Assignment 3

Basic Image Processing
Fall 2019

The problem

This problem is a real life scenario. People get paid for such programs! :)

Given many scans of Emmental cheese slices, we want to

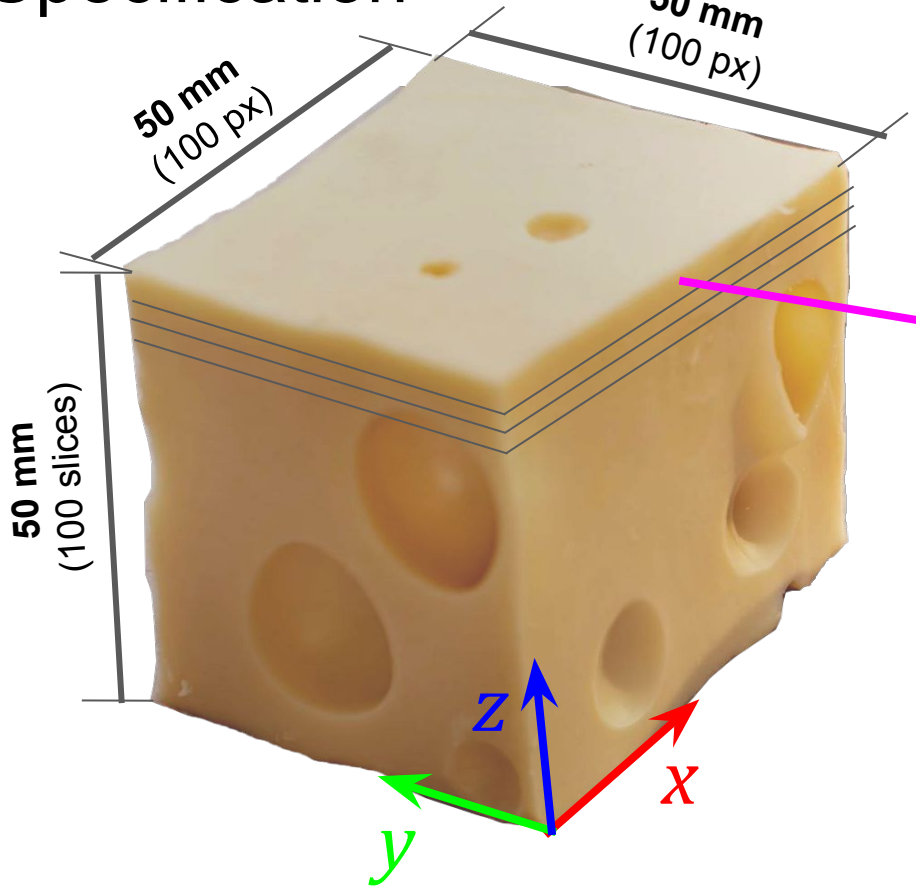
- segment the images (non-slice region, “cheese material”, hole in the slice),
- align the scans to reconstruct the 3D cheese block,
- tell the number of holes in the block,
- tell the volume of the holes.

Specification

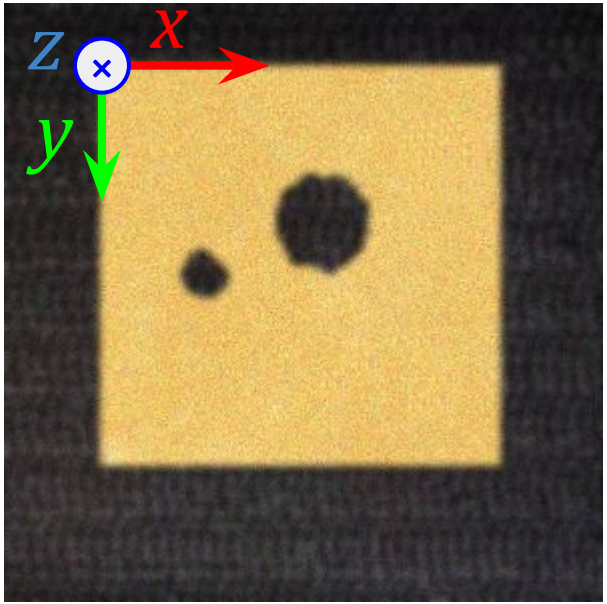
You can be sure that:

- Every scan contains exactly one slice.
- A ‘block’ is built up from 100 slices.
- The slices are put on a gray conveyor belt.
- The slices are squares and always the same size.
- The camera is fixed (i.e. viewport angle, distance is constant).
- The slices have different locations on the scans (consecutive slices are not aligned), however, there are no rotated or flipped slices.
- In one ‘block’ of data, the slice images are in order (1.png is the first slice, 2.png is the second, … 100.png is the last one, the top slice.)
- The contrast of the cheese images varies in a wide range.

Specification



Top slice
(z = 100)



Theory – Thresholding the images

To find the cheese slice on the background you should use thresholding on the RED channel (red is a dominant channel in the yellow cheese). Due to the varying contrast, a single hard-coded threshold value won’t work; adaptive threshold is needed.

We will use Otsu’s method to determine the optimal threshold for each scan separately.

Original scan

Red channel

Otsu threshold result

Theory – Otsu's method

Otsu's method automatically determines the optimal global threshold by minimizing the within-class (intra-class) variance of the black and white areas. Minimizing the within-class variance is the same as maximizing the between-class (inter-class) variance; which is defined as follows:

$$\sigma_b^2(k) = \sigma^2 - \sigma_w^2(k) = \omega_1(k)\omega_2(k)\Big(\mu_1(k) - \mu_2(k)\Big)^2$$

Where

$\sigma_b^2(k)$ between-class variance, $\omega_i(k)$ probability ,

$\sigma_w^2(k)$ within-class variance, $\mu_i(k)$ mean,

of the two classes separated by the threshold k.

Theory – Otsu's method

To calculate ω_i and μ_i the **normalized histogram** of the image is used. Normalized means that the sum of the histogram vector must be 1.

$$\omega_1(k) = \sum_{i=1}^{k} p_i \qquad \omega_2(k) = \sum_{i=k+1}^{N} p_i$$

$$\mu_1(k) = \frac{1}{\omega_1(k)} \sum_{i=1}^{k} i p_i \qquad \mu_2(k) = \frac{1}{\omega_2(k)} \sum_{i=k+1}^{N} i p_i$$

Where p_i is the i-th entry in the N-bin normalized histogram.

Theory – Otsu's method

The thresholding algorithm is the following:

1. The N-bin histogram of the grayscale input image is computed.
2. The histogram vector is divided by its sum (normalization).

3. For $k = 1, 2, \ldots, N$
   Compute and store $\sigma_b^2(k)$ using the formula:

$$\sigma_b^2(k) = \sigma^2 - \sigma_w^2(k) = \omega_1(k)\omega_2(k)\Big(\mu_1(k) - \mu_2(k)\Big)^2$$

4. After computing all the between-class variances, find k where $\sigma_b^2(k)$ is maximal.

The threshold level is the intensity level the k-th bin corresponds to.

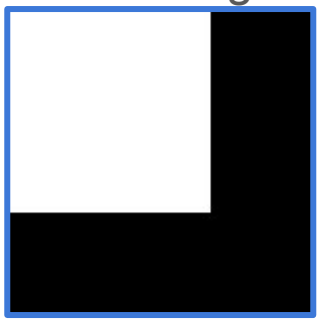
Theory – Slice alignment

Unfortunately the cheese slices are porous and therefore the center-of-mass method is not applicable to find the center of the slices. A different method is needed.

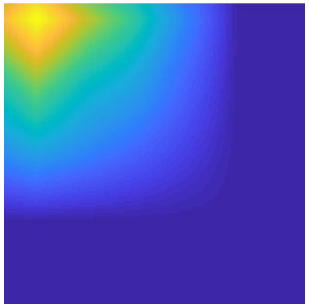
Let’s compute the cross-correlation of the thresholded slice image with an idealized artificial slice. The result of the cross-correlation is a ‘heatmap’, the location of the maximum tells the slice’s offset from the top-left corner.

