MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper No. 281 December 1985

Vision Utilities

Harry Voorhees

Abstract. This paper documents a collection of Lisp utilities which I have written while doing vision programming on a Symbolics Lisp machine. Many of these functions are useful both as interactive commands invoked from the Lisp Listener and as "building blocks" for constructing larger programs. Utilities documented here include functions for loading, storing, and displaying images, for creating synthetic images, for convolving and processing arrays, for making histograms, and for plotting data.

Contents

1 Introduction

This paper documents a collection of Lisp utilities which I have written while doing vision programming on a Symbolics Lisp machine. Many of these functions are useful both as interactive commands invoked from the Lisp Listener and as "building blocks" for constructing larger programs. Utilities documented here include functions for loading, storing, and displaying images, for creating synthetic images, processing arrays, for convolving images, for making histograms, and for plotting data.

The purpose of this paper is two-fold: first, to document these utilities so that others can use them, and second, to contribute to the development of a common utilities package which everyone in the vision group can use conveniently. Hopefully, many of the functions described in this paper merit inclusion in the vision package. This paper may also serve as a prototype for the documentation of the forthcoming set of utilities. In any case, the files where the functions currently reside are noted so that anyone can use them now. All files reside in directory *pig:[hlv.utils]*.

Writing general "building blocks" has the advantage of speeding code development, since duplication of code is reduced. Here, utility functions have even been used to write other utility functions quickly. This presents no problem to the user who loads all the utilities (file *utils* does this). However, selectively loading certain functions can be somewhat awkward, because a desired function may require a function defined in another file. To solve this problem, a file which requires other files automatically loads them. To prevent loading the same file over and over again, a unique global variable is defined by a file as it is loaded. Other files which require this file only load it if the global variable is not bound. For example, file *apl* begins with the Lisp forms:

```
(defvar *apl-loaded* t)
(unless (boundp '*prim-loaded*) (load "pig:[hlv.utils]prim"))
(eval-when (compile)
  (unless (boundp '*map-array-loaded*) (load "pig:[hlv.utils]maparray")))
```

Since functions in this file require functions defined in file *prim*, that file is automatically loaded when *apl* is loaded if it hasn't been loaded already. Of course, *prim* may in turn load other files. By binding *apl-loaded* first, any circular dependencies will not cause an infinite loop. In this example, some functions in *apl* require a macro defined in *maparray*; this file need only be loaded when functions in *apl* are recompiled. The simple convention described here frees the user from keeping track of file definition dependencies.

1.1 A Note about Notation

This paper uses a format like that of the Lisp Machine Manual (*e.g.* Symbolics [1985]): the documentation for each function, macro, variable, or method starts with a header line containing its name, arguments, and type. For clarity, the default values of optional arguments are often omitted from the header line and are described in the documentation instead. One should *not* assume that the absence of a default value of an optional variable in the header line means that the default value is **nil**.

The examples use the notation of Steele [1984]: The symbol ==> denotes evaluation; `(between 4 0 10) ==> T` means that evaluating `(between 4 0 10)` returns the value `T`. Often, in examples which emphasize the side effects of a function (*e.g.* a print function), the value returned is not shown. The symbol --> denotes macro expansion; `(between 4 0 10) --> (and (>= 4 0) (<= 4 10))` means that the expression `(between 4 0 10)` macro expands to the code `(and (>= 4 0) (<= 4 10))`.

1.2 The Vision Package

To avoid name conflicts with user programs, all functions described here are defined in package **vision**, unless otherwise indicated. For convenience, the nicknames **vis** and **v** can be used. Thus, the function **vision:add** can also be referenced as **v:add** or **vis:add** from outside package **vision**. Of course, the function can simply be referenced as **add** within its package.

2 Primitives

The file *prim* contains a number of functions which serve as an extension of Lisp primitives and which are not particular to any application. Some of the functions listed below are more generally useful than others, but almost all are used by one or more of the utilities described in this paper.

2.1 Mathematical Functions and Constants

pi *Variable*
2pi *Variable*
pi//2 *Variable*
sqrt2pi *Variable*

Variables **pi**, **2pi**, **pi//2**, and **sqrt2pi** are bound to single-precision values of π, 2π, $\pi/2$, and $\sqrt{2\pi}$. In Release 6, the Lisp variable **si:pi** is bound to a double-precision value, which can slow down computations significantly.

infinity *Variable*
–infinity *Variable*

Variables **infinity** and **–infinity** are bound to **+1e∞** and **–1e∞**, which are hard to type.

square *x* *Subst*

Function **square** computes x^2.

divide *dividend divisor* *Subst*

Function **divide** divides *dividend* by *divisor*, but unlike the Lisp primitive **//**, it always returns a floating-point number:

```
(divide 3 2) ==> 1.5
(// 3 2) ==> 1
```

factorial *n* *Function*

Function **factorial** computes $n!$ where n is a non-negative integer.

average *list* *Function*

Function **average** computes the average of a list of numbers, returning a floating-point number.

log2 *x* *Function*
log10 *x* *Function*

Functions **log2** and **log10** return the logarithm of x to bases 2 and 10, respectively.

roundto *x* &optional *(multiple 1)* *Function*
roundup *x* &optional *(multiple 1)* *Function*
rounddown *x* &optional *(multiple 1)* *Function*

Function **roundto** rounds x to the nearest multiple of *multiple*, which defaults to 1. Functions **roundup** and **rounddown** do the same, rounding up or down. For multiples other than 1, these functions are more convenient than the Lisp primitives **round**, **floor**, and **ceiling** which return the quotient and remainder of the two arguments.

```
(roundto 8 3.2) ==> 6.4
(rounddown 5.9999) ==> 5
(roundup 3.3 1\3) ==> 10\3
```

multiple-of *radix number* *Function*

Predicate **multiple-of** returns t if *number* is an integral multiple of *radix* and nil otherwise.

```
(multiple-of 192 32) ==> T
(multiple-of 216 32) ==> NIL
```

between *x low high* *Subst*

Predicate **between** returns **t** if $low \leq x \leq high$ and **nil** otherwise.

```
(between 4 0 10) --> (and (<= 0 4) (<= 4 10)) ==> T
```

compare-all *predicate list* *Function*

Predicate **compare-all** applies function *predicate* between every pair of adjacent elements in *list* and returns **t** if no comparisons yield **nil**; it returns **nil** otherwise.

```
(compare-all #'< '(1 4 9 16)) ==> T
```

eqall &rest *args* *Function*
=all &rest *args* *Function*

Predicate **eq-all** is a version of **eq** which takes any number of arguments; it returns **t** if all *args* are **eq** and **nil** otherwise. Similarly, predicate **=all** is a version of **=** which takes any number of arguments.

2.2 Logical Functions

and* &rest *args* *Function*
or* &rest *args* *Function*

and* and **or*** are logical *functions* which take any number of arguments. Unlike Lisp special forms **and** and **or**, **and*** and **or*** can be used as mapping functions.

```
(apply #'or* '(nil t nil)) ==> T
```

2.3 List Processing Functions

one-of &rest *args* *Function*

Function **one-of** is used to enumerate legal values of a variable or an argument of a function for documentation purposes. It returns the first argument, which is the default value of the variable or argument. For example, in the definition

```
(defun plot (... &key (curve (one-of :spline :line :none)) ...) ...)
```

the function **one-of** declares that *:spline* is the default value of keyword argument *curve* and that *:line* and *:none* are other legal values.

list-pairs *cars cadrs* *Function*

Function **list-pairs** constructs an association list of lists (not dotted pairs, which is what Lisp primitive **pairlis** does).

```
(list-pairs '(A B C) '(1 2 3)) ==> ((A 1) (B 2) (C 3))
```

lastcar *list* *Subst*

Function **lastcar** returns the last element of a list.

rcons *list element* *Function*

Function **rcons** does the reverse of function **cons**: it adds an element to the *end* of a list, returning a new list.

```
(rcons '(1 2) 3) ==> (1 2 3)
```

filter-mask *mask list* &optional *predicate* *Function*

Function **filter-mask** takes a mask and a list of the same length as arguments. It returns those elements of *list* for which the corresponding elements of *mask* satisfy a predicate. The predicate defaults to `(lambda (x) (not (null x)))`.

```
(filter-mask '(t nil t) '(50 -1 40)) ==> (50 40)
(filter-mask '(1 0 4) '(X Y Z) #'zerop) ==> (Y)
```

remove-elements *of-list from-list* *Function*

Function **remove-elements** removes all occurences of elements in *of-list* from *from-list*.

```
(remove-elements '(1 2) '(7 1 1 8 2)) ==> (7 8)
```

list-non-nil &rest *elements* *Function*

Returns a list of *elements* which are not **nil**.

2.4 Sorting and Indexing Functions

min-n *list n* *Function*

Function **min-n** returns the minimum *n* elements of *list* by recursively finding and removing the smallest remaining element.

```
(min-n '(30 20 10 40) 2) ==> (10 20)
```

sort-positions *list* &optional *(predicate #'<)* *Function*

Function **sort-positions** sorts *list*, returning the original indices of the sorted elements instead of the elements themselves. The function *predicate* is used for comparisons, which defaults to #'<. Indexing starts at 0. Function **sort-positions** is like the APL operator "sort".

```
(sort-positions '(67 3 10 99)) ==> (2 3 0 4)
```

find-positions-in-list= *n list* *Function*

Function **find-positions-in-list=** returns a list of all indices of elements in *list* which = *n*. If no such elements are found the function returns **nil**. Indexing starts at 0. This function differs from Lisp primitive **find-position-in-list** which returns only the first index, using **eq** for comparisons.

```
(find-positions-in-list= 3 '(3 44 3 5)) ==> (0 2)
```

2.5 Mapping Functions

mapcir *function* &rest *args* *Function*

Function **mapcir** is a "smart" version of **mapcar**. Any arguments after the mapping function which are not lists are coerced to lists of the appropriate length (actually, circular lists). The function is then applied to corresponding elements of each list and a list of results is returned. If no arguments are lists **mapcir** is equivalent to **apply**.

```
(mapcir #'list 'i '(love spayed) 'my '(dog cat)) ==>
  ((I LOVE MY DOG) (I SPAYED MY CAT))
```

mapbetween *function list* *Function*

Function **mapbetween** applies a two-argument function between elements of a list from left to right. It is like the APL operator "insert" ("/").

```
(mapbetween #'^ '(3 2 2)) ==> 81
```

mapexpand *function list* *Function*

Function **mapexpand** works like **mapbetween**, but a list of intermediate results is returned. It is like the APL operator "expand" ("\").

```
(mapexpand #'* '(1 2 3 4)) ==> (1 2 6 24)
```

maptree *function tree* *Function*

Function **maptree** maps a function over atoms of a (nested) list *tree*, returning a list of the same structure.

```
(maptree #'square '(((4 (3 2))) 5)) ==> (((16 (9 4))) 25)
```

cross *function list1 list2* *Function*

Function **cross** applies two-argument function to every pair of elements, one from each of two lists. If *list1* and *list2* contain n and m elements respectively, **cross** returns a list of length nm. The first m elements are the results of applying *function* to the first element of *list1* and each element of *list2*, etc.

```
(cross #'+ '(1 2) (10 20 30)) ==> (11 21 31 12 22 32)
```

2.6 Interpreter Functions

The following functions can be useful for writing macros which generate code and which evaluate their own arguments.

symbol *string* &optional *package* *Function*

Function **symbol** creates a symbol from *string*, interning it to package *package*. If *package* isn't specified, the current package is used.

make-alist-of-bindings &rest *variables* *Macro*
make-plist-of-bindings &rest *variables* *Macro*

Function **make-alist-of-bindings** makes an association list of variable names and their bindings. It is useful for saving function parameters. Each element of the association list is a list, not a dotted pair. Function **make-plist-of-bindings** is similar; it makes a property list of variable names and bindings.

```
(let* ((a 1) (b a)) (make-alist-of-bindings a b)) ==> ((A 1) (B 1))
(let* ((a 1) (b a)) (make-plist-of-bindings a b)) ==> (NIL A 1 B 1)
```

quotedp *expr* *Function*

Predicate **quotedp** returns **t** if *expr* a quoted s-expression; that is, if *expr* is a list whose *car* is the symbol *quote*.

