2D Minimum and Maximum Filters: Algorithms and Implementation Issues

Herman Tulleken
herman.tulleken@gmail.com
http://www.code-spot.co.za

January 23, 2011

Contents

1 The Problem **3**

2 Exact Algorithms **3**

2.1 The Naive Algorithm 3

2.2 The Max-Queue 5

2.3 Implicit Queue Algorithm 7

2.4 The Monotonic Wedge Algorithm 10

3 Approximate Algorithms **12**

3.1 The Power Mean Approximation 13

3.2 The Power Mean Variant Algorithm 13

3.3 The Contra-Harmonic Mean Approximation 14

4 Other Algorithm Concepts **14**

4.1 Separation 14

4.2 Implementing Minimum Filters 19

4.3 Windows with Even Diameters 20

4.4 Filtering a Region of Interest 22

4.5 Maximum and Minimum Filters for Binary Images 26

A Image Containers **28**

A.1 Image Class Interface 28

A.2 Image Loops 29

A.3 Image Iterators 30
A.4 Image Access Modifiers 34

B Fixed-width Deques **38**

C Max-queues **43**

D Summed Area Tables **47**
D.1 Calculating a SAT 47
D.2 Finding a Sum from a SAT 49
D.3 Checking for Overflow 50
D.4 Large SATs 53

This document is about implementing fast, robust and maintainable minimum and maximum filters.

First, we define the problem as it is approach here. Second, a description of a variety of algorithms is given, with tips on implementing them. Third, we look at ways in which algorithms can be modified to accomplish certain goals. Finally, a comparison is made between the different implementations, looking at time and space efficiency, accuracy, and ease of implementation. Appendices at the end give some information on the various data structures used to implement the algorithms.

1 The Problem

I assume you already know what a maximum or minimum filter is. To make sure we are talking about the same thing, I will briefly give the definition used here.

The notation $I(x, y)$ is used to denote the pixel value of an image at integer coordinates (x, y). If the width and height of the image is given by m and n, then $x \in \{0, 1, \ldots m-1\}$ and $y \in \{0, 1, \ldots n-1\}$.

If I is an image, and we filter it with a maximum filter of radius $r \in \mathbb{Z}$, we get the new image $(\max_r I)$ given by

$$(\max_r I)(x, y) = \max_{\substack{x-r \le u \le x+r \\ y-r \le v \le y+r}} I(u, v) \tag{1}$$

Notice that u and v may fall outside the image borders. Usually, when $I(u, v)$ falls outside the image border, its value is taken as a specific color (such as black), or as the closest pixel inside the image. We will not do this here. Instead, we use the slightly altered definition:

$$(\max_r I)(x, y) = \max_{\substack{\max(0,x-r) \le u \le \min(m-1,x+r) \\ \max(0,y-r) \le v \le \min(n-1,y+r)}} I(u, v) \tag{2}$$

All that is different is that u and v are limited so that they will always fall within the image.

The pixel value at (x, y) in new image, thus, is the maximum value of a square window (limited to fall inside the image) around the pixel at (x, y) in the original image. Notice that, except when the window is limited when it falls close enough to the border, the diameter of the window is $2r + 1$; it is always odd. In Section 4.3 we will look at implementing maximum filters for windows with even diameter.

Maximum and minimum filters are as defined above are special cases of erosion and dilation filters.

2 Exact Algorithms

2.1 The Naive Algorithm

The naive algorithm is very straightforward.

```
for each pixel (x,y) in the image
  max_pixel = image(x, y)
  for each pixel (u, v) in the neighbourhood of (x, y)
    if image(u, v) > max_pixel
      max_pixel = image(u, v)
  result(x, y) = max_pixel
```

The algorithm is conceptually clear, but in actual code there are many places to make mistakes, some subtle enough to go unnoticed. Here is the algorithm in more detail:

```
for (int x = 0; x < image.width(); ++x)
{
  int x0 = max(0, x - radius);
  int x1 = min(image.width() - 1, x + radius);

  for (int y = 0; y < image.height(); ++y)
  {
    int y0 = max(0, y - radius);
    int y1 = min(image.height() - 1, y + radius);

    max_pixel = image(x, y);

    for (int u = x0; u <= x1; ++u) //note <=, not <
    {
      for (int v = y0; y <= y1; ++v)
      {
        if (image(u, v) > max_pixel)
        {
          max_pixel = image(u, v);
        }
      }
    }
    result(x, y) = max_pixel;
  }
}
```

Off-by-one errors It is incredibly easy to make off-by-one errors. An easy way to verify the size of the window is to use two test images containing sequential data—one going up and the other going down. This is the image with a sequence going up:

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & \dots & w-1 \\ w & w+1 & w+2 & w+3 & \dots & 2w-1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ (h-1)w & (h-1)w+1 & (h-1)w+2 & (h-1)w+3 & \dots & hw-1 \end{array} \tag{3}$$

On this test data, the maximum is trivially dependent on the window size. For example, the algorithm must give the value $2w + 2$ for the pixel (1, 1) if the window is 3×3 for the upward image, and $(wh - 1) - (2w + 2)$ for the downward image. If we do not get these value, the most likely cause is a miscalculation of the window corner coordinates.

Using unsigned integers In the code above, `int` was used for all the indices and sizes. I prefer to use `unsigned int` instead—not to increase the range, but rather to provide extra information for the reader. When using unsigned values, making certain mistakes is much easier (and for this reason some recommend not using them in the first place). In particular, the window coordinates must be calculated differently:

```
//do not subtract before we are sure it is safe
unsigned int x0 = x > radius ? (x - radius) : 0;

//always safe (at least for images with positive dimensions)
unsigned int x1 = min(image.width() - 1, x + radius);
```

This code still assumes that the image is not an image with zero width or height.

Initialising max-pixel Note that we initialise the max-pixel value with the centre pixel. We could use any pixel in the window range, but using the centre pixel avoid any extra (error prone) calculations.

Another approach is to assign it with an appropriate interpretation of $-\infty$. For example, if we know that all our pixels are non-negative, we might use zero. There are several reasons why this is not a good idea:

- You might later find that it is useful to extend the data range of images. If you forget that you made the assumption about the minimum value of image data, your implementation will be incorrect.
- The algorithm is less generic, that is, you need to do extra work to use the algorithm on other types, especially custom types.

2.2 The Max-Queue

A Max-Queue is a queue that have the following additional properties:

- It has a fixed capacity. When an item is pushed into the queue so that the capacity is exceeded, an item automatically popped from the front.
- It maintains the maximum value of all values in the queue, that is, its supports a `max()` operation that performs in $O(1)$ time.

A max-queue can be used to implement the filter maximum algorithm. The algorithm uses the separability of the problem. There is a whole section about separability under the section *Other Algorithm Concepts*, but for now it is enough to know that we can perform the filtering in two steps:

1. The first step applies a 1D filter to each row.
2. The second step applies the 1D filter to each column of the result of the last step.

We can use the same algorithm for both steps. There are different ways to accomplish this (and these are discussed under the Separation section). The simplest way is to transpose the image before and after the second step:

```
Image filter_max_2d(Image image, int radius)
{
  //pass on rows
  filter_max_queue_rows(image, radius);

  //move rows to columns
  image.transpose();

  //pass on columns (now in rows)
  filter_max_queue_rows(image, radius);

  //move columns back to rows
  image.transpose();
}
```

The basic operation of the `filter_max_queue` function is simple:

1. We start pushing pixels from the row into the queue.
2. At some stage, we have enough information in the queue to start writing output into the results folder. This happens when we have pushed radius + 1 pixels. We keep on pushing pixels as we write the queue maximum to the result row.
3. At some stage, we have pushed all the pixels in the row. We still have enough values in the queue to write output until the result row is full. We pop out the remaining queue values as we write output. When the result vector is full, there are radius values left in the queue.

As always with "simple" algorithms, the devil is in the details. Off-by-one errors is extremely easy to make, and somewhat tricky to spot.

Here is the algorithm in proper loops:

```
unsigned int window_width = 2 * radius + 1;
MaxQueue<T> window(window_width);

for (unsigned int y = 0; y < image.height(), ++y)
{
  unsigned int x_read = 0;//read from image
  unsigned int x_write = 0;//write to result

  // Step 1 - start pushing data into the queue
  for (; x_read < radius; ++x_read)
  {
    window.push(image(x_read, y));
  }

  // Step 2 - keep on pushing data, start writing output
  for (; x_read < image.width(); ++x_read, ++x_write)
  {
    window.push(image(x_read, y));
    result(x_write, y) = window.max();
  }
```

```
  // Step 3 - Keep on removing values from the queue
  // and keep on writing output.
  // We cannot push more data - we are out of pixels.
  for (; x_write < image.width(); ++x_write)
  {
    window.pop();
    result(x_write, y) = window.max();
  }

  // A check to make sure we have not accidentally popped
  // one pixel too many or few.
  assert(queue.size() == radius + 1);

  // If we do not clear values, the values that remain
  // in the queue will be used in the first few
  // calculations of the next row.
  window.clear();
}

return result;
```

Because of the simplistic way in which we use the queue, you might think it is not necessary to use a queue. Indeed, it is not, and in the next section we see how the same algorithm can be implemented without a queue.

