Perimetric Complexity of Binary Digital Images
Notes on Calculation and Relation to Visual Complexity

Andrew B. Watson

Perimetric complexity is a measure of the complexity of binary pictures. It is defined as the sum of inside and outside perimeters of the foreground, squared, divided by the foreground area, divided by 4π. Difficulties arise when this definition is applied to digital images composed of binary pixels. In this article we identify these problems and propose solutions. Perimetric complexity is often used as a measure of visual complexity, in which case it should take into account the limited resolution of the visual system. We propose a measure of visual perimetric complexity that meets this requirement.

■ Background

Perimetric complexity is a measure of the complexity of binary pictures. It is defined as the sum of inside and outside perimeters of the foreground, squared, divided by the foreground area, divided by 4π. The concept of perimetric complexity was first introduced (and called *dispersion*) by Attneave and Arnoult [1] in an effort to explain the apparent perceptual complexity of visual shapes. In the field of image processing, the concept appears as its inverse, *compactness* [2, 3, 4]. The concept was given new life (and a new name) in 2006 by Pelli et al., who showed that the efficiency of letter identification was nearly proportional to perimetric complexity [5]. It has since become a popular metric in a variety of shape analysis applications, including human letter identification [5, 6, 7], handwriting recognition [8], evolution of graphical symbols [9], and design of graphical anti-spam technologies [10, 11, 12].

In this article we develop *Mathematica* functions to compute perimetric complexity of binary digital images and illustrate their application. The code is compatible with Version 8 of *Mathematica*.

Although the concept of perimetric complexity is clear when applied to continuous plane shapes, complications arise when the concept is extended to binary digital images. We discuss these complications and suggest suitable solutions. We also introduce the concept of visual perimetric complexity, which takes into account the blurring action of the human visual system.

We begin by illustrating the application of the function `PerimetricComplexity` to a binary image.

```
cat =        ;
```

```
PerimetricComplexity[cat]
```

```
{2110.08, 89588, 3.95494}
```

The output is a list containing the perimeter (in pixels), the area (in square pixels), and the complexity. In the following sections we describe the derivation of this function, as well as the options that may be used to control its operation.

■ Perimetric Complexity of Geometric Shapes

Perimetric complexity is a measure of the complexity of binary pictures. In a binary picture, one or several regions of the same color (white) are defined as foreground, and the remainder (black) as background. Perimetric complexity C is defined here as the sum of the inside and outside perimeters of the foreground P, squared, divided by the foreground area A, divided by 4π:

$$C = \frac{P^2}{4\pi A}. \tag{1}$$

In the remainder of this article, unless otherwise noted, we use the term complexity as synonymous with perimetric complexity.

We begin with the example of a circular disk with unit radius.

```
Graphics[{White, Disk[]}, PlotRange → 2, Background → Black]
```
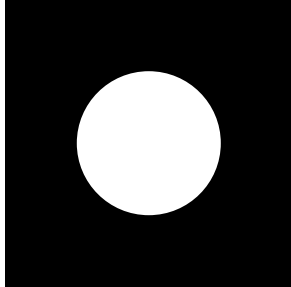

Here the perimeter is 2π and the area is π, so the complexity is

$$C = \frac{(2\pi)^2}{4\pi\pi} = 1. \tag{2}$$

It can be shown that the disk is the shape with the lowest complexity. The normalizing constant 4π in the definition leads to a unit value for this most simple shape. As a consequence, any other value of complexity is easily compared to that of the circular disk. Pelli et al. [5] suggest that complexity is closely related to the number of visual features in a shape. In that sense, we could say that the circular disk has only a single feature.
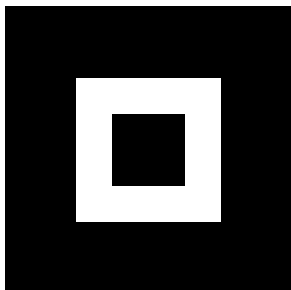
Our next example is a square with unit sides.

The perimeter here is 4, and the area is 1, so the perimetric complexity is

$$C = \frac{4^2}{4\pi} = \frac{4}{\pi} \approx 1.27. \tag{3}$$

If we add a square hole in the center with side length 1/2, there is an interior perimeter as well, as shown here.

```
Graphics[{
   White, Rectangle[-{1, 1} / 2, {1, 1} / 2],
   Black, Rectangle[-{1, 1} / 4, {1, 1} / 4]},
  PlotRange → 1, Background → Black]
```

Now the total perimeter is the sum of inner and outer perimeters and the area is the difference in areas of the squares, so

$$C = \frac{(4+2)^2}{(1-1/4)\,4\,\pi} = \frac{12}{\pi} \approx 3.82. \tag{4}$$

So according to this measure, the square with a hole is about three times as complex as the square.

Some important observations about complexity are: (1) it is dimensionless; (2) it is independent of scale or orientation; and (3) it is additive. By additive we mean that the complexity of a pair of shapes, considered as a single shape, is equal to the sum of their complexities computed separately.

■ Perimetric Complexity of Plane Curves

Although it is beyond the scope of this article, we note for reference that if a shape is defined by a closed parametric curve, its exact complexity can be obtained using calculus methods [13]. Specifically, if over an interval $a \leq t \leq b$ the functions $x(t)$ and $y(t)$ and their derivatives $\dot{x}(t)$ and $\dot{y}(t)$ are continuous, then the curve described has a length

$$P = \int_a^b \sqrt{\dot{x}^2 + \dot{y}^2} \; dt \tag{5}$$

and an area

$$A = -\int_a^b \dot{y}\, x \, dt. \tag{6}$$

■ Perimetric Complexity of Binary Digital Images

A digital image is defined here as a rectangular array of square pixels. A binary digital image contains pixel values of 1 (white) and 0 (black) only. The foreground consists of the white pixels.

The original definition of complexity relies upon the notion of a perimeter, which has no unique analog in the context of digital images. However, two definitions of perimeter are available, as described below.

☐ Using "PerimeterLength"

The first definition we consider is the most straightforward. Consider a binary image consisting of a single white pixel.

```
ImagePad[Image[{{1}}], 1]
```


It seems natural to define the perimeter of this shape as 4 (pixels), and the area as 1 (pixel^2), so $C = 4/\pi$, the same as the square discussed earlier.

Now consider this shape consisting of 3 white pixels.

```
ImagePad[Image[{{1, 0}, {1, 1}}], 1]
```


Here the perimeter, consisting of the exposed pixel faces, is 8, and the area is 3, so $C = 16/(3\,\pi)$.

Extending this idea, we can define the perimeter as the sum of the exposed faces of pixels in the foreground.

Version 8 of *Mathematica* includes a set of functions from the discipline of mathematical morphology. These can be used to easily calculate perimetric complexity. To illustrate this we begin with a binary image with several separated parts.

The MorphologicalComponents function finds connected regions and labels them with integers. The Colorize function visualizes these regions by assigning colors to each label. The CornerNeighbors → False option ensures that only 4-connected neighborhoods are considered.

```
MorphologicalComponents[yi, CornerNeighbors → False] //
  Colorize
```

The `ComponentMeasurements` function returns a selected set of measurements about each region. In this case we are interested in the area and the perimeter length. The results are returned as a set of rules, showing the results for each region.

```
cm = ComponentMeasurements[
   MorphologicalComponents[yi, CornerNeighbors → False],
   {"PerimeterLength", "Count"}, CornerNeighbors → False]
```

{1 → {56, 98}, 2 → {198, 469}, 3 → {138, 371}}

We can combine the perimeters and areas of the several regions, and then compute complexity in the usual way.

```
Total[Last /@ cm]
```

{392, 938}

```
%[[1]]^2 / %[[2]] / (4 Pi) // N
```

13.0365

The preceding calculations are implemented in the function `PerimetricCom‑plexity`, defined in the Appendix. We can obtain the previous result.

```
PerimetricComplexity[yi, Filter → None,
   Method → "PerimeterLength"] // N
```

{392., 938., 13.0365}

A list is returned, containing the perimeter, the area, and the complexity. The option `Method → "PerimeterLength"` ensures that we use the definition of perimeter described above. The option `Filter → None` is explained later.

