Notebook

April 13, 2018

Contents

1 Defining the Flow field

1. With a convolution

   - So the plan with this one is to have a n by n window that goes around the first image, while for every n by n patch, we look at m by m area of the second area of the second image to find the best match (an area that minimizes the mean squared difference).
   - In order to do this I wanted to get the degrees of the movement in terms of a full 360 degrees.

   ```
   degrees :: (Ord a, Floating a, Eq a) ⇒ a → a → a
   degrees 0    0   = 0
   degrees rise 0
     | rise > 0     = 90
     | otherwise    = 180
   degrees rise run
     | rise ≥ 0 ∧ run > 0 = calc
     | rise > 0 ∧ run < 0 = calc + 90
     | rise ≤ 0 ∧ run < 0 = calc + 180
     | rise < 0 ∧ run > 0 = calc + 270
     where calc = abs $ atan (rise / run) * 180 / pi
   ```

     – here we only really have a few cases which are all self explanatory.

   - Now that we got that out of the way, we shall now talk about the main function

   ```
   convfn n m img1 img2 = R.fromFunction newSize f
     where
       sideSize        = n `div` 2
       edgeSize        = m * n + sideSize
       Z :. i :. j     = extent img1
       newSize         = ix2 (i `div` edgeSize) (j `div` edgeSize) -- this gives the boundary so w
       f (Z :. x :. y) = comp
         where
           centerX     = edgeSize + x * n
           centerY     = edgeSize + y * n
           fromMid ι κ = ix2 (centerX + ι) (centerY + κ)
           extractImg  = extract (fromMid 0 0) (ix2 n n)
           current     = R.computeUnboxedS $ extractImg img1
           sameSpotOn2 = R.computeUnboxedS $ extractImg img2
           comp | current == sameSpotOn2 = 0
                | otherwise              = uncurry degrees added
   ```

```
-- if the image moved at all then we have to add everything to a priority queue
added = (fromIntegral lowestI, fromIntegral lowestJ)
  where
    (lowestI, lowestJ) = peek $ foldr insertPQ empty allspots
    allspots           = (,) <$> [negate n*m .. n*m] <*> [negate n*m .. n*m] -- get al
    insertPQ (ι,κ)     = add diff (ι,κ)
      where
        diff = meanDiff current (extract (fromMid ι κ) (ix2 n n) img2)
```

- so we are going to take this function line by line in order to understand how it works
- `R.fromFunction newSize f`
  * This line is creating an array with size `newSize` with default values defined by a function f that takes coordinates and constructs the point
    · Note that every point is therefore independent and thus can be computed in parallel
- sideSize = n ‘div‘ 2
  * here edgeSize is how much an edge goes to either size from the middle, so if our n is 3, then its side size is 1
- edgeSize = m * n + sideSize
  * this part is a little trick, so I don't want the m by m window to go off the edge of the image, so instead of doing bounds checking for edge points, I just ignore the size of n m times and the raidus of n.
- i and j here are just the `extent` (size) of the first image, n and m are already taken, so i and j is the next best bet!
- newSize = ix2 (i ‘div‘ edgeSize) (j ‘div‘ edgeSize)
  * with all these constants defined, we can now define the size of the the output array. Here we just divide i and j by the edgeSize and make a new DIM2 shape
- f (Z :. x :. y) = comp
  * This is the function that will populate the array, since we now have the x and y coordinates, we can start to define what this function does
  * centerX centerY
    · these constants just compute where in the image we are
  * fromMid ι κ = ix2 (centerX + ι) (centerY + κ)
    · this is just an abstraction that adds a distance from the middle and generates a shape
  * extractImg = extract (fromMid 0 0) (ix2 n n)
    · this is yet again another image that takes an array and takes a n by n patch from the middle
  * current = R.computeUnboxedS $ extractImg img1
    · Now we finally have the n by n patch from the first image that we wish to test against
  * So instead of doing a lot of extra work, we define `sameSpotOn2` which grabs the same location in image2, as we can see that in
  * comp . . . = . . .
    · that if the current patch is the same as the same patch in image 2, then we just give back 0, otherwise we do `uncurry degrees added` which just calls degrees on added
  * added = (fromIntegral lowestI, fromIntegral lowestJ)
    · the end result of added is just the min over the m by m window but to see why, we must see the functions inside of added
  * allspots = (,) <$> [negate n*m .. n*m] <*> [negate n*m .. n*m]
    · Here we are doing a little fun trick, where we generate the range -n*m to n*m and then using map (<$>) to make the entire range a 1 argument function
    · (,) <$> [-2..2] : (Enum a, Num a) ⇒ [b → (a, b)]

- and then we use the applicative (one can think of the applicative `<*>` as the cross product that does any arbitrary functions instead of just ,) we get every combination of -n*m to n*m
- (,) <$> [-1..1] <*> [-1..1] = [(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)]

* insertPQ (ι,κ) = add diff (ι,κ)
  - `diff = meanDiff current (extract (fromMid ι κ) (ix2 n n) img2)`
  - I'm going to fold on the above range, but to do so, we must first make a function that takes a single element of the range and adds it to a priority queue. and to do this we just take the meanDiff (defined after this code block section) between the current patch and the new patch around the points ι and κ
  - After we get this diff we add the diff as the key with the value pair (ι,κ)
* Now that we got all this work out of the way we can make sense of
* (lowestI, lowestJ) = peek $ foldr insertPQ empty allspots
  - This function just folds over allspots (the generalized) cross product and starts with an initially empty priority queue with the insertPQ function, now that everything is added to the priority queue, we can now just peek at the queue and take what is lowest in value (lowest in the mean squared difference)

– With all these functions defined now f is defined and the entire function just works! and if one is still confused, try re-reading from top to bottom again, now that you know what each little function/constant means

- I did a few test cases for this, so I'll include 1 of them

```
computeUnboxedS (convfn 3 1 (fromListUnboxed (Z :. (10 :: Int) :. (10 :: Int)) [0..99])
                            (fromListUnboxed (Z :. (10 :: Int) :. (10 :: Int)) [0..99]))
```

  – and thankfully it gave me back the correct size of the output

    * AUnboxed ((Z :. 2) :. 2) [0.0,0.0,0.0,0.0]

- The only thing left to define is the meanDiff I used in convfn

```
meanDiff :: (Source r c, Source r2 c, Floating c) ⇒ Array r DIM2 c → Array r2 DIM2 c → c
meanDiff as = √ ∘ (/ fromIntegral (i * j)) ∘ sumAllS ∘ R.zipWith (\x y → abs (x^2 - y^2)) as
  where Z :. i :. j = R.extent as
```

  – meanDiff just takes 2 arrays and basically just runs the formula RMSE(a,b) = $\sqrt{\frac{\sum_{t=0}^{n-1}((a_t-b_t)^2)}{n}}$

2. With Gradient constraintp