However, the queue approach has distinct advantages that makes using this algorithm useful in certain circumstances:

1. It makes it possible to change the algorithm to an in-place algorithm. The implementation above needs extra memory the size of the image (in addition to the negligible amount of memory needed by the queue and index variables). Of course, this is only useful if the transpose function is also in-place, or we use a access modifier (see the section Image Access Modifiers in the Appendix).
2. It is conceptually easier to implement.
3. Conceptually, it foreshadows the more complicated monotonic wedge algorithm discussed later. Understanding the algorithm makes implementing that algorithm easier.

As we will show, the algorithm performs well with random data, but not so well when the data is piece-wise pseudo-monotonic (i.e., if the pixels in a row (roughly) grows from left to right over for sections of the image. Unfortunately, for real-world images, the latter is often the case.

2.3 Implicit Queue Algorithm

This algorithm is conceptually identical to the one described in the previous section, except that there is no explicit queue. We use indices to keep track of the front and back of the queue, and a value to keep track of the maximum. There is some code added to search for a new maximum whenever the current maximum moves out of the window. This has been hidden in the implementation of the max-queue. (This function looks so long because of all the comments!)

```
unsigned int window_width = 2 * radius + 1;
unsigned int width = image.width();

for_Y (image, y)
{
  unsigned int x_read = 0;//read from image,
                          //end of implicit queue

  unsigned int x_write = 0;//write to result
  unsigned int x_start = 0;//start if implicit queue
  unsigned int x_max = 0; //max in implicit queue

  // Step 1
  for (; x_read < radius; ++x_read)
  {
    if (image(x_read, y) > image(x_max, y))
    {
      x_max = x_read;
    }
  }

  // Step 2
  for (; x_read < width; ++x_read, ++x_write)
  {
    if (image(x_read, y) >= image(x_max, y))
    {
      x_max = x_read;
    }

    //x and y must be swapped
    result(y, x_write) = image(x_max, y);

    if (x_read + 1 >= window_width)
    //only pop if there are enough items in the queue!
    {
      if (x_start == x_max)
      { // The max is moving out of the window next round
        // so find the new max from the pixel after the
        // front of the queue. It would be correct to
        // analyse up to x_read + 1, but it is not necessary,
        // because on the next pass, if the next value to be
        // read (now x_read + 1) is bigger than the max, it
        // will replace it.

          x_max = find_argmax_in_row_in_range(
            image,
            x_start + 1,
            x_read,
            y);
      }

      ++x_start;
```

```
    }
  }

  for (;x_write < image.width(); ++x_write)
  {
    //x and y must be swapped
    result(y, x_write) = image(x_max, y);

    if (x_start == x_max)
    { // The max is moving out of the window next round
      // so find the new max from the pixel after the
      // front of the queue. We do not look past the last
      // pixel in the row!

      x_max = find_argmax_in_row_in_range(
        image,
        x_start + 1,
        image.width() - 1,
        y);
    }

    ++x_start;
  }
}

return result;
```

Below is the implementation of the `find_argmax` function.

```
find_argmax_in_row_in_range(
  image,
  range_start,
  range_stop, //includes last pixel to be analysed
  row)
{
  // Initialise max to the first pixel in the range
  x_max = range_start;

  // Check whether any of the remaining values
  // in range are bigger
  for (unsigned int x = range_start + 1; x <= range_stop; ++x)
  {
    if (image(x, y) >= image(x_max, y))
    {
      x_max = x;
    }
  }

  return x_max;
}
```

A few notes about this implementation:

- In the `filter_max_implicit_queue` function, Step 2, note the line: `if (image(x_read, y) >= image(x_max, y))`. Here we use `>=` instead of `>` as a small step towards a faster filter. By using the rightmost value as the maximum, we reduce the chances of the maximum being moved out of the window, causing us to perform the slow linear search for the new maximum. The same goes for the line `if (image(x, y) >= image(x_max, y))` in the function `find_argmax_in_row_in_range`.
- The indices `x_start`, `x_read`, and `x_write` can be calculated from a single index. However, it is quite tricky to get this right, and makes the program harder to read.

As we will show later, the implicit queue algorithm is *considerably* faster than the explicit queue algorithm. It is somewhat more complicated, but still manageable.

2.4 The Monotonic Wedge Algorithm

The max-queue algorithm is slow when it has to perform many searches. It is possible to have images that performs a search for every pixel popped, in which case the algorithm does no better than the separated naive algorithm. To combat this worse-case scenario, we maintain a more sophisticated presentation of the window: a monotonic wedge that ensures we can find new maximums quickly when they move out of the window. The monotonic wedge is discussed in more detail in the appendix. For now, the following facts will suffice for understanding the algorithm:

Let U_i be a monotonic wedge of the sequence A_i. Then U_i has the following properties:

- U_0 is the index of the maximum element of A_i,
- U_1 is the largest element of all elements to the right of A_{U_0}.
- In general, U_k is the largest element of all elements to the right of $A_{U_{k-1}}$.

As with the queue-based algorithm, we push and pop elements, while maintaining the maximum element (as the first element in the monotonic wedge). When we pop out the maximum, we need to find the new maximum—but instead of having to search for it through the window, we merely take the next element from the wedge (since we also popped the maximum from the wedge, the new maximum will be the new front of the wedge).

The algorithm maintains a wedge for a sliding window—thus the sequene A_i changes in each iteration. The trick is to maintain the wedge efficiently. There are three cases:

- We process an element smaller than the wedge back. In this case, we merely append the element's index to the wedge back.
- We process an element that is larger than the back. In this case, we need to pop out all elements from the back that are smaller than this element, and then push this element onto the wedge. When the element is larger than the wedge front, we will pop all elements before pushing the new index.

Here is an example. Let our source sequence be $1, 3, 5, 4, 2, 6, 3$, and suppose we use a window with width 3. Then the wedge looks like this as we process each element:

i	A	U
0	1	1
1	1 3	3
2	1 3 5	5
3	2 5 4	5 4
4	5 4 2	5 4 2
5	4 2 6	6
6	2 6 3	6 3
7	6 3	6 3

```
deque<unsigned int> wedge;

for (unsigned int y = 0; y < image.height(); ++y)
{
  unsigned int x_read = 0;//read from image
  unsigned int x_write = 0;//write to result

  // Step 1 - start pushing data into the wedge
  for (; x_read < radius; ++x_read)
  {
    while(
      (!wedge.empty()) &&
      (image(wedge.back(), y) <= image(x_read, y)))
    {
      wedge.pop_back();
    }

    wedge.push_back(x_read);
  }

  // Step 2 - keep on pushing data, start writing output
  for (; x_read < image.width(); ++x_read, ++x_write)
  {
    while(
      (!wedge.empty()) &&
      (image(wedge.back(), y) <= image(x_read, y)))
    {
      wedge.pop_back();
    }

    wedge.push_back(x_read);
    result(x_write, y) = image(wedge.front(), y);

    if (x_read + 1 >= window_width)
    //only pop if there are enough items in the queue!
    {
      if (wedge.front() == x_read)
      {
        wedge.pop_front();
```

```
      }

      ++x_start;
    }
  }

  // Step 3 - Keep on removing values from the queue
  // and keep on writing output.
  // We cannot push more data - we are out of pixels.
  for (; x_write < image.width(); ++x_write)
  {
    result(x_write, y) = image(wedge.front(), y);

    if (wedge.front() == x_start)
    {
      wedge.pop_front();
    }

    ++x_start;
  }

  // If we do not clear values, the values that remain
  // in the queue will be used in the first few
  // calculations of the next row.
  wedge.clear();
}

return result;
```

The following table shows how internal variables change in the example above:

	i	`x_read`	`x_write`	A	U
Step 1	0	0	-	1	1
Step 2	1	1	0	1 3	3
	2	2	1	1 3 5	5
	3	3	2	2 5 4	5 4
	4	4	3	5 4 2	5 4 2
	5	5	4	4 2 6	6
	6	6	5	2 6 3	6 3
Step 3	7	-	6	6 3	6 3

3 Approximate Algorithms

In some cases, an approximate maximum filter might be acceptable, or even desirable. Approximate filters can be implemented very efficiently, and in some-cases, they provide smoothness that are better suited for whatever the filter is used for. The maximum function satisfies these two properties:

- $\max(x, y) \geq x$ and $\max(x, y) \geq y$,
- $\max(x, x) = x$.

Ideally, we would like our approximation to also satisfy these two properties. Unfortunately, for continuous approximations, this is not possible. It should be clear that the properties can be trivially satisfied when we allow discontinuities, for example:

$$\max(x, y) \approx \begin{cases} x & x = y \\ x + y & x \neq y \end{cases} \tag{4}$$

There are several approximations that satisfy one of these properties. Here we will consider three that are viable for efficient implementation

$$\max(x_i) \approx \left(\frac{1}{N} \sum x_i^p \right)^{1/p} \qquad \text{(Power Mean)} \tag{5}$$

$$\max(x_i) \approx \left(\sum x_i^p \right)^{1/p} \qquad \text{(Power Mean Variant)} \tag{6}$$

$$\max(x_i) \approx \frac{\sum x_i^{p+1}}{\sum x_i^p} \qquad \text{(Contra-harmonic Mean)} \tag{7}$$

The implementations of all three these algorithms are very similar.

