Received: 28 November 2019 | Revised: 30 April 2020 | Accepted: 10 May 2020

DOI: 10.1002/qre.2689

SPECIAL ISSUE ARTICLE

WILEY

Detection of circlelike overlapping objects in thermal spray images

Dominik Kirchhoff[1] | **Sonja Kuhnt**[1] | **Louise Bloch**[1] | **Christine H. Müller**[2]

[1]Department of Computer Science, Dortmund University of Applied Sciences and Arts, Dortmund, Germany

[2]Faculty of Statistics, TU Dortmund University, Dortmund, Germany

Correspondence
Sonja Kuhnt, Dortmund University of Applied Sciences and Arts, Dortmund, Germany.
Email: sonja.kuhnt@fh-dortmund.de

Funding information
Deutsche Forschungsgemeinschaft, Grant/Award Numbers: SFB823, Project B1/B5

Abstract

In this paper, we present a new algorithm for the detection of distorted and overlapping circlelike objects in noisy grayscale images. Its main step is an edge detection using rotated difference kernel estimators. To the resulting estimated edge points, circles are fitted in an iterative manner using a circular clustering algorithm. A new measure of similarity can assess the performance of algorithms for the detection of circlelike objects, even if the number of detected circles does not coincide with the number of true circles.

We apply the algorithm to scanning electron microscope images of a high-velocity oxygen fuel (HVOF) spray process, which is a popular coating technique. There, a metal powder is fed into a jet, gets accelerated and heated up by means of a mixture of oxygen and fuel, and finally deposits as coating upon a substrate. If the process is stopped before a continuous layer is formed, the molten metal powder solidifies in form of small, almost circular so-called splats, which vary with regard to their shape, size, and structure and can overlap each other. As these properties are challenging for existing image processing algorithms, engineers analyze splat images manually up to now.

We further compare our new algorithm with a baseline approach that uses the Laplacian of Gaussian blob detection. It turns out that our algorithm performs better on a set of test images of round, spattered, and overlapping circles.

KEYWORDS

circular clustering, image processing, object detection, thermal spraying

1 | INTRODUCTION

Here, we introduce a new approach to the detection of circles in noisy grayscale images. The circles are allowed to have (variously) distorted edges, different radii, and varying overlaps. Our work is motivated by an application in thermal spraying, where splats of metal composites coat a surface. In some images of the coating, many splats overlap each other, so that they form a single cluster. In this case, a human tends to mark the most clear splat first, then the second most clear, and so on until the whole cluster is dealt with. An automatic image detection is wanted that emulates this time-consuming human way of handling.

Splats are usually of roughly circular shape. Our algorithm searches—unlike common approaches that are based on the Hough transform[1]—iteratively for circles and takes the goodness of fit of the found circles into account. The detection of circles is also covered by some approaches that concentrate on shapes with radial symmetry.[2,3]

A key element of any such algorithm is the detection of edge points in noisy images, which is an ongoing field of research.[4-7] We base our work on the idea of rotational difference kernel estimators (RDKE) from Qiu,[8,9] further improved by Garlipp and Müller.[10] Chu et al.[11] combine these results with a kernel smoothing method to estimate jump location curves.

The remainder of this paper is organized as follows: In Section 2, we introduce the splat images mentioned above and the challenges in automatic splat detection. Section 3 reviews key methods needed to develop our algorithm. Our new algorithm is introduced step by step in Section 4. We also give some details on the implementation in R[12] and outline the tuning of the algorithm's parameters. We apply the algorithm to splat images and discuss the results in Section 5. In Section 6, we apply our algorithm to a set of 300 test images with different properties. In order to compare our algorithm, we also evaluate a generic approach using the Laplacian of Gaussian blob detection on the same test images. Finally, Section 7 contains a summary.

2 | SPLAT IMAGES OF THERMALLY SPRAYED SURFACES

The aim of every thermal spray process is to obtain a durable and resistant coating with optimal properties. In high-velocity oxygen fuel (HVOF) processes, the quality of the coating depends on a number of parameters, for example, the amount of kerosene and the velocity of the powder feeder. Roughly speaking, the more resources are used in the process, the better the result. Due to the expensiveness of some materials, however, the engineer has to find a trade-off between the goodness of the resulting coating and the maximum acceptable cost.

The coating consists of many so-called splats—that is, droplets of molten powder that hit the substrate and solidify there; see Figure 1 for an example. Tillmann et al.[13] introduce a beam-shutter device that makes it possible to expose the specimen only for a short moment of time to the spray jet. Thereby, only a few splats emerge, which can then be counted and analyzed. After the spraying process, scanning electron microscope (SEM) images are taken of the sample. So far, some engineer had to analyze these images manually. Counting and classifying splats from hundreds of images, however, is not only a time-consuming and quite annoying task, it may also lead to inconsistencies if two or more people share this work, as the classification of splats is not always clear and different people may have different opinions about the same cluster of overlapping splats. The aim of this contribution is to develop an algorithm that identifies splats automatically in order to count them and to analyze their positions and sizes.

The SEM images we work with typically are 8 bit images. That is, the pixel intensities may have values from 0 ($\hat{=}$ black) to $2^8 - 1 = 255$ ($\hat{=}$ white), so there are $2^8 = 256$ possible values.

Figure 2 is an example of an SEM image of some splats on a polished substrate. From this image, we can make some observations: First, splats are lighter than the background, but not everything that has a higher intensity is a splat—there are many small speckles in the background. Also, not everything with a darker shade of gray belongs to the background, as there are, for example, splats with a rough texture, which leads to dark pixels within the foreground.

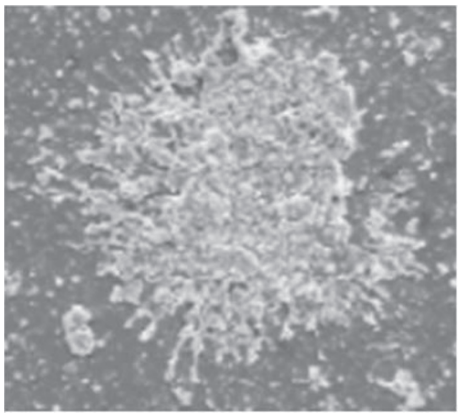


FIGURE 1 An example of a splat on a polished substrate

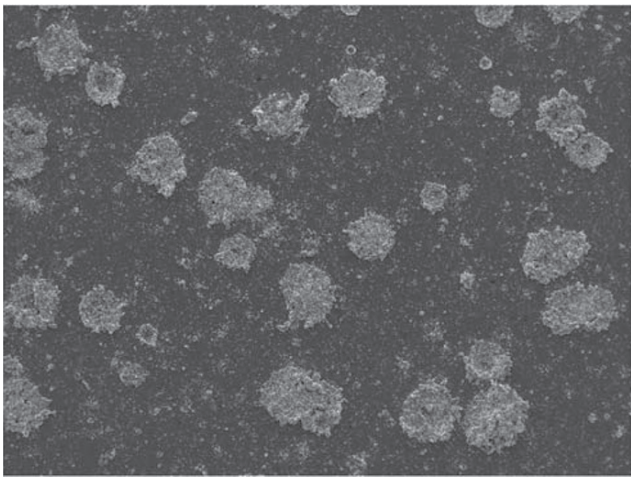


FIGURE 2 Scanning electron microscope image of splats on a polished substrate

Second, splats are about circular, but some are distorted. Third, they do not have all the same size. Last, some splats overlap each other. All these challenges need to be met by an automatic splat detection method.

3 | MEDIAN FILTERING, EDGE DETECTION, CIRCULAR CLUSTERING, AND LAPLACIAN OF GAUSSIAN

In this section, we provide a review of the key elements of our new procedure. These are, essentially, median filtering for noise reduction, edge detection as an important step to identify shapes, and circular clustering to go beyond the sole detection of edges by fitting circles to found edge points.

3.1 | Median filtering

Median filtering is a technique for noise reduction in images while preserving the edges at the same time. This is achieved by replacing each pixel intensity with the median of a certain amount of surrounding pixel values. The pattern of the observed surrounding area is called the **window**. Different window patterns are possible; here, we are going to use a square pattern. The window slides over the entire image, entry by entry, whereby every pixel is the center of the window once. To ensure that there are always enough values to "fill" the window near the boundaries, the image is padded with the closest boundary pixel values. This is simply done by creating a frame surrounding the image that is half as many pixels wide as the window size. Then, the pixel intensities directly at the left and right boundary of the image are copied to the adjacent parts of the frame. After that, the same is done with the pixel intensities at the top and bottom boundaries, including those that have already been copied into the frame.

3.2 | Edge detection

Edge detection aims at finding all pixels close to boundaries of different regions, for example, in our case splats in an image. Such edges can be understood as discontinuities or jumps in a two-dimensional intensity function. We employ a method proposed by Müller and Garlipp[14] and Garlipp and Müller.[10] Based on robust RDKE, each pixel is classified as being close to an edge or not. We sketch the general idea first before giving details. A window is laid over each pixel and rotated by an angle θ (see Figure 3). The window is partitioned into an upper and a lower window in the direction of the angle. Location estimates are derived for the expected value of pixel intensity at the pixel from both windows.

