GERT Manual

THE GROUPING ELEMENTS RENDERING TOOLBOX (v1.30)

Maarten Demeyer
Bart Machilsen

July 6, 2015

Laboratory of Experimental Psychology
University of Leuven (K.U. Leuven)
Tiensestraat 102 bus 3711
3000 Leuven BELGIUM
GERT@gestaltrevision.be
http://www.gestaltrevision.be/GERT/

Contents

PART I

Introduction

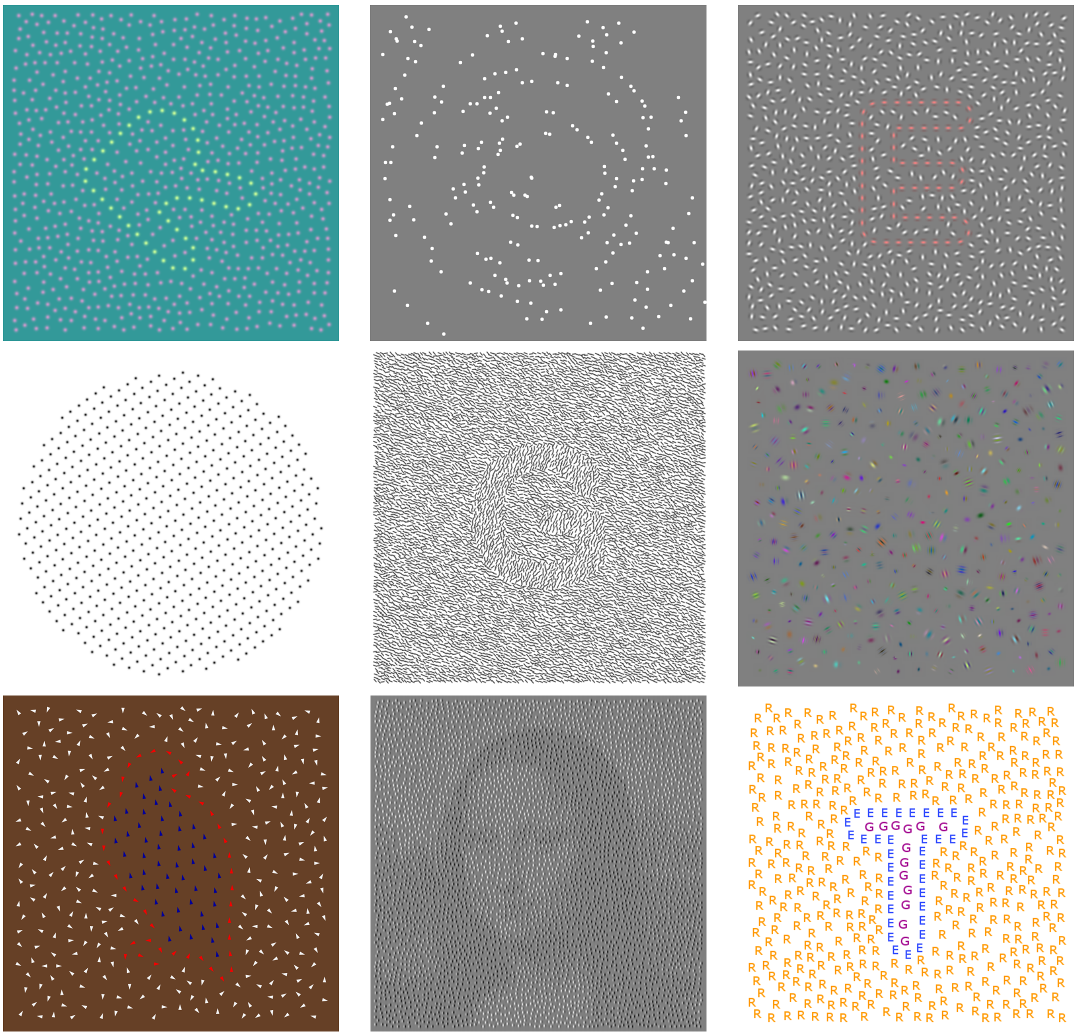

Figure 1.1 – Example stimuli generated with GERT. Running GERT_demo from MATLAB's command prompt should yield similar stimuli. Visit our website for more example stimuli.

1.1 Instant Play

The code below can be executed directly from MATLAB's command prompt to generate Figure 2.1. In the Tutorial below we will walk you through this code one step at a time.

```
% Clear everything and initiate GERT:
  ccc; GERT_Init;

% Ellipse contour definition
  gce_params.hax = 120;
  gce_params.vax = 80;
  C = GERT_GenerateContour_Ellipse(gce_params);

% Shift the (0,0) point to (300,300)
  C.x = C.x + 300;
  C.y = C.y + 300;

% Plot the result
  figure; plot(C.x, C.y, '.'); axis equal;

% Place contour elements
  pec_params.cont_avgdist = 40;
  [E ors] = GERT_PlaceElements_Contour(C, pec_params);

% Place background elements
  peb_params.dims = [1 600 1 600];
  peb_params.min_dist = 35.51;
  Ea = GERT_PlaceElements_Background(E,[],peb_params);

% Plot the result
  figure; hold on;
  plot(Ea.x,Ea.y,'b.'); plot(E.x,E.y,'r.');
  axis([1 600 1 600]); axis equal;

% Indices for contour and background elements
  c_idx = 1:E.n;
  b_idx = E.n+1:Ea.n;

% Parameters for GERT_DrawElement_Gabor
  gabel_params.sigma = 4;
  gabel_params.freq = 0.0771;
  gabel_params.phase = 0;
  gabel_params.or(b_idx) = 2*pi*rand(1,Ea.n-E.n);
  gabel_params.or(c_idx) = ors;

% Image parameters
  img_params.bg_lum = 0.5;

% Render image
  IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor, ...
  Ea, gabel_params, img_params);
  figure; imshow(IMG);
```

1.2 Overview

GERT (Grouping Elements Rendering Toolbox) is an open-source MATLAB toolbox aimed at generating a wide variety of stimulus displays typically used in empirical research on perceptual grouping. In particular, GERT is good at creating displays consisting of large arrays of spatially separate or partially overlapping graphical elements. As a software package it is simple and straightforward, even for novice programmers who are only familiar with the most basic MATLAB concepts. Yet, GERT also offers a great degree of flexibility, extensibility and control for more advanced users. Some example stimuli generated with GERT are shown in Figure 1.1.

This manual begins with a step-by-step Tutorial demonstrating the basic functionality of GERT. By means of a practical example, the generation of an ellipse contour embedded in a field of randomly positioned Gabor elements, we will illustrate GERT's fundaments. This should allow new users to become familiar with the basic structure. In the next part we will delve further into the various options available, and offer a more exact description of the methods introduced in the tutorial. The Tips & Tricks section will discuss a number of specific advanced topics that often recur as questions. For instance, we will describe how to make stimulus generation faster, and how to implement a custom drawing function for the elements. The Full Documentation section compiles all help text available from within the MATLAB command window. All function arguments and parameters will be described in detail. This part in particular is intended as a reference, not as a step-by-step guide.

1.3 Requirements

GERT requires MATLAB 2007b or later, with the Image Processing Toolbox installed. The Statistics toolbox is used for some options, but is not strictly required. GERT was developed on Windows 7 and Windows XP, but should run flawlessly on modern Linux and Mac operating systems as well. For users of older MATLAB versions or the open-source project Octave we provide a separate download, that should provide most of the normal GERT functionality. However some features may not work; for Octave 3.2.4 we have listed known difficulties in the Appendix. Users of newer MATLAB versions may even notice a speed increase of around 15 percent (on `GERT_Demo.m`) by using this alternate download; however the MATLAB variable editor window will not be able to display the contents of GERT's basic data structures.

For novice programmers, we assume familiarity with the following basic MATLAB programming concepts: Variable types (double, char, cell, stuctures, ...), vector and matrix operations (indexing, transposing, arithmetic,...), procedural program flow (if, while, for, ...), and functions (input and output arguments, scope,...). We will also encounter some more advanced concepts in this manual, which we will then briefly explain.

1.4 Website, download and install

Before downloading the GERT toolbox from our website, we ask you to enter your email-address. We will never give this information to third parties (not even for money). In fact, you could enter an invalid email-address, but then you will miss out on the occasional update mailings.

Your download will contain all the source code to run the latest stable release of the GERT toolbox, and can be unpacked into a directory of choice. Just make sure MATLAB knows where to find these files (MATLAB menu: File -> Set path -> Add folder). To check whether everything works as it should, enter GERT_demo at the MATLAB command prompt. This should open a number of Figure windows that look similar to Figure 1.1. If no errors or warnings occur, you are ready to use GERT.

Previous releases will remain available at the website. This will be useful to those who wish to make exact reproductions of stimuli generated with an earlier GERT version.

Additional information will be made available on-line, such as demo code, graphical examples of stimuli, this manual, etc. Known bugs and frequently encountered difficulties will be posted in the Frequently Asked Questions (FAQ) section.

1.5 License and cooperation

GERT is free but copyrighted software. You can redistribute and/or modify it under the terms of the GNU General Public License (Version 3) as published by the Free Software Foundation. GERT is distributed without implied support and without any warranties about its functionality.

We welcome collaboration with other researchers to expand and improve the GERT toolbox. Please send your suggestions and comments to GERT@gestaltrevision.be. In the near future, we will provide space on the website where you can upload your own code for stimulus construction, or your own extensions to the basic GERT functionality. This should increase transparency in the research domain, and provide new users with many useful examples. For the time being, please send us your code (GERT@gestaltrevision.be) and we will upload it manually.

1.6 Citing GERT

When referring to GERT in your scientific publications, please mention the version used (run GERT_-Version at the command prompt), and include a reference to the original paper. Example: "Stimuli were generated using GERT, a toolbox for constructing perceptual grouping displays (Demeyer & Machilsen, 2012)".

Demeyer, M., & Machilsen, B. (2012). The construction of perceptual grouping displays using GERT. *Behavior Research Methods, 44(2)*, 439-446.

PART II

GERT started

In this tutorial we will demonstrate how to use the GERT toolbox to embed an ellipse contour in a field of randomly oriented Gabor patches (Figure 2.1).

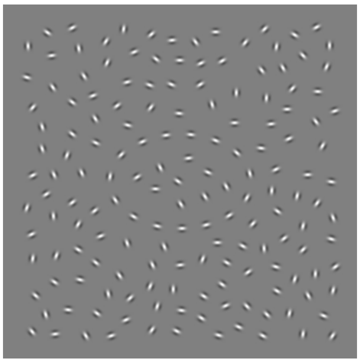

Figure 2.1 – Gabor field with embedded ellipse outline.

Step-by-step, we explore the basic functionality of GERT. We will focus mostly on the **required** input arguments for the GERT functions. **Optional** arguments and more advanced functionality will be explored later in this manual (Part 3, 4, & 5).

This tutorial is divided into 5 sections, reflecting the basic structure of the GERT toolbox:

(1) Generate a contour description;

(2) Place elements along this contour;

(3) Embed the contour in a background of randomly positioned elements;

(4) Check for local density cues in the display;

(5) Render the display.