3.1 The Power Mean Approximation

```
power_sum_table = make_power_sum_table(image, p)
for_XY (image, x, y)
{
  x0 = x - radius;
  x1 = x + radius;
  y0 = x - radius;
  y1 = x + radius;

  power_sum = get_sum(power_sum_table, x0, y0, x1, y1);
  window_size = (x1 - x0)*(y1 - y0);

  result(x, y) = pow(power_sum/window_size, 1.0/p);
}
```

3.2 The Power Mean Variant Algorithm

```
power_sum_table = make_power_sum_table(image, p)
for_XY (image, x, y)
{
  x0 = x - radius;
  x1 = x + radius;
  y0 = x - radius;
  y1 = x + radius;

  power_sum = get_sum(power_sum_table, x0, y0, x1, y1);

  result(x, y) = pow(power_sum, 1.0/p);
}
```

3.3 The Contra-Harmonic Mean Approximation

```
denominator_power_sum_table = make_power_sum_table(image, p)
numerator_power_sum_table = make_power_sum_table(image, p + 1)

for_XY (image, x, y)
{
  x0 = x - radius;
  x1 = x + radius;
  y0 = x - radius;
  y1 = x + radius;

  numerator = get_sum(power_sum_table, x0, y0, x1, y1);
  denominator = get_sum(power_sum_table, x0, y0, x1, y1);

  result(x, y) =
    (denominator > 0) ? nominator / denominator : 0;

  result = pow(power_sum, 1.0/p);
}
```

Of course, we might also calculate the numerator from the denominator with a point-wise product:

```
numerator_power_sum_table =
  denominator_power_sum_table * image;
```

4 Other Algorithm Concepts

4.1 Separation

In Section 2.2 we used the fact that we can implement a 2D maximum filter by applying a 1D maximum filter to all the rows, and then again to all the columns of the result. In this section we look in more detail to this procedure.

The separability comes from the following property of the maximum operator. If S and T are two sets of numbers, then

$$\max(\max(S), \max(T)) = \max(S \cup T)$$

The same applies to the minimum operator:

$$\min(\min(S), \min(T)) = \min(S \cup T)$$

(There are operators that do not follow this law. The median operator is an example. For instance, if $S = \{1, 1, 1\}$ and $T = \{1, 2, 3\}$, then $\text{med}(\text{med}(S), \text{med}(T)) = \text{med}(1, 2) = 1.5$, but $\text{med}(S \cup T) = 1$.)

This property of the maximum operator allow us to rewrite the definition as follows:

$$(\max_r I)(x, y) = \max_{x-r \le u \le x+r} \left[\max_{y-r \le v \le y+r} [I(u, v)] \right] \tag{8}$$

From this it should be clear that we can first find the maximum of the rows, and then the columns (or vice versa).

When separation was first introduced, we structured it as follows:

Here is a separated version of the naive algorithm:

```
filter_max_1d()
{

  for (int x = 0; x < image.width(); ++x)
  {
    int x0 = max(0, x - radius);
    int x1 = min(image.width() - 1, x + radius);

    max_pixel = image(x, y);

    for (int u = x0; u <= x1; ++u) //note <=, not <
    {

      if (image(u, v) > max_pixel)
      {
        max_pixel = image(u, v);
      }
    }

    result(x, y) = max_pixel;
  }
}
```

There are many benefits to use a separated algorithm.

First, it is simpler to implement. In the 1D implementation, there are fewer loops and indices to keep track of, so there is less chance for error.

Second, it is much faster. If we can perform the operation $\max S$ of k elements in $O(f(k))$ time, then the first original form of the definition will execute in $O(mnf(d^2))$, but the second one only in $O(mn \cdot 2f(d))$ where d is the window diameter. If $f(k) = k$, for example, as it is in the case of the naive algorithm, then the original algorithm is $O(mnd^2)$, but the separated one is only $O(2mnd)$. Even for a small diameter $d = 3$, the unseparated version is 1.5 times as slow as the separated one.

Third, for some algorithms less temporary memory is needed. For example, the approximation algorithm discussed here can be structured so that only one row of extra memory is necessary for the SAT, instead of an entire image of extra memory.

Fourth, 1D algorithms are more readily available. Any 1D max filtering algorithm can automatically be turned into a 2D separated max filtering algorithm. It is not even always clear how to define a 2D algorithm without using separation. The Max Queue algorithm is an example of this.

Using transposition This method has been introduced in Section 2.2; recall the implementation:

```
Image filter_max_2d(Image image, int radius)
{
  //pass on rows
  filter_max_1d_rows(image, radius);

  //move rows to columns
  image.transpose();

  //pass on columns (now in rows)
  filter_max_1d_rows(image, radius);

  //move columns back to rows
  image.transpose();
}
```

The transpose function simply replaces rows with columns. Here is a simple implementation:

```
void Image::transpose()
{
  Image result(height, width); //swap dimensions

  for_XY (*this, x, y)
  {
    result(y, x) = (*this)(x, y);
  }

  //Swap this image width and height
  mWidth = result.width();
  mHeight = result.width();

  //Copy result to this image

  for_XY ((*this), x, y)
  {
    (*this)(x, y) = result(x, y);
  }
}
```

In-place transposition The simple implementation above requires extra memory the same size as the image we are transposing. This will defeat any attempt to save memory with an in-place implementation of the max-filter. An in-place transposition function will solve this problem.

The solution below assumes we have all the image data in a 1D array, and access it through calculations. For example, `image(x, y)` is equivalent to `image.mData[x + y * mWidth]`.

For each pixel:

1. We calculate the position `k0` in the array of the pixel

2. and the position `k1` in the array of the pixel that it has to swap with.
3. If `k0 < k1`, we swap the two array values. The check is necessary, otherwise we will perform two identical swaps, leaving the array exactly as it was. Note that we need not swap the values when `k0 == k1`, since then it is exactly the same pixel!

Finally we swap the width and height values of the image.

Here is the algorithm in code:

```
void Image::transpose() //in-place
{
  for_XY (*this, x, y)
  {
    k0 = x + mWidth * y;
    k1 = y + mHeight * x;

    if (k0 < k1)//make sure we only swap once
    {
      //swap
      tmp = mData[k0];
      mData[k0] = mData[k1];
      mData[k1] = tmp;
    }
  }

  //Swap this image width and height
  unsigned int tmp = mWidth;
  mWidth = mHeight;
  mHeight = tmp;
}
```

Implicit transposition We can also implement a separated function without any transposition of image data. A tricky scheme to accomplish this is to perform the transposition implicitly when we write into the result image of the 1D function that operates on rows:

Here is how it looks in the implementation of the separated naive algorithm:

```
for_Y (image, y)
{
  for_X (image, x)
  {
    int x0 = max(0, x - radius);
    int x1 = min(image.width() - 1, x + radius);

    max_pixel = image(x, y);

    for (int u = x0; u <= x1; ++u) //note <=, not <
    {
      if (image(u, v) > max_pixel)
      {
        max_pixel = image(u, v);
```

```
      }
    }

    // Write in reverse index order
    // for implicit transposition
    result(y, x) = max_pixel;
  }
}
```

The critical line is where we write the result—we simply swap the images.

When calling this function, we ensure that the result image its dimensions swapped from the image we wish to process:

```
filter_max_naive_separated(image)
{
  Image result(image.height(), image.width());

  filter_max_naive_rows_in_cols(image, result);
  filter_max_naive_rows_in_cols(result, image);
}
```

Notice that we merely call the function with result and image swapped on the second pass. On the second pass, the column data is in the rows of result, and the algorithm will thus write the column data (rows in result) in the columns of image.

Although this scheme has the appearance of elegance, the requirement to swap the x and y indices in the 1D algorithm might easily be overlooked leading to subtle bugs (especially if no index checking is performed). It may also confuse other developers: they might see it as a mistake and swap them.

Also note that this scheme does not allow in-place implementations. (Not without writing some very tricky code, at least).

Access modifiers An elegant way to solve the problem is to use an image access modifier. An access modifier changes the way pixels are accessed—in this case it will the meaning of x and y indices. There are different ways of doing this, and it is discussed more thoroughly in its own section under Image Containers in the Appendix A. To demonstrate the method, we will use a simple wrapper class that overloads the access function:

```
template<typename T>
class ImageSwapXY
{
  ImageAcess(const & Image<T> image):mImage(image);
  T& operator(unsigned int x, unsigned int y)
  {
    return mImage(y, x);
  }
}
```

We can now write our 2D algorithm as follows:

```
filter_max_naive_separated(image)
```

```
{
  Image result(image.width(), image.height());

  filter_max_naive_rows_in_cols(image, ImageSwapXY(result));
  filter_max_naive_rows_in_cols(result, ImageSwapXY(image));
}
```

The implementation of the function `filter_max_naive_rows_in_cols` can now write the result in rows, which is much more natural. Note that we do not need the ImageSwapXY objects for anything else than to wrap the result images—all the data is written on the original image.