For future reference, to distinguish it from variants that we consider, we call this the "raw" perimetric complexity. Thus the same result can be obtained with the option `Method → "Raw"`.

```
PerimetricComplexity[yi, Method → "Raw"] // N
```

{392., 938., 13.0365}

□ Using "PolygonalLength"

The second definition of the perimeter of a connected region in a binary digital image is to consider the perimeter pixel centers as points on a lattice, and to define the length as the sum of the sides of the polygon defined by those points. This estimate of the perimeter is obtained from `ComponentMeasurements` by using the measurement `"Poly`
`gonalLength"`.

```
cm = ComponentMeasurements[
   MorphologicalComponents[yi, CornerNeighbors → False],
   {"PolygonalLength", "Count"}, CornerNeighbors → False]
```

{1 → {39.6985, 98}, 2 → {159.439, 469}, 3 → {117.012, 371}}

Note that the measures of perimeter length are smaller than before.

And again complexity can be easily computed.

```
N[♯[[1]]^2 / ♯[[2]] / (4 Pi)] & @ Total[Last /@ cm]
```

8.47953

This variant of perimetric complexity is implemented with the following options.

```
PerimetricComplexity[yi, Filter → None,
   Method → "PolygonalLength"] // N
```

{316.149, 938., 8.47953}

Or, since `Method → "PolygonalLength"` is the default, the input can be simplified.

```
PerimetricComplexity[yi, Filter → None] // N
```

{316.149, 938., 8.47953}

■ Approximating Complexity of Continuous Shapes

We introduced the concept of perimetric complexity with a few continuous shapes, such as a square and a circle. In these cases, complexity is easily calculated, because we have simple formulas for the area and perimeter. It might be imagined that complexity of the continuous shape could be approximated by computing the complexity of a discrete sampled image, rendered from the shape. As we shall see, this assumption is not strictly correct.
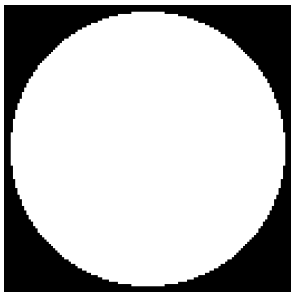
Consider the circular disk. As noted at the beginning, it has $C = 1$.

```
disk = Graphics[{White, Disk[]}, Background → Black]
```
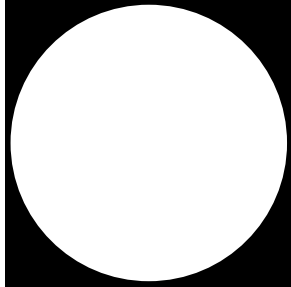

We set the foreground color to white, as is our convention. Now we consider an image rendered from the continuous shape. We render it into a certain size image.

```
size = 2^7 + 1;
diskimage = Image[ disk, "Bit", ImageSize → {1, 1} size,
    ColorSpace → "Grayscale"]
```


If we compute the complexity, we find that it is 62% too large relative to the continuous shape.

```
PerimetricComplexity[diskimage, Method → "Raw"] // N
```

```
{492., 11873., 1.62241}
```

The reason is that the sampled image is actually more complex than the continuous shape. Its contour is jagged, while that of the continuous shape is smooth. It might be imagined that this could be remedied by increasing the resolution of the rendering. Here we show that belief is misplaced. We render at several sizes and plot the results. Size has little effect, and the complexity never approaches the value of 1 corresponding to the continuous shape.

```
points = Table[
    {size = 2^k + 1,
     PerimetricComplexity[
        Image[disk, "Bit", ImageSize → {1, 1} size,
         ColorSpace → "Grayscale"]
        , Method → "Raw"][[3]]}
    , {k, 4, 12}];
ListLogLinearPlot[points, PlotRange → {1, Automatic},
 Frame → True, FrameLabel → {"Size", "Complexity"},
 Joined → True]
```
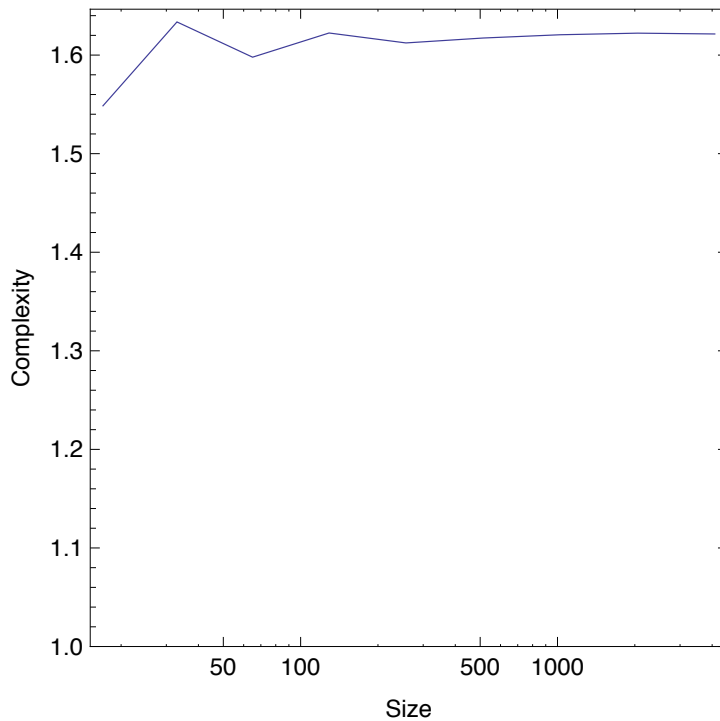

This problem can be somewhat ameliorated by using the `PolygonalLength` measure of perimeter length. Rather than the pure approach of measuring the exposed face of each foreground pixel, this measures the length of the contour that travels between the centers of those pixels.

```
PerimetricComplexity[diskimage, Filter → None,
 Method → "PolygonalLength"]
```

```
{403.647, 11873, 1.09203}
```

Now the difference is reduced to 9%. Here again, the reader might think that this difference could be reduced to zero by enlarging the resolution (number of pixels) in the rendered image, but this is not so. We leave that as an exercise for the reader. The error can

never be zero, because the path between pixel centers must always be vertical, horizontal, or diagonal, so it can never smoothly follow the true circular contour. Put another way, it has a higher fractal dimension than the circle, and thus greater length.

■ Pelli Algorithm

Pelli et al. [2] proposed a method for computing complexity that we quote here in full:

> The ink area is the number of 1's. To measure the perimeter we first replace the image by its outline. (We OR the image with translations of the original, shifted by one pixel left; left and up; up; up and right; right; right and down; down; and down and left; and then bit clear with the original image. This leaves a one-pixel-thick outline.) It might seem enough to just count the 1's in this outline image, but the resulting "lengths" are not Euclidean: diagonal lines have "lengths" equal to that of their base plus height. Instead we first thicken the outline. (We OR the outline image with translations of the original outline, shifted by one pixel left; up; right; and down.) This leaves a three-pixel-thick outline. We then count the number of 1's and divide by 3.