Formally, let z_{ij}, the pixel intensities, be observations of independent random variables Z_{ij} with $E(Z_{ij}) = m(x_{ij}) \in \mathbb{R}$ at equidistant design points (pixels) $x_{ij} = (i,j)^T$ with $1 \le i \le n_1$ and $1 \le j \le n_2$, where n_1 and n_2 are the number of pixels in the two directions. We denote the current pixel of interest by x, which is the center of the rotated window. The rotation by an angle θ is conducted with the rotation matrix $A_\theta = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$.

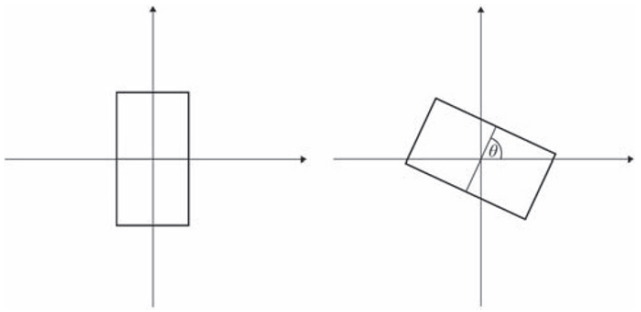


FIGURE 3 Rotation of windows in the edge detection procedure

Estimators $\widehat{m}_k(\theta, x)$ for the two parts of the window are defined as zeros of an objective function $\mathcal{H}_k(z; \theta, x)$, $k \in \{1, 2\}$. The objective function contains two major elements: firstly, the score function $\psi : \mathbb{R} \to \mathbb{R}$ of an M-estimator, secondly, rotated asymmetric Gasser–Müller weights, which depend on the distance of observations to the center of the window. The weight function is based on two one-sided, continuous kernel functions K_1 and K_2, which fulfill the following conditions:

- $K_1(x) = 0$ for $x \notin \left[-\frac{1}{2}, \frac{1}{2}\right] \times [-1, 0]$,
- $K_2(x) = 0$ for $x \notin \left[-\frac{1}{2}, \frac{1}{2}\right] \times [0, 1]$,
- $\int_{[-1,1]^2} K_k(x)\mathrm{d}x = 1$, $k \in \{1,2\}$,
- $K_1(x) \geq 0$, $K_2(x) \geq 0$.

The weight function itself is then defined by

$$\alpha_{ij}^{(k)}(\theta, x) := \int_{\Delta_{ij}} K_k(\theta, u - x)\mathrm{d}u,$$

where Δ_{ij} is a well-chosen neighborhood of x_{ij}. For example, if x_{ij} is the centroid of a pixel, then $\Delta_{ij} = (i-0.5, i+0.5] \times (j - 0.5, j+0.5]$. Furthermore, $K_k(\theta, x) = \frac{1}{h_1 h_2} K_k(H^{-1} A_\theta x)$, where H is a diagonal matrix with diagonal elements h_1 and h_2. Typically used kernel functions for K_k are, for example, the Gaussian kernel or the rectangular kernel. One can also use $\alpha_{ij}^{(k)}(\theta, x) = K_k(\theta, (i, j) - x)$. Then all pixel positions within the window, even those close to the boundary, are equally weighted when the rectangular kernel is used.

The rotated asymmetric M-kernel estimators $\widehat{m}_k(\theta, x)$ of $E(Z) = m(x)$ are finally defined by

$$\widehat{m}_k(\theta, x) \in \{z \in \mathbb{R} : \mathcal{H}_k(z; \theta, x) = 0\}$$

with

$$\mathcal{H}_k(z; \theta, x) := \sum_{i=1}^{n_1} \sum_{i=1}^{n_2} \alpha_{ij}^{(k)}(\theta, x) \psi(Z_{ij} - z).$$

In general, any score function $\psi : \mathbb{R} \to \mathbb{R}$ of an M-estimator can be chosen for a rotated asymmetric M-kernel estimator. One could, for example, use the score functions $\psi(z) = z$ for a rotated difference kernel mean estimator or $\psi(z) = sign(z)$ for a rotated difference kernel median where $sign(z)$ is the sign function. Here, we consider the sign function and the derivative of the density of the standard normal distribution for the score function ψ. Both score functions lead to outlier robust estimators because the score function is bounded.[15]

Choosing $\alpha_{ij}^{(k)}(\theta, x) = K_k(\theta, (i,j) - x)$ together with the rectangular kernel for K_k and the sign function as score function results in the calculation of a simple median in the rotating windows.

The jump height at the pixel x can be estimated by the difference between the estimators from both windows for each angle θ:

$$M(\theta, x) := \widehat{m}_2(\theta, x) - \widehat{m}_1(\theta, x).$$

Overall, the jump height can be estimated by

$$\widehat{C}(x) = \left| M\left(\widehat{\theta}(x), x\right) \right| \text{ with } \widehat{\theta}(x) \in \arg\max_{\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]} |M(\theta, x)|.$$

We classify the pixel x as close to an edge if the estimated jump height exceeds some threshold. Alternatively, a multiple test on the global hypothesis of equal expectations in both windows for every angle as given by Garlipp and Müller[10,16] can be employed.

Note that only points x of the image are used, which allow that all rotated windows lie completely in the window.

3.3 | Circular clustering

Garlipp and Müller[16] propose a method for circular clustering based on a set of N edge points $x_1, ..., x_N$. To fit so-called regression circles to these edge points, they[16] consider the objective function

$$H(a, r) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{s} \rho\left(\frac{||x_i - a|| - r}{s} \right),$$

where ρ is a redescending score function—typically a unimodal density function like the standard normal distribution[14]—that is, the closer the distance between an edge point x_i and a circle with center a and radius r, the higher the value of ρ for a fixed scale parameter s. In this sense, $H(a, r)$ measures the mean "closeness" between a set of N edge points and a circle with parameters a and r. Thus, the local maxima of the objective function H can be used to estimate the parameters a and r of different regression circles.

The local maxima are computed by using the Newton–Raphson method. These are searched iteratively using randomly chosen starting circles until a predefined number of local maxima are found. Where appropriate, a circle may be excluded if its radius leaves a specified interval during the optimization.

In the implementation of the circular clustering we use, ρ is determined as the negative of the density of the standard normal distribution and consequently the objective function H is minimized.

3.4 | Laplacian of Gaussian

The Laplacian of Gaussian (LoG) method is a common method to detect blobs in an image,[17] which is based on the LoG operator

$$\nabla^2 G(x, y) = \frac{x^2 + y^2 - 2\sigma^2}{\pi \sigma^4} \exp\left(-\frac{x^2 + y^2}{2\sigma^2} \right).$$

This operator derives from the execution of two successive convolutions. First, the input image f is convolved by a Gaussian kernel

$$G(x, y; \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2 + y^2}{2\sigma^2} \right)$$

to smooth the image, where σ is the standard deviation of the Gaussian kernel. Second, the Laplacian operator

$$\nabla^2 = \frac{\partial f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

is applied to the image f to highlight regions with a steep intensity gradient. The output signal of the LoG operator is particularly high for blobs of radius $r = \sqrt{2\sigma}$. Using a single Gaussian correspondingly allows the detection of all blobs with a specific radius. In many applications, blobs of different radii should be detected from an image. Therefore, LoG is usually applied multiple times using different scales (σ) to all blobs as it is described in Lindeberg.[18] Then, scale-space optimization is done to find the best blobs.

4 | DETECTION OF OVERLAPPING CIRCLELIKE SHAPES

In this section, we present all steps of our new algorithm. A measure of similarity is developed to judge the performance of the image detection, which can also be used to tune parameters of the algorithm. We conclude this section with some words on the implementation of the algorithm.

4.1 | The new algorithm

To illustrate our algorithm, we first create a synthetic demonstration image with almost circularly shaped objects. To derive edge points of each of these objects, we generate a set of I equidistant angles $\varphi_1, ..., \varphi_I \in [0, 2\pi)$ and sample I points from an autoregressive process, which we denote by $r_1, ..., r_I$, add a sufficiently great number $c \in \mathbb{R}^+$ so that $r_i + c > 0$, $i = 1, ... I$, and multiply the result by a positive scale factor $s \in \mathbb{R}^+$. Using polar coordinate representation, we are able to transform these values into the edge points $(x_i, y_i), i = 1, ..., I$, of a circlelike object. The coordinates are defined as follows:

$$\begin{aligned} x_i &= (r_i + c) \cdot s \cdot \cos(\varphi_i) + x^*, \\ y_i &= (r_i + c) \cdot s \cdot \sin(\varphi_i) + y^*, \end{aligned}$$

where (x^*, y^*) is a random midpoint. By using values from an autoregressive process, the edges have some bulges and are not as scattered as they would be if the distances were sampled independently. In order to produce a demonstration image, we generate 40 circlelike objects with different midpoints, radii, and values of I and c. As a result, we obtain various circlelike objects with different degrees of size, roundness, and the number of bulges of the edges. The objects may overlap, but we pay attention that the overlap does not exceed a certain threshold. The final image is constructed by

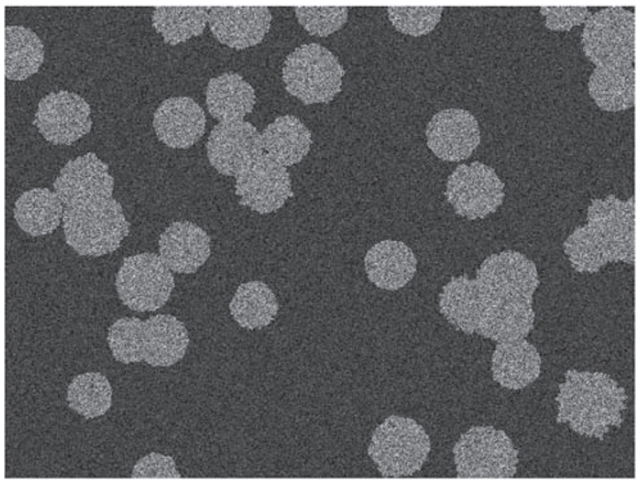


FIGURE 4 The synthetic demonstration image

assigning each foreground pixel a normally distributed intensity with mean 0.4 and a standard deviation of 0.2, and each background pixel a normally distributed intensity with mean 0.1 and standard deviation 0.1. In fact, truncated normal distributions are used in order to ensure that the pixel intensities lie between 0 ($\hat{=}$ black) and 1 ($\hat{=}$ white). The demonstration image is shown in Figure 4.