Thresholded slice image

Artificial slice image



Cross-correlation heatmap



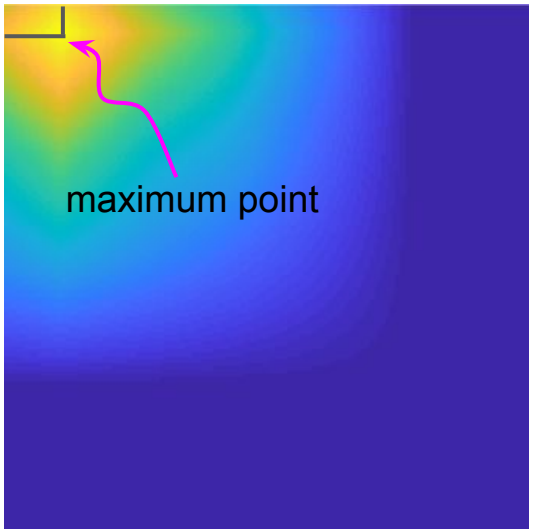
Theory – Slice alignment

Cross-correlation heatmap

column index

row index

maximum point

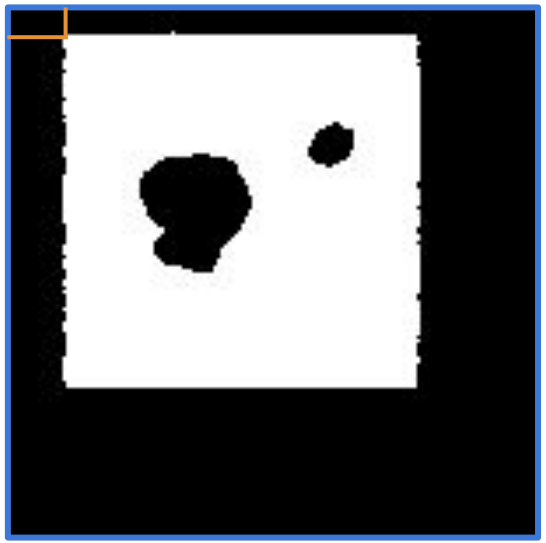


The row and column indices of the maximum correlation are the x and y offset values.