4.2 Implementing Minimum Filters

Given an algorithm for a maximum filter, it is usually easy to convert it to a minimum filter. In the case of exact filters, the appropriate comparissons need to change (for example, "greater than" changes to "less than"). In the case of the aproximate algorithms given, we simply replace p with $-p$.

Minimum filters for free For an arbitrary algorithm, the changes might not be obvious. In this case, we can pre- and post-process the image to get the desired result, using the following formula:

$$\min I = -\max(-I)$$

That is, we invert the image, run it through the maximum filter, and invert the result. The only caveat is that we must ensure that the inverted image fits into our data range. To do this, we use the following conversion instead:

$$\min I = I_0 - \max(I_0 - I),$$

where I_0 is the maximum value in our data range.

A generic implementation We might also want to implement both filters as a single algorithm where a parameter forces the desired behaviour. How this can be done depends on what language features are available. Languages with function datatypes (including function pointers and delegates) can use the appropriate comparator as argument. The naive algorithm can look like this:

```
filter_extreme(
  const Image<float> & image,
  int radius,
  bool (* comparator)(float,float),
  Image<float> & result)
{
  for (int x = 0; x < image.width(); ++x)
  {
    int x0 = max(0, x - radius);
    int x1 = min(image.width() - 1, x + radius);

    for (int y = 0; y < image.height(); ++y)
    {
```

```
      int y0 = max(0, y - radius);
      int y1 = min(image.height() - 1, y + radius);

      extreme_pixel = image(x, y);

      for (int u = x0; u <= x1; ++u) //note <=, not <
      {
        for (int v = y0; y <= y1; ++v)
        {
          if (comparator(image(u, v),extreme_pixel))
          {
            extreme_pixel = image(u, v);
          }
        }
      }

      result(x, y) = extreme_pixel;
    }
  }
}


bool greater-than(float x, float y)
{
  return x > y;
}

bool less-than(float x, float y)
{
  return x < y;
}

///...

// To max filter an image:
filter_extreme(image, result, radius, greater_than);

// To min filter an image:
filter_extreme(image, result, radius, less_than);
```

4.3 Windows with Even Diameters

All the algorithm descriptions are given in terms of the radius, which implies that all the window diameters are odd. It is possible to define the algorithms for even windows too, but the implementation becomes somewhat trickier. In this section, I describe the general way in which you can proceed to turn a odd-window algorithm into one that supports all window sizes.

There are all kinds of interesting (and often rather tricky) ways in which you can adapt an algorithm to support even-sized windows. But there is one which is fairly useful across the board: extend the defintion to work with different radii as follows:

$$(\max_r I)(x,y) = \max_{\substack{x-r_{x0} \le u \le x+r_{x1} \\ y-r_{y0} \le v \le y+r_{y1}}} I(u,v) \tag{9}$$

Instead of a single radius, there are now four radii: one for each direction around a pixel. This definition not only makes it possible to max filters with even diameters, it also allows us to shift the pixel around, so that it is not in the center of the window.

Below is an adaption of the max-queue algorithm:

```
(Image<T> & image,
  unsigfned int left_radius,
  unsigned int right_radius,
  Image<T> & resultimage)
{
  unsigned int window_width = left_radius + right_radius + 1;
  MaxQueue<T> window(window_width);

  for (unsigned int y = 0; y < image.height(), ++y)
  {
    unsigned int x_read = 0;//read from image
    unsigned int x_write = 0;//write to result

    // Step 1 - start pushing data into the queue
    for (; x_read < left_radius; ++x_read)
    {
      window.push(image(x_read, y));
    }

    // Step 2 - keep on pushing data, start writing output
    for (; x_read < image.width(); ++x_read, ++x_write)
    {
      window.push(image(x_read, y));
      result(x_write, y) = window.max();
    }

    // Step 3 - Keep on removing values from the queue
    // and keep on writing output.
    // We cannot push more data - we are out of pixels.
    for (; x_write < image.width(); ++x_write)
    {
      window.pop();
      result(x_write, y) = window.max();
    }

    // A check to make sure we have not accidentally popped
    // one pixel too many or few.
    assert(queue.size() == right_radius + 1);

    // If we do not clear values, the values that remain
    // in the queue will be used in the first few
    // calculations of the next row.
```

```
    window.clear();
  }

return result;

filter_max_2d()
{
  filter_max_1d_rows(image, result, left_radius,
    right_radius);
  result.transpose();

  filter_max_1d_rows(result, image, top_radius,
    bottom_radius);
  result.transpose();
}
```

Notice how relative painless the conversion is. This is also true for the other algorithms—try it!

The corner calculations in the naive algorithm becomes:

```
int x0 = max(0, x - left_radius);
int x1 = min(image.width() - 1, x + right_radius);

int y0 = max(0, y - top_radius);
int y1 = min(image.height() - 1, y + bottom_radius);
```

For square windows centered around the working pixel, the left and right radii can be calculated from the diameter with this formula:

```
left_radius = diameter / 2; // assumes floor division
right_radius = diamater - 1 - left_radius;
```

4.4 Filtering a Region of Interest

In some cases filtering the entire image is wasteful for the purpose at hand. To limit any of the algorithms to a rectangular region is trivial; however, there is an important issue to consider—that is how the borders should be treated.

Before we get to that, let us ignore for the moment the borders and see how we can implement algorithms for ROIs.

There are two basic strategies.

The first is to pass a container rectangle to the algorithm, and limit its operation within that rectangle. In the case of the naive algorithm, the only change is to the loop ranges:

Here, we pass in a `Rect` structure `roi` that is defined like this:

```
struct Rect
{
  int x0; //inclusive
  int x1; //exclusive
```

```
  int y0; //inclusive
  int y1; //exclusive
}


// We might want to check that the
// roi rectangle is indeed in the image
// before we proceed

for (int x = roi.x0; x < roi.x1; ++x)
{
  int x0 = max(0, x - radius);
  int x1 = min(image.width() - 1, x + radius);

  for (int y = roi.y0; y < roi.y1; ++y)
  {
    int y0 = max(0, y - radius);
    int y1 = min(image.height() - 1, y + radius);

    max_pixel = image(x, y);

    for (int u = x0; u <= x1; ++u) //note <=, not <
    {
      for (int v = y0; y <= y1; ++v)
      {
        if (image(u, v) > max_pixel)
        {
          max_pixel = image(u, v);
        }
      }
    }
    result(x, y) = max_pixel;
  }
}
```

Notice that the clamping ranges have been left in tact. This means that the algorithm will use pixels outside the image for its calculation, but only as is necessary.

It should be easy to see that the other algorithms are as easily modified.

The second strategy is to wrap the image in an access modifier, and pass it to the (unmodified) algorithm. The access modifier simply maps coordinates from the region of interest to indices starting at zero.

Here is a naive implementation of this idea:

```
class ROIWrapper : AbstractImage
{
  int mX0;
  int mY0;
  int mWidth;
  int mHeight;
```

```
  AbstractImage & mImage;

  RoiWrapper(AbstractImage & image, const Rect & roi):
    mX0(roi.x0),
    mY0(roi.y0),
    mWidth(roi.x1 - roi.x0),
    mHeight(roi.y1 - roi.y0)
    mImage(image) //copy reference to image
  {}

  const T & operator()(int x, int y) const
  {
    return image(x - mRoi.x0, y - mRoi.y0);
  }

  T & operator()(int x, int y)
  {
    return image(x - mRoi.x0, y - mRoi.y0)&;
  }

  int width()
  {
     return mWidth;
  }

  int height()
  {
    return mHeight;
  }
}
```

To filter part of our image, we simply wrap the image with the ROIWrapper, and pass it to the normal algorithm (which, now, of course, must take an AbstractImage). Note that we wrap our result in exactly the same way.

I said the implementation above of the wrapper is naive, because using it in an algorithm will make it ignore all pixels outside the border, even when they are available. Unfortunately, it is not easy to circumvent this problem.

One way to handle it is to extend the ROI, but limit changes to within the ROI.

```
//somewhere

T * IGNORED_ADRESS = new T();

class ROIWrapper : AbstractImage
{
  int mReadX0;
  int mReadY0;
  int mReadWidth;
```

```
  int mReadHeight;

  Rect mWriteRect;

  AbstractImage & mImage;

  RoiWrapper(
    AbstractImage & image,
    const Rect & roi,
    unsigned int radius
  ):
    mReadX0(max(0, roi.x0 - radius)),
    mReadY0(max(0, roi.y0 - radius)),
    mReadWidth(min(image.width(), roi.x1) - mReadX0),
    mReadHeight(min(image.height(), roi.y1) - mReadY0),
    mWriteRect(roi),
    mImage(image) //copy reference to image
  {}

  const T & operator()(int x, int y) const
  {
    return image(x + mReadX0, y + mReadY0);
  }

  T & operator()(int x, int y)
  {
    int writeX = x + mReadX0;
    int writeY = y + mReadY0;

    if(mWriteRect.contains(writeX, writeY))
    {
      return image(, y - mRoi.y0)&;
    }
    else
    {
      return IGNORED_ADDRESS;
    }
  }

  int width()
  {
     return mReadWidth;
  }

  int height()
  {
    return mReadHeight;
  }
}

ROIWrapper(image, roi, radius);
RoiWrapper(result, roi, radius);
```

```
filter_max(image, result, radius);
```

The `IGNORED_ADRESS` is simply an address where we can write values we wish to ignore to.