This method can be implemented using the `Dilation` function, as we show here. With `verbose == True`, it shows two images: the perimeter in red and the thickened perimeter. It returns the length of the perimeter, the area, and the complexity.

```
PelliMethod[image_, verbose_: False] := Module[
   {tmp0, tmp1, tmp2, tmp3, perimeter, area, result},
   tmp0 = ImagePad[image, 2];
   tmp1 = Dilation[tmp0, BoxMatrix[1]];
   tmp2 = ImageSubtract[tmp1, tmp0];
   tmp3 = Dilation[tmp2, CrossMatrix[1]];
   perimeter = Total[ImageData[tmp3], 2] / 3;
   area = Total[ImageData[tmp0], 2];
   result = {perimeter, area, perimeter^2 / area / (4 Pi)};
   If[verbose, Column[{GraphicsRow[
        {ColorCombine[{tmp1, tmp0, tmp0}], tmp3}], result}],
    result]
  ]
```

We apply this to the three-component Chinese character.

```
PelliMethod[yi, True] // N
```



```
{326.333, 938., 9.03463}
```

The raw method gives the following.

```
PerimetricComplexity[yi, Method → "Raw"] // N
```

{392., 938., 13.0365}

We see that the perimeter is substantially underestimated by the Pelli method in this case. This method has other limitations. It effectively assumes regions that are large in pixel dimensions. For example, consider the case of a single pixel object. As noted above, it has $C = 4/\pi \approx 1.273$. But the Pelli method yields a complexity value more than three times too large.

```
test = ImagePad[Image[{{1}}, "Bit"], 3]
```



```
PelliMethod[test, True] // N
```
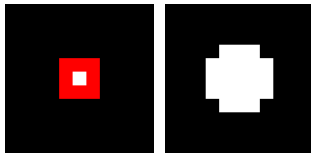

{7., 1., 3.8993}

■ Visual Perimetric Complexity

Much of the motivation for the use of perimetric complexity is the hope that it might provide an approximate measure of the visually perceived complexity of shapes. But this only makes sense if the shape is actually visible. Consider the difference between the continuous circular disk and its sampled image, as discussed above. They have different perimetric complexities, no matter how high the resolution of the sampled version. But of course, at a certain viewing distance, they are indistinguishable.

□ Filtering

Here we propose an approach to dealing with this problem. The idea is to first blur the image, in a manner consistent with visual blur, and then compute perimetric complexity. To make things simple, we use Gaussian blur, although this is not an accurate description of human visual blur. Later we show a more accurate form of blur. We begin with the example of a Chinese character jun (整).

We pad the image slightly, so that the blur is contained, and then magnify, to allow greater flexibility in the filtering. Then we blur the image, in this case by a Gaussian filter with a radius of 8 pixels.

```
tmp0 = ImagePad[jun, 5];
mag = 4;
tmp1 = ImageResize[tmp0, mag ImageDimensions[tmp0]];
tmp2 = GaussianFilter[tmp1, 8]
```


Then we binarize the image. Unfortunately, this requires some method of setting the threshold. Here we use a fixed threshold of 0.5. We use `ImageAdjust` to ensure that the filtered image is amplified to fill the grayscale range before thresholding.

```
tmp3 = Binarize[ImageAdjust[tmp2], 0.5]
```
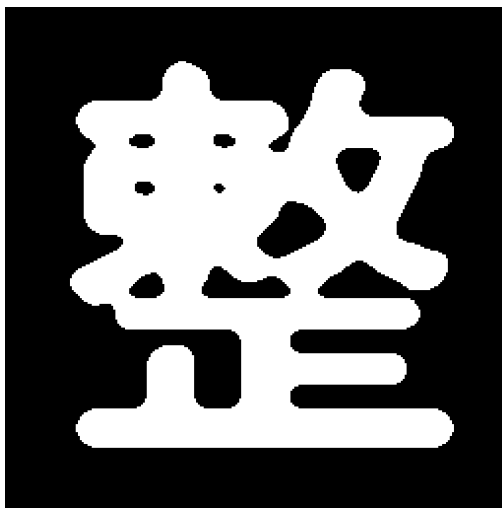
And because we imagine that the image is viewed at such a distance that the pixels are not resolved, we use the (default) `PolygonalLength` method.

```
PerimetricComplexity[tmp3, Filter → None]
```

```
{1944.82, 24 044, 12.5182}
```

We can compare this to the unfiltered raw complexity.

```
PerimetricComplexity[jun, Method → "Raw"] // N
```

```
{606., 1467., 19.9207}
```

The filtered version has substantially lower complexity, as we expect.

□ Visual Filtering Using a Gaussian

For the filtering to approximate visual blur, it must be based on the size of the original image and its distance from the viewer. Obviously, as the shape becomes smaller or farther from the observer, its details are more blurred, less visible, and contribute less to the visual complexity.

The challenge is to determine the appropriate value of the Gaussian filter radius for a given viewing distance. From measurements of visual sensitivity, we know that visual Gaussian blur has a standard deviation of about $B = 0.01549$ degrees of visual angle [11]. But we need to convert this into a radius in pixels. Recall that the image may be magnified by M before filtering. If we express the viewing distance V in terms of pixels (before magnification), then the size of each pixel S (after magnification) is approximately

$$S = \frac{57.3}{MV} \text{ deg.} \tag{7}$$

The constant 57.3 is an approximation to $1 / \arctan(1°)$.

```
1. / ArcTan[1 °]
```

```
57.3016
```

By default, the radius is twice the standard deviation. So the filter radius should be

$$R = \frac{2B}{S} = \frac{2BMV}{57.3} \text{ pixel.} \tag{8}$$

Consider an example. Suppose that our Chinese character image is displayed on a typical computer screen with a resolution of 72 pixels/inch, and viewed at a distance of 48 inches.

```
V = 48 × 72
```

```
3456
```

Recall the value of B.

```
B = 0.01549;
```

Suppose further that the magnification is 4.

```
M = 4;
```

Then we know R.

```
R = 2 B M V / 57.3
```

```
7.47413
```

We now proceed with the steps outlined above for filtering.

```
tmp0 = ImagePad[jun, 1];
tmp1 = ImageResize[tmp0, M ImageDimensions[tmp0]];
tmp2 = GaussianFilter[tmp1, R];
tmp3 = Binarize[ImageAdjust[tmp2], 0.5]
PerimetricComplexity[tmp3, Filter → None]
```



```
{1987.85, 23 932, 13.1395}
```

The preceding steps of padding, magnification, and filtering are built into the function PerimetricComplexity, as shown below. With Verbose → True, the function also shows the original, the filtered version, the binarized version, and the original (in red) with the perimeter of the filtered version (in white within the original and aqua outside the original).

```
PerimetricComplexity[jun, Magnification → 4,
 Filter → "Gaussian", ViewingDistance -> 48 × 72,
 Verbose → True]
```
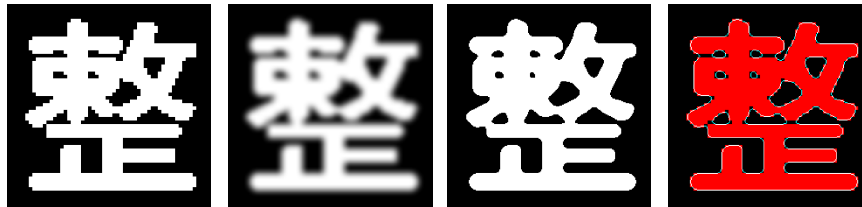

{1987.85, 23 932, 13.1395}

The Gaussian is parameterized by a scale in degrees of visual angle. The default value is 2.33/60 degrees. The user can experiment with different values via the option GaussianScale.

☐ Accurate Visual Filtering with Sech

Visual blur is more accurately represented with filters other than a Gaussian. In one simple form, the kernel is a sech (hyperbolic secant) function [14, 15]. This filter can be selected with an option (Filter → "Sech", the default) in the function Perime tricComplexity.

```
PerimetricComplexity[jun]
```

{1016.14, 5977, 13.7472}

The hyperbolic secant is parameterized by a scale in degrees of visual angle. The default value is 2.16/60 degrees [14, 15]. The user can experiment with different values via the option SechScale.

☐ Accurate Visual Filtering with an Arbitrary Point Spread Function

In real human eyes, blur results not only from low-order aberrations such as defocus and astigmatism, but also from higher-order aberrations. In this example, we use a blur function defined by an array of values representing the filter kernel. This example is an actual estimate of the point spread function for an individual human observer, as measured using a device called a wavefront aberrometer that includes both low-order and high-order aberrations [16].

```
ImageAdjust[Image[psf]]
```
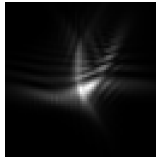

We can use this blur kernel by supplying it directly to the `Filter` option.

```
PerimetricComplexity[jun, Verbose → True, Filter → psf,
 Magnification → 3]
```



```
{851.855, 14 959, 3.86028}
```

In the above calculation we used a magnification of 2. We do not address this topic in detail, but when a kernel is supplied, the pixels of the kernel and of the magnified image must be of the same size, in degrees of visual angle, for the filter to be accurate. The size of the magnified pixels in degrees depends upon the viewing distance and the magnification, as described above.