The MATLAB code used to generate Figure 2.1 (and all other stimuli in this manual) is available from the website. For each step in this tutorial, try changing the parameter values and play with some of the options. Plot, and evaluate the effect of your changes.

2.1 Step 1: Describing the contour

To describe the contour of an ellipse we use the GERT_GenerateContour_Ellipse function. This function takes a single input argument, namely a parameter structure which we will call `gce_params`. Within this structure a number of parameters can be defined. However for the purpose of this tutorial we will stick to the two mandatory fields, `hax` and `vax`, respectively the length of the horizontal and vertical semi-axes of the ellipse. If you leave out either of these fields, GERT will display an error message. The `gce_params` structure can also contain additional, optional fields. If you omit an optional field, GERT will assign a default value. Including new, unexpected parameter fields in the `gce_params` structure will again lead to a GERT error dialog[1].

Now, try to execute this code, line by line:

```
% Clear everything and initiate GERT:
  ccc; GERT_Init;

% Ellipse contour definition
  gce_params.hax = 120;
  gce_params.vax = 80;
  C = GERT_GenerateContour_Ellipse(gce_params);

% Shift the (0,0) point to (300,300)
  C.x = C.x + 300;
  C.y = C.y + 300;

% Plot the result
  figure; plot(C.x, C.y, '.'); axis equal;
```

The above MATLAB code starts with two other commands. The ccc command included in GERT clears all variables from the MATLAB workspace, including some globally persistent variables. It also closes all Figure windows and cleans the command window. The GERT_Init command checks the GERT version, its software dependencies, and runs the required initialization routines (if any). It would be prudent to always run both of these commands at the start of your GERT script.

The GERT_GenerateContour_Ellipse function will return a new variable, `C`, which contains the contour definition. This variable is actually a MATLAB object, but can be treated as a MATLAB structure for most purposes[2]. The contour `C` contains a number of properties: `x, y, cdist, lt, closed`. The first two are the Cartesian XY coordinates of a large set (default: 1000) of successive points along the contour. It is important to realize that this is how GERT will describe any contour: As a discretized

[1] The Full Documentation section lists all requested and optional input arguments for each function.
[2] In MATLAB versions prior to 2008 you might not be able to see the contents of this object in your workspace.

set of points, ordered along a continuous contour, and never identical to any other contour point. The `cdist` property is a MATLAB vector containing the distances along the contour between the starting point of the contour and each point described by the (`x,y`) pairs. The `lt` field contains the orientation of the local tangent line to each of these points. The final property, `closed`, is a logical value indicating whether the contour should be considered as closed (`true`) or open (`false`).

The contour description returned by GERT_GenerateContour_Ellipse is always centered on the (0,0) point. As we plan to embed this contour in a 600×600 display, and we can't have negative pixel positions, we first shift this (0,0) point to (300,300). The generated contour is visualized using the standard MATLAB plot command. The result is shown in the left-hand panel of Figure 2.2.

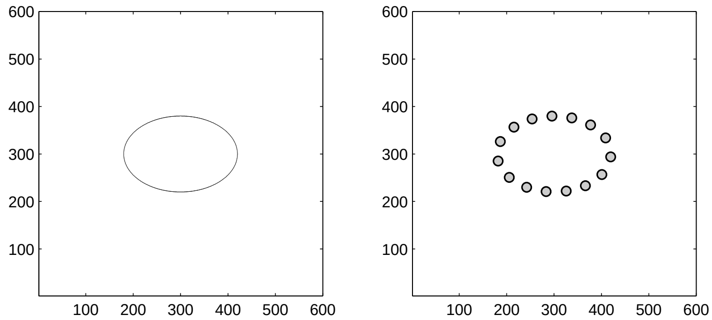

Figure 2.2 – The left-hand panel shows an ellipse contour. This looks like a continuous line, but is actually a discrete set of 1000 points on the contour. The right-hand panel shows how a small number of elements can be placed on this contour, at equidistant locations along the contour.

2.2 Step 2: Placing elements on the contour

Once we have obtained a discretized but detailed description of the contour, we can place a small number of *elements* on the contour. These elements indicate where the Gabor image patches will be drawn. In this example, we want to place the elements at equidistant locations along the contour outline. Importantly, these elements do not have to coincide with any of the 1000 points of the full contour description. GERT interpolates between these 1000 contour points to calculate the position of the requested elements.

We use the function GERT_PlaceElements_Contour to find the correct X and Y coordinates of the elements placed on the contour. Again, these elements will be separated by a constant distance *along the contour*, which is not the same as the actual Euclidean distance between the points. As its arguments, this function requires the full contour description, which we obtained in the previous section, and a parameter structure. The user has a choice as to how to determine the distance between two successive elements along the contour. In the example below, we set the `pec_params.cont_avgdist` parameter, and let the function figure out to what number of equidistantly placed elements this most closely corresponds. Alternatively, the `pec_params.el_n` parameter can be set to fix the number of elements, and let the function figure out the corresponding distance between successive elements. The equidistant method which we use here is the default element placement method, and does not need to be specified explicitly.

```
% Place contour elements
  pec_params.cont_avgdist = 40;
  [E ors] = GERT_PlaceElements_Contour(C, pec_params);
```

Two output arguments are returned: `E` is an elements object similar to the above contour object `C`. It contains the Cartesian coordinates for the elements that we have just placed, in two separate `x` and `y` vectors. These elements are plotted in the right-hand side of Figure 2.2. The field `n` indicates the number of elements (in this example: 15).

The second output variable `ors` is a vector which, for each element in the `E` object, contains the orientation of the local contour tangent. In our example, where we will render the display using Gabor patches, the orientations contained in `ors` will be used to orient the Gabor patches so that they are smoothly aligned to the contour.

2.3 Step 3: Placing elements in the background

Next, we will fill up the remainder of the display with background elements. Before embedding the contour in randomly positioned elements, a number of choices need to be made. First, we have to tell GERT how large our display should be. We already told you that we wanted a 600×600 display, but GERT is still ignorant about this. We inform GERT with `peb_params.dims`. Second, we will have to specify a minimum Euclidean distance to keep between the elements placed, using the `peb_params.min_dist` parameter. Here we set it to 35.51. In this example GERT will continue placing elements until the display is full, but we could also set the the total number of elements using `peb_params.bg_n`.

```
% Place background elements
  peb_params.dims = [1 600 1 600];
  peb_params.min_dist = 35.51;
  Ea = GERT_PlaceElements_Background(E,[],peb_params);
```

```
% Plot the result
  figure; hold on;
  plot(Ea.x,Ea.y,'b.'); plot(E.x,E.y,'r.');
  axis([1 600 1 600]); axis equal;
```

We then call the function GERT_PlaceElements_Background with the `E` object, computed above, as the first argument. GERT respects these contour element positions and will not place background elements closer than `peb_params.min_dist`. The second argument remains empty. It is used to explicitly define a limited region for GERT to place background elements, but in this case we want to fill the entire rectangular display. This option will be discussed further in 3.3. The third argument is the `peb_params` structure containing the background element placement parameters.

As output, we receive a new 'elements' object, `Ea`, containing the X and Y coordinates of all elements in the display (i.e., for both contour and background elements). It also contains the total number of elements (`Ea.n`) and the image dimensions (`Ea.dims`).

The result of the above MATLAB code is plotted in the left-hand panel of Figure 2.3. We used standard MATLAB plotting commands to generate this Figure. You can also use GERT's built-in plot command for a quick impression of the result. At MATLAB's command prompt, type: `plot(Ea)` or `Ea.plot` (both commands are equivalent in MATLAB 2007b and above).

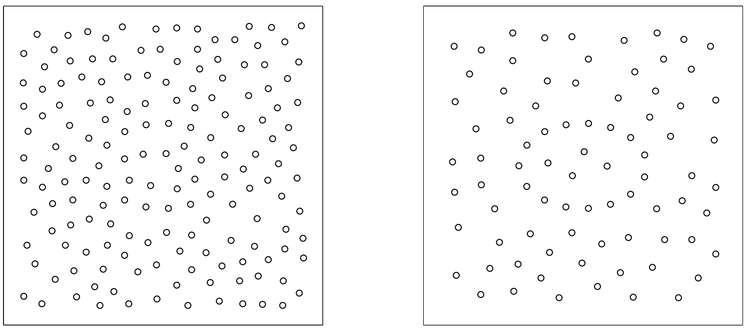

Figure 2.3 – The left-hand panel shows the ellipse contour embedded in a background of randomly positioned elements, respecting a minimal distance of 35.51. The right-hand panel shows the result when the minimal distance is set to 50.

2.4 Step 4: Local density cues

When studying perceptual grouping cues other than proximity, researchers often want to avoid that the embedded contour or figure can be detected merely because these foreground elements are positioned closer together than the background elements. In other words, they want to avoid a *local* *density* cue in the stimulus display. For instance, in the right-hand panel of Figure 2.3 the ellipse contour is clearly visible, while it is much harder[3] to see in the left-hand panel. The reason for this is that the local density distribution of contour and background elements is more similar in the left-hand panel. We achieved this by carefully choosing the value of `peb_params.min_dist` (you probably already guessed that 35.51 was not an entirely random value).

Whether the local density cue is truly eliminated depends in this example on the relative values of `pec_params.cont_avgdist` and `peb_params.min_dist`, as well as the random outcome of the element placement procedure. Depending on the sort of figure that is being embedded in a perceptual grouping display, it is not always trivial to balance these two parameters. GERT contains functionality to explicitly check for and reduce local density cues. The next part of this manual will provide a more detailed look into this (see 3.4).

2.5 Step 5: Rendering the display

As we are now satisfied with the spatial lay-out of our stimulus, we can start to draw Gabor patches at these element positions. We want all of these Gabors to be identical to one another, except for their orientation. Contour elements will be aligned to the local tangent lines of the contour, and background elements will be given an entirely random orientation.

To accomplish this we call the GERT_RenderDisplay function, with four input arguments. The first argument is a function handle to the element drawing function. Function handles are a type of variable in MATLAB that contain a 'link' to a specific function. This obviously brings a lot of flexibility to the rendering function: Merely by providing a different drawing function handle, the exact same element positions can be rendered as something else than Gabors. But for now, we stick to Gabor elements, using the function handle @GERT_DrawElement_Gabor. The second argument defines which elements need to be rendered (in our example: `Ea`). The third input argument is the parameter structure for the element drawing function. The exact fields depend on the specific drawing function, in this case GERT_DrawElement_Gabor. Here we will define the following parameters:

1. `sigma`: The standard deviation of the Gaussian envelope (in pixels).
2. `freq`: The frequency of the sinusoidal grating (in cycles/pixel).
3. `phase`: The phase offset of the sinusoidal grating.
4. `or`: The orientation of the Gabor patch.

[3] But not impossible, due to the compactness of the figure and the regularity of the element spacing.

