Mateusz Moderhak. 05.01.2026

Task 1.1 Edge detection

1. Which state-of-the-art approach would you choose to robustly detect relevant edges? And why?

Because of the complexity of microscopic images and the physical phenomena affecting them (such as reflections, specular highlights and different surface textures), I would evaluate a set of different approaches rather than selecting a single one. I expect that, more advanced, convolutional neural-network-based edge detectors will outperform classical methods such as Canny or Sobel, mainly due to their robustness to noise, illumination variations and their ability to suppress non-relevant texture edges.

In particular, I would consider:

- **HED – Holistically Nested Edge Detection**
  (available in OpenCV, TensorFlow / Keras, PyTorch)
  Well-established, stable and widely used, with good multi-scale edge representation.
- **BDCN – Bi-Directional Cascade Network**
  (available in PyTorch / Caffe)
  Known for improved localization accuracy and better handling of challenging edge structures.
- **DexiNed**
  A more recent approach with promising benchmarks, though pretrained models are less commonly available; if necessary, I would consider training or fine-tuning it myself.

My plan would be to first benchmark these CNN-based methods as well as classic methods (Canny, Sobel, etc. to establish the baseline, mainly due to my personal curiosity) on representative microscopic data. If performance is insufficient, I would perform transfer learning and fine-tuning using domain-specific microscopic images to adapt the CNN models to the real operating conditions of the application. This should further improve robustness, edge continuity and relevance of detected edges for metrology tasks.

2. Sketch how you would implement your solution. How would you validate your approach in practice?

Implementation stages:

- **Analysis**
  First, I would analyze the main requirements, such as image resolution, expected FPS, and acceptable processing time. I would also investigate the hardware (CPU/GPU), how much memory is available, and what computing resources can be used. Based on this, I would choose the right algorithms and optimizations to make the system efficient. Then I would test it in real conditions to confirm that it meets these requirements.

- **Dataset and ground truth preparation**
  I would gather sufficiently large dataset of microscopic PCB images with reliable ground truth or reference measurements. Ideally, the dataset should cover different materials and textures, reflective solder joints and metallic surfaces, varying illumination conditions, and typical scenarios encountered in PCB inspection.
- **CNN-based edge detection**
  I would evaluate pretrained edge detectors such as HED, BDCN, or DexiNed. If needed transfer learning or fine-tuning with company-specific data would adapt convolutional filters to domain-specific edges and textures. Tiles would be processed independently and then merged to reconstruct the full edge map.
- **Post processing of edge detection results**
  Thresholding (probability-based) would convert edge probabilities to binary edges if needed. Or Morphological refinement, such as closing or skeletonization, would ensure precision for metrology requirements.
- **Validation strategy: edge detection quality and runtime performance**
  - Edge quality would be evaluated using ground truth dataset with Precision, Recall, F1-score, and PR curves. Robustness would be tested against noise, illumination variations, reflections, and partial visibility. If possible, in simulated and real test cases.
  - Performance validation would measure processing times and memory usage. Preprocessing and postprocessing could be tuned to optimize trade-offs between accuracy and speed.

3. Please also address how you envision the deployment into a product software that is written in C++.

If needed, I would first train, test, and experiment in Python (Keras / TensorFlow / PyTorch). The trained model can then be exported to ONNX for use in C++. Preprocessing and postprocessing steps would be reimplemented in C++ using e.g.: OpenCV, ensuring compatibility and efficient execution within the product pipeline. Optional FP16 or INT8 quantization could accelerate GPU execution.

4. How do you achieve best speed and leverage modern CPU/GPU hardware?

To improve speed and efficiently use CPU/GPU resources, I would consider:

- In my opinion, the most important part: proper pipeline configuration batching / prefetching / caching / paralel processing: Implement an efficient input pipeline to feed tiles or batches of tiles to the network without stalling the GPU or CPU. Pre- and post-processing should run in parallel, ensuring that the GPU always receives ready-to-infer images.
- Lightweight feature extractors: replace heavier backbones like VGG16 or Inception with lighter networks such as MobileNetV2/V3. This may require model assembly and training from scratch using only pretrained backbone.

- Precision reduction / quantization: convert the model from FP32 to FP16 or INT8 to reduce memory usage and accelerate inference.
- Model pruning: remove redundant weights to decrease computational complexity.
- Knowledge distillation: train a smaller, faster student model using a large teacher model.

5. Where do you see challenges for your approach and how would you address those?

Dataset preparation

CNN networks require a significant amount of labeled data. It can be challenging to obtain a sufficient amount of data. However, fine-tuning an existing model with data augmentation may significantly reduce the amount of data needed.