Algorithm 1 Detection of circlelike objects

```
1:  procedure DETECTOBJECTS(img, size, comp, min.size.cluster, estimator, kernel, use.fill.hull, ratio.thresh)
2:      img ← MEDIANFILTER(img, size)
3:      img ← COMPRESSING(img, comp)
4:      img ← NORMALIZING(img)
5:      threshold ← HUANG(img)
6:      img.bin ← BINARIZEIMAGE(img, threshold)
7:      img ← FILLHOLES(img.bin)
8:      if use.fill.hull == TRUE then
9:          img.bin ← img
10:     end if
11:     cluster.list ← CLUSTERING(img)                          ▷ see Algorithm 3
12:     cluster.list ← REMOVESMALLCLUSTERS(cluster.list, min.size.cluster)
13:     circle.list ← INITIALIZELIST                            ▷ this will be the list of all found circles
14:     for each cluster in cluster.list do
15:         img.cut.out ← CUTOUT(img, cluster)
16:         edge.points ← DETECTEDGES(img.cut.out, estimator, kernel)
17:         continue.loop ← TRUE
18:         while continue.loop == TRUE do
19:             circle ← FINDBESTFITTINGCIRCLE(edge.points, img.bin, ratio.thresh)
20:             if GOODNESSOFFITTOOLOW(circle) then
21:                 continue.loop ← FALSE
22:             else
23:                 circle.list ← ADDROW(circle.list, circle)
24:                 edge.points ← REMOVEEDGEPOINTS(edge.points, circle)
25:             end if
26:             if NUMBEROFELEMENTSTOOLOW(edge.points) then
27:                 continue.loop ← FALSE
28:             end if
29:         end while
30:     end for
31:     return circle.list
32: end procedure
```

We will now go through all steps of our new algorithm, whose pseudo code is given in Algorithm 1. The image to be analyzed enters the algorithm as input `img`.

4.1.1 | Median filtering

We start by preprocessing the image. As mentioned before, the noise—in terms of lighter pixels in the background and darker pixels in the foreground—might compound the problem of finding the correct circlelike objects in an image. Therefore, we would like to reduce the noise first. We achieve this by applying a median filter (see Section 3.1). The chosen window size enters the algorithm as parameter `size`. Figure 5 results from applying a median filter with window size 2 and window size 4 to the demonstration image. The bigger the window size, the blurrier and more homogeneous the image gets.

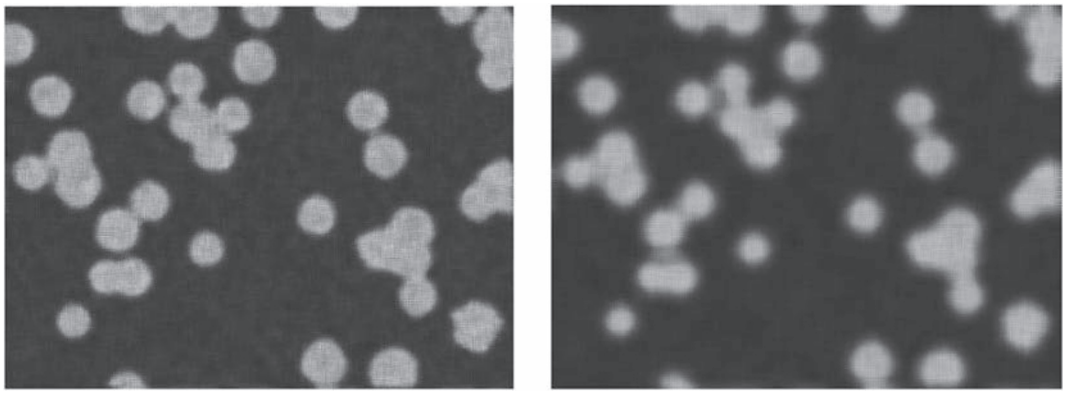


FIGURE 5 The demonstration image after applying median filters of size 2 (left panel) and 4 (right panel), respectively

4.1.2 | Compressing

At this point, it might be desirable to "compress" the image, for example, by removing every second pixel in order to reduce the algorithm's run time. The parameter `comp` $\in \{0, 2, 3, \ldots\}$ defines that every `comp` th pixel is removed, where for `comp = 0` no reduction of pixels takes place.

4.1.3 | Normalizing

It is conceivable that images at hand may have different color depths. That is, the ranges of pixel values might not be equal for all images. To avoid any problems with the subsequent analysis, images are normalized to the interval [0, 1] as the last step of the preprocessing. Hence, we replace the intensity $z_{i,j}$ of pixel (i, j) by

$$z_{i,j}^{norm} = \frac{z_{i,j} - \min\limits_{i,j}\{z_{i,j}\}}{\max\limits_{i,j}\{z_{i,j}\} - \min\limits_{i,j}\{z_{i,j}\}}.$$

Note that this is only valid for $\min\{z_{i,j}\} \neq \max\{z_{i,j}\}$, which is true if there are at least two different pixel intensities. We ignore the case of having an image where all pixels share the same intensity due to obvious reasons.

4.1.4 | Binarizing

We assume that pixels belonging to circlelike shapes are in average lighter than the background pixels. Therefore, we aim at separating the foreground from the background. This can be done by thresholding (or binarizing) the image, that is, whitening all pixels that are lighter than a certain threshold, and blackening all other pixels. The choice of the threshold value is crucial. There are a variety of algorithms that determine a threshold automatically (see, for example, previous works[19-30]).

One of these methods is the so-called Huang thresholding.[31] This algorithm uses a measure of fuzziness that represents the difference between the original image and its binary version. For every pixel and a given threshold value, the difference between the original pixel intensity and the mean original intensity of the region (back- or foreground) it belongs to is computed. Then, a so-called membership function is defined upon this difference where a smaller difference leads to a higher membership value. The fuzziness is defined as Shannon's entropy function, applied to the membership functions. The threshold minimizing this fuzziness is considered the optimal value.

4.1.5 | Hole filling

Because of the foreground's heterogeneity, it is likely that there are falsely blackened pixels within a single object or a conglomeration of objects. We fill these holes by whitening all black pixels that are completely surrounded by white

FIGURE 6 One cluster of the demonstration image after applying a median filter of size 2 and binarizing (left panel), and after additionally whitening black pixels located within the clusters of white pixels (right panel)

pixels. Figure 6 shows this step for one cluster of the demonstration image. As we can see, there are only a few pixels inside the cluster that are black. This is because in this image, the foreground's pixel intensities' variance is relatively small, so that the foreground can be separated well from the background. Nonetheless, on the right panel, these black pixels are whitened.

4.1.6 | Clustering

After binarizing the image, it seems sensible that each of the bigger clusters of connected white pixels like the one in Figure 6 contains at least one circlelike object—if two or more objects are overlapping or touching each other, they belong to the same cluster. In applications, it can occur that very small clusters of connected white pixels are speckles and not supposed to be detected as circlelike objects. We therefore first apply a fast stacked scanline algorithm (see Algorithm 3) to find out which white pixels belong to the same cluster, count the number of pixels in each cluster and omit those clusters that have too few pixels. Of course, the minimum number of pixels an object must have is not clear—we will leave this as a tuning parameter called `min.size.cluster`.

As mentioned above, it is still problematic that a cluster may contain more than one circlelike object. Fortunately, we know that each cluster contains an integer number of objects. We also know that they are almost circular. Therefore, we can achieve our goal by iterating over all clusters and fitting one circle after the other to each of them so that all clusters are, in a way, covered.

It would be possible to fit the circles directly using only the contours of the clusters, but then we would not use any extra information from the original image, which could be helpful in finding the correct positions. Because of this, we rather use the clusters as stencils for cutting out the corresponding pieces of the preprocessed original image.

Note the ambiguity of the word “clustering”: We first find connected white pixels containing one splat or more than one overlapping splats. In the latter case, the cluster's shape might not be circular at all. Within these clusters of arbitrary shapes, we aim to find one or more circles using the circular clustering algorithm.

4.1.7 | Edge detection

We now apply the edge detection as described in Section 3.2 on the cut-out. The parameters `estimator` and `kernel` determine the details. Figure 7 shows the estimated edge points of one of the clusters of our demonstration image. In this case, we use the simple median by setting `estimator = "median"` and `kernel = "mean"`, where the option `"median"` provides the sign function as score function and the option `"mean"` gives the rectangular kernel.