Notice in the MATLAB code below that the gabel_params structure contains both scalar fields (i.e., one single value) and vector fields (i.e., an array of multiple values). Vectors (in this example, or) should always have the same length as the number of elements to be drawn (Ea.n). GERT will then apply the different values in the vector to their corresponding elements in the Ea object. If a parameter field contains only a scalar, GERT_RenderDisplay will use the same constant value for each element. For instance in the example below, all Gabor patches will have a phase offset of 0.

```
% Indices for contour and background elements
  c_idx = 1:E.n;
  b_idx = E.n+1:Ea.n;

% Parameters for GERT_DrawElement_Gabor
  gabel_params.sigma = 4;
  gabel_params.freq = 0.0771;
  gabel_params.phase = 0;
  gabel_params.or(b_idx) = 2*pi*rand(1,Ea.n-E.n);
  gabel_params.or(c_idx) = ors;

% Image parameters
  img_params.bg_lum = 0.5;

% Render image
  IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor, ...
  Ea, gabel_params, img_params);
  figure; imshow(IMG);
```

To distinguish between contour and background elements in the gabel_params structure, we will for now make use of this simple fact: The GERT_PlaceElements_Background function will place the background elements *after* the already present contour elements in the Ea.x and Ea.y vectors. A smarter method will be introduced later, in the Tips & Tricks section. Knowing the rank numbers of contour and background elements in the Ea object, we can then use these indices to separately manipulate their orientation (along the contour, or random).

The final input argument for GERT_RenderDisplay is img_params, a structure containing the parameters for the GERT_RenderDisplay function itself. In this example, we have only defined one field: bg_lum, the luminance of the background (between 0 and 1). This parameter structure is optional; even when omitted, the background luminance will default to 0.5.

The result of the above MATLAB code is illustrated in the left-hand panel of Figure 2.4. It is easy to manipulate more aspects of the Gabor patches, as illustrated in the other panels of Figure 2.4. To generate these images, start from the above MATLAB code, and add one extra line before running GERT_RenderDisplay, as shown in the MATLAB code below.

```
% Image 2: same orientation for all background elements
gabel_params.or(b_idx) = 2*pi*rand;

% Image 3: random phase offset for all Gabor patches
gabel_params.phase = 2*pi*rand(1,Ea.n);

% Image 4: random orientation for all Gabor patches (incl. contour)
gabel_params.or = 2*pi*rand(1,Ea.n);
```

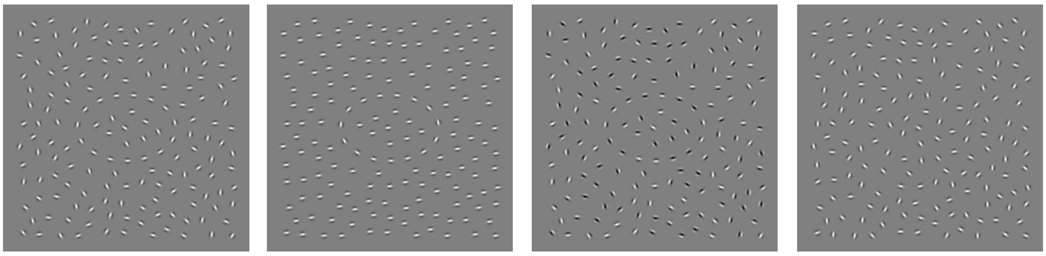

Figure 2.4 – An ellipse contour rendered with Gabor patches, embedded in a field of randomly positioned Gabor patches. From left to right: (1) Contour elements aligned with the contour, and background elements have a random orientation; (2) Same as (1), but all background elements now have the same orientation; (3) Same as (1), but with random phase offsets for all elements; (4) Same as (1), but with randomly oriented contour elements.

2.6 Concluding remarks

We have shown how GERT generates perceptual grouping displays. It did not require advanced programming skills, and the code should have executed very quickly. We did not emphasize how GERT covertly used default methods and parameter values in many functions. In the next part of this manual, we will learn more about these methods and parameters. We will illustrate how GERT can be used to create a greater variety of displays. For now, we have only discussed GERT's basic functionality: (1) Define a full contour; (2) Place elements on the contour; (3) Add background elements; (4) Check the local density cue; (5) Render the display.

PART III

GERT more experienced

Here, we will demonstrate how the basic functionality of GERT can be extended. We will follow the same logical structure as before. First, we will describe different ways of defining a contour. Next, more options for placing elements on contours and in the background are illustrated. Third, we discuss how local density cues can be checked and minimized. The final part will be about GERT's flexible display rendering techniques.

3.1 Describing the contour

In the Tutorial we illustrated how GERT can generate an ellipse contour description. Although ellipses (and circles, when `hax` equals `vax`) do look nice, often we want to embed more complex shapes. GERT contains several functions to do this. We will first discuss the exact nature of the contour object that is returned by these functions. We will then let GERT generate radial frequency patterns, and finally we will show how pre-defined contours can be read from TXT and SVG files.

3.1.1 The GContour object

We mentioned before that the contour descriptions were in fact MATLAB *objects*, but could for the time being be treated as if they were structures. However, you have undoubtedly noticed that MATLAB displayed its type as GContour. We will now inspect these objects in more detail, and explain why they are not and should not be mere structures. For this, however, it is necessary that the reader is familiar with the basics of object-oriented programming. For those with no prior OOP experience, we offer a brief introduction in the Appendix.

GContour properties

Each GContour object will contain two row vectors, `x` and `y`, to hold the Cartesian coordinates defining the contour. These vectors are allowed to have different lengths, but any function using a GContour object will quit on an error if this is the case. The property `closed` is by default set to `false`, and indicates whether the GContour should be considered to be closed or open. The property `n` contains the number of contour definition points. It will always be equal to the length of vector `x`, and cannot be set to a different value by the user.

Properties `cdist` and `lt` contain the distance of each contour definition point *along* the contour and the local tangent orientation to the contour at this point, respectively. Depending on the contour generation function used, these properties might be automatically filled in or empty. As explained below, the GContour class contains methods to fill them. Finally, the `clength` property is dependent on `cdist`, and contains the total length of the contour.

GContour methods

The `validate` method checks whether the values contained within the object can be considered to reflect a valid contour. For instance, whether both coordinate vectors are of equal length. The `plot` method generates a graphical display of these data. Useful for closed contours are the `centroid` and `main_axis` methods, computing and returning the centre-of-mass point and the main axis orientation of the contour. If no `cdist` or `lt` values are present in the GContour object, the `compute_cdist` and `compute_lt` methods allow the user to fill them in using default methods that are applicable to most types of contours. Do note however that the contour definition points always have to be ordered along the contour in order to use these functions.

GContour constructors

Not passing any arguments to the constructor will leave the coordinate vectors initially empty. Three other options are available, however. First, when passing a 2×N numerical vector as the first argument, these two rows will be used as the initial `x` and `y` vectors, respectively. Second, passing a GElements object (see below) as the first argument will use the `x` and `y` vectors contained within this object to fill their counterparts in the new GContour object. In both cases, a second argument may optionally be provided, which should be a 1×1 logical value (`true` or `false`). This value will then be used for the `closed` property of the contour. If it is omitted, the property is set to `false`. Third, another GContour object may be passed on to the constructor, together with an indices vector. A new GContour object which is a subset of the existing GContour object will then be created, also copying its `closed` property.

3.1.2 RFP Contours

GERT_GenerateContour_RFP is used to generate a particular type of contours known as Radial Frequency Patterns (Wilkinson, Wilson, and Habak (1998)). These stimuli are often used in vision science, because they allow the creation of a great variety of smooth, closed contours, that can nevertheless be described using only a few parameters. A Radial Frequency Pattern (RFP) is composed of a number of radial frequency components. Each component is described as a sine function with a frequency of k, that is, a wavelength of $2\pi/k$. Smooth contours are generated by plotting the sum of these sine waves in a polar coordinate system. A large variety of contours can then be created by changing the number of radial frequency components, and their frequencies, phases and amplitudes.

A simple example using only two radial frequency components illustrates the creation of a RFP. The following MATLAB code was used to generate Figure 3.1.

```
ccc;
GERT_Init;

% Define the Radial Frequency Components
gcr_params.freq = [2 3];
gcr_params.amp = [1 1.8];
gcr_params.ph = [rand rand]*2*pi;
gcr_params.baser = 10;

% Construct the Radial Frequency Pattern contour from them
C = GERT_GenerateContour_RFP(gcr_params);
plot(C);
```

The `gcr_params.freq` field defines the frequencies of the constituent components. In our example, we have two radial frequency components, with k=2, and k=3. These two components are plotted as grey sine waves in the left-hand panel of Figure 3.1. The `amp` and `ph` fields should be vectors of the same length as the `freq` field, as they define respectively the amplitudes and phases of the components defined in `freq`. Unequal sizes for these three fields will result in an error. In this specific example, the amplitudes remain fixed, and the phases are randomized. Play around with the number of components and the amplitudes and phases to see how this effects the resulting RFPs. The last field, `baser`, defines the radius of the base circle to which the sum of the sine waves is applied in the polar plot (see right-hand panel of Figure 3.1). Using the optional fields `rot` and `scale`, the contour can be rotated and scaled.

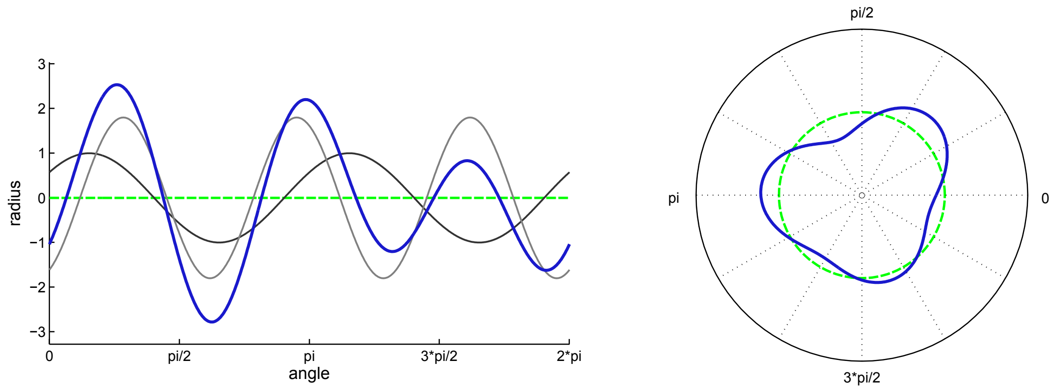

Figure 3.1 – The left-hand panel displays two sine waves (dark grey: k=2; light grey: k=3), and their sum (blue). This sum is shown in the polar plot on the right as the deformation of a base circle (green; radius defined by `baser`). Where the blue line in the left-hand panel is maximal (at about $\pi/4$), the deformation from the base circle in the right-hand panel is also maximal. Note that you will probably not be able to exactly replicate this figure, as we randomly chose the phase offset for each sine wave.

The object outputted by the function is of the type GContour. As can be seen in the MATLAB workspace, it contains the X and Y coordinates of 1000 points (`C.n`) along the contour[4]. These 1000 points constitute the full contour definition that GERT uses for all subsequent calculations. If you somehow feel this is too poor a resolution, increase the value of the optional parameter `th_n`. The GContour object `C` contains a `closed` flag, indicating whether the contour is closed (`true`) or not (`false`). This particular contour is indeed closed. However with the ellipse contours described in 2.1 and the RFP contours described here, generating an open contour is very simple. Because both the ellipses and the RFP contours are actually defined in polar coordinates (angle and radius), there is an easy way to define only a contour segment. We illustrate this here for the RFP example. Adding `gcr_params.th_range = [pi/4 3*pi/4]` in the above MATLAB code restricts the contour between polar angles of $\pi/4$ and $3\pi/4$. The resulting contour segment is depicted in red in Figure 3.2. The `closed` flag will now be `false` for the contour that is returned (displayed in red). Note that non-closed contours cannot compute their `centroid` and `main_axis` properties.