The downside of this implementation is that the algorithm will do more work than needed. On the upside, it makes it possible to use any image algorithm in a ROI without the need to modify the algorithm.

4.5 Maximum and Minimum Filters for Binary Images

For binary images, the implementation can be significantly optimised.

First note that for binary values, $\max(a, b) = a \,\mathrm{OR}\, b$ and $\min(a, b) = a \,\mathrm{AND}\, b$.

This allows us to define the maxfilter as follows:

$$(\max_r I)(x, y) = \mathrm{OR}_{\substack{x-r \le u \le x+r \\ y-r \le v \le y+r}} I(u, v) \tag{10}$$

From this, we can modify our naive algorithm to bail out as soon as a 1 is found.

```
max_filter()

for (int x = 0; x < image.width(); ++x)
{
  int x0 = max(0, x - radius);
  int x1 = min(image.width() - 1, x + radius);

  for (int y = 0; y < image.height(); ++y)
  {
    int y0 = max(0, y - radius);
    int y1 = min(image.height() - 1, y + radius);

    max_pixel = 0;

    for (int u = x0; (u <= x1) & (max_pixel == 0); ++u)
    {
      for (int v = y0; (y <= y1) & (max_pixel == 0); ++v)
      {
        if (image(u, v) == 1)
        {
          max_pixel = 1;
        }
      }
    }

    result(x, y) = max_pixel;
  }
}
```

The max-queue algorithm needs no adaption; we only adapt the max-queue. This is done in Section C, but you will already get the gist with the adaption

we make for the implicit queue algorithm. The main difference is that we need not search for maximum. We will never pop the maximum if it is 0, and if we pop the maximum when it is 1, it must be the only one in the window—which means the new maximum is the right-most 0.

```
unsigned int window_width = 2 * radius + 1;
unsigned int width = image.width();

for_Y (image, y)
{
  unsigned int x_read = 0;//read from image,
                          //end of implicit queue

  unsigned int x_write = 0;//write to result
  unsigned int x_start = 0;//start if implicit queue
  unsigned int x_max = 0; //max in implicit queue

  // Step 1
  for (; x_read < radius; ++x_read)
  {
    if (image(x_read, y) >= image(x_max, y))
    {
      x_max = x_read;
    }
  }

  // Step 2
  for (; x_read < width; ++x_read, ++x_write)
  {
    if (image(x_read, y) >= image(x_max, y))
    {
      x_max = x_read;
    }

    result(x_write, y) = image(x_max, y);

    if (x_read + 1 >= window_width)
    //only pop if there are enough items in the queue!
    {
      if (x_start == x_max)
        // There are no 1s in the window, so
        // set the index to the right-most 0
        x_max = x_read;
      }

      ++x_start;
    }
  }

  for (;x_write < image.width(); ++x_write)
  {
    result(x_write, y) = image(x_max, y);
```

```
    if (x_start == x_max)
    {
      // There are no 1s in the window, so
      // set the index to the right-most 0
      x_max = image.width() - 1;
    }

    ++x_start;
  }
}

return result;
```

Note, this algorithm does not have the same worse-case performance as the one for arbitrary values. The worse-case performance of the binary algorithm is the same as the average case. In fact, a monotonic wedge is automatically maintained here—so this algorithm is essentially equivalent to the monotonic wedge algorithm for binary values.

A Image Containers

The way in which an algorithm is implemented depends heavily on the implementation of the data structure used to represent an image. Consider, for example, all the small differences in a Matlab implementation of the naive algorithm compared to the C++-like one given earlier:

```
[height, width] = size(image);

for x=1:width
  x0 = max(1, x - radius);
  x1 = min(width, x + radius)

  for y=1:height
    y0 = max(1, y - radius);
    y1 = min(height, y + radius);

    window = image(y0:y1, x0:x1);

    result(y, x) = max(window(:));
  end
end
```

The goal of this section is not to look at all the different ways images can be represented, and how these affect the implementation of the algorithms. Rather, this section will look at the presentation assumed in the algorithm descriptions given above, and give some general enhancements that you should be able to implement on top of whatever presentation you are using.

A.1 Image Class Interface

```
class Image<T> //T is the image pixel data type - int or float
{
```

```
public:
  unsigned int width(); //returns the width of the image
  unsigned int height(); //returns the height of the image

  //Gets value of pixel at (x, y)
  const T & operator(unsigned int x, unsigned int y) const;

  //Gets a writeable reference to the pixel at (x, y)
  T & operator(unsigned int x, unsigned int y);

}
```

If you define your own class, you might also find it useful to provide accessors to the last x- and y-indices.

A.2 Image Loops

The *CImage* library defines several loops that are extremely convenient.

The following loop iterates over all pixels in the image:

```
for_XY(image, x, y)
{
  do_something_with(image(x, y));
}
```

It is equivalent to the more verbose and error-prone

```
for(int x = 0; x < image.width(); i++)
{
  for(int j = 0; j < image.height(); j++)
  {
    do_something_with(image(x, y));
  }
}
```

The definitions of the macros are simple. Here they are:

```
#define for_1(i, end) for(unsigned i = 0; i < (end); i++)
#define for_X(image, x) for_1(x, (image).width())
#define for_Y(image, y) for_1(y, (image).height())

#define for_XY(image, x, y) \
  for_X((image), x) for_Y((image), y)
```

These are easy to extend to images of more than 2 dimensions (including multi-channel images).

These types of macros can also be used to implement loops over windows of images. Here is an example:

```
#define for_2(i, begin, end)                  \
  for(unsigned i = begin; i < end; i++)

#define for_win_X(image, x, u, radius)        \
  for_2(u,                                    \
```

```
        x - radius >= 0 ? x - radius : 0,     \
        min(image.width() - 1, x + radius))

#define for_win_Y(image, y, v, radius)        \
  for_2(v,                                    \
        y - radius >= 0 ? y - radius : 0,     \
        min(image.width() - 1, y + radius))

#define for_win_XY(image, x, y, u, v, radius) \
  for_win_X(image, x, u, radius)              \
  for_win_Y(image, y, v, radius)
```

Using these we can write the naive algorithm as follows:

```
for_XY(image, x, y)
{
  max_pixel = image(x, y);

  for_win_XY(image, x, y, u, v, radius)
  {
    if(image(u, v) > max_pixel)
    {
      max_pixel = image(u, v);
    }

    result(x, y) = max_pixel;
  }
}
```

A.3 Image Iterators

The looping mechanisms shown above can of course also be implemented with iterators.

These are simple to implement in the idioms of any language, except for C++.

For example, in Java

```
class XYImageIterator implements Iterator
{
  int x;
  int y;
  Image image;

  XYImageIterator(Image image)
  {
    this.image = image;
    x = 0;
    y = 0;
  }

  boolean hasNext()
  {
```

```
    return (y < image.height());
  }

  float next()
  {
    float value = image(x, y);

    x++;

    if(x >= image.width())
    {
      x = 0;
      y++;
    }
  }
}

class XYWindowIterator()
{
  int x;
  int y;
  int u;
  int v;
  Image image;
  int radius;

  int uBegin;
  int uEnd;

  int vEnd;
  // vBegin is not necessary


  XYWindowIterator(Image image, int radius)
  {
    x = 0;
    y = 0;

    calculateWindowLimits()

    this.image = image;
    this.radius = radius;
  }

  bool hasNextWindow()
  {
  }

  bool hasNextPixel()
  {
    return v < vEnd;
  }
```

```
  float nextWindow()
  {
    float pixel;
    x++;

    if (x >= image.width())
    {
      x = 0;
      y++;
    }

    calculateWindowLimits();

    return pixel;
  }

  void calculateWindowLimits()
  {
    uBegin = max(0, x - radius);
    uEnd = min(image.getWidth() - 1, x + radius);

    float vBegin = max(0, y - radius);
    vEnd = min(image.getHeight() - 1, y + radius);

    u = uBegin;
    v = vBegin;
  }

  float nextPixel()
  {
    float pixel = image(u, v);

    u++;

    if (u > uEnd) // Not >=
    {
      u = uBegin;
      v++;
    }

    return pixel;
  }
}
```

The window iterator allows us to implement the naive algorithm as follows:

```
for(XYWindowIterator i; i.hasNextWindow(); )
{
  float maxPixel = i.nextWindow();

  for(; i.hasNextPixel();)
```

```
  {
    float pixel = i.nextPixel();

    if(pixel > maxPixel)
    {
      maxPixel = pixel;
    }
  }
}
```

This type of iterator is unfortunately only useful in a limited set of circumstances (we cannot, for example, conveniently add two images and put the result in a third). More useful are iterators that return indeces, instead of values:

```
class XYIndex
{
  public int x;//no use protecting these
  public int y;
}

class XYIndexIterator implements Iterator
{
  int x;
  int y;
  Image image;

  XYImageIterator(Image image)
  {
    this.image = image;
    x = 0;
    y = 0;
  }

  boolean hasNext()
  {
    return (y < image.height());
  }

  XYIndex next()
  {
    float index = new XYIndex(x, y);

    x++;

    if(x >= image.width())
    {
      x = 0;
      y++;
    }
  }
}
```

Although we could write `image.getPixel(index.x, index.y)`, it is a lot neater to implement the Image class to take `XYIndex` objects as parameters for pixel accessors, so that we can write `image.getPixel(index)`.