2.7 Stream Functions

tyi-now &optional *stream* *Function*

Function **tyi-now** is like function **tyi** but it clears any previous inputs first. If *stream* is not specified **standard-stream** is used by default.

listen &optional *stream* *Function*

Function **listen** tells whether any inputs are in a stream's input buffer. It is used by functions which run until the user types a character or clicks the mouse. If *stream* is not specified **standard-stream** is used by default.

2.8 System Utilities

clock &body *body* *Macro*

Macro **clock** executes form(s) or forms, while timing how long they take to execute. The time is printed, and value of the last form is returned.

```
(clock (make-array '(1000 1000) :type art-8b))
Time elapsed = 9.2 seconds.
<#ART-8B-1000-1000 2345234>
```

apropos-msgs *object substring* *Function*

Function **apropos-msgs** returns a list of all valid messages to *object* which contain *substring.* Argument *object* is an instantiation of a Flavor.

```
(apropos-msgs tv:initial-lisp-listener "save") ==>  (:SAVE-BITS
  :SAVE-INPUT-BUFFER :SET-SAVE-BITS :SAVE-RUBOUT-HANDLER-BUFFER)
```

warning *format-string* &rest *format-args* *Function*

Function **warning** (usually) prints a warning message on the screen. The function takes the same arguments as **format**—a format string followed by arguments which specify the formatted message string. The action taken depends on the value of global variable ***warning-action***, which can be one of the following:

:just-warn The warning message is printed and processing continues. This is the default case.

:wait The warning message is printed. Processing waits until the user types a space. This insures that the user sees the message and gives him a chance to abort processing.

:ignore No message is printed and processing continues.

:error An error is signalled.

2.9 Print Functions

Some functions for printing objects, defined in file *print*, are described here.

pp *array* &key *(tiny nil) (field '(9 3)) format* *Function*

Function **pp** pretty-prints a one- or two-dimensional array. Assuming that *foo* is bound to a 2×3 element array containing the integers 1 through 3 in each row,

```
(pp foo)
    1  2  3
    1  2  3
```

Optional arguments control how array elements are printed. By default *art-nb* and *art-fixnum* arrays are printed using a column width just large enough to hold the largest possible value. Type *art-q* arrays are displayed as floating point numbers; the total field width and digits after the decimal point is specified by the two-element list *field*, which defaults to *(9 3)*. By default type *art-boolean* arrays are dislayed using letters T and F. Alternatively, a format string *format* can be specified for any type of array. Also, if argument *tiny* is t a tiny-sized font instead of the window's current font is used; this option is useful for printing large arrays.

The function **pp** should not usually be used for printing image arrays because such arrays are displayed in transposed order. **pp** follows the convention that an $m \times n$ array contains m rows and n columns; on the other hand, an $m \times n$ image array has width m columns and height n rows. For this reason, function **pp-image** should be used for printing image arrays instead. This incompatibility will be eliminated in Release 7, when arrays will be represented in row-major instead of column-major order.

pp-image *array* &key *from-pos size (tiny t) (field '(9 3)) format* *Function*

Function **pp-image** pretty prints an image array. Keyword arguments can be used to print a portion of the array of size *size* starting at position *from-pos*. Like function **pp**, keyword arguments *field* and *format* control how array elements are printed. By default a tiny-sized font is used unless keyword *tiny* is **nil**. For printing mathematical matrices, use function **pp** instead.

pp-list *list* *Function*

Function **pp-list** pretty-prints a list, one element per line.

```
(pp-list '(THIS IS (A LIST)))
  THIS
  IS
  (A LIST)
```

pp-alist *alist* *Function*
pp-plist *plist* *Function*

Function **pp-alist** pretty-prints an association list (of lists, not dotted pairs; see function **list-pairs**).

```
(pp-alist '((image-name "Pookie") (width 192) (height 200)))
    IMAGE-NAME: Pookie
    WIDTH:      192
    HEIGHT:     200
```

Function **pp-plist** pretty-prints a property list using the same format.

print-values &rest *exprs* *Macro*
print-more-values &rest *exprs* *Macro*

Macro **print-values** takes one or more variables (or expressions) as arguments and prints their names and values on a single line. This macro is useful for debugging programs if you don't want to bother figuring out how to use the debugger.

```
(let ((name "Zippy") (hundred 100)) (print-values name (1+ hundred)))
NAME="Pookie"  (1+ HUNDRED)=101
```

Macro **print-more-values** does the same thing but prints the values on the current output line rather than on a new line.

3 APL Functions

A number of functions based on the programming language APL [Polivka and Pakin, 1979] are defined in file *apl*. A major feature of this language is that its operators are highly generic.

ravel *elements* *Function*

Function **ravel** takes an array, list, or atom *elements* and returns a list of the elements. It does not fringe nested lists or arrays. It is a generalization of **listarray** and **list**.

```
(pp foo)
  1 2 3
  1 2 3
(ravel foo) ==> (1 2 3 4 5 6)
(ravel '((1 2) 3)) ==> ((1 2) 3)
(ravel 3) ==> (3)
```

shape *elements* *Function*

Function **shape** is a generalization of functions **length** and **array-dimensions**. It returns the shape of its argument *elements*: given an array, it returns a list of its array dimensions; given a list, it returns a scalar which is its length (**nil** is treated as an empty list); given a scalar, it returns **nil**.

```
(shape '(1 2 3)) ==> 3
(shape nil) ==> 0
(shape foo) ==> (2 3)
(shape 3) ==> nil
```

rho *shape elements* &optional *array-type* *Function*

Function **rho** is a generalization of functions **list** and **make-array**. It returns a scalar, list, or array whose dimensions are specified by *shape* (see function **shape** above). The item returned is filled with elements of *elements*, which may be a scalar, list, or array; *elements* is ravelled and repeated until the item is filled. If an array is to be returned, an *array-type*, which defaults to *art-q*, can be specified.

```
(rho 3 '(1 2)) ==> (1 2 1)
(rho 3 '((1 2))) ==> ((1 2) (1 2) (1 2))
(pp (rho '(3) '(1 2))) ;; a 1d array
   1 2 1
(pp (setq foo (rho '(2 3) (1 2 3) art-2b)))
   1 2 3
   1 2 3
(pp (rho '(3 2) foo))
   1 2
   3 1
   2 3
```

The function **ident**, which returns an identity matrix of specified size, illustrates the usefulness of function **rho**:

```
(defun ident (n) (rho (list n n) (cons 1 (rho n 0))))
(pp (ident 3))
   1 0 0
   0 1 0
   0 0 1
```

vector &rest *elements* — *Function*

Function **vector** is like **list**, but it creates a one-dimensional array.

```
(pp (vector 1 2 3))
   1 2 3
```

vectorize *elements* &optional *array-type* — *Function*

Function **vectorize** converts a list or array into a one-dimensional array containing the same elements. Array type *art-q* is used unless *array-type* is specified otherwise. Note that **ravel** can be used to perform the inverse operation.

```
(pp (vectorize foo))
   1 2 3 1 2 3
(pp (vectorize '(1 2 3)))
   1 2 3
(pp (vectorize '((1 2) 3))) ;; an array of 2 elements
   (1 2) 3
```

iota *n* *Function*
iotav *n* *Function*

Function **iota** generates a list of n integers from 0 to $n-1$.

```
(iota 5) ==> (0 1 2 3 4)
```

Function **iotav** is like **iota** except that it returns a vector (a one-dimensional array) instead of a list.

shift-list *n elements* *Function*

Function **shift-list** shifts the elements of list using wrap-around. If n is positive elements are shifted by n to the right; if n is negative elements are shifted by $-n$ to the left.

```
(shift-list 2 (iota 5)) ==> (3 4 0 1 2)
(shift-list -3 (iota 5)) ==> (3 4 0 1 2)
```

interval *start stop* &optional *(increment 1)* *Function*

Function **interval** is a generalization of **iota**, returning a list of integers between *start* and *stop*. If *increment* is specified, then a list of multiples of the increment on the interval is returned.

```
(interval -1.1 1.9) ==> (-1 0 1)
(interval -1.1 1.9 0.5) ==> (-1.0 -0.5 0.0 .5 1.0 1.5)
(interval -1.1 1.9 1\2) ==> (-1 -1\2 0 1\2 1 3\2)
```

drop *n elements* *Function*

Function **drop** drops the first n or last $-n$ elements of a list or one-dimensional array *elements*.

```
(drop 2 (drop -1 '(1 2 3 4 5))) ==> (3 4)
```

index *indices list-or-vector* *Function*

Function **index** is a generic function for indexing a list or vector *list-or-vector*. If *indices* is a scalar number, a scalar is returned; if *indicies* is a list of numbers, a list of the same length as *indices* is returned. Indexing is zero-based.

```
(index 2 '(THIS IS A LIST)) ==> A
(index '(0 3) '(THIS IS A LIST)) ==> (THIS LIST)
```

row *indices matrix* *Function*
col *indices matrix* *Function*

Function **row** extracts rows from a matrix (a two-dimensional array). If a scalar row index is specified, a vector is returned; if a list of row indices are specified, a matrix is returned. Function **col** works like **row** but extracts columns instead. (Because image arrays are displayed in transpose order, **row** extracts *columns* and **col** extracts *rows* of a displayed image array.)

```
(pp (row '(1 3) (ident 4)))
    0 1 0 0
    0 0 0 1
```

add &rest *addends* *Function*
sub *minuend subtrahend* *Function*
mul &rest *multiplicands* *Function*
div *dividend divisor* *Function*

Functions **add**, **sub**, **mul**, and **div** are generic arithmetic functions which operate on scalars, lists, and arrays. A scalar can be coerced to a list or array; otherwise, arguments must be of the same size and type. Any array result returned is of array type *art-q*, regardless of the array types of the arguments. Functions **add** and **mul** take any number of arguments; **sub** and **div** take exactly two. Function **div** calls function **divide**, which always returns a floating point number.

```
(pp (add foo foo))
   2 4 6
   2 4 6
(sub '(10 20 30) '(1 2 3)) ==> (9 18 27)
```

```
(pp (sub (mul 3 (ident 3)) 1))
   2 -1 -1
  -1  2 -1
  -1 -1  2
(div '(1 2 3) 2) ==> (0.5 1.0 1.5)
```

invert *matrix* *Subst*

Function **invert** inverts a matrix. It is another name for Lisp primitive **math:invert-matrix.**

trans *matrix* *Subst*

Function **trans** transposes a matrix. It is another name for Lisp primitive **math:transpose-matrix.**

det *matrix* *Subst*

Function **det** computes the determinant of a matrix. It is another name for Lisp primitive **math:determinant.**

mul-mat-2 *matrix1 matrix2* *Subst*

Function **mul-mat-2** performs matrix multiplication on *matrix1* and *matrix2*. It is another name for Lisp primitive **math:multiply-matrices.** Note that function **mul** computes a different result.

Functions **sort-positions, mapbetween** and **mapexpand**, described in section 2, are also based on APL operators.

