The Climb User Manual

The Common Lisp Image Manipulation Bundle, Version 0.1 beta 1

The Climb Development Team

Table of Contents

1 Introduction

An introduction here.

2 Quick Start

In this chapter, we build a very short program to demonstrate the simplicity and some of the capabilities of Climb. This program will load an image from the hard drive, compute the histogram of its blue component and save this histogram to the disc.

First of all, you need to load Climb. We assume that you have properly installed it using ASDF. Climb is contained in a package called `fr.epita.lrde.climb`. We will `use-package` it for convenience.

```
(in-package :cl-user)
(require :asdf)

(asdf:operate 'asdf:load-op :fr.epita.lrde.climb)
(use-package :fr.epita.lrde.climb)
```

In Climb, images are represented by objects of the class `image`. In order to load one from the hard drive, we will call the function `image-load`, which can open image files in most common encoding formats.

```
(let ((ima (image-load "imagefile.jpg")))
```

Next, we want to build a histogram of the blue component. This means that we retrieve a vector which, using the possible values for the blue component as index, gives the number of pixels which have this value.

Histograms are built using the function `histogram`.

```
  (let ((histo (histogram ima
                          (lambda (x) (aref x 2)))))
```

The value returned by `histogram` is a vector of 256 elements. This is perfect to perform additional calculations on it, but not for visualization. We therefore use `histogram->image` which creates a (very basic) bar histogram image.

```
    (let ((visual-histo (histogram->image histo :height 100)))
```

Finally, we save the resulting image to the hard drive.

```
      (image-save visual-histo "histogram.png"))))
```

Here is the final and complete code, with imbricated `let`s replaced with a single `let*`:

```
(let* ((ima (image-load "imagefile.jpg"))
       (histo (histogram ima (lambda (x) (aref x 2)) 256))
       (visual-histo (histogram->image histo :height 100)))
  (image-save visual-histo "histogram.png"))
```

And *voila*, you have a representation of the histogram you can visualize with your favorite image viewer.

3 Images

3.1 Image Creation

First of all, in order to work on an image, you need to create an image. Creating an image from scratch is done using the function `make-image`.

`make-image` *DIMENSIONS VALUE-TYPE* [Function]

Create an image with uninitialized data. *DIMENSIONS* is the list of the dimensions of the image. *VALUE-TYPE* is a string or a symbol describing the channels of the image. Each character indicates one channel of the image. The accepted components are:

- `I` indicates an intensity component. It is used for grayscale images.
- `RGB` indicate a red, green and blue component, respectively. These three components are meant to be used together to represent a color in the same way as it is displayed by a computer screen.
- `CMYK` indicate a cyan, magenta, yellow and key (or black) component, respectively. These four components are meant to be used together to represent a color in the same way as it is rendered by a printer.
- `A` indicates an alpha (a.k.a transparency) component. It can be added to any of the previous encodings to add transparency to the image.

Therefore, the valid values for *VALUE-TYPE* are the following symbols: `:i`, `:ia`, `:rgb`, `:rgba`, `:cmyk`, `cmyka`.

3.2 File Input And Output

Another way to provide Climb with an image is to load it from your hard drive. You can do so by simply calling `image-load`.

`image-load` *FILENAME* [`:method` *METHOD*] [Function]

Load the image stored in the file *FILENAME*. The library used for input is chosen using *METHOD*. The default (and currently unique) value is `:magick` which tells Climb to use ImageMagick[1]. This means that most popular image formats are supported: PNG, JPEG, GIF, BMP, TIFF, etc[2].

Saving an image to your hard drive is just as simple:

`image-save` *IMAGE FILENAME* [Function]

Save *IMAGE* to *FILENAME*. Like image loading, image saving is currently always done using ImageMagick. The encoding is determined by the filename extension.

[1] Climb uses lisp-magick, a set of Common Lisp bindings for the C ImageMagick API.

[2] For a full list, see ImageMagick's web page.

4 Built-in Algorithms

Climb provides you with a number of common image processing algorithms.