☐ Magnification Parameter

The `Magnification` parameter should be set with a value that ensures that a filter kernel, if present, has enough pixels in it to adequately represent the filter. For the `Sech` filter, we generally want a width of least three times the scale, and at least 8 pixels. This means that for the magnification,

$$M \geq \frac{8 \times 57.3}{3 \, V \, B} \, . \tag{9}$$

This rule is effectively implemented by the default `Magnification → Automatic` option.

☐ Binarization

After applying visual blur, it is necessary to binarize the image before calculating the perimetric complexity. There are many ways to binarize an image and *Mathematica* offers many of them as options. The simplest is to use a fixed threshold. Since our images are ini-

tially defined as 0 or 1, a natural choice of threshold is 0.5. One drawback of this choice is that as images become severely blurred, no pixels may remain that exceed the threshold. From a perceptual point of view, the mean might appear a reasonable choice. As the image blurs, all pixels revert towards the mean, but some always remain above the mean until a uniform image is reached. A drawback of the mean is that it is influenced by the area of the background. For this reason, we adopt the fixed value of 0.5 as the default threshold for binarization.

We should acknowledge that a more valid visual thresholding scheme might be devised that better reflects our perceptual segregation of areas into light and dark. This is a topic for future research.

It should also be acknowledged that for severely blurred images, considerable grayscale information remains that is lost in binarization. Thus we should question whether perimetric complexity is an appropriate measure for such images.

To illustrate that problem, we show an example of a character viewed at 10 feet on a display with 100 pixels/inch. Note that the blurred image displays internal grayscale structure that is not conveyed by the binarized version.

```
PerimetricComplexity[jun, ViewingDistance → 10 × 12 × 100,
 Verbose → True]
```
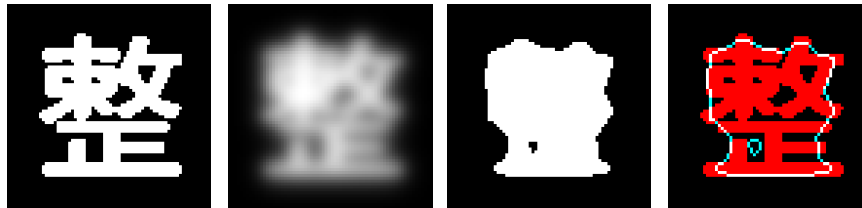


```
{195.782, 1706, 1.78795}
```

☐ Viewing Distance

For a given shape, complexity declines with viewing distance as a result of visual blur. Here we illustrate this effect with an example Chinese character. First we find the raw complexity.

```
raw = PerimetricComplexity[jun, Method → "Raw"][[3]] // N
```

```
19.9207
```

Now we compute complexity for viewing distances ranging from 3 inches to 10 feet, assuming a display with a resolution of 100 pixels/inch. For reference, we show as red lines the raw complexity and the theoretical limit of 1 (a circular disk). As it should, the visual complexity proceeds from one of these limits to the other as distance increases.

```
{min, max} = {3, 10 × 12};
points =
   {#, PerimetricComplexity[jun, ViewingDistance → # 100][[
        3]]} & /@ Range[min, max, 3] // N;
ListPlot[points, Frame → True, Joined → True,
 Epilog → {Red, Line[{{min, raw}, {max, raw}}],
   Line[{{min, 1}, {max, 1}}]},
 PlotRange → {{0, Automatic}, {0, 22}},
 PlotStyle → Blue, Axes → False,
 FrameLabel → {"Distance (inches)", "Complexity"}]
```
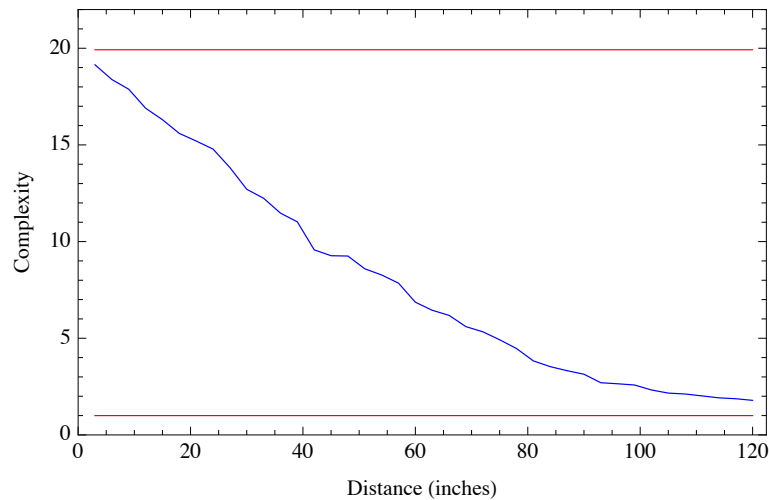

At very small viewing distances (in pixels) the blur has little effect on each pixel, so the visual complexity approaches the raw value. As a rule of thumb, this asymptote is approached when the size of each pixel exceeds 1/4 degree.

It is reassuring to know that the algorithm does approach the correct asymptote as distance increases. Here we show the intermediate images for a case of extreme blur (distance = 30 feet).

```
PerimetricComplexity[jun, Verbose → True,
 ViewingDistance → 30 × 12 × 100]
```
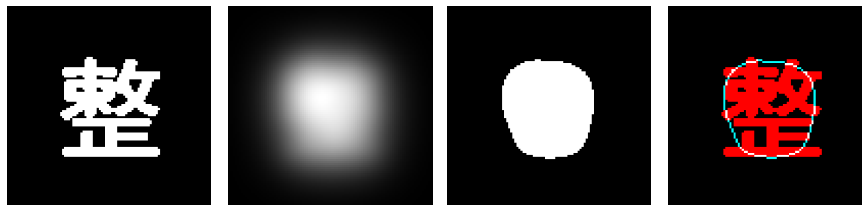


```
{158.811, 1801, 1.11439}
```

But a note of caution is warranted. Consider the effect of distance on our other example character yi ( ). We show it here along with the previous plot. Note that the curves cross, so that at large distances (large blurs) the "simpler" character becomes the more complex of the two.

```
points2 =
  {#, PerimetricComplexity[yi, ViewingDistance → # 100][[
      3]]} & /@ Range[min, max, 3] // N;
ListPlot[{points, points2}, Frame → True, Joined → True,
 Epilog → {Red, Line[{{min, raw}, {max, raw}}],
   Line[{{min, 1}, {max, 1}}]]},
 PlotRange → {{0, Automatic}, {0, 22}},
 PlotStyle → {Blue, Green}, Axes → False,
 FrameLabel → {"Distance (inches)", "Complexity"}]
```
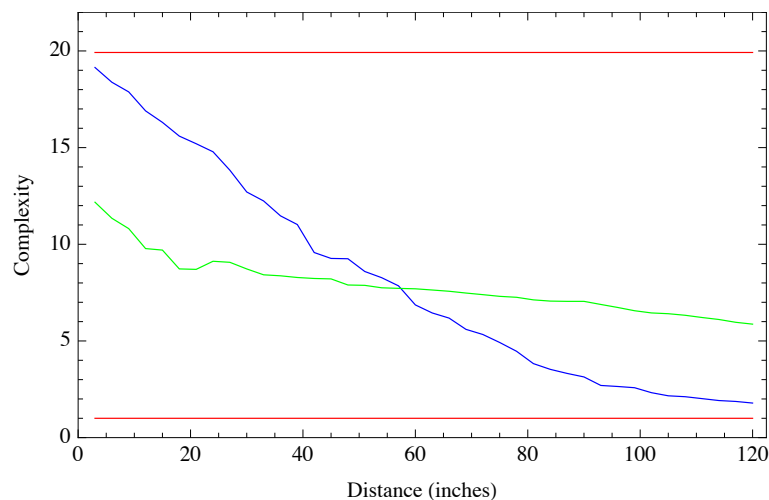