4 Image Array Functions

This section describes functions for manipulating image arrays, which are defined in file *array*.

Functions which compute arrays can be used in two ways: either as pure functions, which do not have side effects, or as mutators, which reuse arrays. Such functions have an optional argument for their result. If the result is not specified, a new array of the appropriate size is automatically allocated. This method of programming, which frees the programmer from the chore of managing storage, is also quite efficient if the ephemeral garbage collector is on. Alternatively, a result array can be passed to the function, allowing storage to be reused. Passing the same array as the input and result arguments is thus equivalent to calling a function which mutates its input. This implementation strategy thus supports both functional and side-effect models of programming conveniently.

same-size-array *array* &optional *array-type* *Function*

Function **same-size-array** allocates and returns an array of the same size as *array*. Its array type is the same, too, unless specified otherwise by *array-type*. This function is commonly used for the default value of the *result* argument of array processing functions which create an array if one is not specified.

```
(defun find-edges (image-array &optional
                   (result (same-size-array image-array art-1b)))
  ;; body left as an exercise for the reader
  )
```

4.1 Making and Copying Arrays

copy-array *array* &key *to-array size from-pos to-pos* *Function*

Function **copy-array** returns a copy of *array*. Optional arguments act as constraints; *size*, *from-pos*, and *to-pos* must be lists whose lengths equal the dimensionality of *array*. The whole array is copied by default, but a portion can be copied by specifying *size* (e.g. the width and height) and/or *from-pos*, the position to start copying from. If a destination array *to-array* is specified, the result gets copied to it, starting at an optionally specified position *to-pos*;

otherwise, an array of the appropriate size is created. In other words, the function always does the logical thing by default.

For efficiency, this function uses **bitblt** if it can, that is, if both arrays are *bitbltable* (see function **bitbltable?** below) and are of the same type; otherwise, it calls **copy-array-portion** which uses compiler array registers for efficiency. The arrays need not be two-dimensional.

```
(pp (copy-array (setq bar (rho '(3 4) (add 1 (iota 12))))))
    1  2  3  4
    5  6  7  8
    9 10 11 12 ;; a new copy of bar
(pp (copy-array bar :from-pos '(1 1)))
    6  7  8
   10 11 12
(pp (copy-array bar :to-array (rho '(4 5) 0) :to-pos '(1 2)))
    0  0  0  0  0
    0  0  1  2  3
    0  0  5  6  7
    0  0  9 10 11
```

zero-array *array* *Function*

Function **zero-array** fills *array* with zeros and returns it. It uses **bitblt** if possible.

make-displaced-array *size to-position to-array* *Function*

Function **make-displaced-array** makes and returns a conformally-displaced array of shape *size* displaced to *to-array* starting at position *to-position*. Both *size* and *to-position* are lists whose lengths equal the dimensionality of *to-array*.

bit-blit *array* &key *to-array size from-pos to-pos alu* *Function*

Function **bit-blit** is a version of Lisp primitive **bitblt** which uses the convenient argument syntax and defaults of function **copy-array**. Unlike **copy-array**, this function always calls **bitblit**, so it should be used by applications which always require fast copying. It also has an optional argument to specify the *alu* for combining elements, which defaults to **tv:alu-seta** (copy).

bitbltable? *array* *Function*

Function **bitbltable?** returns t if *array* is a valid argument to *bitblt*, that is, if array is two-dimensional and its width times number of bits per element is a multiple of 32. It returns nil otherwise.

bitbltable-shape *shape type* *Function*

Function **bitbltable-shape** rounds up *shape*, a list of two numbers representing array dimensions, so that an array of type *type* would be *bitbltable*. It returns the new shape.

make-bitbltable-array *shape type* &rest *keywords-and-args* *Function*

Function **make-bitbltable-array** is like **make-array** except that is rounds up *shape*, if necessary, to return a *bitbltable* array.

bitbltablize-array *array* *Function*

Function **bitbltablize-array** copies *array* to a new array, which it returns, if *array* is not *bitbltable*; otherwise, it just returns *array*.

convert-array-type *array array-type* *Function*

Function **convert-array-type** copies an integer array to a new array of specified array type, returning the new array.

4.2 Allocating Temporary Arrays

The file *temparray* defines a utility for allocating temporary arrays. It is especially designed to allow image processing functions to temporarily allocate big arrays for holding intermediate results. The arrays are allocated from a stack and hence do not require garbage collection. The file *temparray* defines a special area of memory called ***temporary-array-area*** for allocating temporary arrays, over which the garbage collector is turned off.

with-temporary-array *(name shape* &rest *options)* &body *body)* — *Macro*

Macro **with-temporary-array** allocates a temporary array of specified *shape* and *options*, which can be any keywords and arguments to function **make-array**. While executing form or forms *body*, *name* is bound to the temporary array. The last form of *body* is returned. The temporary array must only be referenced dynamically within *body*, so *body* must not return the array as a value or set any non-local variable to it.

4.3 Array Bookkeeping Functions

bits-per-element *array-or-type* — *Function*

Function **bits-per-element** takes an array, an array-type symbol (*e.g. 'art-8b*), or an array-type code (e.g. *art-8b*), and returns the number of bits per element (e.g. *8*). For array-type *art-q* it returns **nil**.

max-value-of-array-type *array-or-type* — *Function*
min-value-of-array-type *array-or-type* — *Function*

Functions **min-value-of-array-type** and **max-value-of-array-type** return the minimum and maximum values, respectively, which can be held in an array of specified array type. As with function **bits-per-element**, the argument *array-or-type* can be an array, an array-type symbol, or an array-type code. The minimum and maximum values of array-type *art-q* are $-1e\infty$ and $+1e\infty$, respectively.

```
(max-value-of-array-type 'art-8b) ==> 255
(min-value-of-array-type art-8b) ==> 0
```

array-type-p *array array-type* — *Function*

Predicate **array-type-p** returns t if *array* is of array type *array-type*, and **nil** otherwise.

bit-array? *array* *Function*

Predicate **bit-array?** returns t if *array* is of one of the array types *art*-n*b* or *art-fixnum*, and nil otherwise.

width *array* *Function*
height *array* *Function*

Functions **width** and **height** return the width and height of an image array, which are the first and second dimensions, respectively, of the two dimensional array *array*.

flip-image *array* &optional *result* *Function*
flip-image-rows *array* &optional *result* *Function*
flip-image-cols *array* &optional *result* *Function*

Function **flip-image**, also called **flip-image-rows**, reflects an image by reversing the order of rows of an image array. It is used for flipping images represented in a right-handed coordinate system (*e.g.* Keith Nishihara's format) to the left-handed coordinate system format currently used.

Function **flip-image-cols** is similar, but it reverses the order of columns of an image array.

4.4 Array Processing Functions

complement-array *array* &optional *array-type result* *Function*
uncomplement-array *array* &optional *amplify result* *Function*

Function **complement-array** converts an *art-q* array of integers *array* to an *art*-n*b* array in two's complement form, which can be stored more efficiently. An *art*-n*b* array can store integers from -2^{n-1} to $2^{n-1}-1$. A result array of type *art-fixnum* is allocated, unless either *result* or *array-type* specify otherwise.

Function **uncomplement-array** performs the opposite function. It returns an *art-q* array *result* which is optionally multiplied by an amplification factor *amplify*.

scale-array *array* &key *offset factor type result* *Function*

Function **scale-array** linearly scales an array by an offset and scaling factor optionally specified by keywords *:offset* and *:factor*. The resulting array or its array-type can be specified using keywords *:result* or *:type*; otherwise, a result array of the same type as *array* is returned.

```
(pp (scale-array (vector 1 2 3) :offset 2 :factor 10))
   30  40  60
```

enhance-array *array* &key *type result min-result max-result* *Function*

Function **enhance-array** also scales an array, automatically choosing the constants for function **scale-array** so that the full range of the result is used. For example, for an *art-8b* result, the minumum value of *array* is mapped to *0* and the maximum value is mapped to *255*. If neither *result* nor its *type* are specified, a result array whose array type is that of the input array is returned.

One can also specify the range of the result using keyword arguments *min-result* and *max-result*. These values can fall outside the range of the result array type, in which case any scaled array values out of range are thresholded.

```
(pp (enhance-array (vectorize '(-1000 0 1000)) :type art-8b))
   0 128 255
```

Function **enhance-array** returns three values: the enhanced array and the offset and amplification factor used to enhance the array.

shift-array *array cols-rows result* *Function*

Function **shift-array** shifts a two-dimensional array by the number of columns and rows specified by the two-element list *cols-rows*, returning a new array *result*. Like the APL operator "rotate" ("ϕ"), **shift-array** performs wrap-around, so the result has the same size as *array* and is completely filled. Since this function is implemented using function **copy-array**, it uses **bitblt** if possible.

```
(pp bar)
   1  2  3  4
   5  6  7  8
   9 10 11 12
(pp (shift-array bar '(2 -1)))
   7  8  5  6
  11 12  9 10
   3  4  1  2
```

4.5 Binary Array Functions

The following functions are designed for creating or manipulating binary arrays—1-bit arrays which represent boolean values. The value 1 represents "true" and 0 represents "false". Type *art-1b* arrays are used instead of *art-boolean* arrays since the former can be displayed directly (e.g. by using **bitblt**).

binary-array *boolean-array* *Function*

Function **binary-array** converts an *art-boolean* array to an *art-1b* array by returning an *art-1b* array which is displaced onto *boolean-array.*

threshold-array *array threshold* &optional *result* *Function*

Function **threshold-array** returns an array *result* (of type *art-1b* by default), where 1's indicate values of $result > threshold$ and 0's indicate otherwise.

threshold-array< *array threshold* &optional *result* *Function*

Function **threshold-array<** returns an array *result* (of type *art-1b* by default), where 1's indicate values of $result < threshold$ and 0's indicate otherwise.

equal-array *array value* &optional *result* *Function*

Function **equal-array** returns an array *result* (of type *art-1b* by default), where 1's indicate values of $array = value$ and 0's indicate otherwise.

and-array *array1 array2* &optional *result* *Function*

Function **and-array** maps the logical function AND over two *art-1b* arrays *array1* and *array2*. It returns an array *result* (of type *art-1b* by default), where 1's indicate locations where *array1* $= 1$ and *array2* $= 1$ and 0's indicate otherwise.

4.6 Computing Array Statistics

sum-array *array* *Function*
average-array *array* *Function*
max-array *array* *Function*
min-array *array* *Function*
min-and-max-array *array* *Function*

Functions **sum-array**, **average-array**, **max-array**, and **min-array** return the sum, average, maximum value, and minimum value, respectively, of the elements in an array. If both the minimum and maximum elements are to be computed, function **min-and-max-array** should be used instead, which returns both values while only looping through the array once. All of these functions are implemented using compiler array registers for efficiency.

array-mean *array* *Function*
array-variance *array* *Function*
array-standard-deviation *array* *Function*
array-variance-and-mean *array* *Function*
array-standard-deviation-and-mean *array* *Function*

Functions **array-mean**, **array-variance**, and **array-standard-deviation** compute the mean, variance, and standard deviation (square root of the variance) of the elements of *array*. Function **array-mean** is the same as function **average-array**. Functions **array-variance-and-mean** and **array-standard-deviation-and-mean** return two values, only looping over the array once, and hence are more efficient than calling the single-valued functions separately. In all cases compiler array registers are used for efficiency.

count-elements *array n* *Function*

Function **count-elements** counts the number of elements of *array* which = number *n*.