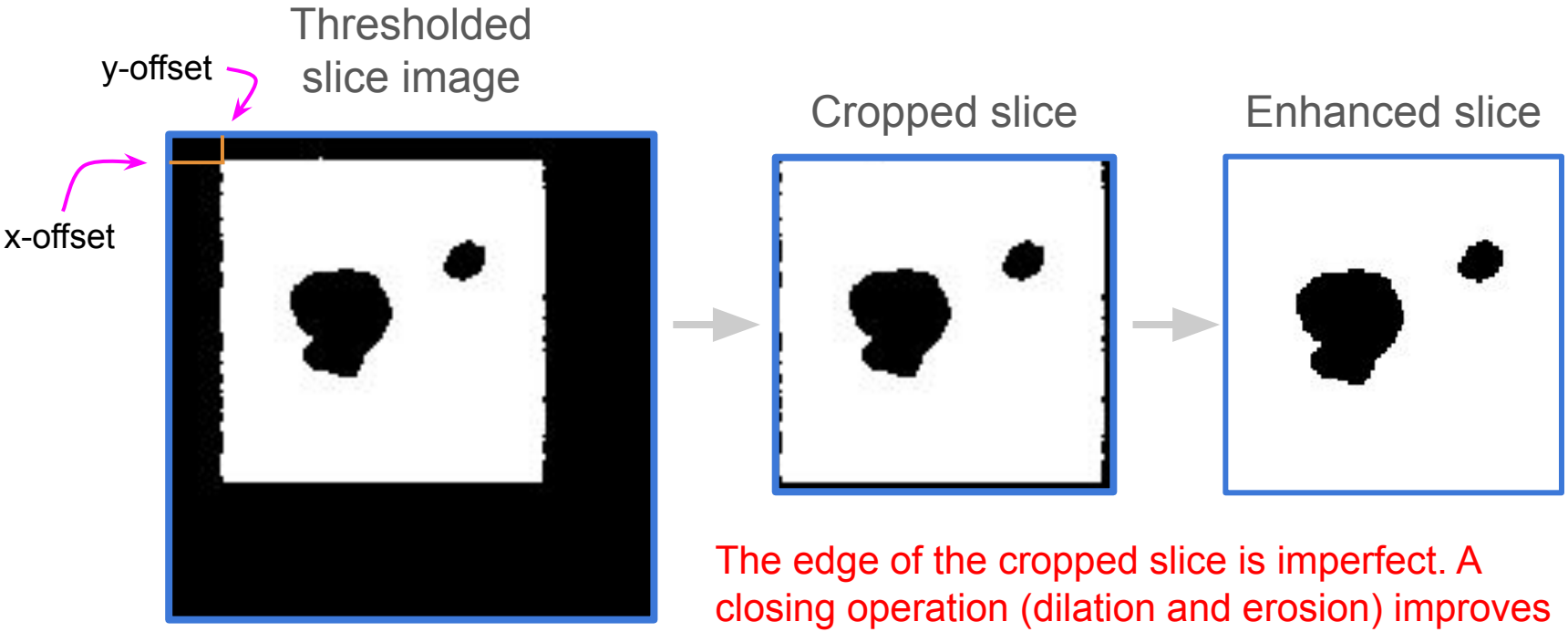
Thresholded slice image

y-offset

x-offset



Theory – Slice cropping and enhancement



The edge of the cropped slice is imperfect. A closing operation (dilation and erosion) improves the quality (and keeps the holes intact.)

Theory – Counting holes

Counting the holes and their volumes require an algorithm which scans the whole block of cheese voxel by voxel. (A voxel is the 3D version of a pixel; a “cube”.)

Our hole-filling algorithm uses a **queue** structure (a FIFO type list).

QUEUE:
A queue has two methods:

Enqueue: adds an item to the BACK of the queue.
Dequeue: returns the FIRST item and then deletes it from the queue.

Theory – Counting holes | Queue example

Let there be an empty queue:

```
HEAD [] TAIL
```

Let us enqueue the value ‘2’:

```
HEAD [2] TAIL
```

Let us also enqueue the values ‘4’ and then ‘8’:

```
HEAD [2, 4, 8] TAIL
```

If we call the dequeue operation, then the returned element will be the value ‘2’ and it will be removed from the queue:

```
HEAD [4, 8] TAIL
```

Then enqueue(1) yields

```
HEAD [4, 8, 1] TAIL
```

etc...

Theory – Counting holes

The hole-counting algorithm can perform two actions:

SCANNING

Go and check the next voxel. If this voxel is a hole, then increment the hole counter and enqueue the voxel's location into the FIFO.

FILLING

Dequeue an item from the FIFO, check if it is a 'hole' voxel. If yes then fill it with 'cheese' and increment the hole volume counter. Also, check the neighbors of this voxel; enqueue the location of the 'hole' type neighbors into the FIFO.

Switch between the modes depending on the FIFO. If the FIFO contains something then we do a FILLING action, or else when the FIFO is empty, then we're in SCANNING mode.

Please

download the ‘Assignment 3’ code package

from the

submission system

The maximum score of this assignment is

6 points

The points will be given in 0.25 point units.
(Meaning that you can get 0, 0.25, 0.5, 0.75, 1, 1.25 etc. points).

Exercise 1

Implement the function `otsu_th` in which:

- The inputs are the grayscale image `I` and the number of histogram bins `N`.
- The output is the thresholded binary image as a logical array.
- Threshold level is determined using Otsu's method.