Computational complexity / runtime

Heavy CNN–based edge detectors may be too slow for real-time or high-throughput industrial environments. Apart from the optimizations discussed in point 4 (precision reduction, pruning, lightweight backbones, tiling), additional optimization of pre- and post- processing is required to ensure deterministic, high-speed performance.

Generalization to PCB microscope images

Out-of-the-box CNN edge detectors may not be optimal for this specific domain. Robust performance may require:

- creation of a representative dataset,
- transfer learning and fine-tuning,
- experimentation with hyperparameters, data augmentation and regularization,
- a reliable evaluation protocol with well-defined metrics.

Challenging imaging conditions

Highly reflective solder joints, specular highlights, noise and varying lighting may still degrade performance. In such cases, algorithmic robustness might not be enough and improvements in the image acquisition chain may be needed, e.g. use of polarization filters, multi-exposure strategies, or HDR / dynamic-range-enhancement techniques to stabilize image contrast and reduce extreme reflections.

Task 1.2: Geometry estimation

1. How do you approach the fitting of circles based on these edge maps? And why?

A classical approach for fitting circles from edge probability maps is to use the Hough Transform. The original form is computationally expensive, since it maps a 2D image into a 3D parameter space (center coordinates+ radius). In this application, where circles have significantly different sizes and some are only partially visible, a multi-scale, gradient-assisted Hough Transform seems appropriate.

As an alternative approach, a CNN model trained on synthetic data could predict approximate circle parameters (x, y, r) or its bounding box. These estimates could then be refined using a localized Hough method or least-squares fitting to achieve high accuracy. The CNN model would either process raw images or edge probability maps.

2. What is the best way to enable a fit that is robust to spurious edges?

In my opinion the best way to enable fit to spurious edges is edge map preprocessing:

1. histogram based preprocesing – to threshold out the low probability pixels, the images characterize by bi-modal histograms where hi probability = edge; low probability = noise
2. apply spatial and morphological filtering to eliminate or mask out spurious edges.
3. Gradient-based line masking: to reduce the number of points beeing processed by the Hough Transform

CNN aproach: The influence of spurious edges could also be minimized by using a circle detector to identify existing circles in the image. YOLO or Mask R-CNN could be used to predict bounding boxes around circles. The detected circles could then be preprocessed and processed independently, reducing the level of noise and spurious edges. For instance, in 14.png this would eliminate lot of ‘background edge noise’.

3. What are the runtime and memory requirements for your approach? What are the limitations?

My approach uses the gradient-based Hough Transform, where the complexity can be estimated as $O(N \times R)$, with N being the number of edge points and R the number of possible radii. It allocates an accumulator matrix of size (image width, image height, number of radii).

Limitations:

- Hough methods can be slow and memory-consuming for large images, wide radius ranges, and images with many edge points.
- A different approach may be needed for detecting circles with centers outside the image – otherwise it would severely increase the search space.

4. How would you address the memory limitations?

- The provided images suggest creating different scenarios for different image types. This would significantly reduce the search space and memory requirements. For each scenario, different min and max radii, as well as accumulator thresholds, can be set. This way, we won't have to search, for example, for large circles in an image where only small ones are present. i.e. The scenario for big circles with centers outside the image would involve image resizing and zero-padding before initiating search.
- Another way is to decimate the accumulator space. Using the dp parameter = 2.0 reduces each dimension by half, leaving the accumulator 4× smaller than the original.
- Finally, using unsigned 8-bit precision for processing can further reduce memory usage.

5. How can you optimize your approach with respect to runtime (e.g. parallel programming paradigms)?

I would partition the image and analyze each part separately. The edge points from each part would vote into the accumulator.

For the scoring step, I would parallelize by partitioning the circle set beeing scored. Each core or process would compute the scores for its subset of circles.

Parallelization would also be possible if a circle detector were used to select existing circles in the image. YOLO or Mask R-CNN could be used to predict bounding boxes around circles. The detected circles could then be processed independently.

6. How do you ensure the correctness of your implementation in practice? Which tool set are you using?

This particular problem is quite easy to test using synthetic data, since it uses edge probabilities instead of grayscale images. I would generate circles according to the circle equation and prepare synthetic edge probability maps with various backgrounds, image features, noise, and distortions that resemble real-life scenarios – essentially "test it where you use it."

To evaluate correctness, I would use quantitative metrics such as MSE of center estimation, MSE of radius estimation, and precision/recall of detected centers to assess both localization accuracy and completeness of detection.