4.7 Mapping Functions Over Arrays

Functions and macros for mapping functions over arrays are defined in file *maparray*.

map-array *fn-or-expr* &rest *arg-arrays* { :to &rest *to-arrays*} *Macro*

Macro **map-array** maps a function over elements of an array or arrays (*arg-arrays*), returning an array or arrays (*to-arrays*) containing the results. All arrays should be of the same size; they need not be two-dimensional, however.

If keyword *:to* is not specified, a single array of the same size and type as the first *arg-array* is created and returned as the resulting *to-array*. The specification of the *n*-argument function is followed by *n* arrays *arg-arrays*. For example, `(map-array '+ a b)` adds corresponding elements of arrays *a* and *b*, returning a new array containing the results. `(map-array '+ a b :to c)` does the same thing, but stores the results in array *c*. The function can be specified as a function, a symbol, a quoted symbol, or a quoted lambda expression.

If *fn-or-expr* is a Lisp function (*e.g.* `#'square` or `(lambda (x y) (* 2 x y))`) or a symbol (*e.g.* `f`) the resulting code uses **funcall** to compute each value.

```
(map-array f a b :to c) -->
(LET ((%ARG-ARRAY-0 A) (%ARG-ARRAY-1 B) (%VAL-ARRAY-0 C))
  (DECLARE
    (SYS:ARRAY-REGISTER-1D %ARG-ARRAY-0 %ARG-ARRAY-1 %VAL-ARRAY-0))
  (LET ((ARRAY-LENGTH (ARRAY-LENGTH A))
        (INDEX 0)
        %VALUE-0)
    (LOOP REPEAT ARRAY-LENGTH DO
      (MULTIPLE-VALUE (%VALUE-0)
      (FUNCALL F (SYS:%1D-AREF %ARG-ARRAY-0 INDEX)
                 (SYS:%1D-AREF %ARG-ARRAY-1 INDEX)))
        (SYS:%1D-ASET %VALUE-0 %VAL-ARRAY-0 INDEX)
        (INCF INDEX))
    (VALUES %VAL-ARRAY-0)))
```

It is faster if the body of the function (or the name of a primitive function) can be coded inside the loop instead of doing a **funcall** at each iteration. If *fn-or-expr* is a *quoted* lambda expression (*e.g.* `'(lambda (x y) (* 2 x` y))), then the body of the lambda expression is macro-expanded inside the loop, using **setqs**.

```
(map-array '(lambda (x y) (* 2 x y)) a b :to c) -->
(LET ((%ARG-ARRAY-0 A) (%ARG-ARRAY-1 B) (%VAL-ARRAY-0 C))
  (DECLARE
    (SYS:ARRAY-REGISTER-1D %ARG-ARRAY-0 %ARG-ARRAY-1 %VAL-ARRAY-0))
  (LET ((ARRAY-LENGTH (ARRAY-LENGTH A))
        (INDEX 0)
        X
        Y)
    (LOOP REPEAT ARRAY-LENGTH DO
      (SETQ X (SYS:%1D-AREF %ARG-ARRAY-0 INDEX))
      (SETQ Y (SYS:%1D-AREF %ARG-ARRAY-1 INDEX))
      (SYS:%1D-ASET (* 2 X Y) %VAL-ARRAY-0 INDEX)
      (INCF INDEX))
    (VALUES %VAL-ARRAY-0)))
```

For convenience, a primitive function can be specified by simply quoting its name (*e.g.* '+), in which case the function name is macro-expanded inside the loop without using **setqs**.

```
(map-array '+ a b :to c) -->
(LET ((%ARG-ARRAY-0 A) (%ARG-ARRAY-1 B) (%VAL-ARRAY-0 C))
  (DECLARE
    (SYS:ARRAY-REGISTER-1D %ARG-ARRAY-0 %ARG-ARRAY-1 %VAL-ARRAY-0))
  (LET ((ARRAY-LENGTH (ARRAY-LENGTH A))
        (INDEX 0)
        %VALUE-0)
    (LOOP REPEAT ARRAY-LENGTH DO
      (MULTIPLE-VALUE (%VALUE-0)
        (+ (SYS:%1D-AREF %ARG-ARRAY-0 INDEX)
           (SYS:%1D-AREF %ARG-ARRAY-1 INDEX)))
      (SYS:%1D-ASET %VALUE-0 %VAL-ARRAY-0 INDEX)
      (INCF INDEX))
    (VALUES %VAL-ARRAY-0)))
```

Of course, this macro-expanded code will only work fast if compiled. For convenience, though, **map-array** dispatches to pre-compiled functions using **funcall** in the common cases of unary or binary mapping functions which return a single value. Thus an expression which maps a compiled unary or binary function by name can still be evaluated efficiently from a Lisp Listener (*e.g.* `(map-array #'+ a b)`), although not quite as efficiently as if compiled open coded. In all cases, array registers are used for efficiency.

Mapping functions which return multiple values can be used by specifying more than one *to-array* after keyword **:to**. If a quoted lambda expression is used to specify the mapping function, its last outermost sub-expression must be of the form `(values` ⟨*value-expr-1*⟩ ... ⟨*value-expr-m*⟩`)` where m is the number of values returned. Therefore the function specified by
`'(lambda (x y) (incf sum x) (values (+ x y) (* x 2)))`
returns multiple values while the function specified by
`'(lambda (x y) (progn (incf sum x) (values (+ x y) (* x 2))))`
does not.

Scalar attributes of arrays (e.g. the maximum value of an array) can be efficiently computed by using mapping functions which return zero values. Such functions are executed for side-effect, setting some variable or variables defined outside the function's scope. A quoted lambda expression representing a zero-valued function has as its last outermost sub-expression `(values)`. The keyword **:to** is used followed by nothing to indicate that no array should be created for returning values. For example, function **min-and-max-array** is efficiently implemented as follows:

```
(defun min-and-max-array (array)
  (let ((max -infinity) (min infinity))
    (map-array
      '(lambda (x)
          (if (< x min) (setq min x))
          (if (> x max) (setq max x))
          (values))
      array :to)
    (values min max)))
```

Finally, it should be noted that functions of zero arguments can be used as well, with no *arg-arrays* specified. Of course the user must specify at least one *to-array* in this case. For example, `(map-array #'random :to a)` fills array *a* with random numbers.

map-array-offset *fn-or-expr* &rest *arg-arrays* { **:to** &rest *to-arrays* } *Macro*

Macro **map-array-offset** is a special version of macro **map-array** for two-dimensional arrays which allows arrays to be offset relative to one another. Each array is specified by a list (not quoted) consisting of the array optionally followed by a list of two values representing the x and y offsets used for referencing the array. These offsets may be negative, and one cannot be specified without the other. For example, the expression
`(map-array-offset '- (a) (a (1 0)) :to (b))`
computes the first difference of array a in the x direction, storing the result in b.

This macro does not employ wrap-around, so generally values near certain borders of *to-arrays* are not set. Each array must be two-dimensional and of the same size.

Like macro **map-array**, multiple- or zero-valued functions can be used, and the code generated depends on whether *fn-or-expr* is a function or variable, quoted lambda expression, or quoted function name. If keyword **:to** is not present an array the same size as the first is created.

For efficiency, array registers are used. The two-dimensional arrays are addressed linearly (as one-dimensional arrays) without using multiplication to compute their linear indices.

map-over-array *array function* &optional *identity-element* *Function*

Function **map-over-array** maps a function over a single array, combining results and returning a scalar. The identity element for the function must be provided unless the function applied to no arguments is so defined. For example, functions **sum-array** and **max-array** could be implemented using **map-over-array** as follows:

```
(defun sum-array (array) (map-over-array array #'+))
(defun max-array (array) (map-over-array array #'max -infinity))
```

Since this macro calls **funcall** for each array element, it is not as efficient as the in-line code generated by macro **map-array**.

4.8 Synthetic Image Arrays

File *synth* defines functions for creating synthetic image arrays. An array of size $n \times m$ is created by applying a function over a grid of points (x_i, y_j), for $i = 1 \ldots n$ and $j = 1 \ldots m$.

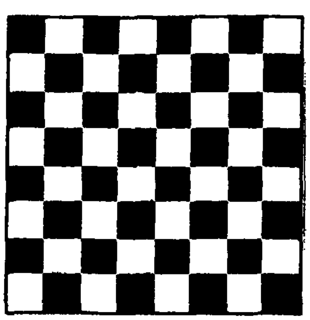
make-syn *function* &key *x-range y-range size type amplify integer result* *Function*

Function **make-syn** makes a synthetic image array. The mapping function specified must take two arguments. Keyword argument *size* specifies the size of the array *(i, j)*, a two-element list which defaults to *(256 256)*. The x and y ranges of the grid points are specified as lists of the form *(low high)* using keyword arguments *x-range* and *y-range*; *x-range* defaults to the interval *(0 1)* and *y-range* defaults to *x-range*. To be precise, the interval [*low high*] is used as the grid range in each dimension. The grid range is independent of the size of the array; the grid range corresponds to the section of the surface constructed, while the array size corresponds to the resolution of the discrete representation.

Function *function* is applied to every element of the grid with result stored in array *result*. Either *result* or its *type*, which defaults to *art-8b*, can be specified using keyword arguments.

To make the range of *function* match the range of array *result*, the values are rounded to the nearest integer if *result* is a *bit-array* (but are not if *result* is an *art-q* array); they are also multiplied by *amplify*, which defaults to 1.

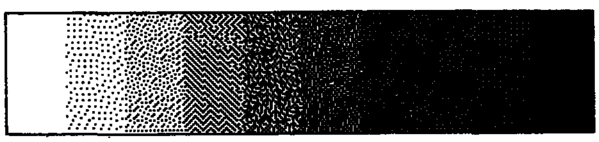
```
(defun check (x y)
  (if (eq (evenp (rounddown x)) (evenp (rounddown y))) 1 0))
(display-array (make-syn #'check :x-range '(0 8) :type art-1b))
```



map-syn-1 *function* &key *x-range size type amplify integer result* *Function*

Function **make-syn-1** makes a synthetic image array which varies in only one dimension. A mapping function of one argument is specified which is applied in the x direction. Like function **make-syn**, keyword arguments specify the range of x; the result array, its size, and/or type; amplification factor, and whether *result* should contain only integer values. This function has the same defaults as function **make-syn**, except that *size* defaults to *'(256 128)*.

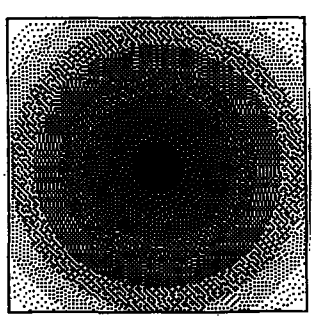
```
(defun staircase (x)
   (cond ((< x 0) 0)
         ((< x 1) (// (fix (* 10.0 x)) 10.0))
         (t 1)))
(display-array (make-syn-1 #'staircase :size '(200 50) :amplify 255))
```



make-syn-r *function* &key *range size type amplify integer result* *Function*

Function **make-syn-r** makes a square synthetic image array which is rotationally symmetric. A mapping function of one argument (r) is specified and applied in the radial direction. Keyword argument *range* specifies the upper bound of r in the x direction, which defaults to 1; to be precise, the interval (0 *range*] is used. The *size* of the square array returned can be specified as a single integer; it defaults to *256*. As with function **make-syn** keywords can also be used to specify the *result* array, amplification factor *amplify* and whether *result* should contain only integer values.

```
(display-array (make-syn-r #'staircase :range 0.6))
```



4.9 Histograms

Functions for making and using histograms are defined in file *histogram*. Histograms are implemented using Flavors.