A.4 Image Access Modifiers

An access modifier is a scheme that modifies the way pixel data is accessed—usually to hide ugly an error prone calculations. In the separation section (Section 4.1), we have seen how access modifiers can be used to swap x and y coordinates, allows elegant implementations of 2D functions in terms of 1D functions. There, we have implemented the access modifier as a wrapper class.

Access modifiers are useful for the following reasons:

- they make algorithms more reusable,
- they make it easier to write test code based on invariants.

Access modifiers can be implemented in several ways:

- wrapper classes;
- function pointers, delegates or other function data types (including functors); and
- iterators.

Implementation with Wrapper Classes

```
class ImageRow
{
  ImageRow(Image &, unsigned int row = 0):
    mRow(row)
  {};

  set_row(unsigned int row){mRow = row;}

  inc_row(){mRow++;}

  operator[](unsigned int x)
  {
    return mImage(x, row);
  }
}
```

Notice that we overloaded operator [] instead of operator (). The former is much more common in the world of 1D algorithms.

This allows us to use any 1D function on our images:

```
for_Y (image, y)
{
  some_1d_algorithm(ImageRow(image, y), ...);
}
```

or if we do not like to create a new instance for each row, we could call

```
ImageRow image_row(image)

for_Y (image, y)
{
  image_row.set_row(y);
  some_1d_algorithm(image_row, ...);
}
```

A third, and perhaps much cleaner implementation:

```
class ImageRow
{
  ImageRow(Image & image):
    mImage(image),
    mRow(0);
  set_row(unsigned int row);

private:
  const Image & mImage;
  unsigned int mRow;
}


class ImageRows
{
  ImageRows(Image &):
    mImageRow(image)
  {};

  ImageRow & operator[](unsigned int y)
  {
    mImageRow.set_row(y);
    return mImageRow;
  }
}

...

some_2d_function(image, ...)
{
  ImageRows image_rows(image)

  for_Y (image, y)
  {
    some_1d_algorithm(image_rows[y], ...);
  }
}
```

Implementation function pointers Another way to implement access modifiers is to build the functionality directly into the image container. We modify access by calling a access modifier function, which changes the

current accessor being used. Here is a simple implementation.

```
class Image
{
  typedef T (* AccessFunction)(Image<T> image, T x, T y);

  void set_acces_swap_xy()
  {
    access_function =
  };

  void set_access_straight()
  {
    access_function = access_swap_xy();
  }

  AccessFunction access_function;
}

T access_straight(image, x, y)
{
  return image(x, y);
}

T access_swap_xy(image, x, y)
{
  return image(x, y);
}
```

It is also possible to implement access modifiers into the algorithms, instead of into the classes, as has been done in this implementation of the naive algorithm:

```
filter_max(
  Image image,
  Image result,
  unsigned int radius,
  T (* accessor)(image, unsigned int, unsigned int))
{
  for_Y(image, y)
  {
   for_X(image, x)
   {
     x0 = ...
     x1 = ...

     max_pixel = accessor(image, x, y);

     for_X(image, u)
     {
       T pixel = accessor(image, x, y);
```

```
      if(pixel > max_pixel)
      {
        max_pixel = pixel;
      }
    }

    acessor(result, x, y) = max_pixel;
  }
 }
}
```

Implementation with iterators Implementation through iterators is straightforward. Iteration itself does not change—only the pixel that is returned.

Here is a Java implementation of an iterator that swaps x and y:

```
class SwapXYImageIterator implements Iterator
{
  int x;
  int y;
  Image image;

  XYImageIterator(Image image)
  {
    this.image = image;
    x = 0;
    y = 0;
  }

  boolean hasNext()
  {
    return (y < image.height());
  }

  float next()
  {
    float value = image(y, x); //simply swap them

    x++;

    if(x >= image.width())
    {
      x = 0;
      y++;
    }
  }
}
```

B Fixed-width Deques

A fixed-width deque is a deque with a fixed capacity: whenever an item is pushed into the deque so that the capacity is exceeded, an item is popped from the other side of the deque automatically.

Standard-deque-based implementation It is easy to build a FWD from a standard by overloading the push operations:

```
template<class T>
class FixedWidthDeque : deque
{
public:
  FixedWidthDeque(unsigned int capacity);
  void push_back(T item);
  void push_front(T item);
private:
  mCapacity;
};

template<class T>
FixedWidthDeque<T>::FixedWidthDeque(capacity):
  mCapacity(capacity)
{
  //nothing to do here
}

template<class T>
void FixedWidthDeque<T>::push_back(T item)
{
  deque::push_back(item);

  if (size() > mCapacity)
  {
    pop_front();
  }
}

template<class T>
void FixedWidthDeque<T>::push_front(T item)
{
  deque::push_front(item);

  if (size() > mCapacity)
  {
    pop_back();
  }
}
```

Array-based implementation We can significantly improve the performance of the container by building it on top of a simple array instead. The following implementation maintains front and back indices.

```
template<typename T>
class FixedWidthQueue
{
public:
  FixedWidthQueue(unsigned int w);
  ~FixedWidthQueue();

  void clear();

  void push_back(const T& item);
  void push_front(const T& item);
  T pop_front();
  T pop_back();

  const T& back() const;
  const T& front() const;

  unsigned int size() const;
private:
  T * mData;
  unsigned int mFrontIndex;
  unsigned int mBackIndex;
  unsigned int mSize;
  const unsigned int mCapacity;

  // No implementation for these
  // are provided - they are made
  // private to prevent their use.
  T& operator=(const T& other);
  FixedWidthQueue(const FixedWidthQueue<T> & other);
};

template<typename T>
FixedWidthQueue<T>::FixedWidthQueue(unsigned int width):
  mSize(0),
  mFrontIndex(0),
  mBackIndex(0),
  mCapacity(width)
{
  mData = new T[width];
}

template<typename T>
FixedWidthQueue<T>::~FixedWidthQueue()
{
  delete [] mData;
}

template<typename T>
unsigned int FixedWidthQueue<T>::size() const
{
  return mSize;
```

```
}

template<typename T>
void FixedWidthQueue<T>::push_back(const T& item)
{
  if (mSize == 0)
  {
    mData[0] = item;
    mSize++;
  }
  else
  {
    mBackIndex++;

    if (mBackIndex >= mCapacity)
    {
      mBackIndex = 0;
    }

    mData[mBackIndex] = item;

    if (mSize < mCapacity)
    {
      mSize++;
    }
    else
    {
      mFrontIndex++;

      if (mFrontIndex >= mCapacity)
      {
        mFrontIndex = 0;
      }
    }
  }
}

template<typename T>
void FixedWidthQueue<T>::push_front(const T& item)
{
  if (mSize == 0)
  {
    mData[0] = item;
    mSize++;
  }
  else
  {
    if (mFrontIndex > 0)
    {
      mFrontIndex--;
    }
    else
```

```
    {
      mFrontIndex = mCapacity - 1;
    }

    mData[mFrontIndex] = item;

    if (mSize < mCapacity)
    {
      mSize++
    }
    else
    {
      if (mBackIndex > 0)
      {
        mBackIndex--;
      }
      else
      {
        mBackIndex = mCapacity - 1;
      }
    }
  }
}

template<typename T>
void FixedWidthQueue<T>::clear()
{
  mFrontIndex = 0;
  mBackIndex = 0;
  mSize = 0;
}

template<typename T>
T FixedWidthQueue<T>::pop_front()
{
  assert(mSize > 0);

  T return_val = front();

  if (mSize == 1)
  {
    clear();
  }
  else
  {
    mFrontIndex++;

    if (mFrontIndex >= mCapacity)
    {
      mFrontIndex = 0;
    }
```

```
    mSize--;
  }

  return return_val;
}

template<typename T>
T FixedWidthQueue<T>::pop_back()
{
  assert(mSize > 0);

  T return_val = back();

  if (mSize == 1)
  {
    clear();
  }
  else
  {
    if (mBackIndex > 0)
    {
      mBackIndex--;
    }
    else
    {
      mBackIndex = mCapacity - 1;
    }

    mSize--;
  }

  return return_val;
}

template<typename T>
const T& FixedWidthQueue<T>::back() const
{
  assert(mSize > 0);

  return mData[mBackIndex];
}

template<typename T>
const T& FixedWidthQueue<T>::front() const
{
  assert(mSize > 0);

  return mData[mFrontIndex];
}
```

C Max-queues

A max-queue is a queue that also maintains the maximum element in the queue.

The maximum is maintained by making the following adjustments to the standard queue operations:

- When the first element is pushed into the queue, it becomes the max-element.
- Whenever a new element is pushed into the queue that is larger than the current max-element, it replaces the max-element.
- Whenever we push an element from the queue, we check whether it is the max-element. If it is, we do a linear search through all elements in the queue for the largest one—this now becomes the max-element.

To make it possible to know when we pop the maximum element from the queue, we also maintain its position in the queue.