4.1.8 | Circular clustering

Based on the list of detected edge points, we finally can run the circular clustering algorithm from Section 3.3. The value of the objective function introduced there can be used to quantify a circle's goodness of fit. Because of the algorithm's random choice of starting points, it is advisable to search for many—say 30—circles and select the one with

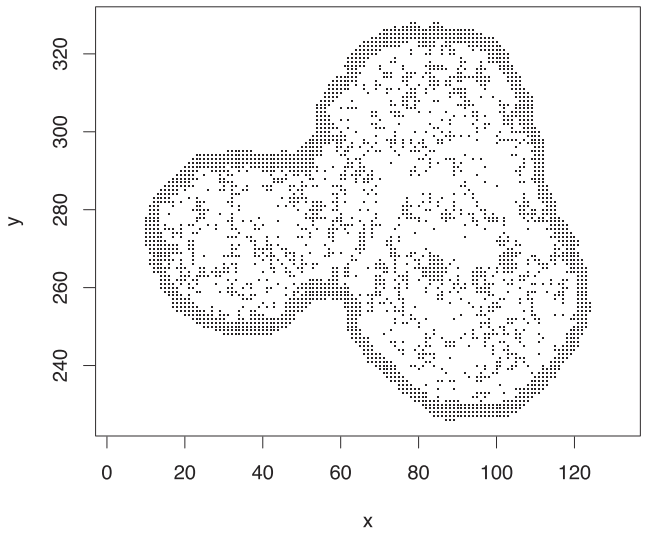


FIGURE 7 Detected edge points

the best fit that also fulfills the following two conditions: First, it must not be too similar to a circle already found. We consider two circles as too similar if their

$$\text{IoU} = \frac{\text{Area of inter section of the circles}}{\text{Area of union of the circles}}$$

exceeds 0.95, where IoU is short for Intersection over Union. Second, the pixels within the circle should be mainly foreground pixels. Hence, we need another threshold for the ratio of the number of foreground pixels in all pixels covered by the circle. This threshold can be set using parameter `ratio.thresh`. One can use both the binarized image as well as the binarized image with filled foreground holes to check this ratio. This leads us to another parameter called `use.fill.hull`. As mentioned above, we decided to proceed in an iterative manner. Repeating the search for thirty circles, though, would possibly yield the same circle representing the same circlelike object. This not only provides no new information, it might lead to a situation in which the algorithm does not terminate. Therefore, we memorize the best circle's parameters (the *x*- and *y*-coordinates as well as the radius) and remove the points that probably belong to the associated circlelike object afterwards. We define that an edge point belongs to a circle if its Euclidean distance to the midpoint is less than 1.2 times the radius.

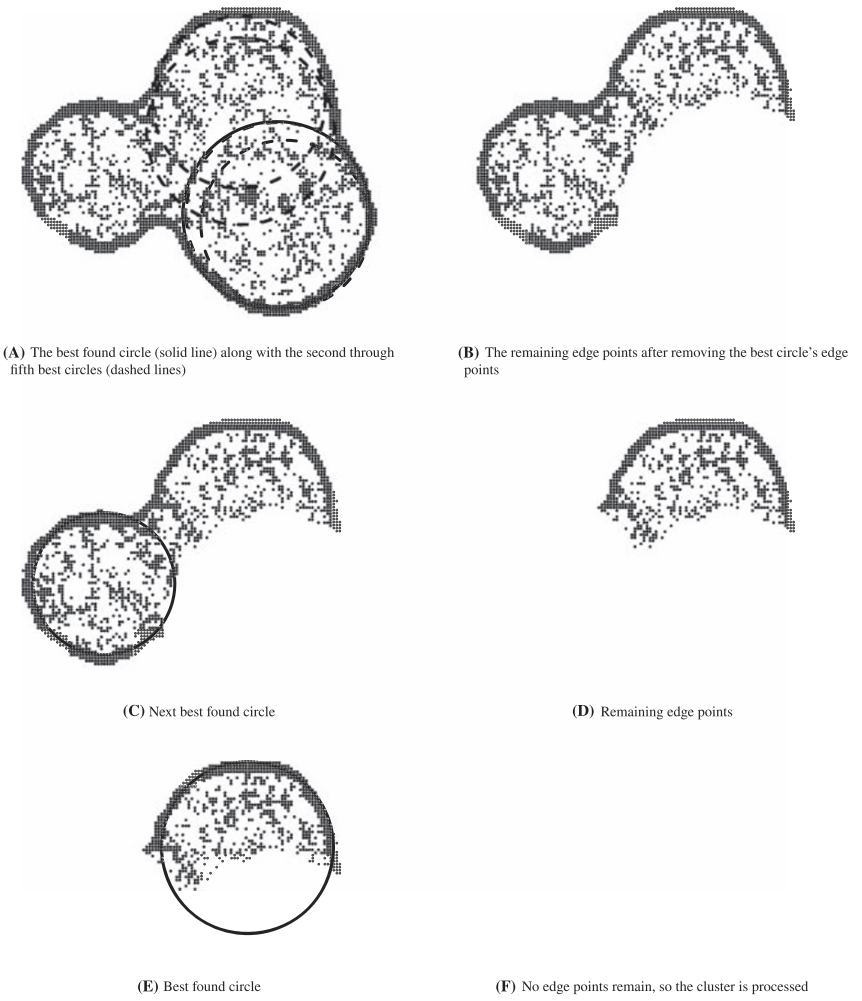
By using the factor 1.2, we ensure that points that might belong to an object but are lying outside the circle are also removed while preserving most points of other objects. Of course, this value should be considered a rule of thumb and might not be the ideal one, depending on the variability of the circlelike objects' shapes.

After finding the best fitting circle and removing its edge points, we repeat the search for the next best out of thirty circles over the remaining edge points and iterate until a stopping criterion is met. This criterion could, for example, be based on the number of remaining edge points or the goodness-of-fit of the detected circles. Figure 8 shows the whole procedure for one cluster.

In this manner, all clusters are processed. This can be done in parallel to reduce the time the algorithm takes. Finally, the algorithm outputs a list of the *x*- and *y*-coordinates of the centers as well as the radii of the circles.

4.2 | A measure of similarity

In order to assess the performance of the algorithm with a specific parameter setting and hence to be able to tune the algorithm's parameters adequately, we need a measure that compares the output to a set of handmarked “true” circles. If for every true circlelike object there would be exactly one identified circle overlapping it, we could just measure each pair's IoU. However, the algorithm will often produce slightly more or less circles than there are true circlelike objects. For example, two circles might be fitted to a single object. Then it is not clear which circle should be compared with the object.



(A) The best found circle (solid line) along with the second through fifth best circles (dashed lines)

(B) The remaining edge points after removing the best circle's edge points

(C) Next best found circle

(D) Remaining edge points

(E) Best found circle

(F) No edge points remain, so the cluster is processed

FIGURE 8 Processing one cluster step by step: Repeatedly finding the best circle and removing its edge points until no acceptable circle can be found anymore

To solve this problem, we first consider every pair of one circlelike object and one circle that are touching each other. For these, we compute the similarity by means of the IoU. Ideally, this value is 1, meaning the detected circle matches the handmarked circle perfectly. We can use these similarity measures as weights for finding a maximum weighted bipartite matching by running a maximum flow algorithm.[32] Here, this would result in an assignment of identified and true circles so that the total weight of the matching is maximal. This leads to maximum matching weights $w_1, ..., w_{n_{\text{matching}}}$, where n_{matching} is the number of such pairs. This number is always smaller than or equal to both the number n_{objects} of manually determined circles and the number n_{circles} of circles returned by the algorithm.

Of course, this is a very optimistic procedure—if the algorithm returned many circles at random positions and with various radii, some of them would match the real circlelike objects coincidentally, which would lead to a relatively high weight of a maximum matching and thus to a high value of the performance measure. Therefore, we should include the ratio of the circles that actually could be matched with a circlelike object and also the ratio of the circlelike objects that could be matched with a circle—to penalize outputs with either too many or too few circles.

With these considerations, we propose the following measure:

$$m := \frac{w_{\text{matching}}}{n_{\text{matching}}} \cdot \frac{n_{\text{matching}}}{n_{\text{objects}}} \cdot \frac{n_{\text{matching}}}{n_{\text{circles}}},$$

where n_{matching} is the number of pairs in the maximum matching, w_{matching} is the sum of all weights in the maximum matching, $w_1, ..., w_{n_{\text{matching}}}$, n_{objects} is the number of manually determined circles, and n_{circles} is the number of circles generated algorithmically.

The overall weight w_{matching} can range from 0 to n_{matching} because it is the sum of each pair's weight, which in turn is a number in [0, 1]. Thus, the first factor can only have values in [0, 1]. This is also true for the other factors because

$0 \leq n_{\text{matching}} \leq \min\{n_{\text{objects}}, n_{\text{circles}}\}$. Therefore, m can take values in $[0,1]$, where $m=1$ if and only if $n_{\text{objects}} = n_{\text{circles}} = n_{\text{matching}} = w_{\text{matching}} > 0$; that is, the output is exactly the same as the set of circles that have been determined manually, and $m=0$ if and only if $n_{\text{matching}} = 0$.