This makes sense, since the densely packed features of the "complex" character blur onto each other, while the more widely separated features of the "simple" character remain distinct. This is illustrated in the following, which shows the intermediate images for the two characters when highly blurred (distance = 10 feet).

```
PerimetricComplexity[#, ViewingDistance → 10 × 12 × 100,
    Verbose → True] & /@ {jun, yi}
```



```
{195.782, 1706, 1.78795}
```

```
{274.551, 1022, 5.8693}
```
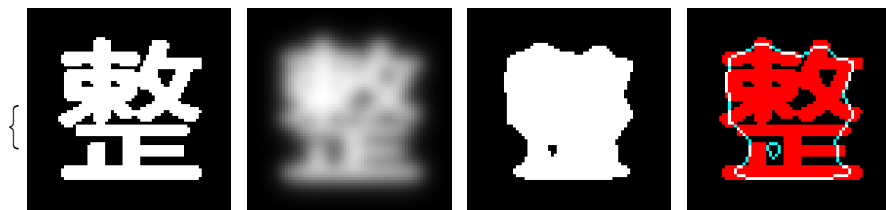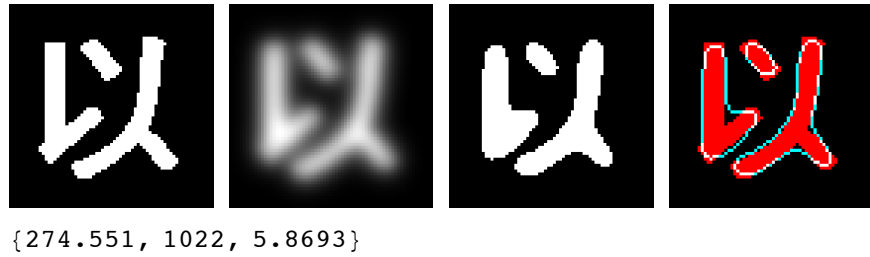
But the conclusion we must draw is that even the *relative* complexity of different shapes cannot be known without specifying the viewing distance.

We use the quantity `ViewingDistance → 48 / ArcTan[1°]` as a default value. This corresponds to a visual resolution of 48 pixels/degree. It is a commonly encountered resolution, about that achieved by a display with 100 pixels/inch viewed from 27 inches. But in actual use, it is advised to use the actual viewing distance rather than relying on this default.

Here is a `Manipulate` that lets you experiment with different viewing distances. The complexity and the diagnostic images are shown.

```
Manipulate[
 Column[{Text@Row[{"complexity = ", #[[3]] // N}],
      GraphicsRow[#[[4]], ImageSize → 400]},
     Alignment → Center] &@
  PerimetricComplexity[jun, ReturnImages → True,
    ViewingDistance → inches 100],
 {inches, 1, 200, Appearance → "Labeled"},
 SaveDefinitions → True]
```
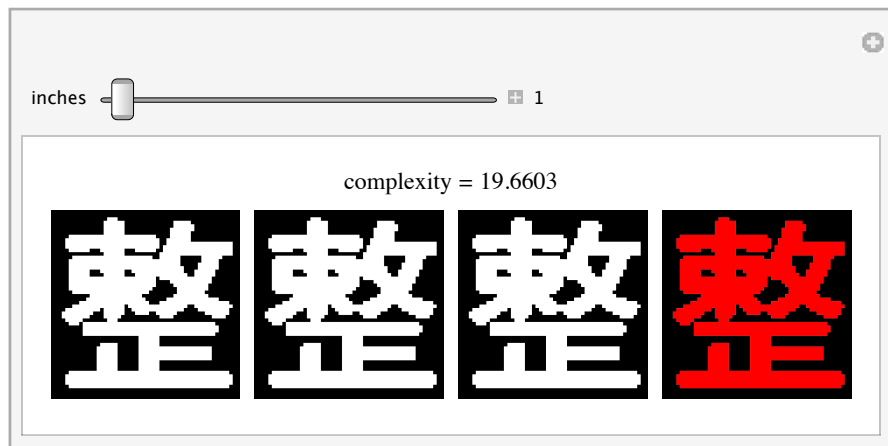
■ Recommended Practice

In general, we recommend using the function with default parameters, shown here as a reminder.

    Options[PerimetricComplexity]

    {Magnification → Automatic, Pad → Automatic, Verbose → False,
     Threshold → 0.5, Method → PolygonalLength, Filter → Sech,
     Normalized → True, ViewingDistance → 2750.48,
     MorphologicalOperators → True, SechScale → 0.036,
     GaussianScale → 0.0388333, ReturnImages → False}

The only option that should be specified for the typical use of this function is `ViewingDistance`. This specifies the distance from the eye to the image in pixels. The default is `ViewingDistance → 2750.48`, consistent with a display having 96 pixels/inch viewed at 28.65 inches (equal to a display with 48 pixels/degree of visual angle).

It is difficult to imagine a case in which vision, in some form, would not be used to view the shape in question. If such a case arises, however, the "raw" complexity can be measured with the following options.

    SetOptions[PerimetricComplexity, Method -> "Raw"]

    {Magnification → Automatic, Pad → Automatic,
     Verbose → False, Threshold → 0.5, Method → Raw,
     Filter → Sech, Normalized → True, ViewingDistance → 2750.48,
     MorphologicalOperators → True, SechScale → 0.036,
     GaussianScale → 0.0388333, ReturnImages → False}

This is also an appropriate measure when the pixels are very large (larger than 1/4 degree).

When trying to approximate the raw complexity of a continuous shape by means of a sampled representation (e.g., a circle via an image of a circle), the following options yield the lowest error. But as noted above, the error is still significant.

    SetOptions[PerimetricComplexity, Filter → None]

    {Magnification → Automatic, Pad → Automatic,
     Verbose → False, Threshold → 0.5, Method → Raw,
     Filter → None, Normalized → True, ViewingDistance → 2750.48,
     MorphologicalOperators → True, SechScale → 0.036,
     GaussianScale → 0.0388333, ReturnImages → False}

■ Examples

We conclude with two examples of the application of `PerimetricComplexity`.

The first example is a set of three binary images. Below each image we print the complexity.

```
pictures = {cat, horse, family};


Grid[
 {pictures,
  Text@Style[NumberForm[Last[N@PerimetricComplexity[#]],
        3], 14] & /@pictures}
 , ItemSize → 10, Spacings → 2]
```
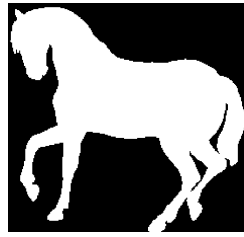

3.95   8.62   22.1

The second example is an array of characters. This array was created as part of an experiment on the effect of complexity on visual acuity [6]. The first row is the Sloan letters, a well-known set of letter acuity targets [5]. The remaining six rows are sets of Chinese characters selected so as to be of equal complexity within a row, but increasing in complexity from row to row [6]. The metric of complexity used for selection was different from that developed in this article.

```
GraphicsGrid[chararray]
```

We first apply the function to each character and plot the results, with a different curve for each set.

```
c = Map[Last[PerimetricComplexity[#]] &, chararray, {2}];
ListLinePlot[c, PlotRange → {{.7, 10.3}, {0, 18}},
 FrameLabel → {None, "Complexity"}]
```
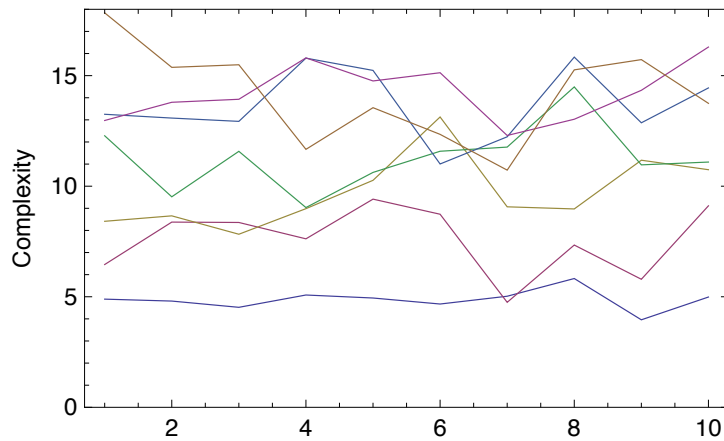