4.1 Mathematical Morphology

4.2 Histograms

A histogram is a powerful tool to study the repartition of a given characteristic of the values of an image. It can be used for any characteristic which has a finite set of values: a color component, the (rounded) luminance value, etc.

The histogram is a vector with one cell for each value of the characteristic. In this cell is the number of sites for which the characteristic has this value. For example, if the characteristic you are studying is the red component, then `(aref histogram 42)` will return the number of pixels whose red component is 42.

4.2.0.1 Creating an histogram

You can create a histogram using the function `histogram`.

`histogram` *IMAGE FUNCTION* **&optional** *SIZE* [Function]

Create an histogram of *IMAGE*. The characteristic is retrieved by passing each value of the image to *FUNCTION*.

`histogram` cannot guess the range of values returned by *FUNCTION*. Therefore, it creates the shortest possible array and extends it whenever it calculates a characteristic greater than the size of the array. If you know that the array will be at least *SIZE* long, you can pass it to `histogram` to avoid some array resizing.

As an example, if `my-image` is an `RGB` image, then the following computes the histogram of its blue component:

```
(histogram my-image
           (lambda (v) (aref v 2))
           256)
```

4.2.1 Visualizing an histogram

Histograms have two uses. If you use the histogram as part of an image processing algorithm, then the vector returned by `histogram` is a convenient representation.

But a histogram can also be created to be viewed by humans. In this case, it is better to have an image representing the "bar histogram". To make it, you can pass your histogram to the function `histogram->image`.

`histogram->image` *HISTOGRAM* [`:height` *HEIGHT*] [`:value-type` *VALUE-TYPE*] [`:fg` *FG-COLOR*] [`:bg` *BG-COLOR*] [Function]

Create a human-friendly image representing *HISTOGRAM*.

If *HEIGHT* is provided, then the values are scaled to fit in this height. This is useful to avoid having a disproportionately tall image if many pixels have the same characteristic.

By default, the output image is grayscale (value type `"I"`), with black bars (value `#(0)`) on a white background (value `#(255)`). You can change this by providing *VALUE-TYPE*, *FG-COLOR* and *BG-COLOR*, respectively.

As an example, the following creates a bar histogram scaled to a height of 100 pixels, with green bars on a dark blue background.

```
(histogram->image histo :height 100 :value-type "RGB"
                        :fg #(0 250 0) :bg #(0 0 40))
```

5 Sites

5.1 Definition

In Climb, we rarely use the term "pixel", because it is ambiguous. Does it refer to a location on an image? The color at this location? Both? Instead, we separate the notions of "site" and "value".

A site denotes a fundamental location on an image. On a classical, regular grid image[1], a site is a point: it can be assimilated to its cartesian coordinates.

To create a point, you can use the variadic function `make-point`:

`make-point` *COORDINATE1 COORDINATE2...* [Function]

Create a point with given *COORDINATE*s.

Note that, since a point site refers to a position on a grid, only integer coordinates make sense.

In order to retrieve the coordinates of an existing point, you can use either `point-coordinates` or `point-coordinate`. See their usage:

`point-coordinates` *POINT* [Function]

Retrieve the vector of coordinates of *POINT*.

`point-coordinate` *POINT COORD* [Function]

Retrieve the *COORD*-th coordinate of *POINT*. This is equivalent to:

```
(aref (point-coordinates POINT) COORD)
```

`point-coordinate` can also be used in conjunction with `setf` to modify a point. If you want to create a new point, you can use `point-copy`.

Climb also provides arithmetic operations on points:

```
(site+ (make-point 1 2) (make-point 6 4)) ;; Equivalent to (make-point 7 6)
(site- (make-point 4) (make-point 7))     ;; Equivalent to (make-point -3)
```

5.2 Image Dereferencing

The whole purpose of sites is to characterize the location of data within an image. To retrieve this data, use the function `iref`.

`iref` *IMAGE SITE* [Function]

Retrieve the data associated with SITE in IMAGE.