The first factor in m is the arithmetic mean of the weights of all pairs in the matching. By substituting this term with the median of the weights, we obtain a more robust alternative

$$m^{\text{robust}} := \text{median}\{w_1, ..., w_{n_{\text{matching}}}\} \cdot \frac{n_{\text{matching}}}{n_{\text{objects}}} \cdot \frac{n_{\text{matching}}}{n_{\text{circles}}}.$$

4.3 | Implementation

The algorithm, similarity measure, and hyperparameter optimization have been implemented in the free software environment R.[12] In this subsection, we give details on packages, functions, and parameters we used.

We start by going through the pseudo code of our algorithm (see Algorithm 1). All function calls that are not described here have been implemented without the help of special packages.

4.3.1 | `medianFilter`

Function `medianFilter` of package **EBImage**,[33] which uses a square pattern with edge length $(2 \cdot s + 1)$, where s is an integer "size."

4.3.2 | `fillHoles`

Function `fillHull` of package **EBImage**.

4.3.3 | `clustering`

Function `bwlabel` of package **EBImage**, see Algorithm 3.

4.3.4 | `detectEdges`

Function `edgepoints` of package **edci**[34] with a bandwidth of `h1n = h2n = 0.005` and score function `score = "gauss"` with standard deviation `sigma = 1`. The function `edgepoints` further needs the arguments `estimator` and `kernel`, which are input parameters of our algorithm.

The argument `kernel` determines the used kernel function K_k and `estimator` the estimator type, which could be `"median"`, `"M_mean"`, and `"M_median"`. If `estimator = "median"` is used, an M-kernel estimator with the sign function as score function is used, so that the argument of the parameter `score` of `edgepoints` is irrelevant. Moreover, the kernel is always the rectangular kernel (`kernel = "mean"`) for `estimator = "median"`. The estimator types `"M_mean"` and `"M_median"` concern M-kernel estimators where the score function is the derivative of the density of the standard normal distribution, that is, `score = "gauss"` is really used. Becayse this leads to so-called redescending M-estimators, the function $\mathcal{H}_k$ of Section 3 has several roots. The arguments `"M_mean"` and `"M_median"` determine that the mean and the median, respectively, are used as starting values for calculating the root.

The final result of the function `edgepoints` is a matrix of estimated jump heights. These jump heights are converted into estimated edge points by means of function `eplist` of the same package, where we use the 60% quantile of the nonzero estimated jump heights as the threshold `maxval`. Hence, points above this quantile are detected as edge points.

4.3.5 | findBestFittingCircle

Function `circMclust` of package **edci** generates a list of possible circles, where `datax` and `datay` are the x- and y-coordinates of the unclassified edge points, the scale parameter is `bw = 0.3` (which is called *s* in Section 3), the starting values for the maximization are picked randomly (`method = "prob"`), `prec = 1`, the radii may lie between `brminr = 3.5` and `brmaxr = 5`, and the search goes on until `nc = 30` candidate circles have been found. Notice that the help page of this function indicates that parameter `nc` is only relevant for `method = "const"`, which is not true. Function `bestMclust` of the same package sorts the candidates by their objective value. We remove those circles from this list for which the IoU with a best circle of a previous iteration is at least 0.95. In other words, we remove circles that have already been detected. After that, we choose the circle with the best objective value from the remaining candidate circles.

Next, we move to the calculation of the similarity measure and the adding of circles to images.

4.3.6 | Maximum weighted bipartite matching

Package **igraph**[35] provides data types and functions for graph algorithms. We first define a weighted bipartite graph using the functions `make_empty_graph`, `add_vertices`, and `add_edges`, where the vertices are marked true or identified circles, and each touching or overlapping pair of one true circle and one identified circle gets an edge. The weight of an edge is the similarity of the corresponding handmarked and estimated circles. We then compute the maximum matching using the function `maximum.bipartite.matching`.

4.3.7 | Visualization

Package **plotrix**[36] provides the function `draw.circle` to draw a circle on an existing plot.

5 | APPLICATION TO THERMAL SPRAY IMAGES

In this chapter, we apply our algorithm to splat images. The main goal is to find an appropriate parameter setting as in the future one would want to have a determined parameter setting for detecting the splats in an unseen image. Thus, it is important to either have one setting for all images that produces good results or to have a model for adjusting the parameters depending on some image, substrate, powder, and/or in-flight properties. First, we have a look at the parameters that we have to find appropriate settings for. These are listed in Table 1 with their associated data types, some possible values we want to consider during the optimization, and a short description. Apart from these tuning parameters, we have set the compression parameter `comp` to 2; that is, we remove every second pixel from the image in order to shorten the algorithm's run time. For the window size of the median filter, the settings 0, 2, 4, 6, 8, and 10 are compared, where 0 results in no filtering at all. For the minimal cluster size, the values 200, 400, or 600 seem to be sensible choices based on a rough examination of splat images. Apart from these tuning parameters, some have been fixed, which can be found in Table 2 along with their settings.

TABLE 1 The algorithm's hyperparameters to be tuned

Parameter	Type	Values	Description
`size`	integer(1)	`0, 2, 4, 6, 8, 10`	Window size of the median filter
`min.size.cluster`	integer(1)	`200, 400, 600`	Minimal cluster size
`estimator`	character(1)	`"median", "M_median", "M_mean"`	Defines how the edge points are estimated
`kernel`	character(1)	`"mean", "gauss"`	Kernel function for `estimator = "M_median"` or `"M_mean"`
`use.fill.hull`	boolean(1)	`FALSE, TRUE`	Use `fillHull` when checking the pixel ratio?
`ratio.thresh`	numeric(1)	`0.6, 0.65, 0.7`	Threshold for checking the pixel ratio

TABLE 2 List of the algorithm's fixed parameters

Parameter	Value
Window pattern of median filter	Square
Compression parameter `comp`	2
Edge detection parameter `maxval`	60% quantile of nonzero estimated jump heights
Number of randomly allocated circles per cluster	30
Threshold for IoU	0.95
Factor multiplied with estimated radius for removing edge points	1.2
Scale parameter s of objective function H in the circular clustering method	0.3

Concerning the edge detection, we compare different estimation methods. First, the simple median for each window is used (`estimator = "median"`, `kernel = "mean"`), as already considered in our running example in Section 4. Next, we consider the redescending M-estimators given by the score function `"gauss"`, that is, the derivative of the density of the standard normal distribution, with the rectangular (`kernel = "mean"`) or the Gaussian (`kernel = "gauss"`) kernel. For both kernels, the objective function needs to be optimized numerically, where starting values are either based on the mean (`estimator = "M_mean"`) or the median (`estimator = "M_median"`), leading to four combinations of used kernels and starting values. Hence, we compare five different edge detection procedures altogether. To decide whether a circle belongs to the foreground, the pixel ratio is checked using the binarized image with (`use.fill.hull = TRUE`) or without (`use.fill.hull = FALSE`) filled holes. As threshold values for checking the pixel ratio, we compare 0.6, 0.65, and 0.7.

To find a good parameter setting within the thereby given possible combinations and to test the algorithm, we optimize over a set of ten images with handmarked splats. As the objective function, we consider the mean of the measure m defined in Section 4 on all 10 images. In a separate optimization, we also examine the robust measure m^{robust}. The optimization is carried out with a meta-model-based approach using the R package **mlrMBO**[37] and a random forest, which is the only implemented model that is suitable for our case with both quantitative and qualitative input variables. The optimization procedure is as follows: First, a random Latin hypercube design (LHD) with 28 points is evaluated using the algorithm. The random forest model is fit to the resulting data. Then, a focus search is employed: Therefore, another random LHD with 10,000 points is generated. In each point, the expected improvement (EI) of the model is computed. Around the point with the highest EI among these points, $\boldsymbol{x}^* = (x_1^*, x_2^*, \dots)$, the parameter space is shrunk. For continuous variables, this is done by updating the bounds l_i and u_i: $l_i := \max\{l_i, x_i^* - \frac{1}{4}(u_i - l_i)\}$, and $l_i := \min\{u_i, x_i^* + \frac{1}{4}(u_i - l_i)\}$. For a categorical variable with more than 2 levels, one of the levels other than x_i^* is removed randomly. Then, another LHD with 10,000 points is generated within the new parameter space for which the EIs of the model are computed and the parameter space is further shrunk. The steps of generating an LHD and shrinking the parameter space are repeated five times in total. Finally, the objective function is evaluated at the point with the greatest expected improvement and is added to the original design, whereupon the whole process of building a random forest model and searching a new point is repeated altogether 10 times.

With respect to implementing the procedure, we first define the considered parameter space using the functions `makeParamSet` in combination with `makeDiscreteParam` and `makeLogicalParam` from the R package **ParamHelpers**.[38] Then we define the random forest we want to use as the meta-model with the function `makeLearner` from the package **mlr**,[39] and set the parameter `predict.type = "se"` in order to get an error estimate of the prediction, which is needed for computing the expected improvement. The optimization is conducted using the function `mbo` from the package **mlrMBO**,[37] which runs the optimization procedure described in Section 5. We therefor set the parameter `crit = "ei"` of the function `setMBOControlInfill` in order to consider the expected improvement during the optimization. The objective function is defined using `makeSingleObjectiveFunction`, where we pass a function in which our algorithm is run for the respective parameter setting on all ten training images and the mean of the resulting measures is returned.