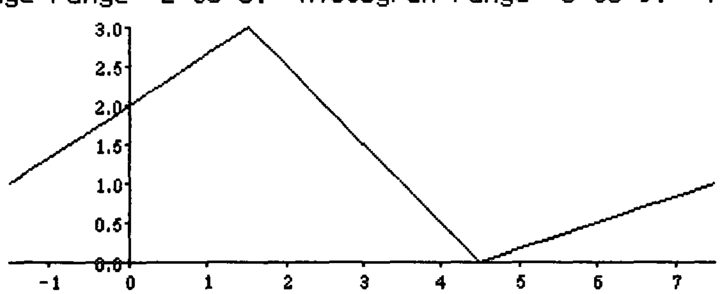
make-histogram *image* &key *range num-buckets amp-factor min-image max-image graph* *Function*

Function **make-histogram** makes and returns a histogram of array *image*, which need not be two-dimensional. The *range* of values histogrammed defaults to an interval which is large enough to contain the minimum and maximum values of *image*, which are computed unless specified by keyword arguments *min-image* and *max-image*. The actual default range may be larger to insure that each bucket size is an integral value, preventing uneven bucket sizes for fixed-point image arrays. Alternatively, *range* can be specified as a list of two numbers, in which case the minimum and maximum value of *image* need not be computed. Exactly *num-buckets* buckets are used, which defaults to 100. A graph of the histogram is plotted (the default) unless *graph* is **nil**.

Argument *amp-factor* is recorded, but is ignored by all histogram methods; it is used by functions which call these methods when *image* is a scaled version of the actual array desired, such as a fixed-point array output by the convolver.

The following simple histogram is to illustrate the methods described below.

```
> (setq hist (make-histogram (vector -2 8 0 1 1) :num-buckets 4))
Image range=-2 to 8.  Histogram range=-3 to 9.  4 buckets of size 3.
```



make-positive-histogram *image* &key *range num-buckets amp-factor min-image max-image graph* *Function*

Function **make-positive-histogram** is a version of function **make-histogram** especially tailored for positive arrays (specifically, bit arrays). The range starts at 0 by default, so only the maximum-value of *image* is computed if neither *range* nor *max-image* is specified.

:image-min *Method of* **histogram**
:image-max *Method of* **histogram**

Methods **:image-min** and **:image-max** return, respectively, the minimum and maximum values of the array histogrammed, if they were ever computed.

```
(send hist :image-min) ==> -2
(send hist :image-max) ==> 8
```

:num-buckets *Method of* **histogram**
:bucket-size *Method of* **histogram**

Methods **:num-buckets** and **:bucket-size** return, respectively, the number of buckets and size of buckets in the histogram.

```
(send hist :num-buckets) ==> 4
(send hist :bucket-size) ==> 3
```

:buckets *Method of* **histogram**

Method **:buckets** returns a one-dimensional array of length *:num-buckets* containing the number of elements in each bucket.

```
(pp (send hist :buckets))
  1  3  0  1
```

:bucket *value* *Method of* **histogram**

Method **bucket** returns the number of elements of the bucket which represents image value *value*, *i.e.*, the bucket which would be incremented for image value *value*. Note that `(send h :bucket value)` does not return the same result as `(aref (send h :buckets) value)`.

```
(send hist :bucket 0.5) ==> 3
```

:total *Method of* **histogram**

Method **:total** returns the total number of elements histogrammed.

```
(send hist :total) ==> 5
```

:cumulative-buckets *Method of* **histogram**

Method **:cumulative-buckets** returns an array of length *num-buckets* which represents the cumulative distribution of *buckets*.

```
(pp (send hist :cumulative-buckets))
  1  4  4  5
```

:bucket-bounds *Method of* **histogram**

Method **:bucket-bounds** returns an array of length 1+*num-buckets* which represents boundaries of ranges of each bucket. $Bucket_i$ represents values x such that $bucket\text{-}bound_i \leq x < bucket\text{-}bound_{i+1}$.

```
(pp (send hist :bucket-bounds))
  -3  0  3  6  9
```

:bucket-midpoints *Method of* **histogram**

Method **:bucket-midpoints** returns an array of length *num-buckets* representing the midpoints of ranges of each bucket.

```
(pp (send hist :bucket-midpoints))
  -1.5  1.5  4.5  7.5
```

:amp-factor *Method of* **histogram**

Method **amp-factor** returns value *amp-factor* specified when creating the histogram.

```
(send hist :amp-factor) ==> 1
```

:percentile-of-value *value* *Method of* **histogram**

Method **:percentile-of-value** returns the approximate percentage of image values less than *value.* The percentage is expressed as a fraction between 0 and 1.

```
(send hist :percentile-of-value 3) ==> 0.8
```

:value-at-percentile *fraction* *Method of* **histogram**

Method **:value-at-percentile** is the inverse of method **:percentile-at-value.** It returns the image value at a percentile specified by a *fraction* between 0 and 1, *i.e.*, a value such that about $100 \times fraction$ percent of image values are less than value.

```
(send hist :value-at-percentile 0.8) ==> 3.0
```

:graph &key *size y-origin* &rest *keywords-and-args* *Method of* **histogram**

Method **:graph** graphs the histogram on the current window. Its keyword arguments and their defaults are the same as those of function **plot**, described in Section 9, except that *size* defaults to *'(400 150)* and *y-origin* defaults to *0.*

5 Image Representation

This section describes how images are represented and stored. Here, the word "image" refers not only to grey-level images, but to any description of a scene. The functions documented here are defined in file *load*.

5.1 Image Representation in Lisp

Images are represented as objects, using Flavors. A grey-level image is represented by flavor **grey-image**, which contains slots *image-type*, *name*, *documentation*, *width*, *height*, *array-type*, and *image-array*. This description is somewhat redundant, since array dimensions and type can be determined from the image array, but it lets an image be clearly described by the **describe** function, for example:

```
IMAGE-TYPE:    GREY-IMAGE
NAME:          Westminster Abbey
DOCUMENTATION: Taken by John Canny on 04-01-82
WIDTH:         512
HEIGHT:        600
ARRAY-TYPE:    ART-8B
```

Any of these slots can be accessed by sending a message to the object.

make-grey-image *name array documentation* — *Function*

> Function **make-grey-image** constructs and returns an instance of flavor **grey-image**, using image array *array* with specified *name* and *documentation* strings. For example, if *westminster* is bound to the image array, the **grey-image** described above could have been created by evaluating
>
> ```
> (make-grey-image "Westminster Abbey" westminster "Taken by John Canny
> on 04-01-82")
> ```

Special-purpose image flavors can be defined in a similar fashion. For example, a flavor **fingerprint** is defined for representing multi-scale edge images. It includes slots for storing a list of edge images at different scales, a list of scales, and the number of scales. Methods are also defined for accessing the three-dimensional scale-space image in a variety of ways; this is why flavors are used instead of structures to represent images. Details and the design philosophy of this representation of images are discussed in Section 2 of Voorhees [1984].

The image processing functions described in this paper operate on image *arrays*, not *images*. Only the array data are relevant to these functions, and the construction of images with associated parameters is unnecessary at this stage. If desired, higher-level functions which operate on images can be defined which call the lower-level functions described here.

5.2 A More Flexible Image Representation

In the future, a more general image representation may be implemented. Images would be represented by a structure of two elements. The first element would be an association list or property list of attributes and values, and the second element would be the image intensity array (or arrays). In addition to required attributes, such as image size and name, any other attributes, such as image statistics would be stored in the attribute list. This more flexible representation allows any attributes to be added to an image without changing the definition of the image structure. Functions which operate on images which require optional image attributes would first check to see if the attribute had been computed; if not, the attribute, once computed, would be memoized by adding it to the image attribute list, so it need not be computed again. Furthermore, a facility which keeps track of and saves modified images should be implemented.

5.3 Image Representation in Files

Images are stored in a file in two parts: a header and the data part. The header contains information about the object in the file, such as its type, name, documentation and size. It provides documentation about the contents of the file for the user as well as information needed to read the rest of the file.

Functions for loading, saving, and describing images, defined in file *load*, are described here. Each function is "smart" about pathname completion; an incomplete pathname is merged with the last pathname specified, and the initial pathname is merged with a default (determined by the global variable ***default-image-pathname***). Thus the user usually needs to specify only the first name of a file.

save-image *image pathname* *Function*

Function **save-image** saves an image into a file. Currently defined image types are **grey-images** and **fingerprints**. The data part is written to the file as a single block for efficiency. The complete pathname is returned.

If *dectalk* is loaded, Dectalk reads the image name and documentation as the image is loading, possibly distracting the user from noticing the time taken to read the image from disk.

load-image *pathname* *Function*

Function **load-image** reads an image from a file. The type of image is determined from the file contents automatically. The data part is read from the file as a single block for efficiency. The image object is returned.

load-image! *pathname* &optional *symbol* *Function*

Function **load-image!** is a special version of function **load-image** for loading **grey-images**. It binds the image array to a symbol which defaults to the first name of the file. The same symbol catenated with "-gr" is bound to the entire image object.

```
(load-image! "pookie") -->
(set 'pookie (send (set 'pookie-gr (load-image "pookie"))
:image-array))

(load-image! "pookie" 'poo) -->
(set 'poo (send (set 'poo (load-image "pookie")) :image-array))
```

print-header *pathname* *Function*

Function **print-header** is used to examine a description of the contents of an image file, without taking the time or storage to load the actual image data. This is the main advantage of storing the image header separate from the image data in the file.

Details and the design philosophy of image representation in files is discussed in Section 3 of Voorhees [1984].

6 Talking Functions

System *dectalk* defines functions for using the Dectalk speech synthesizer. The file *dectalk* loads this system and defines some interface functions outside the package **dectalk**. Of course, this file should be loaded only when a Dectalk is connected to the Lisp machine.

say *string* *Function*

Function **say** makes Dectalk say *string*, unless silenced.

voice *new-voice* *Function*

Function **voice** changes Dectalk's voice to *new-voice*. Defined voices are **:hal**, **:betty**, **:harry**, **:frank**, **:kit**, and **:val**.

shut-up *Function*

Function **shut-up** makes function **say** a no-op, silencing the Dectalk in the future.

speak *Function*

Function **speak** undoes the effect of function **shut-up**, reactivating function **say**.

If a Dectalk is not connected or used with the Lisp machine, file *nodectalk* should be loaded instead. This file defines function **say** and **voice** as no-ops. This allows functions which call **say** and **voice** to function property without the Dectalk system.

7 Screen Display Utilities

This section describes functions for displaying results on Noble Larson's 8-bit color "grey screen" and the Lisp machine console.

7.1 Grey Screen Display

A number of functions have been written to simplify use of Noble Larson's 8-bit color "grey screen". These functions are defined in file *grey*, which automatically loads file *greycolor* and Noble's code which drives the grey screen. The function **grey:help** documents Noble's functions.

display-array *array* &optional *(x 0) (y 0)* *Function*

> Function **display-array** displays a two-dimensional array on the grey screen at position *(x, y)* on the grey screen, which defaults to the upper left-hand corner. Type *art-1b* arrays are displayed using the brightest and darkest intensities (255 and 0); *art-2b* and *art-4b* arrays are displayed using the most significant bits of the screen; and the upper 8 bits of *art-16b* or *art-fixnum* arrays are displayed. Type *art-8b* arrays, of course, are displayed as is. Type *art-q* arrays are displayed by calling function **enhance-array**, which maps the minimum and maximum values of the array to the darkest and brightest intensities, scaling all values linearly.
>
> If the default scaling method is not appropriate, the user can scale the array himself to an *art-8b* array before calling **display-array**. Functions **convert-array-type**, **scale-array**, and **enhance-array** are useful for this purpose.

overlay-array *array* &optional *(x 0) (y 0) plane* *Function*

> Function **overlay-array** is used for displaying an *art-1b* array on a particular plane of the screen at position *(x, y)*. It is commonly used for overlays. The default value of *plane* is **grey:grey-array0**, the least significant bit plane.
>
> Using a straight grey map (Noble's function **grey:straight-map**), a 1-bit image (e.g. an edge image) can be overlaid onto the least significant bit plane of an 8-bit image without affecting the perceived intensity values. The overlaid plane can be highlighted by calling (**grey:highlight-plane** *plane r g b*), or it

can be viewed exclusively by calling (**grey:select-plane** *plane r g b*), where *r*, *g*, and *b* are optional intensity values; if not specified, they default to produce white.