Here is the implementation of a fixed-width max-queue, built on top of a deque which allows random access:

```
class MaxQueue<T>
{
  unsigned int max_index;
  unsigned int max_value;

  deque<T> data;

  public void push(T item)
  {
   if(empty)
   {
     max_value = item;
     max_index = 0;
   }
    else if(item >= max_value) \\>= is better than >
    {
      max_value = item;
      max_index = data.size();
    }

    data.push_back(item);

    //This makes the max-queue fixed-width.
    if(size() > max_size)
    {
     pop();
    }
  }

  public T pop()
  {
```

```
    if(empty())
    {
       //ERROR!
    }
    else
    {
      T item = data.pop_front();

      if(max_index == 0)// We are popping the max element
      {
       if(size() > 0)
       {
         max_value = front();// data[0]
         max_index = 0;

         for(i = 1; i < size(); i++)
         {
           if(data[i] >= max_value)
           {
             max_value = data[i];
             max_index = i;
           }
         }
       }
       else //The max element moves forward
       {
         max_index--;
       }

       return item;
      }
    }

    public void max()
    {
     return max_value;
    }

    public void size()
    {
      return data.size();
    }
}
```

If the deque does no allow random access, but only sequential access through an iterator, we replace the search code with this:

```
if(size() > 0)
{
  Iterator itr = deque.iterator();
  max_value = itr*;
  max_index = 0;
```

```
  for(i = 1; i < size(); i++)
  {
    itr++;

    if(itr* >= max_value)
    {
      max_value = itr*;
      max_index = i;
    }
  }
}
```

Or in the Java idiom:

```
if(size() > 0)
{
  Iterator itr = deque.iterator();

  max_value = itr.next();
  max_index = 0;


  for(i = 1; i < size(); i++)
  {
    T item = itr.next();

    if(item >= max_value)
    {
      max_value = item;
      max_index = i;
    }
  }
}
```

The algorithm is can be optimised for binary values. In this case, there is no need to search for a maximum value if it is popped. The maximum value will only be popped if it the leftmost value is a 1, and the only 1 in the window, which means we can just set the new maximum value to the front of the queue.

```
class MaxQueue<T>
{
  unsigned int max_index;
  unsigned int max_value;

  deque<T> data;

  public void push(T item)
  {
   if(empty)
   {
     max_value = item;
```

```
   max_index = 0;
 }
  else if(item >= max_value) \\>= is essential
  {
    max_value = item;
    max_index = data.size();
  }

  data.push_back(item);

  //This makes the max-queue fixed-width.
  if(size() > max_size)
  {
   pop();
  }
}

public T pop()
{
  if(empty())
  {
     //ERROR!
  }
  else
  {
    T item = data.pop_front();

    if(max_index == 0)// We are popping the max element
    {
     if(size() > 0)
     {
       max_value = data.back();
       assert(max_value == 0);
       max_index = size() - 1;
     }
   }
   else //The max element moves forward
   {
     max_index--;
   }

   return item;
  }
}

public void max()
{
 return max_value;
}

public void size()
{
```

```
    return data.size();
  }
}
```

D Summed Area Tables

A Summed Area Table (also called image integral) of an image I is another image of the same size S such that:

$$S(x,y) = \sum_{u=0}^{x} \sum_{v=0}^{y} I(x,y) \tag{11}$$

A related structure—and one which we will find more useful—is the augmented SAT, where an extra row and column of zeros makes implementation much cleaner. The augmented SAT is given by

$$S^*(x,y) = \begin{cases} 0 & x = 0 \text{ or } y = 0 \\ \sum_{u=0}^{x-1} \sum_{v=0}^{y-1} I(x,y) & \text{otherwise} \end{cases}$$

The SAT makes it possible to calculate the sum of all pixels in a region in constant time—that is, it does not depend on the rectangle side. It is therefore useful for implementing fast box-filters, local standard deviation filters, and so on. It is also makes it possible the implement these filters on graphics cards, which makes them suitable for real-time implementation (in games, for example).

D.1 Calculating a SAT

Although the expressions above can be used to calculate the SAT directly, it is quite slow. The following recursion can be used to speed things up:

$$S(x,y) = I(x,y) + S(x,y-1) + S(x-1,y) - S(x-1,y-1) \tag{12}$$

A similar recursion holds for the augmented SAT:

$$S^*(x,y) = I(x-1,y-1) + S(x,y-1) + S(x-1,y) - S(x-1,y-1) \tag{13}$$

SAT

```
image_sum(0, 0) = image(0, 0);

for (unsigned int i = 1; i < image_sum.width(); i++)
{
  image_sum(i, 0) = image_sum(i - 1, 0) - image(i, 0);
}

//we already assigned image_sum(0, 0); loop starts at 1
for (unsigned int j = 1; j < image_sum.height(); j++)
{
  image_sum(0, j) = image_sum(0, j - 1) - image(0, j);
```

```
}

for (unsigned int i = 1; i < image_sum.width(); i++)
{
  for (unsigned int j = 1; j < image_sum.height(); j++)
  {
    image_sum(i, j) =
      (image_sum(i, j - 1) - image_sum(i - 1, j - 1))
      + image_sum(i - 1, j) + image(i, j);
  }
}
```

Notice the order of the calculations in the main loop: we are making sure that:

- The subtraction is always positive (which is necessary if the implementation uses the unsigned data type).
- We subtract early to reduce the risk of overflow. If we subtract only after adding the other values, we risk overflow in the intermediate calculations even though the final result itself might still be in range.

ASAT The implementation for the ASAT is very similar, the only differences are that:

- the first row and column are assigned to zero; and
- the image is accessed one index less than the SAT being calculated.

```
for (unsigned int i = 0; i < image_sum.width(); i++)
{
  image_sum(i, 0) = 0;
}

//we already assigned image_sum(0, 0); loop starts at 1
for (unsigned int j = 1; j < image_sum.height(); j++)
{
  image_sum(0, j) = 0;
}

for (unsigned int i = 1; i < image_sum.width(); i++)
{
  for (unsigned int j = 1; j < image_sum.height(); j++)
  {
    image_sum(i, j) =
      (image_sum(i, j - 1) - image_sum(i - 1, j - 1))
      + image_sum(i - 1, j) + image(i - 1, j - 1);
  }
}
```

If you are concerned about all the subtractions in the main loop, it is easy to use additions instead:

```
for (unsigned int i = 0; i < image.width(); i++)
{
```

```
  for (unsigned int j = 0; j < image.height(); j++)
  {
    image_sum(i + 1, j + 1) =
      (image_sum(i + 1, j) - image_sum(i, j))
      + image_sum(i, j + 1) + image(i, j);
  }
}
```

D.2 Finding a Sum from a SAT

The sum of a rectangular region in an image (from (x_0, y_0) to (and including) (x_1, y_1)) is given by

$$\sum_{x=x_0}^{x_1} \sum_{y=y_0}^{y_1} I(x,y) = S(x_1,y_1) - S(x_1,y_0-1) - S(x_0-1,y_0-1) + S(x_0-1,y_0-1) \tag{14}$$

provided that $x_0 > 0$ and $y_0 > 0$. Any term $S(x,y)$ for which $x = -1$ or $y = -1$ falls away in the calculation above. It is this that makes this structure so inconvenient:

```
if (x0 > 0)
{
  if (y0 > 0)
  {
    return
      ((image_sum(x1, y1) - image_sum(x1, y0 - 1))
      + image_sum(x0 - 1, y0 - 1)
      - image_sum(x0 - 1, y1);
  }
  else //y0 == 0
  {
    return
      image_sum(x1, y1) - image_sum(x0, y1);
  }
}
else //x0 == 0
{
  if (y0 > 0)
  {
    return
      ((image_sum(x1, y1) - image_sum(x1, y0 - 1));
  }
  else //y0 == 0
  {
    return image_sum(x1, y1);
  }
}
```

In terms of the ASAT, the sum of a rectangular region of an image is given by:

$$\sum_{x=x_0}^{x_1} \sum_{y=y_0}^{y_1} I(x,y) = S(x_1+1,y_1+1) - S(x_1+1,y_0) - S(x_0,y_0) + S(x_0,y_0) \tag{15}$$

for any (x_0, y_0) and (x_1, y_1) that are legal image indices.

The computation is much cleaner (and indeed, faster):

```
return
  ((image_sum(x1 + 1, y1 + 1) - image_sum(x1 + 1, y0))
  + image_sum(x0, y0)
  - image_sum(x0, y1 + 1);
```

D.3 Checking for Overflow

We probably do not want to check each and every calculation for overflow—it will slow down out computations considerably. But during development, it is useful to have a few asserts to make sure we are not thrown off track by corrupt results caused by overflow (especially when dealing with large SATS discussed in the next section).

Thus, here we give a few tests to ensure results of calculations stay in range.

Safe addition and subtraction The following test obviously is useless:

```
z = x + y;
assert(z <= MAX_VALUE); //useless
```

If `z` overflows, it will still be smaller than `MAX_VALUE`; the test will always pass.