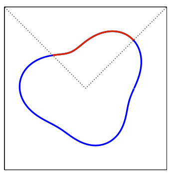

Figure 3.2 – A contour segment (red, from `grc_params.th_range = [pi/4 3*pi/4]`) overlaid on the full RFP contour description (blue). The dotted lines show the polar angles.

[4] With a 2007 MATLAB version (type `ver('Matlab')` at the command prompt) you won't see the contents of the `C` object. To list the properties and methods associated with the GContour object, type `C.` at the command prompt, followed by a tab.

3.1.3 Reading contours from a TXT file

We have now discussed how to generate ellipse contours and RFP contours. If you do not like ellipses or RFPs, you could define your own GContour object by hand in MATLAB using the supplied constructors, or write your own GERT contour generation function. However, it is also possible to simply read in the Cartesian contour point coordinates from a TXT file, as long as it contains one single continuous contour definition. Each line in this file should contain a separate pair of X (first column) and Y (second column) coordinates, separated by a delimiter of your choice. In the following example, we will draw the shape outline of a car (stimulus 47 in the Snodgrass and Vanderwart (1980) set of everyday objects).

We use the GERT_GenerateContour_FileTXT function with three input arguments: (1) The filename of the TXT file containing the coordinates; (2) The `closed` flag, indicating whether the contour should be considered closed or open; and (3) The delimiter to be used (default: space, ' ').

```
ccc;
GERT_Init;

C = GERT_GenerateContour_FileTXT('car.txt',true,' ');
C = GERT_Transform_Rotate(C, pi, 'Centroid');

plot(C);
```

The output of the GERT_GenerateContour_FileTXT is, again, a GContour object. The resolution of the GContour object is now defined by the precision of the TXT file. In this example, we have 1106 coordinate pairs in the TXT file. However, the text file contained an upside-down shape outline of a car. To get it wheels down we first call the GERT_Transform_Rotate function to rotate the coordinates 180° around the centroid of the closed shape. The result is shown in the left-hand panel of Figure 3.3.

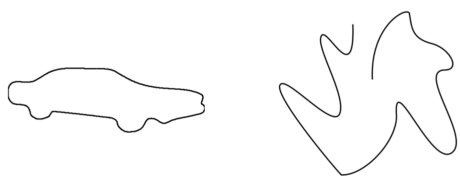

Figure 3.3 – Left: Shape outline of a car. The X and Y coordinates for the full contour definition are read from a TXT file. Right: Random contour read from an SVG file.

3.1.4 Reading contours from an SVG file

GERT supports the creation of GContour objects from a vector graphics file format, namely plain SVG. However, it is limited to reading in one single `<path>`, consisting of a combination of lines, Bézier curves and ellipse arcs. The path may be discontinuous, in which case a vector of GContour objects will be returned. Plain SVG files can easily be created using the free program Inkscape. GERT will read the file, and discretize the contour definition to a precision of the user's choice, set through the second argument, `res`. Here however, we call the GERT_GenerateContour_FileSVG function with just its one required input argument, namely the file name of the SVG file. The right-hand panel of Figure 3.3 shows the result. Open the file in Inkscape, manipulate the contour, and read it in again through GERT to see how easy it is to create contours outside MATLAB.

```
ccc;
GERT_Init;
C = GERT_GenerateContour_FileSVG('test.svg');
plot(C);
```

3.2 Placing elements on the contour

3.2.1 The GElements object

Analogous to the GContour class, we have defined a GElements class to hold information on element positions in a display. It is this type of data that is returned by all GERT element placement functions.

GElements properties

The most basic properties of the GElements class are again `x`, `y`, and `n`. The latter property is again dependent on the length of the `x` vector, and cannot be set by the user. A GElements definition also contains a `dims` property, a 1×4 double vector containing the dimensions of the rectangular display inside which the element positions should be situated. `dims` is allowed to be empty, but should be set before the display is rendered, either by hand or through the GERT_PlaceElements_Background function.

GElements methods

Like the GContour class, the GElements class also defines a `validate`, `rmid`, and `plot` method. The latter will color the element positions differently depending on whether they belong to a contour or to the background, and also draw the display dimensions to provide a useful graphical overview of the data within the object. The `settag` and `gettag` methods are discussed in more detail in the Tips & Tricks section.

GElements constructors

If the constructor arguments are not empty, it must contain one of three data types as the first argument. First, in case of a scalar GContour object, its `x` and `y` vectors will be used to fill in the corresponding vectors of the GElements object. Second, in case of a 2×N double matrix, these two rows will be taken as the `x` and `y` vectors. In both cases, a 1×4 vector may optionally be specified as the second argument, to set the dimensions. Third, another GElements object may be passed, followed by a row vector of indices as the second argument. The constructor will then create a new GElements object which is a subset of the original GElements object provided, and copy its `dims` property.

3.2.2 GERT_PlaceElements_Contour

Once we have obtained a GContour object containing a contour definition, the placement of elements on the contour can commence. As in 2.2, we use the GERT_PlaceElements_Contour function for this.

The default element placement method is `'ParallelEquidistant'`. As already illustrated in 2.2, this method divides the contour in parts of equal length, respecting the requested distance between elements along the contour (`cont_avgdist`). This method is useful if you have a simple, smooth contour, where different parts of the contour do not come too close to one another. Panel A of Figure 3.4 illustrates the `'ParallelEquidistant'` method for the RFP generated above (3.1.2).

```
ccc;
GERT_Init;

grc_params.freq = [2 3];
grc_params.amp = [1 1.8];
grc_params.ph = [0.6103 5.1739];
grc_params.baser = 10;
grc_params.scale = 8;
C = GERT_GenerateContour_RFP(grc_params);

pec_params.cont_avgdist = 40;
pec_params.method = 'ParallelEquidistant';
[E ors] = GERT_PlaceElements_Contour(C, pec_params);

plot(E);
```

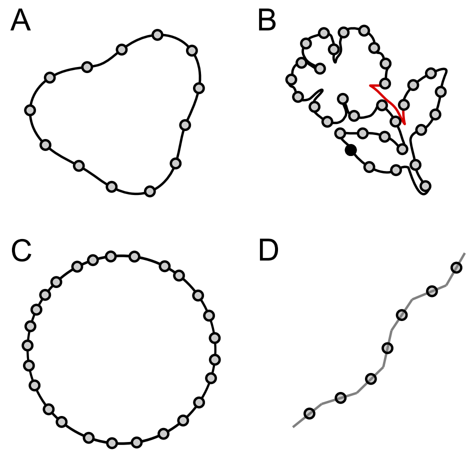

Figure 3.4 – In panel A, an RFP contour was populated with equidistantly spaced contour elements. The element positions have been calculated using the `'ParallelEquidistant'` method. Panel B shows a flower outline with element positions obtained using the `'SerialEquidistant'` method. The black element is the first element that was placed by GERT. Subsequent elements were placed in clockwise direction along the contour, trying to respect equidistance until the `eucl_mindist` could no longer be respected (as in the red segment). Panel C illustrates the `'Random'` element placement method for a circle contour. In panel D, finally, elements were placed on a 'snake' of connected line segements.

When the contours are relatively complex, such that parts of the contour are too near to one another, the `'SerialEquidistant'` element placement method should be used to approximate equidistant placement as closely as possible. GERT will first try to respect the spacing requested by `cont_avgdist`. However, as soon as the Euclidean distance between two successive points becomes smaller than the required parameter field `eucl_mindist` GERT increases the average distance along the contour, until a point along the contour is found that respects the minimal Euclidean distance to all previously placed points. We illustrate this here for the contour of a flower, which was read in from a TXT file (Figure 3.4).

```
ccc;
GERT_Init;

C = GERT_GenerateContour_FileTXT('flower.txt',true,' ');
C = GERT_Transform_Rotate(C, pi, 'Centroid');

pec_params.method = 'SerialEquidistant';
pec_params.cont_avgdist = 40;
pec_params.eucl_mindist = 30;
[E ors] = GERT_PlaceElements_Contour(C, pec_params);

plot(E);
```

But while striving for equidistant element placement results in salient and recognizable contours, your specific research question might preclude the presence of such a regularity cue to contour presence (see 3.4.3). One should then opt for entirely random placement of elements along the contour, using the `'Random'` method. The `eucl_mindist` parameter field must be set to specify the minimum Euclidean distance to be kept between the various elements. Element placement will continue until no more element positions are available, or until the a given number of elements (`el_n` parameter) is reached. This method is illustrated in panel C of Figure 3.4, up to a fixed number of elements.

```
ccc;
GERT_Init;

params.hax = 100;
params.vax = 100;
C = GERT_GenerateContour_Ellipse(params);

pec_params.method = 'Random';
pec_params.eucl_mindist = 10;
pec_params.el_n = 15;
E = GERT_PlaceElements_Contour(C, pec_params);

plot(E);
```

All three methods offer the option of position jittering. For this, see Tips & Tricks.

3.2.3 GERT_PlaceElements_Snake

One often-used type of contour to embed in randomly placed background elements, is a *snake*. The snake generation methods used in GERT are similar to those of Hess and Dakin (1999). This is a somewhat exceptional application for GERT, as it does not generate the element positions from a prior contour definition, but creates the snake element positions right away.

```
ccc;
GERT_Init;

pes_params.seg_n = 7;
pes_params.seg_len = 5;
pes_params.seg_or_avgang = pi/6;

[E, ors] = GERT_PlaceElements_Snake(pes_params);

plot(E);
```

The GERT_PlaceElements_Snake function constructs a snake as a series of `seg_n` connected line segments, each having a length of `seg_len`. The angle between successive segments equals `seg_or_avgang`, either pointing to the left or to the right of the previous segment. The snake elements are then placed on the midpoints of these segments. The function returns `E`, a GElements object containing the positions of the snake elements, and `ors`, a MATLAB double vector containing the orientation of the segments on which the elements were placed. These are also the suggested orientations to use in order to make the snake perceptually smooth. The resulting element positions for the above MATLAB code are plotted in panel D of Figure 3.4. Optionally, position jitter and more variation in the angle between successive segments can be introduced, as described in the Full Documentation.

3.3 Placing elements in the background

To fill up the display, GERT samples background positions iteratively from the collection of remaining element positions that are sufficiently far from previously placed contour or background elements. In addition, GERT's batched position sampling scheme (see Tips & Tricks) considerably reduces computing time for large collections of elements. Background element placement is handled by GERT_PlaceElements_Background.