Philippe Brou's grey screen functions, not documented here, provide a cleaner method of performing grey screen overlays.

erase-array *array* &optional *(x 0) (y 0)* *Function*

Function **erase-array** erases an area at position *(x, y)* on the grey screen. The size of the area erased is that of the array.

erase-plane *array* &optional *(x 0) (y 0) zero plane* *Function*

Function **erase-plane** erases an area at position *(x, y)* of a plane of the grey screen, which defaults to **grey:grey-array0**. The size of the area erased is that of the array. The area is filled with zeros by default, but if *zero* is **nil** it is filled with ones instead.

extract-array &optional *width-multiple height-multiple* *Function*

Function **extract-array** extracts a portion of the grey screen array using the mouse, returning an array. Upon invoking the function, a rubber box cursor appears on the grey screen, which the user sizes with the mouse. After clicking left once the user positions the box and clicks left again to extract the array. The operation can be aborted by clicking middle instead.

The box size is rounded up to the nearest multiple of optional arguments *width-multiple* and *height-multiple*, which default to 32 and 1, so that any arrays of this size are *bitbltable*. These arguments can also be used to extract an array of a desired size (*e.g.* 256×256).

It is often convenient to display a number of images side by side on the grey screen, without having to manually compute their positions. A number of functions for automatically displaying images of the same size have been implemented. These functions are convenient for displaying a sequence of images of the same size, such as intermediate results of a computation.

make-grid *array* &optional *(x-margin 8) (y-margin x-margin)* *Function*

Function **make-grid** computes a list of coordinates for displaying arrays whose size is that of *array* on the grey screen. Optional arguments specify the widths of margins between the displayed arrays; extra arrays can sometimes be squeezed onto the screen by using negative margin widths. Argument *array* can be either an array or an image.

auto-display *array* *Function*

Function **auto-display** automatically displays an array (or image) at the next location on the grey screen grid constructed by function **make-grid**. It cycles back to the first location if the screen is full. (If **make-grid** has not been called, it always displays *array* at the upper left-hand corner of the screen). If *array* is an image, its name is also printed over a corner of the array.

auto-erase *array* *Function*

Function **auto-erase** automatically erases an array (or image) where the last one was drawn. The next call to **auto-display** will use this position. The size of the area to be erased is determined from *array*.

7.2 TV Console Display

File *plot* contains a function for displaying images on the Lisp machine TV console.

tv:display-array *array* &optional *position* *Function*

Function **tv:display-array** displays an array on the Lisp machine console. The array is displayed on the currently selected window at *position* (a list of two numbers), which defaults to the position directly under the cursor. If the default position is used the cursor is repositioned under the displayed array so the display will not be overwritten.

Currently, this function only supports *art-1b* arrays, although this limitation will be removed in the future by using a dithering algorithm.

8 Convolution Utilities

A number of utilities exist for performing convolutions either with or without a hardware convolver. File *gcon* defines a number of functions for using Noble Larson's digital hardware convolver to do convolutions using Gaussian and difference-of-Gaussian (DOG) masks. File *softcon* defines functions for doing convolution in Lisp without special-purpose hardware. Both files can be loaded simultaneously, since each set of functions has different capabilities. Both files *gcon* and *softcon* load file *dog*, which defines generic Gaussian and DOG functions which dispatch to the hardware functions if loaded, and to the software functions otherwise. These generic functions are described first.

8.1 Generic Gaussian and DOG Convolution

convolve-gauss *image* σ &key *zero-bc show-progress use-hardware result* *Function*

Function **convolve-gauss** convolves an *art-8b* image array *image* with a 2-D Gaussian mask,

$$G(\sigma; x, y) = \frac{1}{2\pi\sigma^2} exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

of specified scale σ pixels. The eight-bit result S and a scaling factor k are returned such that $I(x,y) * G(\sigma; x, y) = kS(x,y)$ where $I(x,y)$ is the eight-bit image array. If the hardware convolver is used, σ must be less than 4.65 pixels.

The following keyword argments can be specified:

:use-hardware (**t** or **nil**) tells whether to use the hardware convolver. By default the hardware convolver is used if file *gcon* is loaded.

:zero-bc (**t** or **nil**) tells whether to use zero boundary conditions. Its default value is **nil**, in which case the borders of *result* should be ignored.

:result specifies an *art-8b* array for returning the result, which must be the same size as *image*. If not specified, one is allocated.

:show-progress (**t** or **nil**) tells whether to print a message on the screen which shows the progress of the software convolvution. Its default value is **t**.

If software convolution code is used, the legal values of certain arguments are less restricted. Also, the exact result values returned by **convolve-gauss** depend on whether hardware or software code is used. See functions **gcon-gauss** and **softcon-gauss** for their respective restrictions and effects.

convolve-dog *image* σ &key *sign-bits zero-bc show-progress use-hardware result* *Function*

Function **convolve-dog** convolves an *art-8b* image array *image* with a 2-D difference-of-Gaussian (DOG) mask $G(\sigma_+; x, y) - G(\sigma_-; x, y)$ which approximates a Laplacian of Gaussian of size σ, $\nabla^2 G(\sigma; x, y)$ [Marr and Hildreth, 1980]. In addition to the result S, a scaling factor k is returned such that $I(x, y) * \nabla^2 G(\sigma; x, y) = kS(x, y)$ where $I(x, y)$ is the eight-bit image array.

The following keyword arguments can be specified:

:sign-bits (**t** or **nil**). If **t**, the default, only sign bits are returned in an *art-1b* array; if **nil**, signed integers are returned in an **art-q** array. (The second value returned, k, is only meaningful when *sign-bits* is **nil**.)

:zero-bc (**t** or **nil**) tells whether to use zero boundary conditions. If **nil**, the default, zero boundary conditions are not used and the borders of *result* should be ignored.

:result specifies an *art-1b* or *art-q* array, depending on the value of *sign-bits*, for returning the result, which must be the same size as *image*. If not specified, an array of the appropriate size and type is allocated.

:show-progress (**t** or **nil**) tells whether to print a message on the screen which shows the progress of the convolvution. Its default value is **t**.

If software convolution code is used, the legal values of certain arguments are less restricted. Also, the exact result values returned by **convolve-dog** depend on whether hardware or software code is used. See function **gcon-dog** and **softcon-dog** for their respective restrictions and effects. Table 1 shows the amounts of time taken to perform sample DOG convolutions with and without hardware.

The global variable ***space-constant-ratio*** controls the ratio σ_-/σ_+ [Marr and Hildreth, 1980]. Its default value is 1.6. Different values can be used when doing convolutions in software, but the hardware code is not currently guaranteed to work for values other than 1.6.

Table 1: Sample software and hardware DOG convolution times for 576 x 454 image

Scale	$\sigma = 1.0$	$\sigma = 4.0$
Software mask size (pixels)	9×9	35×35
Hardware iterations	1	2
Software convolution time*	36.0 sec	97.0 sec
Hardware convolution time		
Sign bits, quick b.c.	0.6 sec	1.3 sec
Sign bits, zero b.c.	0.9 sec	2.0 sec
Signed integers, quick b.c.	1.2 sec	1.9 sec
Signed integers, zero b.c.	3.6 sec	4.8 sec

*Sign bit mode and boundary conditions do not affect software convolution time significantly.

8.2 Sign and Zero Crossing Functions

sign-array *array* &optional *result* *Function*

Function **sign-array** returns an *art-1b* array *result* containing sign bits of *art-q* array of numbers *array*. Elements of *result* equal 1 when corresponding elements of *array* are negative, and 0 otherwise. If *result* is not specified, an *art-1b* array the same size as *array* is allocated.

zc-array *sign-array* &optional *result* *Function*

Function **zc-array** returns a zero crossing array *result* corresponding to a sign bit array *sign-array*. Zero crossings are marked (using 1's) in *result* at points where the value of *sign-array* differs from its east or south neighbors; *result* is zero everywhere else. If *result* is not specified, an *art-1b* array the same size as *sign-array* is allocated.

8.3 Software Convolution

Functions defined in file *softcon* perform convolution without using special-purpose hardware. First, functions for convolving in one and two dimensions with arbitrary masks are described.

convolve *image mask* &key *result zero-bc bit-pos show-progress* *Function*

Function **convolve** convolves image array *image* with a general mask represented by a 2-D array *mask*, and returns *result*. Both arrays *image* and *mask* can be of any array type, but the function works fastest if both are fixed-point arrays.

The following keywords can be specified:

:result specifies an array for returning result, which must be of the same size as *image*, but of any type. If not specified, an array whose type is that of *image* is allocated.

:bit-pos The integer *bit-pos* least significant bits are dropped from the convolution values before being stored in array *result*. If not specified, function **bit-position-for-mask** is called to choose a bit position based on *mask* and the array types of *image* and *result*.

:zero-bc tells whether to use zero boundary conditions. If **nil**, the default, convolution values are not computed at points where *mask* would extend beyond the boundary of *image*. Values of *result* at these points near the border are zero. Alternatively, if *zero-bc* is **t**, zero boundary conditions are used by padding the image array with zeros, and convolution values are computed at every point of *result*.

:show-progress (**t** or **nil**) tells whether to print on the screen the row number currently being convolved. Its default value is **t**.

convolve-1d *signal mask* &key *result zero-bc bit-pos* *Function*

Function **convolve-1d** convolves a 1-D array *signal* with a 1-D array *mask*. Both arrays *signal* and *mask* can be of any array type, but the functions works fastest if both are fixed-point arrays. As with function **convolve**, keywords **:result**, **:bit-pos**, and **:zero-bc** specify the 1-D result array, the number of bits to be dropped, and boundary conditions.

bit-position-for-mask *mask image-array-type result-array-type* *Function*

Function **bit-position-for-mask** computes the number of least significant bits to be dropped when convolving an image of type *image-array-type* with *mask* to a result of type *result-array-type*. The bit-position increases as the area of the mask increases and as the *image-array-type* increases, and it decreases as the *result-array-type* increases. The function assumes that the full range of the image and result arrays are used. For convenience, *image-array-type* and *result-array-type* can be arrays rather than array types.

Functions for convolving with 2-D Gaussian and difference-of-Gaussian masks are described next.

softcon-gauss *image* σ &key *zero-bc show-progress result* *Function*

Function **softcon-gauss** convolves an image array *image* with a 2-D gaussian mask of scale σ pixels. Arrays *image* and *result* can be of any fixed-point array type. Like function **convolve-gauss**, **softcon-gauss** returns both *result* and a scaling factor. Unless the convolution is always to be done in software, function **convolve-gauss** should be used instead.

To perform the 2-D Gaussian convolution, the function convolves *image* in the x and y directions with an eight-bit 1-D Gaussian mask. Unlike function **gcon-gauss**, the mask is of odd length so the result is perfectly aligned with the image.