Your algorithm should be built up like the following:

1. Compute the `N`-bin histogram of the grayscale image. Call the `histcounts` function with two arguments; the first one is the image, the second is the number of bins. Request two return values: the first will be the histogram vector, the second is the vector of intensity levels for each bin's edge.

```
[hist_vect, bin_edges] = histcounts(I(:), N)
```

Description continues on the next slide...

Exercise 1

2. Normalize the histogram vector (divide `hist_vect` by the sum of `hist_vect`).
3. For $k = 1, 2, \ldots, N$
   a. Compute `omega_1_k`
   b. Compute `omega_2_k`
   c. Compute `mu_1_k`
   d. Compute `mu_2_k`
   e. Compute `sigma_b_k` and store its value in a vector.
4. Get the arg. max of the `sigma_b` vector (`[~, arg_max_sigma_b] = max(...)`)
5. Get the appropriate bin edge; this will be the threshold level:

```
th = bin_edges(arg_max_sigma_b + 1)
```

6. Threshold the input image with the threshold `th` and return the result.

You can test your function by running `test1.m`.

Exercise 2

Implement the function `calc_offset` in which:

- The input is the grayscale image `I`.
- The output is a 2 element vector containing the y and x offsets (in this order).
- Offset values should be determined using cross-correlation.

Your algorithm should be built up like the following:

1. Create a matrix of zeros which has the same size as the input image.
2. Fill this matrix with ones starting from the location (1,1) to (100,100). This forms the artificial cheese image.
3. Compute the cross-correlation of the input and the artificial image:

```
CrossCorr = conv2(I, rot90(conj(artificial_image), 2));
```

Description continues on the next slide...

Exercise 2

4. Crop the result of the cross-correlation. We are only interested in this part:

```
CrossCorr(size(I,1):end, size(I,2):end)
```

5. Turn this `CrossCorr` matrix into a vector using `CrossCorr(:)` and find the arg. abs. max. of this vector (`[~, arg_max] = max(abs(...))`)
6. Translate the found linear index (`arg_max`) into two offset coordinates using the `ind2sub` function:

```
[off_y, off_x] = ind2sub(size(CrossCorr), arg_max)
```

7. Return the offset coordinates in a vector. The order is `[off_y, off_x]`.

You can test your function by running `test2.m`.

Exercise 3

Implement the function `crop_and_close` in which:

- The input is the binary image `I`.
- The output is the cropped and closed image.
- The returned image should be a 100×100 logical array.

Your algorithm should be built up like the following:

1. Compute the offset vector using your previously implemented function.
2. Crop a 100×100 size region from the input image starting from the location given by the offset coordinates. The slice should fit into this 100×100 matrix.
3. Create a ‘disk’ type structuring element with radius 1 (use `strel`)
4. First dilate then erode the cropped image (`imdilate` and `imerode` functions).
5. Return the result of the closing.

You can test your function by running `test3.m`.

Exercise 4

Implement the function `fifo_enqueue` in which:

- The inputs are the queue structure `FIFO` and the item to be added (`item`).
- The output is the updated queue structure (`FIFO`)

FIFO is a cell type, every element of FIFO is a cell containing an item. The new element should be inserted at the position `end+1` (use curly braces `{}` to index the cell).

You can test your function by running `test4.m`.

Exercise 5

Implement the function `fifo_dequeue` in which:

- The input is the queue structure `FIFO`.
- The outputs are the updated queue structure (`FIFO`) and the dequeued item.

FIFO is a cell type, every element of FIFO is a cell containing item.

Please check whether FIFO is empty or not. In the former case you should return an empty vector: `[]`

If FIFO is not empty, fetch its first item (use curly braces `{}` to index the cell); then shift the whole content of FIFO. (Item at position p will be moved to position $p-1$, and the length of the FIFO is decreased by 1).

You can test your function by running `test5.m`.

Exercise 6

Implement the function `process_the_block_of_cheese` in which:

- The input is the cheese block `cube`.
- The output is a vector containing the volumes of the found holes.