The resulting optimal parameter settings are `size = 8`, `min.size.cluster = 200`, `use.fill.hull = TRUE`, `estimator = "median"`, and `ratio.thresh = 0.7`. Note that for `estimator = "median"` always the rectangular kernel is used.

To get an idea of how good the algorithm works with this parameter setting, we test it on five validation images that have not been used for the tuning. Figure 9 shows the best and the worst prediction. In this case, it actually does not

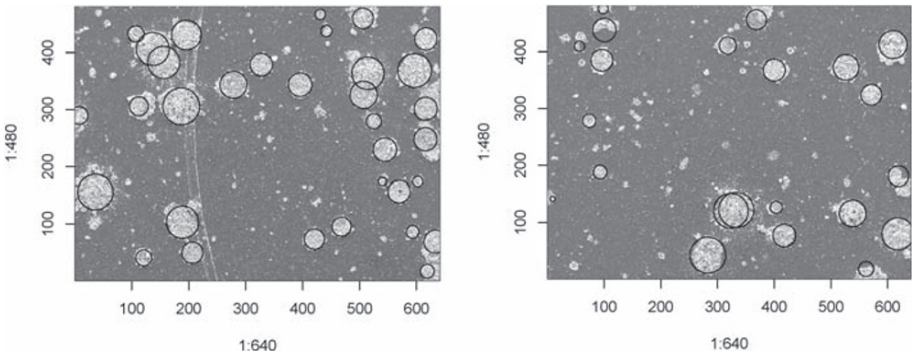


FIGURE 9 The best (left panel) and worst (right panel) predictions of all validation images with the optimal parameters

make a difference whether the mean or the median maximum weighted bipartite matching's weight between hand-marked circles and predicted circles was considered. On the left-hand side, the estimated circles of the best prediction using the optimal parameter setting is depicted, which has values of $m = 0.41$ and $m^{\text{robust}} = 0.43$, whereas the panel on the right-hand side shows the estimated circles of the worst prediction of all validation images, having values of $m = 0.22$ and $m^{\text{robust}} = 0.24$. It can be easily seen that even for an apparently low value of m around 0.2, the identified circles seem to fit well. This might be due to the fact that, especially in the case of many overlapping circles, there are many different alternative solutions to the same image that can be equally valid. Then, accurate fits might lead to low values of m. We will find in the next section that these values are, however, well in accordance with the general performance of the algorithm in cases of overlapping circlelike objects.

6 | COMPARISON TO A BASELINE

Next, we compare our algorithm to an algorithm that is compounded of what can be considered standard approaches. We first describe the baseline algorithm and its implementation. Then, the images we compare the algorithms on are introduced, followed by the results.

6.1 | Baseline algorithm and implementation

The pseudo code of the baseline is given in Algorithm 2. The algorithm mainly consists of the application and scale-space optimization of the multiscale LoG operator to detect blobs in a median filtered, downsampled image. We decided to use this approach rather than operator-independent approaches as introduced by Lankton et al.[40] and Comelli et al.[41] because there each cluster would have to be initialized by the practitioner. Also, because individual splats within the same cluster are in general not distinguishable by considering differences in pixel intensities, these algorithms seem less appropriate for this application.

Algorithm 2 Baseline

```
1: procedure BASELINE(img, filter.size, min.sigma, max.sigma, n.sigma, threshold, overlap)
2:     img ← MEDIANFILTER(img, filter.size)
3:     circles ← LOGBLOBDETECTION(img, min.sigma, max.sigma, n.sigma, threshold, overlap)
4:     return circles
5: end procedure
```

The baseline approach has been implemented using Python 3.6.[42] Python is distributed under the Python Software Foundation License, which is compatible with the GNU General Public License (GPL).

We now go through the pseudo code of the baseline algorithm. The function calls that are not described here have been implemented without the help of special classes.

6.1.1 | medianFilter

Function `median_filter` of class **scipy.ndimage**, parameter `filter.size` is set to 13.

6.1.2 | LoGBlobDetection

Function `blob_log` of class **skimage.feature**, argument `min.sigma` is set to $\frac{10}{\sqrt{2}}$ to detect circles of minimal radius of 10, argument `max.sigma` is set in dependence of image size, the maximal radius should be half of the smaller image dimension, argument `n.sigma` determines the number of Gaussian kernels with different standard deviations used in this approach, which is set to 100. For argument `threshold` a grid search is performed as a parameter tuning, conducted on the same ten images as in Section 5. We considered the 20 values 0.00, 0.01, 0.02, ..., 0.19 and found that a threshold of 0.07 works best for our application. Argument `overlap` is set to 0.95: If two detected blobs overlap more than 95 %, the smaller one is removed from results.

6.1.3 | Visualization

Function `draw.ellipse` of class **PIL.ImageDraw**.

6.2 | Test images

For a thorough comparison of the two approaches, we define six classes of different properties and generate 50 test images for each class. As summarized in Table 3, we consider the following classes of test images: First, a default setting with 40 circlelike objects per image, which comes close to the original splat images considered in Section 5. Second, test images with 60 emulated splats, which leads to an increased number of overlapping objects. Third, test images where objects are prevented from overlapping. Fourth, a class in which the edges of the objects are round and undistorted. Fifth, test images with very distorted, "spattered" objects. Finally, the same setting as the default without any noise added to the images, which leaves the images binary. Figure 10 shows one of the images for each of the different classes. The midpoints of the circlelike objects of the ith image are identical among the classes "Default," "Round," "Spattered," and "NoNoise." This enables us to ascribe the differences in measures between two of these classes to their different properties alone and not to randomly occuring effects due to different midpoints. Because class "MuchOverlap" has more circlelike objects than the other classes, and overlapping circles are removed from class "NoOverlap," these two classes do not have the same midpoints as the other classes.

6.3 | Results of the comparison

We now compute the measure m as defined above for both the new approach presented in this paper as well as the baseline applied to each of the test images. We use the parameter settings obtained from the parameter tuning in

TABLE 3 The six test image classes used to compare the algorithms

Class	Note
Default	40 emulated splats
MuchOverlap	60 emulated splats, so more of them overlap
NoOverlap	Overlapping emulated splats are removed
Round	Edges are not distorted
Spattered	Edges are very distorted
NoNoise	Black and white images without any noise

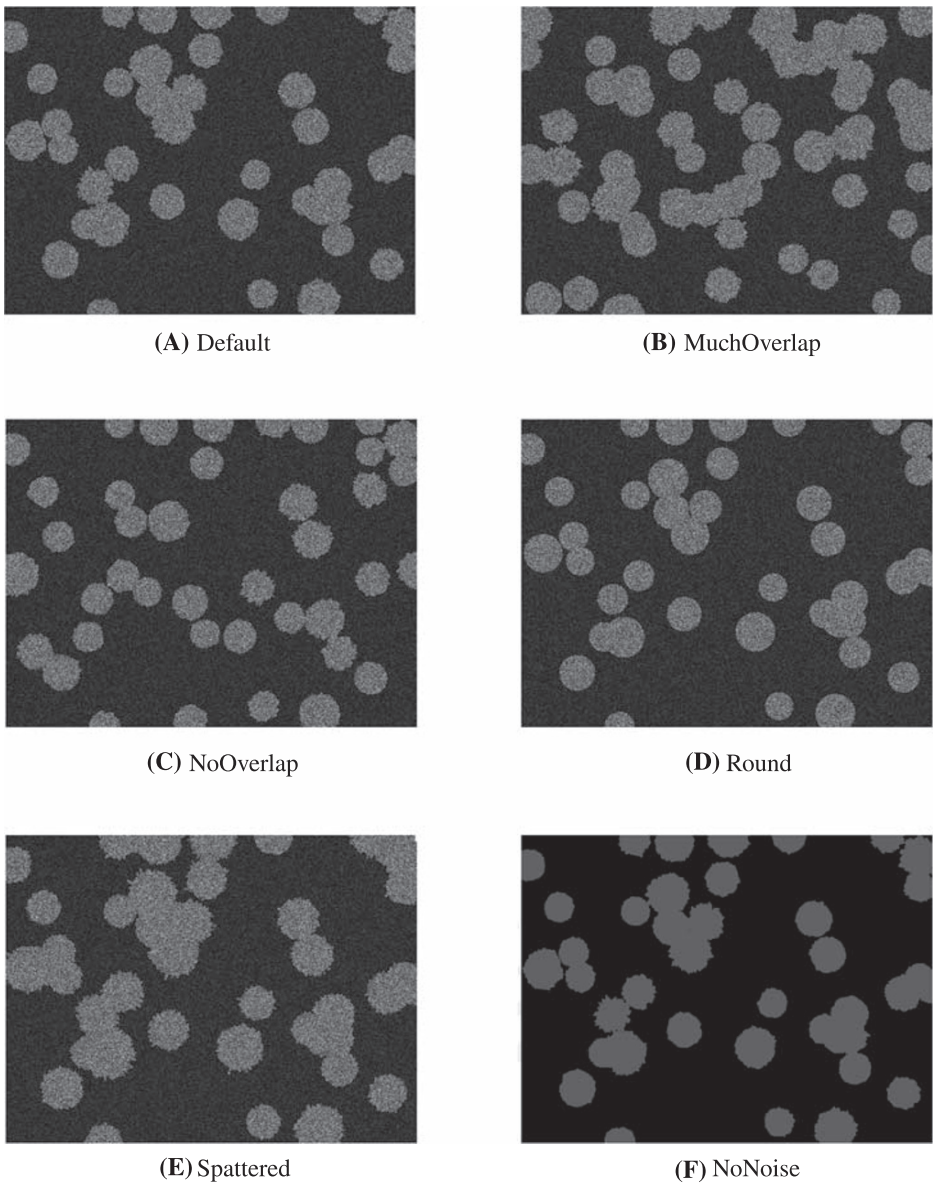


FIGURE 10 One of the test images for each of the different classes. Note that the center points of the objects on Panels (A), (D), (E), and (F) are identical

Section 5 for the new algorithm and the ones described in the introduction of the baseline algorithm in Section 6.1. Figure 11 shows the corresponding boxplots. Comparing the results between the different classes, we can see that the amount of overlap has the biggest effect on the detection accuracy, whereas circles with distorted edges can be detected only slightly worse than round circles. Adding noise to the images in this case obviously had no great effect on the results, as we can see by comparing the measures of classes "NoNoise" and "Default". In fact, both approaches performed even slightly better on the default images with noise. This might be due to the fact that the parameters were tuned to the noisy images of our application.