3.3.1 First argument: Pre-existing GElements or GContour objects

The first input argument for GERT_PlaceElements_Background can be either a GElements or a GContour object. The first case has already been illustrated in 2.3. There, we called the GERT_PlaceElements_Background with the GElements object `E` as first input argument. This GElements object contained the positions of contour elements on an ellipse contour. The following MATLAB code does more or less the same thing. It starts with generating an ellipse contour, places elements on the contour, and then calls the GERT_PlaceElements_Background function with the GElements object `E` as first input argument. The resulting output `Ea` is a new GElements object containing both the positions of the original contour elements and of the newly placed background elements. We use GERT's built-in plot command for GElements objects to illustrate the result (left-hand panel of Figure 3.5).

```
ccc;
GERT_Init;

% Generate contour (GContour object)
gce_params.hax = 120;
gce_params.vax = 80;
C = GERT_GenerateContour_Ellipse(gce_params);
C = GERT_Transform_Shift(C,[300 300]);

% Place elements on contour (GElements object)
```

```
pec_params.cont_avgdist = 40;
E = GERT_PlaceElements_Contour(C,pec_params);
E.dims = [1 600 1 600];

% Background placement with GElements object
peb_params.min_dist = 20;
Ea = GERT_PlaceElements_Background(E,[],peb_params);
plot(Ea);

% Background placement with GContour object
peb_params.dims = E.dims;
Ea = GERT_PlaceElements_Background(C,[],peb_params);
Ea = GERT_MergeElements({Ea E});
plot(Ea);
```

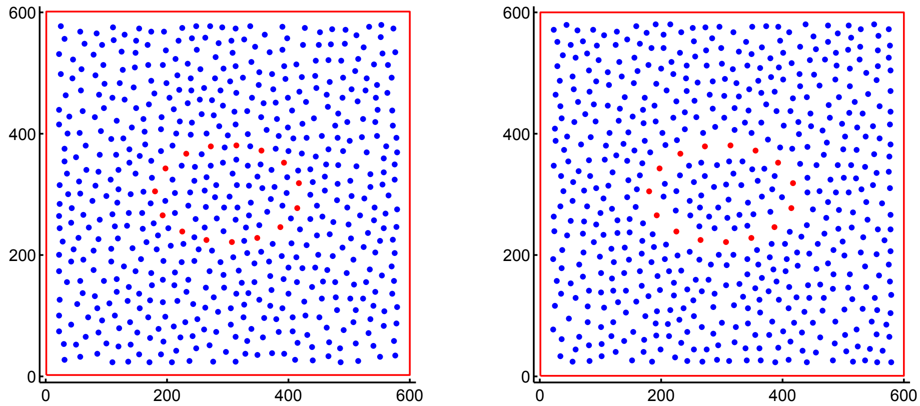

Figure 3.5 – Background element placement. Left: Element object as input argument. Right: Contour object as input argument (see MATLAB code above).

After plotting the result, in this example the GERT_PlaceElements_Background is called again, but now with a GContour object as its first parameter. This will cause GERT to keep a minimal distance from all contour definition points, thus, effectively the entire contour. However these contour definition points will not be used as element positions; the resulting GElements object `Ea` now only contains the *background* elements. If we want to, we can merge the original contour elements and the newly placed background elements using the GERT_MergeElements function. The result is plotted in the right-hand panel of Figure 3.5.

Finally, it is possible to place the background elements around more than one pre-existing GContour or GElements objects, through providing a vector of these objects instead. If a mixture of GContour and GElements needs to be processed, they should be passed as a vector of cells, each containing a single object of either type.

3.3.2 Second argument: Regions

Until now, we have always filled the entire rectangular display. The second argument to GERT_PlaceElements_Background allows the specification of a custom region inside or outside which to place the elements, through a closed GContour object. The region `R` in the MATLAB code below is identical to the ellipse GContour object `C`, except that we scaled it to twice its original size using the GERT_Transform_Scale function. Whether the elements should be placed inside or outside the specified region, is set through the parameter field `in_region`, which can be set to either `true` (default) or `false`.

```
% Define the region (a GContour object)
R = GERT_Transform_Scale(C,2);

% Place background elements INSIDE REGION
peb_params.min_dist = 20;
Ea = GERT_PlaceElements_Background(E,R,peb_params);
plot(Ea);

% Place background elements OUTSIDE REGION
peb_params.in_region = false;
Ea = GERT_PlaceElements_Background(E,R,peb_params);
plot(Ea);
```

The result is shown in Figure 3.6.

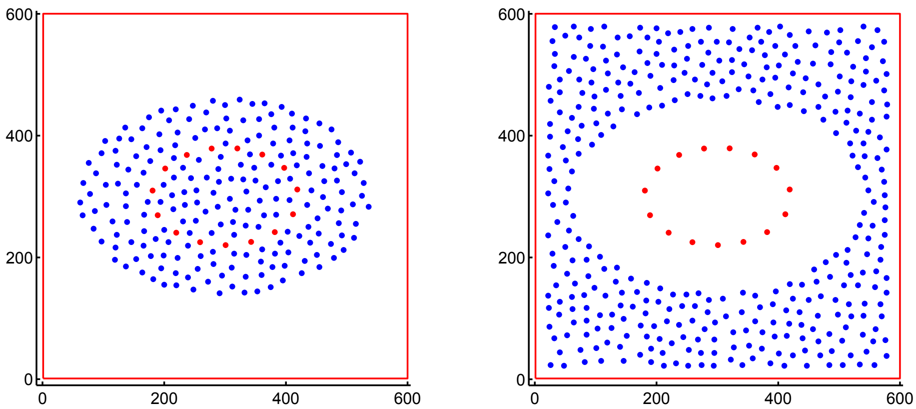

Figure 3.6 – Background element placement inside (left) or outside (right) a pre-defined region.

Again, we could do the same thing with a GContour object as the first argument instead of a GElements object (see 3.3.1). It is also possible to define multiple regions within the same display. To do this, a vector of GContour objects needs to be provided. The MATLAB code below illustrates this. Contrary to the previous examples, we leave the first input argument empty. We call GERT_PlaceElements_Background with a vector of two regions as the second argument, and the parameters for the element placement (peb_params) as the third. The first region is defined as an ellipse contour, the second region as the car outline defined before (3.1.3). The result is shown in Figure 3.7.

```
ccc;
GERT_Init;

% Define region 1:
gce_params.hax = 100;
gce_params.vax = 60;
R1 = GERT_GenerateContour_Ellipse(gce_params);
R1 = GERT_Transform_Shift(R1,[200 400]);

% Define region 2:
R2 = GERT_GenerateContour_FileTXT('car.txt',true,' ');
R2 = GERT_Transform_Shift(R2,[300 400]);
R2 = GERT_Transform_Rotate(R2, pi, 'Centroid');

% Background element placement OUTSIDE regions:
peb_params.min_dist = 10;
peb_params.dims     = [1 1000 1 1000];
peb_params.in_region = false;
Ea = GERT_PlaceElements_Background([],{R1,R2},peb_params);
plot(Ea);
```

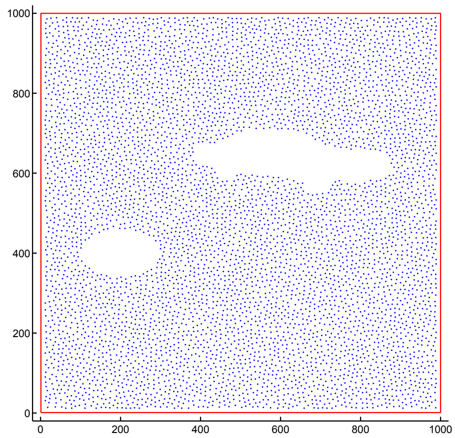

Figure 3.7 – Background element placement with multiple regions (see MATLAB code above).

3.3.3 Third argument: Parameter structure

So far, we have discussed the parameters min_dist and dims. In the previous section, we have also introduced the in_region field. Here, we will discuss how the total number of background elements and the distance to the image border can be controlled. Performance-related parameters are explained in detail in the Tips & Tricks.

By default, GERT_PlaceElements_Background continues placing background elements until not a single background element can be placed anymore that is at least `min_dist` away from previously placed contour or background elements. If you rather prefer to have a fixed number of background elements in your stimulus display, GERT can do this, too. It is sufficient to add one extra field to the `peb_params` structure: `bg_n`, the number of background elements. In Figure 3.8 we ask for exactly 250 background elements.

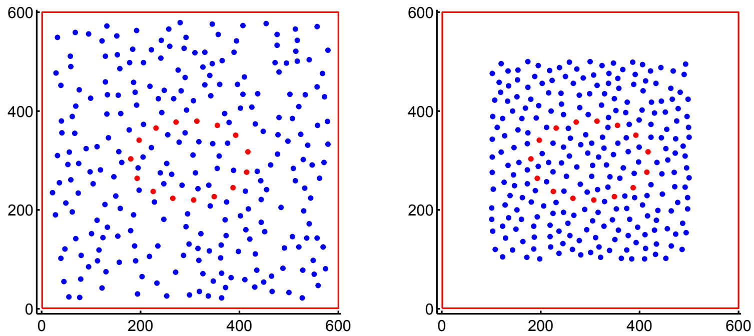

Figure 3.8 – Background element placement with a fixed number of background elements (250). Left: Default border distance of `min_dist + 1`. Right: Using a border distance of 100.

Perhaps you already noticed that no elements have been placed near the border of the display. By default, GERT does not place elements that are within a distance of `min_dist + 1` to the display border. This distance can be changed through setting the `border_dist` field (see MATLAB code below). The right-hand panel of Figure 3.8 illustrates this for a border distance of 100 (while keeping the number of elements fixed).

```
%(continuing on the code of section 3.3.1)

% Fixed number of background elements
peb_params.bg_n = 250;
Ea = GERT_PlaceElements_Background(E,[],peb_params);
plot(Ea);

% Distance to stimulus border
peb_params.border_dist = 100;
Ea = GERT_PlaceElements_Background(E,[],peb_params);
plot(Ea);
```

3.4 Local density cues

Once the positions of the contour and background elements have been computed, GERT can explicitly check for the presence of a *local density cue* in the display. A local density cue is present whenever the element density around the embedded contour is different from the element density in the background. If the cue is strong enough, it can be used to detect the presence of a contour on the basis of the element positions alone (see also Figure 2.3). Often, this cue is unwanted. GERT allows the user to minimize it through the optimization of stimulus creation parameters.

3.4.1 Check for a local density cue

GERT_CheckCue_LocalDensity checks for a local density cue in the display, and returns a value reflecting its severity. This value is similar, but strictly speaking not identical to a p-value of a statistical test, in that a one-sided evaluation is performed where both values below $\alpha/2$ and above $1-(\alpha/2)$ are considered as indicative of a density cue. In the former case, the contour elements are less dense than the background, whereas in the latter case the background is more dense.