The plot shows that there is considerable variation in each set. If we take the mean of each set we see a progression in complexity, except for the last three sets. In the next graphic we show at the bottom an exemplar from each set.

```
ListLinePlot[Mean /@ c, Mesh → Full,
 PlotRange → {{.7, 7.3}, {0, 15}},
 Epilog →
  {Table[Inset[chararray[[k, 1]], {k , .1}, Scaled[{.5, 0}],
     .6], {k, 7}]},
 FrameLabel → {None, "Complexity"}]
```
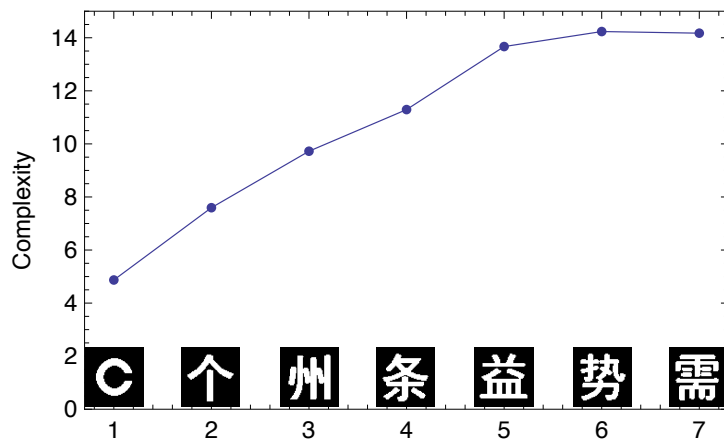
■ Conclusion

We have illustrated several different methods for computing the perimetric complexity of binary digital images. These methods differ in how they compute the perimeter and in whether the image is blurred and binarized before the complexity calculation. We have introduced the concept of visual perimetric complexity and argued that in general it requires blur for a sensible estimate. We have described several methods of implementing visual blur.

The computed value of visual perimetric complexity depends somewhat upon details of the calculation, such as the presumed magnitude and nature of visual blur, and the binarization threshold. In this regard, we have proposed a set of standard default settings and procedures for calculation of visual perimetric complexity.

We have also made the observation that visual perimetric complexity cannot be estimated without specifying the resolution of the display and the viewing distance. As a general rule, the visual perimetric complexity approaches the raw complexity when the width of a pixel exceeds 1/4 degree of visual angle.

■ Appendix

□ Functions

Here we define several functions based on the derivations presented above.

■ *PerimetricComplexity*

```
Options[PerimetricComplexity] =
  {Magnification → Automatic, Pad → Automatic,
   Verbose → False, Threshold → 0.5,
   Method → "PolygonalLength", Filter → "Sech",
   Normalized → True, ViewingDistance → 48 / ArcTan[1.°],
   MorphologicalOperators → True, SechScale → 2.16 / 60,
   GaussianScale → 2.33 / 60, ReturnImages → False};
```

```
PerimetricComplexity::usage =
  "PerimetricComplexity[image_,opts___Rule] computes
    the perimetric complexity of a binary image.
    Perimetric complexity is defined as the square
    of the sum of inner and outer perimeters
    divided by the foreground area, divided by
    4π. By default, foreground pixels are white
    (1). The function returns a list: {perimeter,
    area, complexity}. Optionally, the image can
    be filtered and thresholded before calculation
    of complexity, to provide a better estimate
    of the visual complexity of the image. Options
    available are Pad (width of padding to add
    around image), Magnification (factor by which
    to magnify each pixel before calulation of
    complexity), Method (\"PerimeterLength\",
    \"PolygonalLength\", or \"Raw\", the method
    used to compute the perimeter), Threshold (a
    numerical threshold, or the method to use in
    thresholding operations after filtering), and
    Filter (None: no filter, \"Gaussian\" for
    Gaussian filter, \"Sech\" for a hyperbolic
    secant filter, or an array supplied directly
    as the filter kernel. For the Gaussian and
    the Sech, the size of the filter is determined
    by the ViewingDistance option), ViewingDistance
    (in pixels, used to determine the radius of
    Gaussian and Sech filters), MorphologicalOperators
    (whether to use Mathematica's built-in
    operators), Normalized (whether to normalize
    complexity by the value for a disk, 4π). When
    the Method->\"Raw\" option is used, the filter
    is set to None.";
```

```
PerimetricComplexity[image_, opts___Rule] := Module[
  {tmp, tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, plength,
   area, mag, radius, pad, verbose, fdim, pc, norm,
   threshold, method, filter, vdist, pixelsperdegree,
   sechpixels, sechdegrees, morpho, sechscale, gscale,
   anglescale, rimages, images, result},
  {mag, pad, verbose, threshold, method, filter, norm,
    vdist, morpho, sechscale, gscale, rimages} =
   {Magnification, Pad, Verbose, Threshold, Method,
      Filter, Normalized, ViewingDistance,
      MorphologicalOperators, SechScale, GaussianScale,
      ReturnImages} /. {opts} /.
    Options[PerimetricComplexity];

  If[method === "Raw",
   (filter = None; method = "PerimeterLength")];

  anglescale = ArcTan[1. Degree];
  If[mag === Automatic , mag = Switch[filter
     , None, 1
     , _?ListQ, 1
     , "Gaussian", Ceiling[2. / (vdist anglescale gscale)]
     , "Sech", Ceiling[3. / (vdist anglescale sechscale)]
    ]];
  pixelsperdegree = mag vdist anglescale;
  sechpixels = Ceiling[pixelsperdegree 3 sechscale];
  If[pad === Automatic,
   pad = If[filter === None, 2,
     Ceiling[pixelsperdegree sechscale]]];
  If[EvenQ[sechpixels], sechpixels++];
  sechdegrees = sechpixels / pixelsperdegree;
  radius = 2 gscale pixelsperdegree / Sqrt[2 Pi];
  tmp0 = Binarize[image];
  tmp1 =
   If[mag > 1, ImageResize[tmp0, mag ImageDimensions[tmp0]],
    tmp0];
  tmp2 = ImagePad[tmp1, pad];
  tmp4 = ImageAdjust @ Switch[filter
     , None, tmp2
     , _?ListQ, ImageConvolve[tmp2, filter]
     , "Gaussian", GaussianFilter[tmp2, radius]
     , "Sech",
     ImageConvolve[tmp2, SechKernel2D[sechpixels {1, 1},
       sechdegrees {1, 1}, sechscale]]
    ];
```

```
tmp5 = If[ImageType[tmp4] === "Bit", tmp4,
    If[NumericQ[threshold], Binarize[tmp4, threshold],
     Binarize[tmp4, Method → threshold]]];

{plength, area} = If[morpho,
    Total[
     Last /@
      (tst = ComponentMeasurements[
         MorphologicalComponents[tmp5], {method, "Count"},
         CornerNeighbors → False])]
    ,
    {PerimeterLength[tmp5,
      Switch[method, "PerimeterLength", PixelBorderLength,
       "PolygonalLength", PixelPathLength]],
     Total[ImageData[tmp5], 2]}];
(*Print["mag = ",mag, " ppd = ",pixelsperdegree,
  " sechpixels = ",sechpixels," pad = ", pad];*)
tmp6 = MorphologicalPerimeter[tmp5,
    CornerNeighbors → False];
images = {tmp2, tmp4, tmp5,
    ColorCombine[{tmp2, tmp6, tmp6}]};
result =
  ({plength, area, (plength^2 / area) / If[norm, 4 Pi, 1]} //
     If[rimages, Append[#, images], #] &);
If[verbose,
 Column[{GraphicsRow[images, ImageSize → 400], result}],
 result]
]
```