The following keyword can be specified:

:zero-bc tells whether to use zero boundary conditions. If **nil**, the default, convolution values are not computed at points where *mask* would extend beyond the boundary of *image*. At these points near the border *result* is set to zero. Alternatively, if *zero-bc* is **t**, zero boundary conditions are used by padding the image array with zeros, and convolution values are computed at every point of *result.*

:show-progress (t or **nil**) tells whether to print on the screen the row number currently being convolved. Its default value is **t**.

:result specifies an array for returning the result. It must be of the same size as image, but need not be of the same type. Provided that is a fixed-point array type, the convolution values are scaled appropriately. If not specified, an array of the same type as *image* is allocated and returned.

softcon-dog *image σ* &key *sign-bits zero-bc show-progress interm-result-type result* *Function*

Function **softcon-dog** convolves an image array *image* with a 2-D difference-of-Gaussian (DOG) mask approximating a Laplacian of a Gaussian of scale σ pixels. Like **convolve-dog**, **softcon-dog** returns both a result and a scaling factor. Unless the convolution is always to be done in software, function **convolve-dog** should be used instead.

To perform the DOG convolution, two Gaussian convolutions are performed using decomposition (as described above) to yield two intermediate results, which are subtracted to compute the result. Unlike function **gcon-dog**, the masks are of odd length so the result is perfectly aligned with the image.

The following keywords can be specified:

:sign-bits tells whether to only return sign bits. If **t**, the default, an *art-1b* array of sign bits is returned (1 means "negative"). If **nil**, an *art-q* array of signed integers is returned. Note that the second value returned, the scaling factor, is only meaningful in the latter case.

:zero-bc tells whether to use zero boundary conditions. If **nil**, the default, convolution values are not computed at points where the larger (negative) mask would extend beyond the boundary of *image*. Values of *result* at these points near the border should be ignored. Alternatively, if *zero-bc* is **t**, zero boundary conditions are used by padding the image array with zeros, and convolution values are computed at every point of *result*.

:result specifies an array for returning the result, which must be the same size as *image*. If sign bit mode is used *result* should be of type *art-1b*; otherwise, *result* must be of type *art-q*. If not specified, an array of the same type as *image* is allocated and returned.

:interm-result-type specifies the type of the intermediate result arrays, which should be at least as high as that of the image to maintain precision. Its default value is the next type higher than the type of *image*, *e.g. art-16b* for an *art-8b* image.

:show-progress (**t** or **nil**) tells whether to print on the screen the row or column number currently being convolved for each direction. Its default value is **t**.

8.4 Hardware Convolution

A number of functions are defined for using Noble Larson's digital Gaussian convolver. With this special-purpose hardware attachment to the Lisp machine, one can perform Gaussian convolutions much faster than in software. For example, a 500×500 size image can be convolved with a mask of size 32×32 and sign bits returned in less than one second. (Table 1 on page 46 gives a comparison of DOG convolution times with and without the hardware convolver.) Nishihara and Larson [1981] give an overview of the convolver design.

The file *gcon*, which contains the functions described here, automatically loads file *oz:⟨ngl⟩gcon*, which contains basic functions for running the convolver. The function **gcon:help** describes Noble's functions.

General convolution function are described first. Two steps are needed to make the convolver work: First, the appropriate mask must be loaded. Then the image data is pipelined through the convolver to compute the result.

The convolver convolves the image with two decomposable 2-D masks, and returns the difference of the two. Thus a difference-of-Gaussian convolution can be performed in a single operation. Each of the 2-D masks is specified by two 1-D masks (which may differ), the cross-product of which yields the desired 2-D mask. Since different 1-D masks can be used in the x and y directions, 4 masks are specified altogether. Each of these 1-D masks must be symmetric, so only half the mask is actually specified. Each of these half masks is of length 16, so the effective size of the 2-D mask is 32×32 pixels.

The negative convolution result is divided by two before it is combined with the positive result, so the negative 1-D mask specification must scaled by $\sqrt{2}$. All half-masks are specified as 8-bit integer arrays of length 16. Since the masks contain an even number of elements, the convolution result is shifted by half a pixel in both the horizontal and vertical directions.

The following functions load masks into the convolver to perform convolution with isotropic 2-D masks:

load-masks *mask+ mask−* *Function*

Function **load-masks** loads the convolver with positive and negative half masks. Each of the two masks is used in both the x and y directions.

load-mask *mask* *Function*

Function **load-mask** loads the convolver with a positive half mask. A negative mask of zeros is used, so the entire result is non-negative. The same mask is used in both the x and y directions.

load-gauss-mask σ *Function*

Function **load-gauss-mask** load the convolver with the appropriate masks to yield a 2-D Gaussian mask of specified scale σ. The full amplitude range (0 to 255) is used, and a scaling factor is returned.

load-dog-mask σ *Function*

Function **load-dog-mask** loads the convolver with the masks which yield a 2-D difference-of-Gaussian which approximates a Laplacian of Gaussian of scale σ. The full amplitude range is used, and a scaling factor is returned.

Once the masks are loaded, a convolution routine is called with the image data. Two parameters can be specified: the boundary conditions used and the type of result to be returned.

For each of the functions described below, the keyword **:zero-bc** specifies the boundary conditions to be used. If **t**, zero boundary conditions are implemented by copying *image* to a larger, temporary array, padded with zeros. This large array is fed to the convolver, which returns a large array, from which an image-size array is extracted and returned. By default *zero-bc* is **nil**, and toroidal, time-dependent boundary conditions are used, which avoid the need for extra copying. In this case, the boundary of the result should be ignored. The keyword **:result** specifies an array for returning the result; if not specified, an array of the appropriate size and type is allocated.

Since the image range is 8 bits, the effective 2-D mask range 16 bits, and the effective 2-D mask size $2^5 \times 2^5$, the actual mathematical result would contain $8+16+5+5 = 34$ bits. The convolver drops the 18 least significant bits during its calculation, computing a result of 16 bits per pixel. The convolver can be run in one of four modes, depending on the actual type of result to be returned. The following four functions convolve an image array of any *bitbltable* size up to $1024 \times \infty$.

gcon-half-word *image* &key *zero-bc result* *Function*

The full 16 bits per pixel is returned as an *art-16b* array. Since this is stored in two's complement form (the most significant bit is a sign bit) this format isn't very useful for doing calculations.

gcon-float *image* &key *zero-bc result* *Function*

The 16-bit result is returned as an *art-q* array of positive and negative integers, a useful (although space-consuming) representation.

gcon-byte *image* &key *bit-pos zero-bc result* *Function*

A contiguous 8-bit field is extracted from each 16-bit value, and returned in an *art-8b* array. The position of the lowest-order bit is specified by keyword **:bit-pos**. By default, the highest-order bits are extracted. The result is useful only if an entirely positive mask is used, or if the highest-order bits are extracted, since meaningful sign bits will be dropped otherwise. This mode is useful if the result is to be fed back into the convolver, or for display on the grey screen.

gcon-sign *image* &key *zero-bc result* *Function*

Only the sign bits are returned, as an *art-1b* array. This space-efficient result is useful for computing zero crossings.

The above functions were used to define functions for convolving image arrays with 2-D Gaussian and difference-of-Gaussian masks, which are described next.

gcon-gauss *image* σ &key *zero-bc result* *Function*

Function **gcon-gauss** convolves an 8-bit image array with a 2-D Gaussian mask of specified scale σ. Since the hardware convolve has a limited mask size, σ must be less than 4.65 pixels. Like function **convolve-gauss**, an eight-bit result and a scaling factor are returned. The scaling factor will be approximately one, and *result* will have the same range as *image*, at values of σ given by ***good-gauss-sigmas*** $= 0.94$, 1.48, 2.18, 3.14, and 4.48 pixels. Unless the hardware convolver is always to be used, generic function **convolve-gauss** should be used instead.

Since only even-sized masks can be used, the result is shifted relative to the image down and to the right by one-half pixel.

The following keyword arguments can be specified:

:zero-bc tells whether to use zero boundary conditions. If **nil** (the default) toroidal, time-dependent boundary conditions are used. If t, zero boundary conditions are used by padding *image* with zeros, which involves some extra copying.

:result specifies an *art-8b* array for returning the result, which must be the same size as *image*. If not specified, an array is allocated.

gcon-dog *image* σ &key *sign-bits zero-bc show-progress result* *Function*

Function **gcon-dog** convolves an 8-bit image array with a 2-D difference-of-Gaussian mask. Like function **convolve-dog**, both a result and a scaling factor are returned. Unless the hardware convolver is always to be used, generic function **convolve-dog** should be used instead.

Convolution with large masks is accomplished automatically by smoothing the image with a sequence of Gaussians followed by a smaller difference-of-Gaussian convolution. Only ***good-gauss-sigmas*** are used, so precision is not lost through attenuation. However, roundoff error does accumulate, so no more than six iterations (five Gaussian, one DOG) can be executed. This limits σ to 10.64 pixels or less, corresponding to a maximum central width of $w = 2\sqrt{2}\sigma \approx 30$ pixels. Also, values of σ less than about 0.7 ($w < 2$) usually give noisy results.

Since even-sized masks must be used, the result is shifted relative to the image by one-half pixel down and to the right *for each iteration.*

The following keyword arguments can be specified:

:sign-bits tells whether to use sign bit mode. If t, only sign bits are returned in an *art-1b* array. If **nil**, an *art-q* array of signed integers is returned.

:result specifies an array for returning the result. The array must be of type *art-1b* if *sign-bits* is t and of type *art-q* if *sign-bits* is **nil** and must be the same size as *image*. If not specified, an array of the appropriate size and type is allocated.

:zero-bc tells whether to use zero boundary conditions. If **nil** (the default) toroidal, time-dependent boundary conditions are used. If **t**, zero boundary conditions are used by padding *image* with zeros, which involves some extra copying.

The hack described below shows the hardware convolver running at full speed.[1]

fast-con *channel* &optional *(sigma 3.0) (zc nil)* *Function*

The function **fast-con** shows the hardware convolver running at full speed. It continually grabs images from a TV camera, convolves them with a DOG mask, and displays sign bits or zero crossings in a special window (which it creates automatically) on the TV console.

The TV camera must be connected to channel *channel*, an integer between 0 and 3; see Noble's function **grey:grab-frames**. A DOG of standard deviation *sigma* is used; for speed, *sigma* should be less than 4.65. Sign bits are displayed by default, but zero crossings are displayed if *zc* is **t**.

[1] Actually, the speed is limited by the time the Lisp machine takes to write and read data to and from the convolver; the hardware convolver itself can run considerably faster.

9 Plot Utility

Some functions for plotting one-dimensional data are defined in file *plot*. The functions have been designed so that the plots can be made simply be specifying the data points; details such as axes numbering and graph position can be computed automatically. The user has control over formatting details, if desired, through the use of optional keyword arguments. Many of these optional arguments default to the values of global variables, providing a convenient way of changing the default permanently. For example, the keyword **:title-font** can be used to change the font of the title of a particular graph, or the value of global variable ***title-font*** can be used to change the default font for all graphs.

plot *x-values y-values* &rest *keywords-and-args* *Function*

Function **plot** plots list *x-values* versus list *y-values*.

The following keywords can be used:

:x-range, :y-range, each a list of two numbers, specify the range of data to be plotted on each axis. By default the full range of data is plotted and no more. If a restricted *x-range* is specified but *y-range* is not, *y-range* is adjusted accordingly.

:x-origin, :y-origin, each a number, specify the origin where the axes cross. The default value of each is 0.