Multiple methods are available to quantify local densities and to test for density differences between, for instance, contour and background elements. In the MATLAB code below, we define the local density as the average Euclidean distance (`method_dens = 'AvgDist'`) between each point in the display and its four (`avg_n`) nearest neighbours. Here, we decide on the presence of a local density cue if this average distance differs significantly between contour and background elements. We perform an unpaired t-test (`method_stat = 'T'`) to perform this test[5].

```
ccc;
GERT_Init;

% RFP contour:
rfp_params.freq = [2 4];
rfp_params.amp   = [10 30];
rfp_params.ph    = [0 0];
rfp_params.baser = 120;
C = GERT_GenerateContour_RFP(rfp_params);
C.x = C.x + 250; C.y = C.y + 250;

% Elements on the contour:
pec_params.cont_avgdist = 25;
E = GERT_PlaceElements_Contour(C, pec_params);

% Background elements:
peb_params.min_dist = 25;
peb_params.dims = [1 700 1 600];
Ea = GERT_PlaceElements_Background(E,[],peb_params);
```

[5] This option requires the Statistics Toolbox.

```
% Get indices for contour and background elements:
c_idx = 1:E.n;
b_idx = E.n+1:Ea.n;

% Parameters for the local density check:
cld_params.avg_n = 4;
cld_params.border_dist = 40;
cld_params.method_dens = 'AvgDist';
cld_params.method_stat = 'T';
res = GERT_CheckCue_LocalDensity(Ea,c_idx,b_idx,cld_params);
```

The parameter field `border_dist` restricts which elements are to be included in the analysis, since elements too close to the border naturally have less neighbours. The local density calculation is illustrated in Figure 3.9. Note also how in the above code we pass only the *indices* of the groups of elements to be compared. Thus, the local densities of any two sets of elements contained with the `Ea` object can be compared, not just those of contour and background elements.

The output of the GERT_CheckCue_LocalDensity function is a structure containing two fields: `pm`, the *p*-value of the one-sided statistical test, and `hm`, a logical value indicating whether the null hypothesis ("No difference between the two sets of elements") should be rejected (1) or not (0), given a default alpha-level of 0.1. That is, the null hypothesis will be rejected whenever `pm` lies outside the range 0.05 to 0.95[6]. `pm`-values lower than 0.5 imply that `c_idx` has a lower density than `b_idx`, higher than 0.5 that `b_idx` has a lower density than `c_idx`. In this specific case, the null hypothesis will be rejected (`pm`<0.001). Note that the `res` structure also contains the raw distributions of local densities, as `c1` and `c2`.

These methods should have been intuitive to understand. However, these are not GERT's default methods to check for a local density cue. First, a slightly more sensitive local density metric can often be obtained using `'Voronoi'` as the `method_dens`. This is illustrated in the upper-left panel of Figure 3.10. Instead of defining a fixed number of nearest neighbours to which the distance should be averaged, a Voronoi tesselation is performed to construct for each element a polygon, such that all coordinates within the polygon are closer to that element than to any other element in the display. Each element's local density metric is then computed as the surface of this polygon. Polygons that border the edge of the display are automatically ignored, even when they do exceed the `border_dist`. Importantly, this makes `'Voronoi'` a parameter-free metric. The `AvgDist` metric can also be made parameter-free, by omitting the `avg_n` parameter, or setting it to 0. In that case, a Delaunay triangulation will be performed to determine the natural neighbours of each element.

The third method that is available, is `'RadCount'`. Here, GERT will count the number of other

[6] Note again the subtle difference with a *p*-value as it is commonly used in statistics. We will reject high `pm`-values as well, so that `pm` can have a very natural interpretation in this context (contour density is higher versus background density is higher).

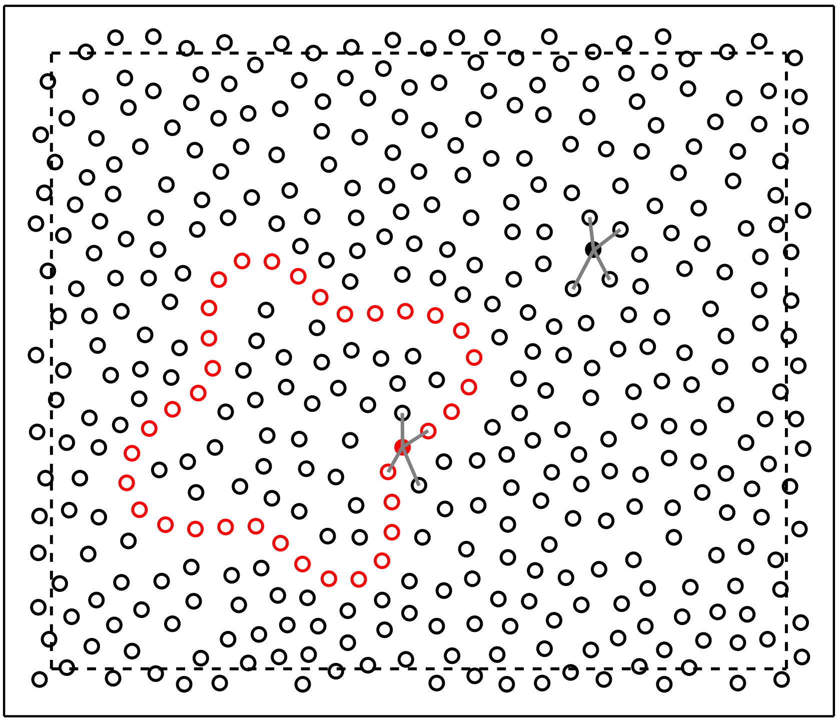

Figure 3.9 – Illustration of the local density metric calculated with the option `'AvgDist'`. For each element within the dotted rectangle (distance of `border_dist` from the display border) GERT calculates the average distance to its 4 (`avg_n`) nearest neighbours. This is visualized here for one contour element and two background elements (full circles). The blue lines connect these elements to their 4 nearest neighbours.

elements within a given radius `rad` of each element (Braun, 1999). This method is generally less sensitive than the other two, but might have specific applications.

The default statistical method defined in `method_stat` is `'MC'`, a non-parametric Monte Carlo permutation test. Compared to a t-test, this has the advantage of not requiring any parametric assumptions on the shape of the density distributions. Rather than directly comparing the observed distributions of local densities for contour and background elements, we now only calculate the difference in means between both observed distributions. Then, we repeatedly re-assign elements to either the contour or the background group, at random, and recalculate the difference in means each time. This will yield a random distribution against which we can compare the actually observed mean difference. The proportion p then reflects the number of resampled differences that are more extreme than the observed difference between the contour density and the background density. By default, GERT resamples 1000 times. This can be changed through the `mc_samples_n` parameter. The Monte Carlo resampling technique is illustrated in the lower part of Figure 3.10.

The result obtained with the default density metric and statistical test again argues against the

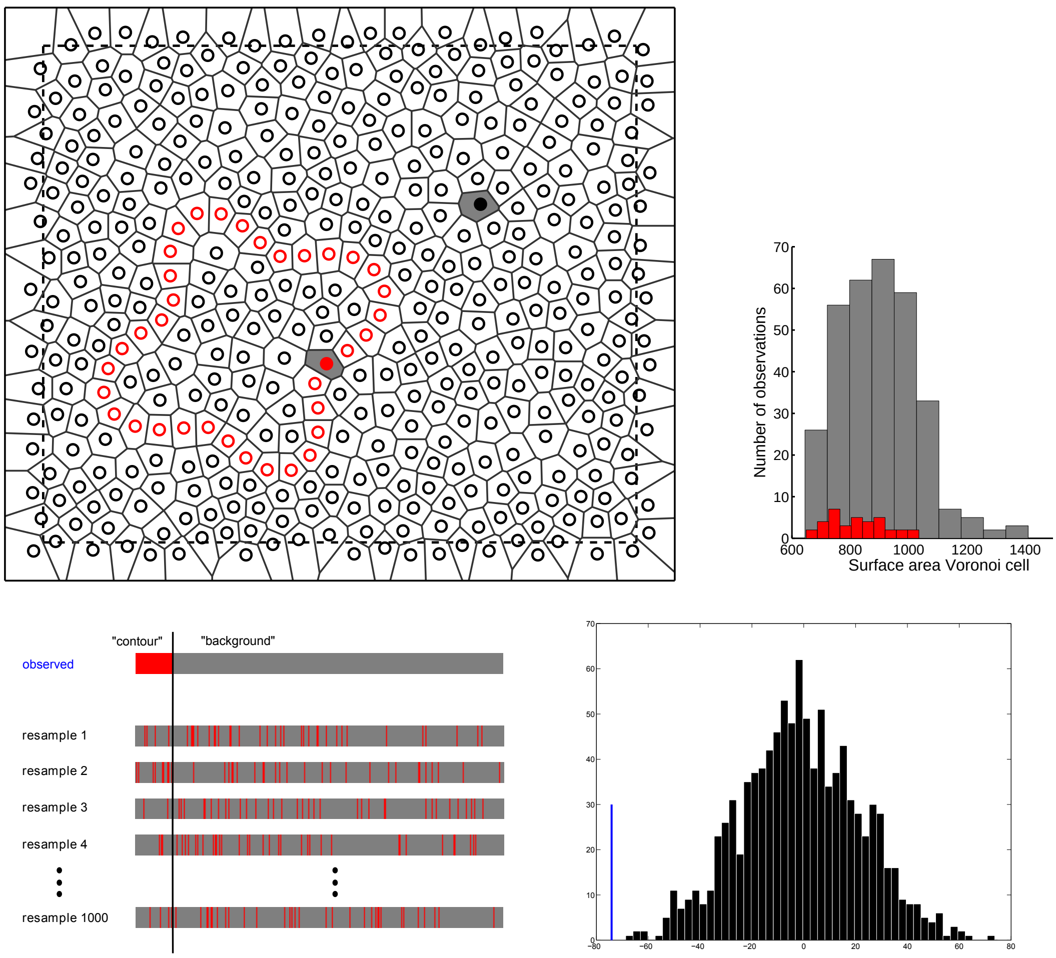

Figure 3.10 – Illustration of the default methods used by GERT_CheckCue_LocalDensity. The upper-left panel shows the `'Voronoi'` method to calculate the local density metric. For each element at least `border_dist` away from the display edge, GERT calculates the surface area of the polygon surrounding this element, unless the polygon borders the display edge. One contour element and one background element polygon are shaded as an illustration. The observed distribution of the polygon surface areas is shown in the upper-right panel, in red for the contour and in gray for the background elements.
The lower panels illustrate GERT's default Monte Carlo resampling technique. Left: Each vertical stripe in the bars represents the surface area of one Voronoi cell. In the top bar all the stripes to the left of the vertical line are contour elements. The stripes on the right of the vertical line are background elements. We subtract the mean polygon area of the background elements from the mean polygon area of the contour elements, to obtain the observed difference in local density. Then we randomly re-assign each element to the 'contour' or the 'background' group. For each of 1000 resamples we re-calculate the difference in mean surface area. Right: The random distribution of the resampled differences. The observed difference is indicated by the vertical blue line. Clearly, the observed difference is very extreme compared to a situation where no systematic difference in local densities is present.

null hypothesis. We can safely conclude that a difference in average local density is present in this display. Contour elements have significantly smaller Voronoi polygons than background elements, that is, element density is higher around the contour. Therefore, we either need to decrease the background element spacing `min_dist` in GERT_PlaceElements_Background, or increase the contour element spacing `cont_avgdist` in GERT_PlaceElements_Contour. But by how much?

```
% Parameters for local density check:
cld_params.border_dist = 40;
cld_params.method_dens = 'Voronoi';
cld_params.method_stat = 'MC';
res = GERT_CheckCue_LocalDensity(Ea,c_idx,b_idx,cld_params);
```

3.4.2 Minimize the local density cue

We have basically three options to avoid a local density cue. Perhaps the embedded figure is very simple, or you have an extraordinary mathematical gift, and then you can compute the correct number analytically. You could also randomly change parameter values and regenerate displays until you find one that does not contain a local density cue. Or, you could systematically vary the parameter values and search for the optimum. This is what GERT_MinimizeCue_LocalDensity has already implemented for you.