Here is an example.

```
PerimetricComplexity[jun, Verbose → True] // N
```



```
{1016.14, 5977., 13.7472}
```

■ *PerimeterLength*

```
PerimeterLength::usage =
  "PerimeterPathLength[image_,verbose_:False] Given
    a binary image that represents a set of
    perimeters, defined by connected white pixels,
    compute the total length of the perimeters.
    The function first locates all foreground
    pixels. It then extracts the 3 x 3 neighborhood
    of each, and using PixelPathLength, it computes
    half the distance to each of the pixels' two
    nearest foreground neighbors. The perimeter
    is the sum of all these distances. If verbose
    is true, it shows a tally of the neighborhoods
    and their corresponding path lengths.";


PerimeterLength[image_, method_: PixelBorderLength,
  verbose_: False] :=
 Module[{perim, positions, neighborhoods, tally, result},
  perim = Perimeter[image];
  positions = Position[ImageData[perim], 1];
  neighborhoods =
   ImageTake[Switch[method, PixelPathLength, perim,
       PixelBorderLength, image],
      Sequence @@ Transpose[{# - 1, # + 1}]] & /@ positions;
  tally = Transpose[Tally[neighborhoods]];
  result = Total[tally[[2]] (method /@ tally[[1]])];
  If[verbose,
   Column[{
     Column[{
       Row [{"pixels = ", Length[positions]}];
       TableForm[Transpose[{tally[[1]], tally[[2]],
          method /@ tally[[1]]}]]
      }],
     result
    }],
   result]
 ]
```

Here is an example.

```
test = ImagePad[Image[{{0, 1}, {1, 1}}, "Bit"], 1]
```

```
PerimeterLength[test, PixelBorderLength, False]
```

8

```
PerimeterLength[test, PixelBorderLength, True]
```


1   3

1   3

1   2

8

▪ *PixelPathLength*

```
PixelPathLength::usage =
  "PixelPathLength[image_] Given a 3×3 binary image,
     finds the coordinates of white pixels not at
     the center, and returns their mean distance
     from the center.";
```

```
PixelPathLength[image_] := Module[{tmp1, tmp2, tmp3},
  tmp1 = DeleteCases[Position[ImageData[image], 1], {2, 2}];
  tmp2 = (# - {2, 2}) & /@ tmp1;
  tmp3 = Sort[Norm /@ tmp2];
  Mean[Take[tmp3, 2]]
 ]
```

Here is an example.

```
tst = Image [{{0, 0, 0}, {0, 1, 1}, {1, 0, 0}}, "Bit"]
PixelPathLength[tst]
```


$$\frac{1}{2} \left(1 + \sqrt{2} \right)$$

■ *PixelBorderLength*

```
PixelBorderLength::usage =
  "PixelBorderLength[image_] Given a 3×3 binary
    image, counts the number of black neighbor
    (4-connected) pixels. This is a measure of
    the length of the exposed border of the
    foreground (white) pixel.";
```

```
PixelBorderLength[image_] :=
 Total[1 - Flatten[ImageData[image]][[{2, 4, 6, 8}]]]
```

Here is an example.

```
Framed[tst = Image [{{0, 0, 0}, {0, 1, 1}, {1, 1, 1}}, "Bit"]]
PixelBorderLength[tst]
```


2

■ *SechKernel2D*

```
SechKernel2D::usage =
  "SechKernel2D[samples_,degrees_,scale_] Computes
    a convolution kernel defined by a hyperbolic
    secant (sech) function of distance from the
    origin. samples is a list {height, width} of
    dimensions of the kernel in pixels, and degrees
    is a list of the corresponding dimensions in
    degrees of visual angle. scale defines the
    width of the kernel in degrees. The origin is
    defined as Floor[samples/2]. The kernel is
    normalized.";
```

```
SechKernel2D[samples_, degrees_, scale_] :=
 (# / Total[#, 2]) &@
  Sech[
   Pi Array[N[Norm[{##} degrees / samples]] &, samples,
      -Floor[samples / 2]] / scale]
```

Here is an example.

```
ImageAdjust[Image[SechKernel2D[{32, 32}, {1, 1}, .4]]]
```


■ *Perimeter*

```
Perimeter[image_] := Module[{padded, positions},
  padded = ImagePad[image, 1];
  positions = Select[Position[ImageData[padded], 1],
    PixelBorderLength[ImageTake[padded,
        Sequence @@ Transpose[{# - 1, # + 1}]]] > 0 &];
  ImagePad[
   Image[ReplacePart[ImageData[padded] 0,
     # → 1 & /@ positions], "Bit"], -1]]
```

□ **Initializations**

```
SetOptions[Graphics, ImageSize → 128];

SetOptions[ListLinePlot, Axes → False,
  BaseStyle → {10, FontFamily → "Helvetica",
    AbsolutePointSize[4]}, Frame → True, ImageSize → 300];

SetOptions[LogLogPlot, AspectRatio → Automatic,
  Axes → False, BaseStyle → {10, FontFamily → "Helvetica"},
  Frame → True, ImageSize → 300];

SetOptions[ListLogLinearPlot, AspectRatio → Automatic,
  Axes → False, BaseStyle → {10, FontFamily → "Helvetica"},
  Frame → True, ImageSize → 300];

SetOptions[ListLogLinearPlot, AspectRatio → 1,
  Axes → False, BaseStyle → {10, FontFamily → "Helvetica"},
  Frame → True, ImageSize → 300];
```

The next closed cell contains the definition of `psf`, a 64×64 array of numbers.

- ***psf***

- ***characters***

`jun =` 整 `;`

`yi =` 以 `;`

`chararray =`

{{ C , D , H , K , N ,

O , R , S , V , Z },

{ 个 , 文 , 么 , 山 , 少 ,

之 , 十 , 力 , 人 , 天 },

{ 州 , 月 , 占 , 只 , 无 ,
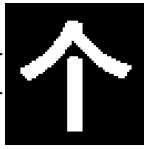
区 , 以 , 世 , 风 , 任

区 以, 世, 风, 任},

{条, 名, 抓, 电, 各,

好, 取, 医, 非, 多},

{益, 售, 很, 资, 建,

做, 学, 深, 张, 事},

{势, 数, 想, 续, 验,

率, 常, 群, 策, 领},

{需, 感, 题, 路, 德,

增, 解, 就, 赛, 整}};

```
cat =              ;
```



```
horse =            ;
```



```
family =           ;
```
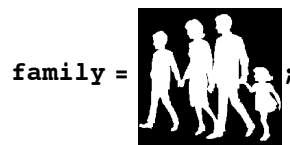

☐ Computing Complexity without Using *Mathematica*'s Morphological Operators

Some readers may wish to write programs in other languages to compute complexity. For that reason we provide an explanation here of how to compute complexity without using 3×3 morphological operators. This amounts to finding alternate methods for computing the perimeter. These are incorporated in the function `PerimeterLength`, defined above, and derived below. These functions can be exercised from within `PerimetricComplexity` by selecting the option `MorphologicalOpera⁚ tors → False`. This is useful mainly for testing.

Consider the following binary image.

```
jun
```


The foreground area is easily obtained, since it is just the sum of all the white pixels.

```
Total[ImageData[jun], 2]
```

```
1467
```

■ *Using "PerimeterLength"*

As noted above, there are two definitions of the perimeter of a binary digital image. The first consists of the sum of the exposed pixel faces. To count the exposed faces we use the function `PixelBorderLength`. This takes a binary image of dimensions {3, 3} and counts the number of black 4-connected neighbors of the center pixel.

Here is an example.

```
Framed[test = Image [{{0, 0, 0}, {0, 1, 1}, {1, 1, 1}}, "Bit"]]
```
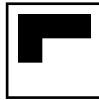


```
PixelBorderLength[test]
```

2

In this example, the center pixel has only two black neighboring pixels.

The total perimeter can be obtained by applying this function to the {3, 3} neighborhood of every white pixel in the image. Pixels in the interior return a value of 0, so only perimeter pixels contribute.