Actually, this looks very much like array indexing. The container, instead of an array, is an image. And the location inside this container, represented by the index in the case of arrays, is here the site. Hence the function name, `iref`.

`iref` also allows you to modify the data inside an image:

```
(setf (iref my-image (make-point 12 34)) #(255 0 127))
```

[1] Regular grid images are actually the only type currently supported by Climb.

5.3 Site Sets

Algorithms often process a set of sites together. In Climb, such a set of sites is called — appropriately enough — a site-set.

Probably the most important kind of site-set is the domain of an image. It consists of all the points on which the image is defined. You can retrieve the domain of an image using the function `image-domain`.

`image-domain` *IMAGE* [Function]

Retrieve the domain of IMAGE.

To create a site-set "from scratch", the most simple way is to use `make-site-set-box`. A box is a rectangular subset of the grid.

`make-site-set-box` *MIN MAX* [Function]

Return the set of all sites whose coordinates range between those of *MIN* and *MAX*.

See the following example:

```
(defvar min (make-point 10 15))
(defvar max (make-point 45 30))
(defvar my-box (make-site-set-box min max))
```

Then `my-box` is the rectangle consisting of all points whose abscissa is between 10 and 45 and whose ordinate is between 15 and 30 (inclusively).

In order to create site-sets with more complex shapes, you need to apply a morpher on a box. See Section 7.3 [Site Set Morphers], page 12.

5.4 Image Browsing

Generally, you will want to act on all sites of a given image or site-set. For such purpose, you can use the macro `do-sites`. This macro looks like the standard macro `dolist`: you provide the site-set, the name you want to give to the currently browsed site, and a body to evaluate for each site.

`do-sites` (*SITE SET*) *BODY* [Macro]

Evaluate BODY once for each SITE in SET. SET can be either a site-set, or an image. In the latter case, the domain of the image is browsed.

The next example copies a rectangular part of `first-image` into `second-image`:

```
(do-sites (site (make-box (make-point 3 5)
                          (make-point 15 12)))
  (setf (iref second-image site) (iref first-image site)))
```

6 Accumulators

Now, you have the basics to implement a number of image processing algorithms: you can browse all or part of an image, retrieving and modifying its data. We will now take a look at a utility provided by Climb, called the accumulator. Accumulators can be considered a generalization of the standard function `reduce`.

6.1 Definition

Reduction is a simple but powerful operation that can be performed on a data collection (typically, in Lisp, a list or an array). The principle is simple: you provide a function F and an initial accumulation value. For each element in the sequence, a new accumulation value is calculated as the result of calling F with the previous accumulation value and the current element as arguments. Finally, when the sequence is exhausted, the last accumulation value is returned.

In Common Lisp, such an operation can be performed on lists and arrays using the function `reduce`. But if we want to reduce another collection (such as, for example, a site-set or an image), we have to write a new version of `reduce` for this type.

Accumulators are objects that allow you to perform reduction on any sequential data. They encapsulate the accumulated data and the accumulation operation.

The most important operations applicable to accumulators are `accu-feed` and `accu-value`.

`accu-feed` *ACCU DATUM* [Function]
Input DATUM into ACCU.

`accu-value` *ACCU* [Function]
Retrieve the value calculated by ACCU.

As an example, the following program calculates the sum of the values in a given image:

```
(let ((accu (make-accu-sum))              ; Create the accumulator.
  (do-sites (s the-image)                 ; For each site s in the image,
    (accu-feed accu (iref the-image s))   ; feed the datum at this position to
                                          ; the accumulator.
  (accu-value accu))))                    ; Finally, return the calculated value.
```

The interesting point in the above code is its genericity. Imagine that instead of summing the values, you want to multiply them, average them, or find their maximum. All you have to do is change the first line: once the accumulator is created, the manipulations are completely independent of what it actually computes.