:x-interval, :y-interval specify the interval between axes numbers (and ticks) on each axis. By default the intervals are chosen so that approximately five ticks (the values of variables ***approx-number-x-, y-ticks***) appear on each axis.

:x-numbers, :y-numbers can be used instead of **:x-interval** and **:y-interval** to explicitly specify a list of numbers along each axis. This option therefore facilitates irregularly-spaced axis numbers.

:x-number-format, :y-number-format specify the format strings for formatting the numbers along each axis.

:x-label, :y-label, and **:title** specify axis labels and a title.

:x-label-font, :y-label-font, and **:title-font** specify the fonts for printing the labels and title; ***label-font*** and ***title-font*** are used by defaultd. If only *x-label-font* is specified, *y-label-font* defaults to it.

:curve specifies how the curve between points is to be drawn. Straight lines are drawn between each pair of data points by default, but the keyword **:curve** can be followed by **:line**, **:spline**, or **:none** to control this option.

:points specifies how data points should be drawn. Points are not drawn by default; but the keyword **:points** can be followed by **:none**, **:dots**, **:circles**, **:triangles**, or **:squares** to control this option.

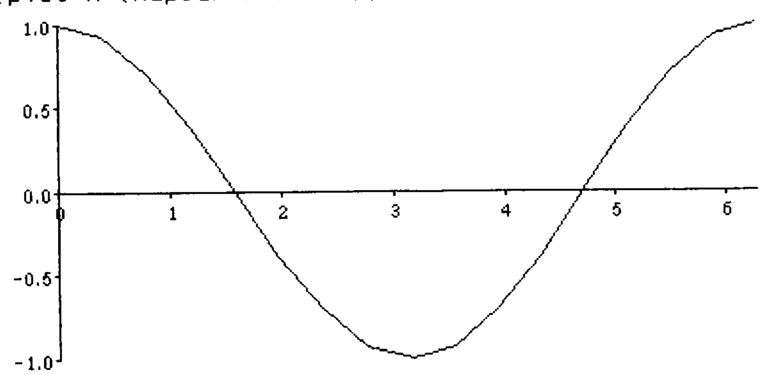
:graph-window and **:position** control the window and position on the window where the graph is drawn. By default the graph is drawn on the currently-selected window under the current cursor position, and the cursor is repositioned under the graph.

:size, a list of two elements, specifies the outer dimensions of the graph in pixels, which defaults to ***overall-graph-size***.

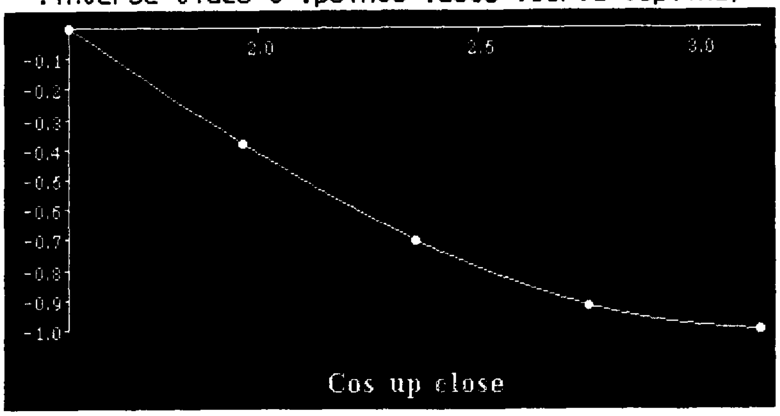
:inverse-video lets the graph be drawn in inverse video mode (the graph background contrasts that of the screen) if t. By default the value of ***inverse-video*** is used.

In addition to default variables listed above, a number of other global variables can be changed to modify the length of tick marks, the widths of various margins between the axes, numbers, labels, and exterior, the number of interpolation points used for drawing splines, and the size of point marks.

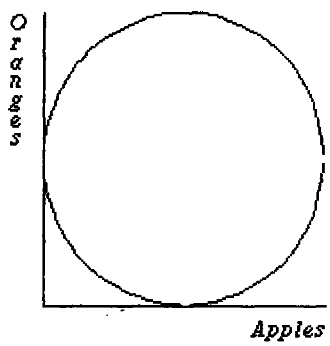
```
(setq x (interval 0 2pi (// pi 8)))
(plot x (mapcar #'cos x))
```



```
(plot x (mapcar #'cos x) :x-range (list pi//2 pi) :title "Cos up close"
  :inverse-video t :points :dots :curve :spline)
```




```
(plot (mapcar #'cos x) (mapcar #'sin x) :curve :spline :size '(180 180)
  :x-origin -1 :y-origin -1 :x-numbers nil :y-numbers nil
  :x-label "Apples" :y-label "Oranges")
```



plot-y *y-values* &rest *keywords-and-args* *Function*

Function **plot-y** works like **plot**, except that *x-values* are not specified; the values $(0, 1, 2, ...)$ are used instead. This function has the same keyword arguments as **plot**.

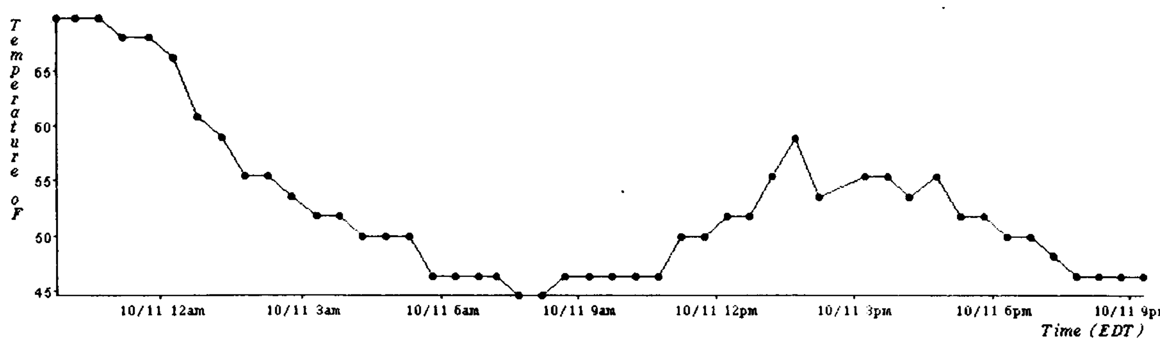
```
(plot-y y ...) --> (plot (iota (length y)) y ...)
```

plot-times *time-values* &key *keywords-and-args* *Function*

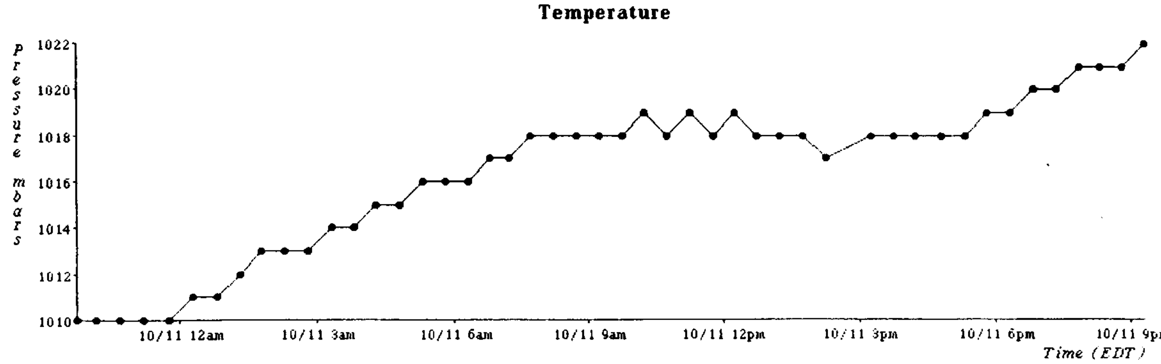
Function **plot-times** plots functions of time. Time values are specified in Zeta-Lisp integer format and time intervals are specified in seconds. This function is like function **plot**, except that keywords **:time-range**, **:time-origin**, **:time-interval**, **:time-numbers**, **:time-number-format**, and **:time-label** are used in place of "**:x-**" keywords. Time intervals default to even multiples of or nice fractions of seconds, minutes, hours, days, and weeks.

For example, the function **plot-weather-data**, defined in file *weather*, uses function **plot-times** to plot weather data from the top of 545 Technology Square over a specified time period:

```
PLOT-WEATHER-DATA: (FROM TO BY)
⇨ (plot-weather-data "1 day ago" "now" "30 minutes")
```



Temperature

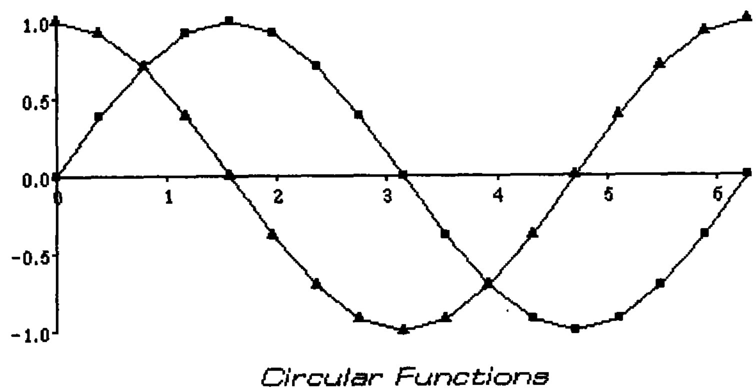


Barometric Pressure

multi-plot *values-spec* &rest *keywords-and-args* *Function*

Function **multi-plot** plots multiple graphs on the same axes. Its first argument is a list, each of whose elements is a list of *x-values*, followed by a list of *y-values*, followed by optional keywords for plotting that particular function. Following this required argument, optional keywords for controlling the overall graph can be specified.

```
(multi-plot '((,x ,(mapcar #'cos x) :points :triangles)
  (,x ,(mapcar #'sin x) :points :squares))
  :title "Circular Functions" :title-font fonts:eurex12i)
```



10 Thoughts on an Image Manipulation Package

Hopefully, many of the functions described in this paper are useful enough for inclusion in a vision utilities package. Many of them can be used both as interactive commands and as "building blocks" for constructing larger vision programs.

The design of a system for displaying and manipulating images, such as Keith Nishihara's *grey** program, is a separate issue from the design of a set of utility functions to be available for vision research. The design of a window-oriented package involves decisions about screen layout, what results are displayed in which windows, etc. The basic functions described in this paper do not require any special windows. I believe it is important to maintain the distinction between a set of basic utilities and a user interface. In particular, the user should be able to use functions for displaying and manipulating images without creating windows, if desired. If properly designed, the window-oriented system can, in fact, call window-free functions like the ones described here.

Acknowledgements

The assistance of Jim Mahoney, Jim Little, and Anita Flynn in proofreading this paper is gratefully acknowledged. Dave Siegel helped fight Latex.

References

Marr, D. and Hildreth, E. "Theory of Edge Detection," *Proc. Royal Society of London, B*, No. 207, pp. 187–217, 1980.

Nishihara, H. K. and Larson, N. G. "Towards a Real-time Implementation of the Marr-Poggio Stereo Mathcer," *Proc. Image Understanding Workshop*, L. Baumann, ed., SAI, College Park, MD, April 1981.

Polivka and Pakin. *APL: The Language and Its Usage*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

Steele, Guy L., Jr. *Common Lisp*. Digital Press, 1984.

Symbolics, Inc. *Reference Guide to Symbolics-Lisp*. Cambridge, MA, 1985.

Voorhees, Harry. "Multi-scale Display Software for the Image Understanding Tool Kit," TASC, EM-2322, 1984.

Index of Definitions