To implement this idea, we first identify the positions of all the white pixels.

```
positions = Position[ImageData[jun], 1];
```

Next we extract all the {3, 3} neighborhoods.

```
neighborhoods =
  ImageTake[jun, Sequence @@ Transpose[{# - 1, # + 1}]] & /@
   positions;
```

We can look at the first five.

```
Framed /@ neighborhoods[[Range[5]]]
```


We can also compute the pixel border length of the first five.

```
PixelBorderLength /@ neighborhoods[[Range[5]]]
```
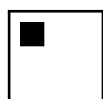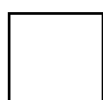
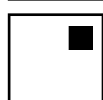{2, 1, 2, 2, 0}

In a large complex image, the same neighborhood might occur many times, so we perform a tally.

```
tally = Tally[neighborhoods];
```

We can take a look at the tally along with the pixel border length for each type of neighborhood. Note that 850 cases consist of all white, drawn from the interior of the foreground, with 0 border length. Here we just look at the first 10 elements of the tally.

```
{Framed[#[[1]]], #[[2]], PixelBorderLength[#[[1]]]} & /@
   tally // Take[#, 10] & // TableForm
```


	10	2
	123	1
	8	2
	12	2
	29	0
	850	0
	24	0
	15	1
	4	2
	1	2

The total perimeter length is the sum of all of the border lengths for all the collected neighborhoods. We use the tallied neighborhoods, so that the pixel border length of each type of neighborhood is computed only once.

```
Total[(#[[2]] PixelBorderLength[#[[1]]] & /@ tally)]
```

```
606
```

This is the complexity.

```
606^2 / 1467 / (4 Pi) // N
```

```
19.9207
```

■ *Using "PolygonalLength"*

The second definition of the length of the perimeter is the sum of sides of the polygon defined by the perimeter pixels considered as points in a lattice.

To extract the perimeter, we use a new function `Perimeter`, defined above. This duplicates a built-in *Mathematica* function. We verify that they yield the same results.

```
perimeter = Perimeter[jun]
```



```
perimeter2 = MorphologicalPerimeter[jun,
   CornerNeighbors → False]
```

```
ColorCombine[{jun, perimeter, perimeter2}]
```


We identify the positions of all of the pixels in the perimeter.

```
Length[positions = Position[ImageData[perimeter], 1]]
```

```
498
```

We extract all of the 3×3 neighborhoods of pixels in the perimeter.

```
neighborhoods =
  ImageTake[perimeter,
    Sequence @@ Transpose[{# - 1, # + 1}]] & /@ positions;
```

Then we use a new function `PixelPathLength`, which looks at a 3×3 binary neighborhood, identifies the two closest white pixels not at the center, and finds the distances from their centers to that of the central pixel. That is the path length corresponding to that neighborhood.

We apply this to all the perimeter pixels and add up the results.

```
Total[PixelPathLength /@ neighborhoods] // N
```

```
547.084
```

We can verify this is the same perimeter length obtained from `PerimetricCom‑plexity` using *Mathematica*'s morphological operators.

```
PerimetricComplexity[jun, Filter → None,
  Method → "PolygonalLength"] // N
```

```
{547.084, 1467., 16.2356}
```

■ Acknowledgments

■ References

[1] F. Attneave and M. D. Arnoult, "The Quantitative Study of Shape and Pattern Perception," *Psychological Bulletin*, **53**(6), 1956 pp. 452–471. psycnet.apa.org/journals/bul/53/6/452.

[2] P. V. Sankar and E. V. Krishnamurthy, "On the Compactness of Subsets of Digital Pictures," *Computer Graphics and Image Processing*, **8**(1), 1978 pp. 136–143. www.sciencedirect.com/science/article/pii/S0146664X78800215.

[3] S. Ullman, "The Visual Analysis of Shape and Form," *The Cognitive Neurosciences* (M. S. Gazzaniga, ed.), Cambridge, MA: MIT Press, 1995 pp. 339–350.

[4] R. Montero and E. Bribiesca, "State of the Art of Compactness and Circularity Measures," *International Mathematical Forum*, **4**(27), 2009 pp. 1305–1335. www.m-hikari.com/imf-password2009/25-28-2009/index.html.

[5] D. G. Pelli, C. W. Burns, B. Farell, and D. C. Moore-Page, "Feature Detection and Letter Identification," *Vision Research*, **46**(28), 2006 pp. 4646–4674. www.psych.nyu.edu/pelli/pubs/pelli2006letters.pdf.

[6] J.-Y. Zhang, T. Zhang, F. Xue, L. Liu, and C. Yu, "Legibility Variations of Chinese Characters and Implications for Visual Acuity Measurement in Chinese Reading Population," *Investigative Ophthalmology & Visual Science*, **48**(5), 2007 pp. 2383–2390. www.iovs.org/content/48/5/2383.short.

[7] A. B. Watson and A. J. Ahumada, Jr., "Modeling Acuity for Optotypes Varying in Complexity," presentation given at *The Association for Research in Vision and Ophthalmology Conference (ARVO 2010)*, Ft. Lauderdale, FL. abstracts.iovs.org//cgi/content/abstract/51/5/5174.

[8] A. Rusu and V. Govindaraju, "The Influence of Image Complexity on Handwriting Recognition," in *Proceediings of the Tenth International Workshop on Frontiers in Handwriting Recognition (IWFHR 2006)*, La Baule (France). hal.inria.fr/inria-00112666/.

[9] S. Garrod, N. Fay, J. Lee, J. Oberlander, and T. MacLeod, "Foundations of Representation: Where Might Graphical Symbol Systems Come From?," *Cognitive Science*, **31**(6), 2007 pp. 961–987. onlinelibrary.wiley.com/doi/10.1080/03640210701703659/abstract.

[10] M. Chew and H. Baird, "BaffleText: A Human Interactive Proof," in *Proceedings of the IS&T/SPIE Document Recognition and Retrieval Conference X (DRR X)*, Santa Clara, CA, 2003 pp. 305-316. www.imaging.org/IST/store/epub.cfm?abstrid=22585.

[11] B. Biggio, G. Fumera, I. Pillai, and F. Roli, "Image Spam Filtering Using Visual Information," in *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP 2007)*, Modena, Italy pp. 105–110. ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4362765.

[12] G. Fumera, I. Pillai, F. Roli, and B. Biggio, "Image Spam Filtering Using Textual and Visual Information," in *Proceedings of the MIT Spam Conference 2007*, Cambridge, MA. projects.csail.mit.edu/spamconf/SC2007/MIT_Spam_Conf _ 2007_Papers.tar.gz.

[13] R. Courant, *Differential and Integral Calculus*, Vol. 1, 2nd ed. (E. J. McShane, trans.), London: Blackie & Son Limited, 1937.

[14] A. B. Watson and A. J. Ahumada, Jr., "A Standard Model for Foveal Detection of Spatial Contrast," *Journal of Vision*, **5**(9), 2005 pp. 717–740. journalofvision.org/5/9/6.

[15] A. B. Watson and A. J. Ahumada, "Blur Clarified: A Review and Synthesis of Blur Discrimination," *Journal of Vision*, **11**(5), 2011. journalofvision.org/11/5/10.

[16] L. N. Thibos, X. Hong, A. Bradley, and X. Cheng, "Statistical Variation of Aberration Structure and Image Quality in a Normal Population of Healthy Eyes," *Journal of the Optical Society of America A: Optics, Image Science, and Vision*, **19**(12), 2002 pp. 2329–2348. www.opticsinfobase.org/abstract.cfm?URI=josaa-19-12-2329.

About the Author

Andrew B. Watson is the Senior Scientist for Vision Research at NASA. He is editor-in-chief of the *Journal of Vision* (journalofvision.org). He is the author of over 150 scientific papers and four patents. He is a Fellow of the Optical Society of America, the Association for Research in Vision and Ophthalmology, and the Society for Information Display.

Andrew B. Watson
MS 262-2
NASA Ames Research Center
Moffett Field, CA 94035
andrew.b.watson@nasa.gov