR CS-91-07 (1991).

11. B. Lucas, *Generalized Image Matching by the Method of Differences,* PhD Dissertation, Dept. of Computer Science, Carnegie Mellon University (1984).

12. P. Meer and I. Weiss, “Smoothed differentiation filters for images,” *ICPR-C*, pp. 121-126 (1990).

13. J. Mundy, T. Binford, T. Boult, A. Hanson, R. Beveridge, R. Haralick, V. Ramesh, C. Kohl, D. Lawton, D. Morgan, K. Price, and T. Strat, “The Image Understanding Environment Program,” *CVPR*, pp. 406-416 (1992).

14. W. K. Pratt, *Digital image processing*, Wiley Intercience (1991).

15. G.X. Ritter and P.D. Gader, “Image algebra techniques for parallel image processing,” *Journal of Parallel and Distributed Computing* **4** pp. 7-44 (1987).

16. G. X. Ritter, J. N. Wilson, and J. L. Davidson, “Image Algebra: An Overview,” *Computer Vision, Graphics and Image Processing* **49** pp. 297-331 (1990).

17. M. E. Spetsakis, “Optical flow estimation using discontinuity conforming filters,” *Submitted to BMVC*, (1994).

18. M. E. Spetsakis and J. Aloimonos, “Optimal Visual Motion Estimation,” *IEEE Transactions on PAMI* **14**(9) pp. 959-964 (1992).

19. J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis,* Springer-Verlang (1980).

20. R. Y. Tsai and T. S. Huang, “Uniqueness and Estimation of Three Dimensional Motion Parameters of Rigid Objects with Curved Surfaces,” *IEEE Trans. PAMI* **6** pp. 13−27 (1984).