For example, you can write the previous code as a version of `reduce` for images which, instead of taking a function and an initial value, only needs an accumulator.

```
(defun image-reduce (accu the-image)
  (do-sites (s the-image)
    (accu-feed accu (iref the-image s))
  (accu-value accu)))

;; An example utilization
(image-reduce (make-accu-sum) the-image)
```

Other useful operations that you can use on an accumulator are `accu-value-p` and `accu-reset`.

`accu-value-p` *ACCU* [Function]

Return a boolean indicating whether a result is available for ACCU. This is one of the features that makes accumulators more powerful than reduction: an accumulator can encapsulate a calculation that needs a minimum number of inputs before returning a value, or any other kind of criterion.

`accu-reset` *ACCU* [Function]

Set ACCU back to its initial state.

6.2 Available Accumulators

Some commonly useful accumulators are provided by Climb.

`make-accu-sum` [Function]

Create an accumulator which operates an arithmetic sum on its inputs.

`make-accu-and` [Function]

Create an accumulator which operates a boolean `and` on its inputs.

`make-accu-or` [Function]

Create an accumulator which operates a boolean `or` on its inputs.

6.3 Create Your Accumulator

There are two ways to create an accumulator. The first one is the most simple, and is directly mapped on the standard `reduce` construct. The second one is more powerful and goes into the internals of how accumulators work.

6.3.1 `reduce`-like Accumulators

To create an accumulator with the same capabilities as `reduce`, you can use the function `make-accu-reduce`.

`make-accu-reduce` *FUNCTION INIT-STATE* [Function]

Create an accumulator which performs a functional reduction on its inputs. FUNCTION is the reduction function:

It takes the reduction function as first argument, and the initial accumulation value as second argument.

As an example, the function `make-accu-sum` can be implemented as follows:

```
(defun make-accu-sum ()
  (make-accu-reduce #'+ 0))
```

6.3.2 General Accumulators

Throughout this chapter, we described accumulation as a reduction independent from the type of sequence. Actually, it can also handle cases for which reduction is insufficient.

One of the limitations of reduction, and thus of accumulators constructed with `make-accu-reduce`, is that the result value is always the last accumulation value.

As an example, let's make an accumulator which averages its inputs (without weight). At any time, it has to keep track of the sum of the previous inputs, and the number of inputs already taken. So the accumulation value is a pair of numbers. However, the desired return value is not this pair of numbers, but their quotient. So we will not be able to use `make-accu-reduce` directly. Of course, a simple wrapper around `make-accu-reduce` can be made (see Section 7.4

[Accumulator Morphers], page 12). But for didactic purposes, we will create the averaging accumulator from scratch.

An accumulator is an object whose class inherits from `accu` and has an implementation for the accumulator operations: `accu-feed`, `accu-value`, `accu-value-p` and `accu-reset`.

First, we create the accumulator class. It has a slot for the sum of the previous inputs, and one for the number of inputs alreday taken.

```
(defclass accu-average (accu)
  ((sum :initform 0)
   (count :initform 0)))
```

Then, we implement `accu-feed`. When inputting data, we add it to the sum and increment the counter.

```
(defmethod accu-feed ((accu accu-average) value)
  (incf (slot-value accu 'sum) value)
  (incf (slot-value accu count)))
```

The method for `accu-feed` calculates the average.

```
(defmethod accu-value ((accu accu-average))
  (/ (slot-value accu 'sum)
     (slot-value accu 'count)))
```

`accu-value-p` is another reason why this accumulator cannot be made using `make-accu-reduce`: if no data has been input, the average is not computable.

```
(defmethod accu-value-p ((accu accu-average))
  (not (zerop (slot-value accu 'count))))
```

Most of the time, resetting an accumulator means setting its slots back to their initial state.

```
(defmethod accu-reset ((accu accu-average))
  (setf (slot-value accu 'sum) 0
        (slot-value accu 'count) 0))
```

Finally, for convenience, we add a constructor.

```
(defun make-accu-average ()
  (make-instance 'accu-average))
```

7 Morphers

7.1 Definition and Examples

Morphers are a powerful utility provided by Climb. A morpher is a wrapper around an object which transforms the data input to it and output from it. We distinguish two types of morphers: content morphers and value morphers.