In five of the six classes, the new approach performed better than the baseline in terms of detection accuracy. On most images without overlapping circlelike objects, however, the baseline outperformed our new approach minimally.

Figure 12 shows the difference of the two algorithms' measures per test image. On 48 out of 50 default images, our algorithm had a higher score than the baseline. The amount of overlap determines which of the approaches is better: A high amount of overlap can be handled better with our approach than with the baseline. On images without overlap, it is the other way around. Without exception, circlelike objects with undistorted edges have been recognized better by the new approach. Only on four out of 50 images with spattered objects and on 11 out of 50 images without any noise, the baseline achieved a higher detection accuracy.

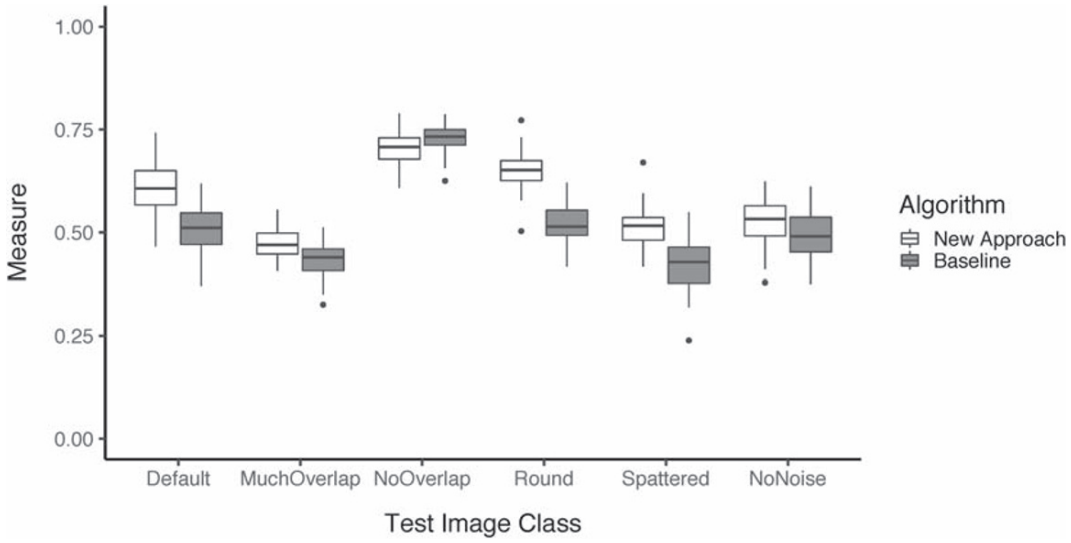


FIGURE 11 The performance of the new approach compared to the baseline (see Algorithm 2) on the different test image classes (see Table 3)

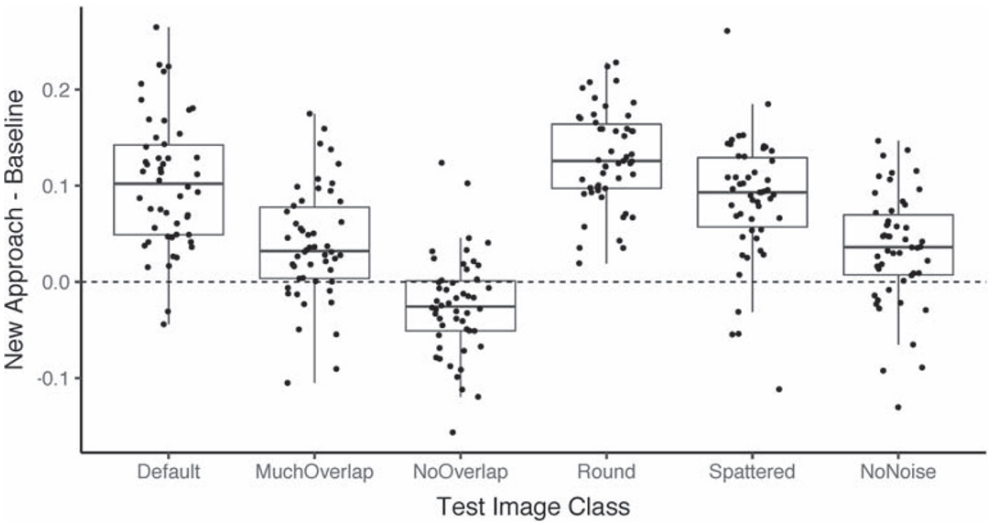


FIGURE 12 The differences between the measures of the two algorithms evaluated on each of the test images. The bigger the numbers, the better the new approach in comparison to the baseline

7 | SUMMARY

Up to now, it is a nontrivial task to detect nearly circular objects of varying sizes in noisy images using open source software. Most of the freely available software with an integrated functionality for automatic object detection requires at least a homogenous background to work decently. SEM images taken of powder splats on the substrate in HVOF thermal spray processes, however, are neither perfectly round nor do they all have equal radii. In addition, the images tend to be quite noisy.

In this contribution, we presented a new approach on how to detect circlelike objects in noisy images. Our algorithm fits circles to circlelike objects in an iterative manner, mainly using RDKE to detect edges and circular clustering to find circles. We developed a measure to quantify the goodness of fit of a set of circles given manually assigned circles.

We created a set of test images, which includes synthetic splat images and modifications with very overlapping, not overlapping, not noisy, round, and spattered circles. For a comparison, we considered a baseline algorithm using the Laplacian of Gaussian blob detection. We found that our approach outperforms the baseline, especially when there is a high amount of overlapping circlelike objects. In case of nonoverlapping objects, however, the baseline is the method of choice.

For a thermal spray application, we tuned the parameters of the new algorithm with a number of splat images and validated its performance on some further images. In future, the new algorithm will replace the time-consuming and subjective manual detection of splats by the engineer.

ACKNOWLEDGMENTS

We thank Birger Hussong for helpful insight in the thermal spray process. This work has been supported by the Collaborative Research Center "Statistical modeling of nonlinear dynamic processes" (SFB 823) of the German Research Foundation (DFG).

ORCID

Dominik Kirchhoff https://orcid.org/0000-0001-6006-790X
Sonja Kuhnt https://orcid.org/0000-0003-1883-8760
Louise Bloch https://orcid.org/0000-0001-7540-4980
Christine H. Müller https://orcid.org/0000-0002-4097-3320

