Counting Targets using the Euler Characteristic, Part 1

D Piponi

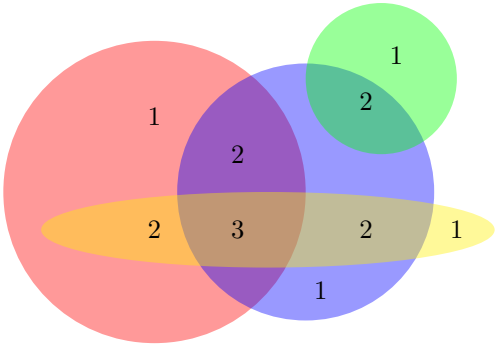
January 10, 2010

1 A Statement of the Problem

The problem I ultimately want to solve, and its solution, is described in the paper Target Enumeration via Euler Characteristic Integrals by Baryshnikov and Ghrist. My goal here is to show how to implement that solution on a computer, and by doing so make it accessible to a wider audience.

Suppose we have a set of targets we want to count. These could be anything from enemy tanks rolling over the plains to electronically tagged wildlife roaming the countryside. Each target has a region of influence which might simply be circular in shape, or might be more complex and depend on the target. Now suppose that we have a high density of sensors scattered over our domain and that each sensor can tell us how many regions of influence it lies in. Roughly speaking, each sensor counts how many targets are nearby. How do we compute how many targets we have in total?

Here's an illustration:



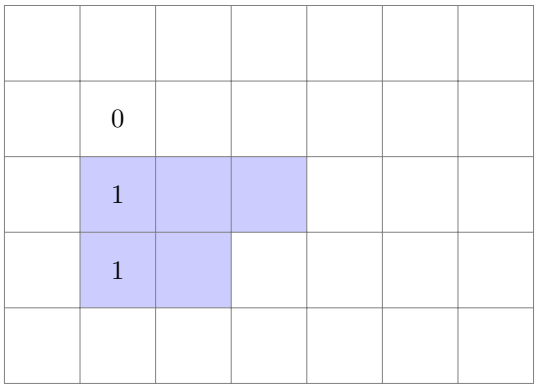
There are four targets. The region of influence for each one is coloured making it easy to see which region is which. I've labelled each region, and intersections between regions, with an integer showing how many targets can be detected there. The idea is that we'd have a very dense scattering of sensors in our domain, each sensor reporting an integer. In effect we'd be getting an

image like a rasterised version of that picture. But we wouldn't be getting the convenient colours, just an integer per pixel.

At first it seems like a trivial problem. The sensors can all count, and if every target is in range of a sensor, every target will be counted. But we can't simply add the numbers from all of the sensors as many sensors will be in the domain of influence of the same target. If we sum all of the numbers we'll be counting each target many times over. We need to be able to subtract off the targets that are counted twice. But some targets will be counted three times and so on. And how do we tell when a target has been counted twice when all we have are counts?

We'll make one simplifying assumption in solving this problem: that the regions of influence are simply connected. In other words, they are basically some kind of shape that doesn't have holes in it. That could mean anything from a square or disk to a shape like the letter 'W'. But it excludes shapes like annuli or the letter 'B'. If we make this assumption then we can solve this problem with a very simple algorithm that will work in almost all cases. In fact, the only time it fails will be situations where no algorithm could possibly work. But there's a little ground to cover before getting to the solution.

We'll make another simplifying assumption for now. That the sensors are arranged in a rectangular grid. So the data we get back from the sensors will be a grid filled with integers. That essentially turns our problem into one of image processing and we can think of sensor values as pixels. Here's a picture where I've drawn one domain of influence and I've indicated the values returned for three of the sensors.



2 Simple Grids

So lets assume the sensors have coordinates given by pairs of integers and that they return integer counts. The state of all the sensors can be represented by a function of this type:

type $Field = Int \to Int \to Int$

The arguments to such a function are the x and x coordinates of the sensor and it returns a count. We'll assume that we get zero if we try to read from beyond

our domain. We can represent a grid of sensors, including the grid's width and height, using:

```
data Grid = Grid Int Int Field
```

For efficiency something of type *Field* ought to read data from an array, but I'll not be assuming arrays here.