Let us first illustrate the problem. The MATLAB code below is used to generate Figure 3.11. We read in a contour description of a bear from a TXT file, place elements on the contour, and populate the remainder of the display. The `cont_avgdist` of the contour element placement and the `min_dist` of the background element placement define the spacing of the elements in the final image. Figure 3.11 shows that the background element spacing is quite homogeneous, but it is still possible to detect the bear because of a local density cue.

```
ccc;
GERT_Init;

% TXT contour:
C = GERT_GenerateContour_FileTXT('bear.txt',true,' ');
C.y = -C.y;
C = GERT_Transform_Center(C,[400 300],'Centroid');

% Elements on the contour:
pec_params.cont_avgdist = 25;
E = GERT_PlaceElements_Contour(C, pec_params);

% Background elements:
peb_params.min_dist = 25;
peb_params.dims = [1 800 1 600];
Ea = GERT_PlaceElements_Background(E,[],peb_params);
```

In the following MATLAB code we optimize the `min_dist` parameter of the background element placement, in order to eliminate the local density cue. The syntax of GERT_MinimizeCue_LocalDensity is a bit more complicated than for other GERT functions. On the positive side, it can

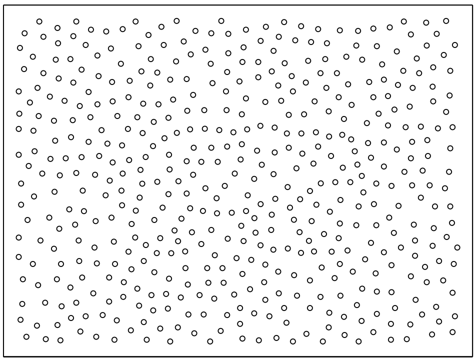

Figure 3.11 – The shape of the bear is visible in the display because of local density differences between contour and background elements.

however be flexibly applied to many types of stimulus displays. The first input argument is a text string, containing the full function by which the contour elements were placed. Alternatively, if these elements are to remain fixed, a GElements object can be provided. In this case, we use the GERT_PlaceElements_Contour function. The second input argument is a cell vector containing the variables that will serve as the arguments to this function call. The third input argument is the parameter structure for the background element placement function GERT_PlaceElements_Background, that will always be used and therefore does not need to be specified explicitly. Fourth, the `cld_params` structure contains the parameters for the local density check (see 3.4.1). In the fifth and final input argument (`optfield`) we tell GERT what parameter should be optimized. In this cell vector, we first provide the number of the function (1 for the function defined in the first input argument, 2 for GERT_PlaceElements_Background function) to which the relevant parameter belongs. Then we provide the name of the parameter, and the range of values to try. In the example below, we optimize the `min_dist` parameter of the background element placement function. From Figure 3.11 it is clear that the distance between background elements (25) was too large. We therefore expect the optimum to be smaller. We ask GERT to search for an optimum somewhere between 15 and 25, in 50 steps of equal size.

```
% Parameters for check local density:
cld_params.border_dist = 40;
cld_params.method_dens = 'Voronoi';
cld_params.method_stat = 'MC';

fnc1  = 'GERT_PlaceElements_Contour(contour, params)';
args1 = [{C},{pec_params}];
args2 = [{[]},{peb_params}];
optfield = [{2},{'min_dist'},{linspace(15,25,50)}];
ansf = GERT_MinimizeCue_LocalDensity(fnc1,args1,args2,cld_params,optfield);
```

GERT will now simply execute the function in the first argument, using the arguments in the second argument, to generate the contour elements. Then it will place the background elements using the arguments in the third argument. At each iteration, the next `optfield` parameter value will be applied to the relevant function, and the local density statistic will be computed given the `cld_-params`. The optimal value for `min_dist` is then the value that will yield a proportion p of 0.5. To find this value, GERT fits a logistic regression line through all computed proportions p, as illustrated in Figure 3.12.

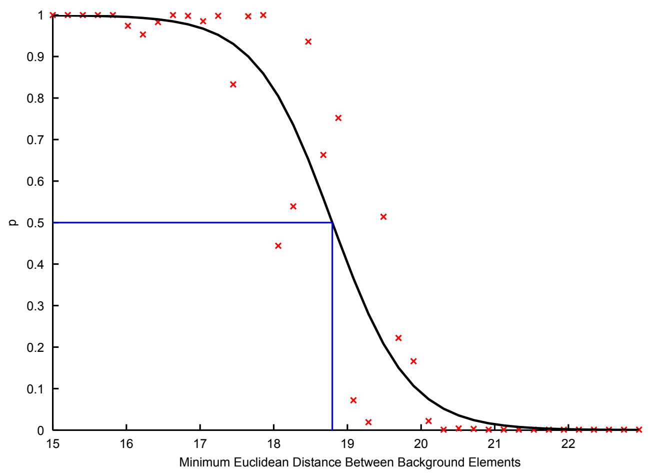

Figure 3.12 – To minimize the local density cue, we need an estimated proportion p of 0.5. The X-axis indicates the 50 steps between 15 and 25 (the `min_dist` range we are interested in). The red crosses represent the proportion p of the local density check. To find the optimal `min_dist` value (i.e., the X value belonging to an Y value of 0.5), we fit a logistic regression through these points.

GERT_MinimizeCue_LocalDensity returns the `min_dist` value estimated to yield a proportion p of 0.5. We obtained a value of 18.79. We generate a new display, changing the `min_dist` value from 25 to 18.79. The result is shown in Figure 3.13. You'll probably have a hard time spotting the bear in this image.

One final note should be made however. Using optimal parameter values does not guarantee that each display generation will indeed have eliminated the local density cue succesfully. Most often there is a random component to the placement of elements. Therefore the local density check should still be performed even when using optimized parameter values.

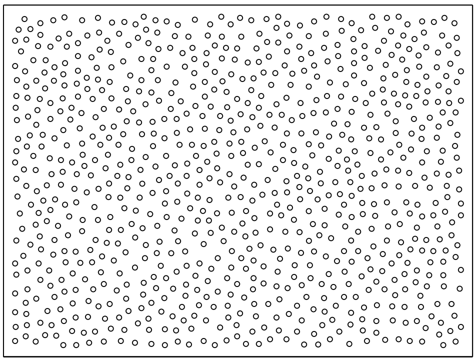

Figure 3.13 – The shape of the bear is no longer visible in the display because the local density cue has been minimized.

3.4.3 Variability in local density

We have so far only controlled for a difference in *average* local density between contour and background elements. However in some cases, the researcher might also want to control differences in the *variability* of these local density metrics, i.e. the differences in regularity between contour element placement and background element placement. In this case, the `'Random'` method of contour element placement is preferable. A `min_dist` value close to the one used in the GERT_PlaceElements_Background function is then most probable to eliminate the local density cue. A variability measure is available for each of the three density metrics introduced. In the case of `'AvgDist'`, a binned histogram is constructed out of the full series of individual element distances, for contour and background elements separately. The summed absolute difference between these two histograms is then evaluated using a Monte Carlo permutation test. The `'Voronoi'` approach is similar, using the full series of polygon areas. Finally, `'RadCount'` counts the number of elements found within expanding radii around each element, before constructing the binned histogram.

To perform a variability check on the local density information, call the GERT_CheckCue_LocalDensity function with 2 output arguments instead of 1. The second `res` structure will then also contain a `hm` and `pm` value, as well as the raw data for the observed histograms in `c1` and `c2`, for visual inspection. Note that the variability check on the local density information can only be performed when the preferred statistical method is Monte Carlo resampling, i.e. only if you specify the `method_stat` field of the local density parameters as `MC`.

3.5 Rendering the display

Once we are satisfied with the element positions, the display can be rendered as an image using the GERT_RenderDisplay function. This function has 4 input arguments:

1. `draw_fnc`: The first input argument is a function handle to a specific drawing function for an individual element. GERT currently has 8 built-in functions to draw Gabor elements, Radial Gabors (new in GERT v1.1), Gaussian blobs, Rectangles, Ellipses, Triangles, Polygons, or custom Images. We will illustrate some of these drawing functions below. For details on the other functions, we refer to the Full Documentation at the end of this manual. It is also possible to write your own element drawing function, as long as it complies with the requirements of GERT_RenderDisplay (see Tips & Tricks).
2. `elements`: A GElements object obtained from an element placement function, containing the element positions.
3. `el_params`: A parameter structure for the individual elements. The fields of this structure can be different for different drawing functions.
4. `img_params`: An optional parameter structure defining the parameters of the rendering function itself .

To illustrate some of the possibilities of GERT_RenderDisplay beyond those already discussed in the Tutorial, we will first generate a dot lattice (Kubovy & Wagemans, 1995). These lattices are often used to investigate the Gestalt principle of proximity.

In the MATLAB code below we define the XY coordinates of the dots in the lattice, using standard MATLAB commands. That is, we set the GElements object manually rather than through GERT functions. Next, we rotate the grid and remove the elements that fall outside a circular window defined by the GContour object `circle`. To achieve this, we use the GERT_Aux_InContour function, which returns the indices of elements falling in or out of the polygon described by the GContour object. We then illustrate how the new GElements object can be constructed directly from these indices.

```
ccc;
GERT_Init;

% Lattice:
X = linspace(1,1000,50); Y = linspace(1,1000,45);
[XX YY] = meshgrid(X,Y);
X = reshape(XX,[1 numel(XX)]);
Y = reshape(YY,[1 numel(YY)]);

% Define GElements object
E = GElements([X;Y],[1 1000 1 1000]);
```

```
E = GERT_Transform_Rotate(E,pi/5);

% Remove elements outside a central circle:
gce_params.hax = 300;
gce_params.vax = 300;
circle = GERT_GenerateContour_Ellipse(gce_params);
circle = GERT_Transform_Shift(circle,[500 500]);
[in_idx, out_idx] = GERT_Aux_InContour(E, circle);
E = GElements(E,in_idx);

plot(E);
```

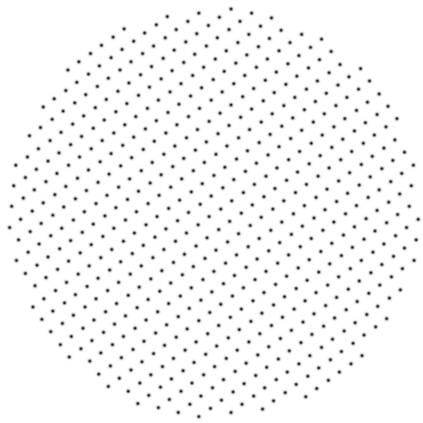

Figure 3.14 – A dot lattice. The dots are Gaussian blobs.