The next level would be testing on real data with ground truth to provide additional validation of the implementation.

For production, I would also write unit tests for individual components (edge map processing, Hough voting, CNN outputs) and integration tests for the full pipeline.

Toolset: OpenCV, NumPy, scikit-image, TensorFlow data.Dataset – for data augmentation and for dataset handling.

Task 2 Parallelization of integral image computation

1. Please sketch the implementation of a single-core version of integral image computation as described above in C++. What is the computational effort for the single-core computation of $I_\Sigma(x, y)$ (two-dimensional case)?

```
for (int y = 0; y < height; y++) {
   int row_sum = 0;
   for (int x = 0; x < width; x++) {
      row_sum += image[y][x];
      integral[y][x] = row_sum;
      if (y > 0) {
         integral[y][x] += integral[y-1][x];
      }
   }
}
```

The computional effort is O(height x width)

2. Please extend your sketch to a multi-core CPU solution. How is the computational effort for this multi-core CPU approach?

We divide the image into N horizontal parts, where N is the number of available CPU cores. Each core computes the partial integral image for its assigned region independently (using the same single-core algorithm as in point 1).

After all threads finish, we combine the partial results. For each region (except the first one), we add the values from the last row of the previous region's integral image to every row in the current region. This correction step ensures that the final result is a valid global integral image.

```
// Assume we have 3 stripes processed in 3 threads: IntegralA, IntegralB, IntegralC

// 1. Correct sums between stripes sequentially

for (int x = 0; x < width; x++) {

   for (int y = 0; y < heightB; y++) {

      IntegralB[y][x] += IntegralA[heightA-1][x];  // add last row of A
```

```
    }
}


for (int x = 0; x < width; x++) {
    for (int y = 0; y < heightC; y++) {
        IntegralC[y][x] += IntegralB[heightB-1][x];  // add last row of B (already includes A)
    }
}
// 2. Combine all stripes into the final global integral image
std::vector<std::vector<int>> Integral;
Integral.insert(Integral.end(), IntegralA.begin(), IntegralA.end());
Integral.insert(Integral.end(), IntegralB.begin(), IntegralB.end());
Integral.insert(Integral.end(), IntegralC.begin(), IntegralC.end());
```

3. What would be the expected speedup for the multi-core implementation?

The single core O(width x height), the multicore would be O (width x hieght / N_cores)- ideally. In practice, the actual speedup is usually lower due to thread creation overhead and the sequential integral image correction between stripes.

4. How would you parallelize the computation on a GPU e.g. using CUDA?

I would estimate that the integral image computation can be parallelized on a GPU in a way similar to how CNNs/DNNs are parallelized. Each pixel or small block of the image can be processed independently in parallel, much like pixels in a convolution operation. Using this approach, the GPU can process many pixels at the same time, achieving significant speedup.

5. What would be the speedup for the GPU implementation compared to the single-core implementation?

I would estimate that the expected speedup on GPU would be similar to what is typically observed when training or running CNNs/DNNs. For large-scale computations, this could be roughly 10–100× faster compared to a single-core CPU. The actual speedup depends on factors like data size, GPU utilization, and memory bandwidth.

6. How would you extend the method above for the computation of the integral image for a three-dimensional volume (3D image)? Is there anything you need to pay attention to - in particular for large volumes?

```
// Compute 3D integral image
for (int z = 0; z < depth; z++) {
    for (int y = 0; y < height; y++) {
        int row_sum = 0;  // cumulative sum along X axis
        for (int x = 0; x < width; x++) {
            row_sum += volume[z][y][x];          // add current voxel
            integral[z][y][x] = row_sum;

            if (y > 0) {
                integral[z][y][x] += integral[z][y-1][x];  // add sum from previous row (Y)
            }
            if (z > 0) {
                integral[z][y][x] += integral[z-1][y][x];  // add sum from previous layer (Z)
            }
            if (y > 0 && z > 0) {
                integral[z][y][x] -= integral[z-1][y-1][x]; // correct double-counted voxels
            }
        }
    }
}
```

Points to pay attention to for large volumes:

Memory usage: The integral volume requires the same amount of memory as the original volume, which can be very large for high-resolution 3D images.

Number of operations: Computation scales as O(depth × height × width), so processing large volumes can be expensive.

Cache and memory access: Access patterns should be as contiguous as possible to avoid cache misses and memory bottlenecks.

Parallelization: To speed up computation, you can split the volume into slices or blocks and compute partial sums in parallel (multi-core CPU or GPU), but you must handle sequential corrections along each axis to maintain correctness.