We can define display and addition of two grids:

```
instance Eq Grid
instance Show Grid where
  show (Grid w h f) = concat
    [[digit (f x y) | x ← [0 .. w − 1]] ++ "\n" | y ← [0 .. h − 1]] where
    digit 0 = '.'
    digit n = chr (48 + n)
instance Num Grid where
  Grid w0 h0 f0 + Grid w1 h1 f1 = Grid
    (w0 `max` w1) (h0 `max` h1)
    (λx y → f0 x y + f1 x y)
```

Our ultimate goal is to define some kind of count function with signature:

```
count :: Grid → Int
```

Now suppose the function f gives the counts corresponding to one set of targets and g is the count corresponding to another. If the region of influence of these two sets of targets is separated by at least one 'pixel' then it should be clear that

```
count f + count g ≡ count (f + g)
```

So at least approximately, *count* is additive. We also need it to be translation invariant. There's only one additive function that has this property, summing up the values at all pixels:

```
gsum (Grid w h f) = sum [f x y | x ← [0 .. w − 1], y ← [0 .. h − 1]]
```

We can implement functions to make some example grids:

```
point x y = Grid (x + 1) (y + 1)
                 (λx0 y0 → if (x0, y0) ≡ (x, y) then 1 else 0)
circle x y r = Grid (x + r + 1) (y + r + 1)
                    (λx0 y0 → if (x − x0) ↑ 2 + (y − y0) ↑ 2 < r ↑ 2 then 1 else 0)
```

And now we can build and display some examples:

```
test1 = circle 10 10 5 + circle 7 13 4 + point 5 5 + point 9 12
test2 = gsum test1
```

Here's a typical output:

```
*Main> test1
................
................
................
................
................
.....1..........
........11111...
.......1111111..
```

```
......111111111.
......111111111.
.....1222211111.
....11222221111.
....11222321111.
....1112222111..
....111122211...
....1111111.....
.....11111......
................
*Main> test2
116
```

It should be pretty clear that this doesn't count the number of targets. So how can we implement something additive and yet count targets?

Another operation we can perform on grids is scale them. Here's an implementation of scaling a grid:

$$scale\ n\ (Grid\ w\ h\ f) = Grid\ (w * n)\ (h * n)\ (\lambda x\ y \rightarrow f\ (x\ `div`\ n)\ (y\ `div`\ n))$$

Scaling up an 'image' shouldn't change the number of targets detected. It should only correspond to the same number of targets with double-sized regions of influence. So we'd also like this property:

$$count\ (n\ `scale`\ f) = count\ f$$

It's easy to see that that *gsum* actually has this property for $n > 0$:

$$gsum\ (n\ `scale`\ f) = n \uparrow 2 * gsum\ f$$

($\uparrow$ is the power function. For some reason lhs2TeX displays it as an up arrow.) These requirements are pretty tough to meet with an additive operation. But there's an amazing transformation we can perform on the data first. Instead of working on a grid with one value for each pixel we'll also store values for the 'edges' between pixels and for the 'vertices' at the corners of pixels.

3 Euler Grids

So lets define a new kind of grid to be a tuple of *Field* functions, one for faces (ie. the pixels), one for horizontal edges, one for vertical edges, and one for vertices.

$$\begin{aligned}&\textbf{data}\ EGrid = EGrid\ \{\\ &\quad eWidth :: Int, eHeight :: Int,\\ &\quad faces :: Field, hedges :: Field, vedges :: Field, vertices :: Field\\ &\quad \}\end{aligned}$$

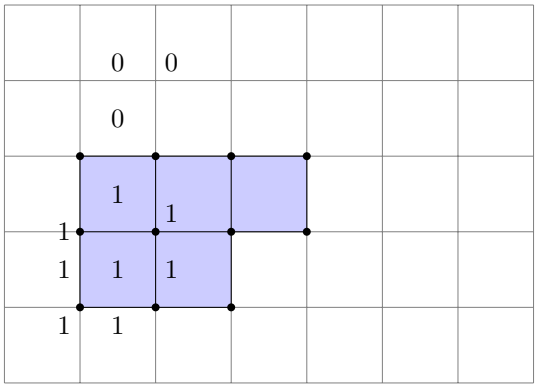
The lower left vertex is $(0, 0)$ but we need to add an extra row and column of vertices on the right. Similarly we'll need an extra row and an extra column of edges. We can now 'resample' our original grid onto one of these new style grids:

$$\begin{aligned}&g2e\ (Grid\ w\ h\ f) = EGrid\ w\ h\\ &\quad f\end{aligned}$$

$(\lambda x\ y \to f\ (x-1)\ y \qquad `max`\ f\ x\ y)$
$(\lambda x\ y \to f\ x \qquad (y-1)\ `max`\ f\ x\ y)$
$(\lambda x\ y \to f\ (x-1)\ (y-1)\ `max`\ f\ (x-1)\ y\ `max`\ f\ x\ (y-1)\ `max`\ f\ x\ y)$

I'm using the rule that the value along an edge will be the maximum of the values on the two impinging faces. Similarly, the vertices acquire the maximum of the four faces they meet.

I'll try to illustrate that here:



I hope you can see, from the placement of the labels, how I've attached values to edges and vertices as well as faces.

We now have a bit more freedom. We have three different types of sum we can carry out:

$fsum\ (EGrid\ w\ h\ f\ \_\ \_\ \_) = gsum\ (Grid\ w\ h\ f)$
$esum\ (EGrid\ w\ h\ \_\ e\ f\ \_) = gsum\ (Grid\ (w+1)\ h\ e) + gsum\ (Grid\ w\ (h+1)\ f)$
$vsum\ (EGrid\ w\ h\ \_\ \_\ \_\ v) = gsum\ (Grid\ (w+1)\ (h+1)\ v)$

(We could sum over horizontal and vertical edges separately too, but if we did that then a 90 degree rotation would give a different target count.)

Now we can define a measurement function that takes three 'weights' and gives us back a weighted sum:

$measure\ a\ b\ c\ g = \mathbf{let}\ e = g2e\ g\ \mathbf{in}\ a * vsum\ e + b * esum\ e + c * fsum\ e$

We can reproduce the *gsum* function as

$area = measure\ 0\ 0\ 1$

Try some examples to test that

$((n \uparrow 2)*) \circ area = area \circ scale\ n$

4 Exercises

Now I can leave you with some challenges:

1. Find some suitable arguments a, b and c to *measure* so that we get:

   $mystery\_property1 = measure\ a\ b\ c$
   $((n \uparrow 1)*) \circ mystery\_property1 = mystery\_property1 \circ scale\ n$

I'll let you assume that there is some choice of values that works.

(Hint: you just need to try applying *mystery_property1* to a few scalings of some chosen shape. You'll quickly find some simultaneous equations in a, b and c to solve. Solve them.)

2. Can you find a simple geometric interpretation for *mystery_property1*? Assume that the original input grid simply consists of zeros and ones, so that it's a binary image. It shouldn't be hard to find a good interpretation. It's a little harder if it isn't a binary image so don't worry too much about that case.

3. Now find some suitable arguments a, b and c to *measure* so that we get:

   $$mystery\_property2 = measure\ a\ b\ c$$
   $$((n \uparrow 0)*) \circ mystery\_property2 = mystery\_property2 \circ scale\ n$$

4. Can you find a simple interpretation for binary images? You might think you have it immediately so work hard to find counterexamples. Have a solid interpretation yet? And can you extend it to images that consist of more general integers?

5. Optimise the code for *mystery_property2* assuming the image is binary and that the input is on a 2D array. Ultimately you should get some code that walks a 2D array doing something very simple at each pixel. Can you understand how it's managing to compute something that fits the interpretation you gave?

6. Define a version of *scale* called *escale* that works on *EGrid*s. Among other things we should expect:

   $$g2e \circ (scale\ n) \equiv escale\ n \circ g2e$$

   and that the invariance properties of the mystery properties should hold with respect to *escale*.

I'll answer most of these questions in my next post. If you find *mystery_property2* you've rediscovered one of the deepest notions in mathematics. Something that appears in fields as diverse as combinatorics, algebraic geometry, algebraic topology, graph theory, group theory, geometric probability, fluid dynamics, and, of course, image processing.