The following test looks better, but it is only valid if `y >= 0`:

```
assert(x <= MAX_VALUE - y); //useful for positive y
z = x + y;
```

To better document the intention of the test, we put the code in a function. This has the additional benefit of allowing us to write specialised code for different data types by overloading functions. This reduces the mental overhead when inserting these tests in code. The following code gives correct tests for non-negative and arbitrary integers respectively:

```
bool safe_to_add(unsigned int x, unsigned int y)
{
  return x <= MAX_UINT - y;
}

bool safe_to_add(int x, int y)
{
  if ((x > 0) && (y > 0))
    return x <= MAX_INT - y;
  if ((x < 0) && (y < 0))
    return x >= MIN_INT - y;

  // if x == 0 or y == 0, it is safe
  // if x and y has opposite signs, it is always safe
  // since then min(x, y) < x + y < max(x, y)
  return true;
}
```

Here are the tests for subtraction.

```
bool safe_to_subtract(unsigned int x, unsigned int y) //x - y
{
  return (x >= y);
}

bool safe_to_subtract(int x, int y) //x - y
{
  if ((x > 0) && (y < 0))
    return x <= MAX_INT + y;

  if ((x < 0) && (y > 0))
    return x >= MIN_INT + y;

  if ((x > 0) && (y > 0) && (x < y))
    return x >= MIN_INT + y;

  if ((x < 0) && (y < 0) && (x < y))
    return MAX_INT + x >= (MAXINT + y) + MIN_INT;

  // if x == 0 or y == 0, it is safe
  // else if (x > y)) return true;

  return true;
}
```

Notice the complicated return expression for the last case (when all signs are negative and `x < y`): it has been carefully crafted to not overflow itself—the quantity `MIN_INT + y` will overflow if `y` is negative; top prevent this, we add `MAX_INT` to both sides. This assumes the compiler will not do anything clever. (If it does, then it should be clever enough to enable overflow checks for debugging, in which case you will not need any of this code).

Also note that the test are only safe if `MIN_VAL` and `MAX_VAL` really are the data type extremes, otherwise the test computations themselves can overflow. Thus, do not use these tests for more general bound checking!

Safe multiplication and division The test for multiplication needs a check for zero to prevent a division by zero error.

The following test is valid for unsigned integer and floating point types:

```
bool safe_to_multiply(unsigned int x, unsigned int y)
//will also work for any unsigned type
{
  if (y == 0)
    return true;

  return (x <= MAX_VALUE / y);
}
```

If we have to deal with negative numbers as well, we have to ensure the

result lies within both ends of the data range. Integer division with negative numbers is a murky area (I do not know, for instance, whether -10/3 will give -3 or -4), so to avoid potential problems, we only perform positive division. The more complicated safe-multiplication test now becomes:

```
safe_to_multiply(int x, int y)
    //will also work for other signed types
    //assumes -MIN_VALUE <= MAX_VALUE
{
  if ((y == 0) || (x == 0))
    return true;

  if ((x > 0) && (y > 0))
    return (x <= MAX_VALUE / y);

  if ((x < 0) && (y < 0))
    return (-x <= MAX_VALUE / -y);

  if ((x > 0) && (y < 0))
    return (-MIN_VALUE / x) >= -y) :

  /*(x < 0) && (y < 0)*/
  return (-MIN_VALUE / y) >= -x);
}
```

The assumption `-MIN_VAL` won't overflow is itself a bit murky, so be sure to check it for your language and compiler.

Except for division by zero cases, integer division can never lead to overflow.

For unsigned floating point types, the following test will work:

```
bool safe_to_divide(unsigned float x, unsigned float y)
{
  //no check for division by zero in this function

  return
    ((x == 0) || (y >= 1)) ? true : x < MAX_VALUE * y);
}
```

Notice that we do not check for division by zero in this function. There are two reasons for this:

1. Using two separate asserts will be more informative—it will immediately tell us *why* it is unsafe.
2. It will enable us to switch off overflow checking without also switching off division-by-zero checking.

In general, we will only want to activate overflow tests in program parts that deals with that issue specifically (such as large SATs discussed below), and only while developing them. Division-by-zero is a more general problem, with a different solution.

For arbitrary floating point types, the following test will work:

Safe exponentiation

```
safe_to_pow(float x, int n)
{
  return x < pow(MAX_VALUE, 1.0/n));
}
```

D.4 Large SATs

Large SATs are bound to overflow. If they are only used for fast sums of image regions, then we can implement a structure that behaves like a proper SAT, but won't overflow, at the expensive of some extra memory. We will call this a Tile SAT (TSAT).

The idea behind the TSAT is that any SAT calculated over only a region of the image will give correct sums for all queries in that region.

That is, if

$$T(x,y) = \sum_{u=u_0}^{u_1} \sum_{v=v_0}^{v_1} I(u,v) \tag{16}$$

then for $u_0 < x_0 \leq x_1 \leq u_1$ and $v_0 < y_0 \leq y_1 \leq v_1$, the following holds:

$$\sum_{x=x_0}^{x_1} \sum_{y=y_1}^{y_1} I(x,y) = T(x_1,y_1) - T(x_1,y_0-1) - T(x_0-1,y_1) + T(x_0-1,y_0-1) \tag{17}$$

The TSAT is a collection of such tiles T that overlap enough so that we can make a query for any rectangle smaller than a $\delta \times \delta$ square. As with the SAT, we can augment tiles to make code cleaner and a bit faster. The non-augmented version will not be discussed here. (Beware though that while the ASAT requires $O(w+h)$ extra memory, the augmented TSAT requires $O(wh)$ extra memory, although it should still be a small fraction of the total memory).

How big should the tiles be? Suppose the maximum value that our data type allows is Z. Then each tile should not have entries that exceed Z. If the maximum value of a pixel is z, then the maximum entry in the table is given by $d^2 z$, where d is the diameter of the (square) tile. Thus, an upper bound for d is given by:

$$d \leq \sqrt{\frac{Z}{z}} \tag{18}$$

How much must tiles overlap? First, note that we cannot make queries for regions larger than or equal to $d \times d$, regardless of the overlap. Thus $\delta < d$. In the extreme case when we want $\delta = d - 1$, we need a tile for almost every pixel in the image, using a considerable amount of memory. So ideally, we want $\delta \ll d$.

It should be easy to see that our tiles need to be computed for regions that overlap by δ pixels. If they do not, there are queries that cannot be calculated from a single tile.

How many tiles will be required? Our tiles are $d \times d$, and they should overlap by δ pixels. Thus, for an image $w \times h$, will need $M \times N$ tiles given by

$$M \times N = \left(\left\lfloor \frac{w-2}{d-\delta} \right\rfloor + 1 \right) \times \left(\left\lfloor \frac{h-2}{d-\delta} \right\rfloor + 1 \right) \tag{19}$$

Using contiguous memory We can still put all our tiles in one big block of memory. Since the tiles in the last columns and rows will typically only be partially used, we need not allocate memory for all of $MN(d+1)^2$. The actual amount of memory we need is given by:

$$d_x = w - (d-\delta)(M-1) \tag{20}$$
$$d_y = h - (d-\delta)(N-1) \tag{21}$$
$$\tag{22}$$

$$w_T = (M-1)(d-1) + M + d_x \tag{23}$$
$$h_T = (N-1)(d-1) + N + d_y \tag{24}$$
$$\tag{25}$$

Constructing the table

```
double cellArea = (double) max_cell_value / max_pixel_value;
mCellDiameter = (unsigned int) floor(sqrt(cellArea));
mNonOverlapDiameter = mCellDiameter - mMaxDifference;

assert(mWidth > 2);//otherwise the following
                   //calculation will bomb out
assert(mHeight > 2);

unsigned int cellCountX
  = (mWidth - 2) / mNonOverlapDiameter + 1;

unsigned int cellCountY
  = (mHeight - 2) / mNonOverlapDiameter + 1;

unsigned int offsetX = mWidth
  - mNonOverlapDiameter * (cellCountX - 1);

unsigned int offsetY = mHeight
  - mNonOverlapDiameter * (cellCountY - 1);

mDataWidth = (cellCountX - 1) * (mCellDiameter - 1)
  + cellCountX + offsetX;

mDataHeight = (cellCountY - 1) * (mCellDiameter - 1)
  + cellCountY + offsetY;

mTableData = new T[mDataWidth*mDataHeight];
```

```
unsigned int readX = mMaxDifference - 1;
unsigned int readY = mMaxDifference - 1;

for (unsigned int x = 0; x < mDataWidth; x++)
{
  readY = mMaxDifference - 1;

  if (x % mCellDiameter == 0)
  { //zero column for augmentation

    //backtrack so that tiles overlap
    //used next round
    readX -= mMaxDifference - 1;

    for (unsigned int y = 0; y < mDataHeight; y++)
    {
      access(x, y) = 0;
    }
  }
  else
  {
    for (unsigned int y = 0; y < mDataHeight; y++)
    {
      if (y % mCellDiameter == 0)
      {//zero row for augmentation

        access(x, y) = 0;

        //backtrack so that tiles overlap
        //used next round
        readY -= mMaxDifference - 1;
      }
      else
      {
        //calculate region sum
        access(x, y) =
          access(x - 1, y)
          + access(x, y - 1)
          - access(x - 1, y - 1)
          + image(readX, readY);

        // did we really calculate everything correctly?
        assert(access(x, y) <= max_cell_value);

        readY++;
      }
    }

    readX++;
  }
}
```