- *Content morphers* change which values are passed to and from the object. It doesn't modify the values themselves, but chooses which are passed and which are ignored.

  For example, a morpher on an array which restricts the size of the array is a content morpher.
- *Value morphers*, on the contrary, modify the values passed to and from the object.

  For example, a morpher on an array (let's call it `array-double`) which, whenever one sets the value of an element, writes the double of the supplied value instead.

It is very important that a morpher behaves exactly like the underlying type: for all intents and purposes, an array morpher *is* an array.

For example, the following holds true for all array `a`, valid index `i` and valid value `x`:

```
(setf (aref a i) x)
(eql x (aref a i))   ; This should always be true!
```

Therefore, in order for `array-double` to be considered a valid morpher, it must also apply the reverse operation when reading data. That is, it must return half the value it reads from the underlying array.

As you can see, the notion of morpher is an abstract concept which can be applied on many objects. Let's have a look at the facilities Climb offers to create morphers on the types it defines.

7.2 Image Morphers

The first and most obvious type of objects you can morph is images. Let's see what "content morpher" and "value morpher" mean in the case of an image:

- An image content morpher adds or removes sites to the image's domain. There is currently no utility to make one.
- An image value morpher changes the image values read and written using `iref`.

  You can build such a morpher using the function `make-image-morpher-value`. It takes as arguments the underlying image and the functions to use for input and output. For example, an image morpher `i2` which triples data into its underlying image `i1` can be written as follows:

  ```
  (defvar i2 (make-image-morpher-value i1
               (lambda (value ima site)
                 (* value 3))
               (lambda (ima site)
                 (/ (iref ima site) 3))))
  ```

Note that in a morpher, the input and output do not need to have the same type. For example, the following takes the multi-component image `original-image` and creates single-component one:

```
(make-image-morpher-value original-image
                          (lambda (value ima site)
                            (vector a))
                          (lambda (ima site)
                            (aref (iref ima site) 0)))
```

7.3 Site Set Morphers

A site-set is actually just a (possibly lazy) collection of sites. So you can also create morphers on site-sets.

- A site-set content morpher can take a subset of the underlying site-set.

  The function `make-site-set-morpher-content` creates such a morpher using a membership predicate. For example, the following creates a set composed of every site from `my-site-set` whose abscissa is even.

  ```
  (make-site-set-morpher-content my-site-set
                                 (lambda (s)
                                    (evenp (point-coordinate s 0))))
  ```

- A site-set value morpher modifies the sites of the set.

  To create one, you can use `make-site-set-morpher-value`. The following code creates a site-set whose sites have their abscissa shifted by 2:

  ```
  (make-site-set-morpher-value my-site-set
                               (lambda (s)
                                 (let ((s2 (point-copy s)))
                                   (incf (point-coordinate s2 0) 2)
                                   s2)))
  ```

7.4 Accumulator Morphers

Accumulators are also "morphable".

- Accumulator value morphers play a double role: they have both an input function, which changes the values taken by the accumulator, and an output function which modifies the result of the accumulation. They are created using `make-accu-morpher`.

  As an example, let's rewrite the average accumulator thanks to a value morpher. The underlying morpher stores the necessary data as a cons cell: the sum of input data in the `car`, and their number in the `cdr`. The morpher takes care of the extra computations needed.

  ```
  (defvar internal-accu (make-accu-reduce
                         (lambda (state input)
                           (cons
                             (+ (car state) input) ;; The sum of inputs
                             (+ (cdr state) 1)))   ;; The number of inputs
                         (cons 0 0)))
  (defvar my-average-accu (make-accu-morpher internal-accu
                           #'identity  ;; We don't need to change the inputs
                           (lambda (result)
                             (/ (car result) (cdr result)))))
  ```

8 Conclusion

Appendix A Indexes

A.1 Concepts

A.2 Functions

A

C

D

H

I

M

P

S

A.3 Variables

(Index is nonexistent)

A.4 Data Types

A

C

I

P

S