If there were 3 holes in the block of cheese then the output of this function should be a 3-element vector containing the volume of each hole. E.g. `[154, 222, 63]`

Check the upcoming slides for more details!

Exercise 6

Your algorithm should be built up like the following:

1. Initialize everything:
    a. Create a variable in which you will store the number of holes you found. Name it `HoleCount` and set its value to 0.
    b. Create an empty vector in which you will store the volumes of the holes. Name it `HoleVolumes`.
    c. Create a variable in which you'll store the index of the last visited voxel. Name it `LastVisited` and set its value to 0.
    d. Create an empty queue as a 0×1 cell. Name it `FIFO`.

Description continues on the next slide...

Exercise 6

2. In an infinite while loop:
    a. Check the number of elements in FIFO. If its length is zero, do a SCANNING step, otherwise do a FILLING step.

The SCANNING step is built up like this:

I. Increment the value of `LastVisited`.
II. If the value of `LastVisited` is greater than the number of voxels in the cube then exit from the infinite loop.
III. Use `ind2sub` to translate the linear index `LastVisited` into `[py, px, pz]` coordinates of the `cube`.
IV. Check the voxel's value at `(py, px, pz)`. If it is 1 (which means 'hole') then enqueue the vector `[py, px, pz]` into the `FIFO`; increment the `HoleCount` counter and set `HoleVolumes(HoleCount)` to zero.

Description continues on the next slide...

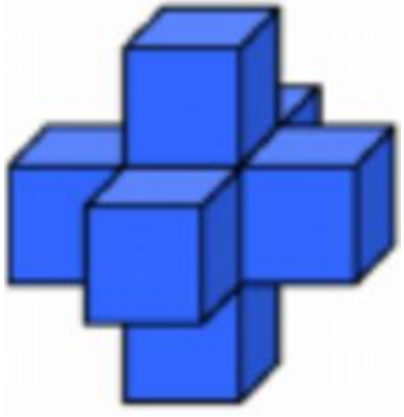
Exercise 6

The FILLING step is built up like this:

I. Dequeue the next element from FIFO. This item is a vector of three coordinates `(py, px, pz)`.
II. Check the voxel's value at this location. If it is 0 (which means it is not a hole) then skip the rest of this step; use the `continue` command to jump to the next iteration of the infinite outer loop.
III. If the voxel is 1 ('hole') then increment the value of `HoleVolumes` at `HoleCount` by one; and set this voxel of cube to 0 (fill the voxel of the hole).
IV. Check all neighbors of the current voxel. If a neighbor is a 'hole' (1) then enqueue its coordinates into the FIFO. If a neighbor is out of the block (i.e. negative or zero index; index > 100) or the voxel's value is 0 then leave it alone.

Description continues on the next slide...

Exercise 6



The neighborhood of a voxel (in the middle) looks like this:
(So a voxel has 6 neighbors: above, below, north, east, south, and west.)

After you have scanned all voxels and filled all holes
you should return the vector `HoleVolumes`.

Please feel free to use the functions `fifo_print` and `plot_cube` for debugging and to better understand what is going on in the code.

When you are ready with the implementation of this function, run `test6.m`.

Finally, after completing all the exercises please run the `batch_processor.m` script to process 5 different cheese samples.

A good result...

```
Results on block #1
    Holes found:  20 / 10
   The error is:  3901 voxels (0.044942%)

Results on block #2
    Holes found:  18 / 13
   The error is:  26286 voxels (0.25571%)

Results on block #3
    Holes found:  9 / 2
   The error is:  3662 voxels (0.02851%)

Results on block #4
    Holes found:  44 / 41
   The error is:  5041 voxels (0.13669%)

Results on block #5
    Holes found:  25 / 16
   The error is:  5654 voxels (0.058132%)
```

The script will print the number of holes found by your script *slash* the hole count of the ground truth.

Also, the script computes an error measure (the relative hole-wise difference of your volume estimation and the ground truth volume vector).

If you observe error values less than 0.3 % then your solution is very good.

THE END