REFERENCES

1. Hough PV. Method and means for recognizing complex patterns. U.S. Patent Nr 3 069 654. 1962.
2. Wu G, Liu W, Xie X, Wei Q. A shape detection method based on the radial symmetry nature and direction-discriminated voting. In: 2007 IEEE International Conference on Image Processing, Vol. 6. San Antonio, Texas; 2007:169-172.
3. Liu H, Wang Z. PLDD: Point-lines distance distribution for detection of arbitrary triangles, regular polygons and circles. *J Visual Commun Image Represent*. 2014;25:273-284.
4. Pajouhi Z, Roy K. Image edge detection based on swarm intelligence using memristive networks. *IEEE Trans Comput-Aided Design Integ Circuits Syst*. 2018;37:1774-1787.
5. Mafi M, Rajaei H, Cabrerizo M, Adjouadi M. A robust edge detection approach in the presence of high impulse noise intensity through switching adaptive median and fixed weighted mean filtering. *IEEE Trans Image Proces*. 2018;27:5475-5490.
6. Flores-Vidal PA, Olaso P, Gómez D, Guada C. A new edge detection method based on global evaluation using fuzzy clustering. *Soft Comput*. 2018;23(6):1809–1821. https://doi.org/10.1007/s00500-018-3540-z
7. Liu Y, Cheng M, Hu X, Bian J, Zhang L, Bai X, Tang J. Richer convolutional features for edge detection. *IEEE Trans Pattern Anal Machine Intel*. 2019;41:1939-1946.
8. Qiu P. Nonparametric estimation of jump surface. *Sankhyā: The Indian J Stat*. 1997;2(59):268-294. http://www.jstor.org/stable/25051155
9. Qiu P. Jump surface estimation, edge detection, and image restoration. *J Am Stat Assoc*. 2007;102(478):745-756. http://www.jstor.org/stable/27639903
10. Garlipp T, Müller CH. Robust jump detection in regression surface. *Sankhyā: The Indian J Stat (2003–2007)*. 2001;69(1):55-86. http://www.jstor.org/stable/25664543
11. Chu C-K, Siao J-S, Wang L-C, Deng W-S. Estimation of 2D jump location curve and 3d jump location surface in nonparametric regression. *Stat Comput*. 2012;22(1):17-31. https://doi.org/10.1007/s11222-010-9203-2
12. R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing; 2016. https://www.R-project.org/
13. Tillmann W, Hussong B, Priggemeier T, Kuhnt S, Rudak N, Weinert H. Influence of parameter variations on WC-Co splat formation in an HVOF process using a new beam-shutter device. *J Thermal Spray Technol*. 2013;22(2):250-262. http://doi.org/10.1007/s11666-012-9881-8
14. Müller CH, Garlipp T. Simple consistent cluster methods based on redescending M-estimators with an application to edge identification in images. *J Multivariate Anal*. 2005;92(2):359-385.
15. Huber PJ. *Robust Statistics*, Wiley Series in Probability and Statistics. Hoboken, New Jersey: John Wiley & Sons, Inc.; 2004. https://books.google.de/books?id=e62RhdqIdMkC
16. Garlipp T, Müller CH. Detection of linear and circular shapes in image analysis. *Comput Stat Data Anal*. 2006;51(3):1479-1490.
17. Kong H, Akakin HC, Sarma SE. A generalized Laplacian of Gaussian filter for blob detection and its applications. *IEEE Trans Cybern*. 2013;43(6):1719-1733.
18. Lindeberg T. Feature detection with automatic scale selection. *Int J Comput Vision*. 1998;30(2):79-116.
19. Prewitt JMS, Mendelsohn ML. The analysis of cell images. *Ann New York Acad Sci*. 1966;128(3):1035-1053. http://doi.org/10.1111/j.1749-6632.1965.tb11715.x
20. Ridler T, Calvard S. Picture thresholding using an iterative selection method. *IEEE Trans Syst Man Cybern*. 1978;8(8):630-632.
21. Li C, Tam PK-S. An iterative algorithm for minimum cross entropy thresholding. *Pattern Recog Lett*. 1998;19(8):771-776.
22. Kapur JN, Sahoo PK, Wong AK. A new method for gray-level picture thresholding using the entropy of the histogram. *Comput Vision, Graph Image Process*. 1985;29(3):273-285.
23. Glasbey CA. An analysis of histogram-based thresholding algorithms. *CVGIP: Graph Models Image Process*. 1993;55(6):532-537.
24. Kittler J, Illingworth J. Minimum error thresholding. *Pattern recognition*. 1986;19(1):41-47.

25. Tsai W-H. Moment-preserving thresolding: A new approach. *Comput Vision Graph Image Process*. 1985;29(3):377-393.
26. Otsu N. A threshold selection method form gray-level histograms. *IEEE Trans Syst Man Cybern*. 1979;9(1):1979.
27. Doyle W. Operations useful for similarity-invariant pattern recognition. *J ACM (JACM)*. 1962;9(2):259-267.
28. Shanbhag AG. Utilization of information measure as a means of image thresholding. *CVGIP: Graph Models Image Process*. 1994;56(5): 414-419.
29. Zack G, Rogers W, Latt S. Automatic measurement of sister chromatid exchange frequency. *J Histochem Cytochem*. 1977;25(7):741-753.
30. Yen J-C, Chang F-J, Chang S. A new criterion for automatic multilevel thresholding. *IEEE Trans Image Process*. 1995;4(3):370-378.
31. Huang L-K, Wang M-JJ. Image thresholding by minimizing the measures of fuzziness. *Pattern Recog*. 1995;28(1):41-51.
32. Goldberg AV, Tarjan RE. A new approach to the maximum-flow problem. *J ACM (JACM)*. 1988;35(4):921-940.
33. Pau G, Fuchs F, Sklyar O, Boutros M, Huber W. EBImage—An R package for image processing with applications to cellular phenotypes. *Bioinformatics*. 2010;26(7):979-981.
34. Garlipp T. edci: Edge detection and clustering in images. https://CRAN.R-project.org/package=edci. R package version 1.1-1; 2010.
35. Csardi G, Nepusz T. The igraph software package for complex network research. *Inter J Complex Syst*. 2006;1695:1-9. http://igraph.org
36. J L. Plotrix: A package in the red light district of R. *R-News*. 2006;6(4):8-12.
37. Bischl B, Bossek J, Horn D, Lang M. mlrMBO: Model-based optimization for mlr. https://github.com/berndbischl/mlrMBO. R package version 1.0; 2016.
38. Bischl B, Lang M, Richter J, Bossek J, Horn D, Kerschke P. Paramhelpers: Helpers for parameters in black-box optimization, tuning and machine learning. https://CRAN.R-project.org/package=ParamHelpers. R package version 1.11; 2018.
39. Bischl KSRSC, Jones. mlr: Machine Learning in R. http://jmlr.org/papers/v17/15-066.html; 2016.
40. Lankton S, Nain D, Yezzi A, Tannenbaum A. Hybrid geodesic region-based curve evolutions for image segmentation. *Medical Imaging 2007: Physics of medical imaging*, Vol. 6510. Bellingham, Washington: International Society for Optics and Photonics; 2007:65104U.
41. Comelli A, Stefano A, Russo G, et al. A smart and operator independent system to delineate tumours in positron emission tomography scans. *Comput Biol Med*. 2018;102:1-15.
42. Van Rossum G, Drake Jr FL. *Python Tutorial*. The Netherlands: Centrum voor Wiskunde en Informatica Amsterdam; 1995.

AUTHOR BIOGRAPHIES

Dominik Kirchhoff is a research assistant for Mathematical Statistics in the Department of Computer Science at Dortmund University of Applied Sciences and Arts. He received his master's degree in Data Science from TU Dortmund University in 2015. His research interests include meta-model-based optimization for both continuous and categorical inputs.

Sonja Kuhnt is a professor for Mathematical Statistics in the Department of Computer Science at Dortmund University of Applied Sciences and Arts. Her research interests include the Design and Analysis of Real and Computer Experiments, Categorical Data Analysis and Robust Statistics. She received her PhD and Habilitation in Statistics from TU Dortmund University.

Louise Bloch is a research assistant for Medical Computer Science in the Department of Computer Science at Dortmund University of Applied Sciences and Arts. She received her master's degree in Medical Computer Science from Dortmund University of Applied Sciences and Arts in 2020. Her research interests include image processing, machine learning and deep learning in particular in medical contexts.

Christine H. Müller is full professor of Statistics with Applications in Engineering Sciences at the Department of Statistics at TU Dortmund University. She received her PhD in mathematics at the Free University of Berlin. Before she became a professor in Dortmund, she was a lecturer in Göttingen and a professor in Oldenburg and Kassel. She is Coordinating Editor-in-Chief of Statistical Papers. From 2013 to 2019, she was the Chairwoman of the Deutsche Arbeitsgemeinschaft Statistik (DAGStat), an umbrella association of 14 German societies in statistics. Her research areas are technometrics, experimental design, and robust statistics.

How to cite this article: Kirchhoff D, Kuhnt S, Bloch L, Müller CH. Detection of circlelike overlapping objects in thermal spray images. *Qual Reliab Engng Int*. 2020;36:2639–2659. https://doi.org/10.1002/qre.2689

APPENDIX A: FAST STACKED SCANLINE ALGORITHM

Algorithm 3 Clustering

```
1:  procedure CLUSTERING(img)
2:      img ← BINARIZEIMAGE(img, threshold)
3:      clusterList ← INITIALIZECLUSTERLIST
4:      unprocessedForegroundPixelList ← GETALLUNPROCESSEDFOREGROUNDPIXELS(img)
5:      while unprocessedForegroundPixelList ≠ ∅ do
6:          cluster ← INITIALIZECLUSTER
7:          currentPixel ← GETPIXEL(unprocessedForegroundPixelList)
8:          stack ← INITIALIZESTACK
9:          stack ← PUSH(stack, currentPixel)
10:         while stack ≠ ∅ do
11:             currentPixel ← POP(stack)
12:             upperPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'top')
13:             while upperPixel ∈ unprocessedForegroundPixelList do
14:                 currentPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'top')
15:                 upperPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'top')
16:             end while
17:             while currentPixel ∈ unprocessedForegroundPixelList do
18:                 cluster ← ADDPIXELTOCLUSTER(cluster, currentPixel)
19:                 unprocessedForegroundPixelList ← DELETEPIXELFROMLIST(unprocessedForegroundPixelList, currentPixel)
20:                 leftPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'left')
21:                 if leftPixel ∈ unprocessedForegroundPixelList then
22:                     if leftPixel ∉ stack then
23:                         stack ← PUSH(stack, leftPixel)
24:                     end if
25:                 end if
26:                 rightPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'right')
27:                 if rightPixel ∈ unprocessedForegroundPixels then
28:                     if rightPixel ∉ stack then
29:                         stack ← PUSH(stack, rightPixel)
30:                     end if
31:                 end if
32:                 currentPixel ← GETNEIGHBOURPIXEL(img, currentPixel, 'bottom')
33:             end while
34:         end while
35:         clusterList ← PUSH(clusterList, cluster)
36:     end while
37:     return clusterList
38: end procedure
```