Below we do the actual rendering. We tell GERT_RenderDisplay to use the GERT_DrawElement_Gaussian function to draw Gaussian blobs at the element positions in `E`. The parameters of the individual image patches are defined in `el_params`. As in 2.5, it is possible to define each parameter field as either a scalar (if all elements have the same value for this parameter), or as a vector (if different elements have different parameter values). In this case all elements can remain identical. However, we want a white background and black Gaussians. The white background is easily achieved, by setting `img_params.bg_lum` to 1 instead of the default 0.5. To make the Gaussians as dark against this background, we make use of the optional `lum_bounds` element parameter, which defines respectively the background and the peak luminance of the Gaussian image patch. Since constant parameter values should always be scalar, we pack this 1×2 vector into a 1×1 cell variable, as required by the GERT_DrawElement_Gaussian drawing function, and pass the values as `[1 0]`. The result is shown in Figure 3.14.

```
% Gaussian elements parameters:
el_params.sigmax = 1.2;
el_params.sigmay = 1.2;
el_params.size = 5;
el_params.lum_bounds = {[1 0]};
```

```
% Image parameters
img_params.bg_lum = 1;
IMG = GERT_RenderDisplay(@GERT_DrawElement_Gaussian,E,el_params,img_params);
figure; imshow(IMG);
```

Up until now we did not explicitly specify the pixel dimensions of the final image to the GERT_-RenderDisplay function. This seems logical, since these dimensions were already contained within the GElements object `E`. However we have been hiding something from you: The `dims` field of a GElements object is not necessarily defined in pixel units, but can be specified in any arbitrary units. But since we did not explicitly specify the pixel dimensions of the final image, GERT interpreted the dimensions in the GElements object `E` as pixel dimensions. It is however possible to add a `dims` field to the `img_params` as well, specifying the pixel dimensions of the final image. The previously specified dimensions in arbitrary units will then be converted to these new pixel dimensions, as will all element positions. We will demonstrate this in the following example.

```
ccc; GERT_Init;

% (1) Read in and position the eagle contour:
C = GERT_GenerateContour_FileTXT('eagle.txt',true,' ');
C = GERT_Transform_Flip(C,0,'Centroid');
C = GERT_Transform_Center(C, [0 0], 'Centroid');

% (2) Place contour elements:
pec_params.method = 'SerialEquidistant';
pec_params.cont_avgdist = 25;
pec_params.eucl_mindist = 20;
[E ors] = GERT_PlaceElements_Contour(C,pec_params);
E.dims = [-250 250 -250 250];

% (3) Place background elements:
peb_params.min_dist = 20;
Ea = GERT_PlaceElements_Background(E,[],peb_params);

% (4) Retrieve which elements inside, outside and on the contour
[in_idx, out_idx, on_idx] = GERT_Aux_InContour(Ea, E);

% (5) Define the triangle patches:
el_params.width = 10;
el_params.height = 17;
el_params.scale = 1;
el_params.size = 20;
el_params.aa = 8;
el_params.or(in_idx) = repmat(main_axis(C),[1 numel(in_idx)]);
el_params.or(on_idx) = ors;
el_params.or(out_idx) = 2*pi*rand([1 numel(out_idx)]);
el_params.lum_bounds(in_idx) = {[0.4 0; 0.25 0; 0.15 0.6]};
el_params.lum_bounds(on_idx) = {[0.4 .95; 0.25 0; 0.15 0]};
el_params.lum_bounds(out_idx) = {[0.4 .95; 0.25 .95; 0.15 .95]};

% (6) Define the image parameters:
img_params.bg_lum = [0.4 0.25 0.15];
img_params.dims = [1000 1000];
```

```
% (7) Render:
IMG = GERT_RenderDisplay(@GERT_DrawElement_Triangle,Ea,el_params,img_params);
figure; imshow(IMG);
```

In (1) we read in the shape outline of an eagle as a GContour object. We introduce the GERT_Transform_Flip function here to flip the eagle around the horizontal axis through its centroid, and position the centroid at (0,0). We then (2) place elements on this contour using the 'SerialEquidistant' method. The display dimensions are specified as [-250 250 -250 250], obviously not pixel values. (3) We now fill the display with background elements. Note that we do not control for local density cues here. In (4) we retrieve the indices of the elements inside, outside and on the shape outline. This is a slightly different use of GERT_Aux_InContour, in that we specify the relevant polygon through a GElements object. This is why we can also retrieve the indices of elements exactly on the outline. The actual rendering starts from (5), where we specify the parameters of the image patches, in this case triangles. Many of these parameters should be self-explanatory. In the `or` field we specify the orientations of the individual triangles. Triangles inside the contour are oriented along the main axis (`C.main_axis`) of the contour, triangles outside the contour have random orientations, and triangles on the contour are set using the previously obtained `ors` variable. The `lum_bounds` field again specificies the luminance boundaries of the image patches. But now, we are going to make a color image. Normally, we would need a 1×2 vector to specify foreground and background luminance. To create color, we specify a 3×2 matrix, one pair of values for the R, the G, and the B layer. Remember that this matrix should always be put into a cell or, in this case, a vector of cells equal in length to the number of elements. Similarly, the image background luminance `img_params.bg_lum` is set to a triplet of values to create color. Finally, we also specify the final pixel dimensions of the image, and render the full display. The resulting image is shown in Figure 3.15.

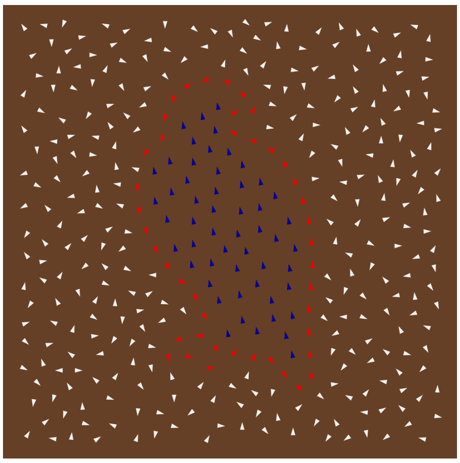

Figure 3.15 – The shape outline of an eagle rendered with colored triangles.

Maybe you have tried making denser displays already. In that case, you will probably have noticed that the square element image patch can 'cut into' neighbouring elements, since they are simply pasted on top of previously placed elements. Such slight overlaps can be taken care of by setting `img_params.blend_mode` to `'MaxDiff'`. For each pixel, GERT will now retain the value that deviates the most from the background luminance. In the MATLAB code below, we do not have a contour description. We have randomly placed background elements (more than 3000 of them!) into an empty display equal in size to a JPG image of a famous painting. We then render the elements as Gabor patches, such that their phase offset corresponds to the JPG pixel luminance at the same location within the image. The individual Gabor patches are 25×25 pixels large (`size`×2+1) and partially overlap. Thanks to the `'MaxDiff'` blending mode however, no such artefacts can be noted. You may set the `blend_mode` to `'None'` to see what would otherwise have happened.

```
ccc;

% Read in image:
im = imread('MonaLisa.jpg');
im = im';
im(:,1:end) = im(:,end:-1:1);
dims = size(im);

% Place background elements:
peb_params.min_dist = 6;
peb_params.dims = [1 dims(1) 1 dims(2)];
E = GERT_PlaceElements_Background([],[],peb_params);

% Gabor patches:
el_params.or = pi/2;
el_params.sigma = 3;
el_params.freq = 0.09;
el_params.size = 12;
el_params.scale = 0.5;
lind = sub2ind(dims,round(E.x),round(E.y));
lums = double(im(lind))/255;
el_params.phase = pi - (lums*pi);

% Image parameters:
img_params.blend_mode = 'MaxDiff';

% Render:
IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor,E,el_params,img_params);
figure; imshow(IMG);
```

3.6 Logging the stimulus creation process

Many of you are probably familiar with the following situation: A suitable stimulus set was created for an experiment, and then months or even years later, you need to know how they were generated exactly. Somewhere in the messy depths of your backup folders, seven different versions of stimu-

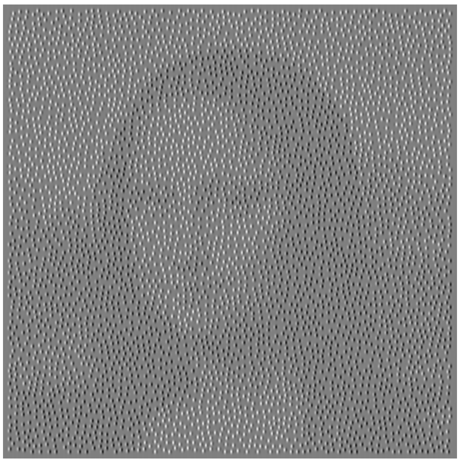

Figure 3.16 – A painting rendered with overlapping Gabor patches. Each Gabor patch has a phase offset that corresponds to the corresponding pixel intensity of the underlying JPG image.

lus generation scripts are found, none of which seem to correspond exactly to what you did for that particular study. And the method section of your article was not detailed enough to make exact reproductions.

GERT aims to provide a solution to this common problem, through its *logging* functionality. When enabled, your main script will be saved as it was upon execution, and many GERT functions will store their exact input and output variables. For each stimulus, a .mat data file can then be saved along with the final stimulus, containing all this information.

3.6.1 The GLog class

Logging is implemented as a single class, called GLog. In the GERT_Init routine, a global log object named `GERT_log` is created, to which all GERT functions will write. The GLog class contains three properties. The first, `Info`, is a struct containing general information on the status of the user's computer and MATLAB installation. The second, `Functions`, is a struct grouping all stored messages and variables per function that used them as an input or output, and per call to that function. A timestamp is also saved. The third, `Files`, stores the contents of entire (plain-text) files into the log.

The implemented methods are fairly straightforward. `start` enables logging, creates the `Info` struct, and stores the .m file from which it was called into the `Files` struct. Thus, if you call `start` from your main script, all the MATLAB code in this main script will automatically be saved. `stop` disables the logging, and clears all the logged information. `add` allows the user to add specific entries to the log, either messages, variables or files (see below for an example). `group` is a method that is mostly used internally, and allows the grouping of variables under a single function call within

Functions, rather than lumping everything together when calling the same function twice. Finally, fetch retrieves the log as a struct that can be saved to a .mat file. It is important to note that the fetched structure is only a copy of the log. Changing values in this structure will not affect the GLog object itself, nor will clearing the log remove the fetched copy.

3.6.2 An example

The following example illustrates how logging works in GERT. First, logging is enabled through calling GERT_log = start(GERT_log). Do not forget to always re-assign the outcome of any class method back to GERT_log. The next GERT function calls will then automatically log all their input and output activity. After the final stimulus image has been created, we illustrate the manual adding of three types of information: Messages, Variables, and Files. We also illustrate how the message and the variable can be grouped together in the log structure using the group method. Try running the script without the grouping to note the difference. Finally, the log is fetched and saved to a .mat file. After the fetching, the full log structure can be browsed in the workspace explorer in MATLAB.

```
% Clear everything, initialize GERT
ccc; GERT_Init;

% Start logging, the current m-file is now automatically added to the log
GERT_log = start(GERT_log);

% Create a contour (now automatically logged)
gce_par.hax = 100; gce_par.vax = 50;
C = GERT_GenerateContour_Ellipse(gce_par);
C = GERT_Transform_Center(C,[200 200],'Custom',[0 0]);

% Put elements on the contour (now automatically logged)
pec_par.cont_avgdist = 25;
Ec = GERT_PlaceElements_Contour(C,pec_par);

% Put elements on the background (now automatically logged)
peb_par.min_dist = 40;
peb_par.dims = [1 400 1 400];
Ea = GERT_PlaceElements_Background(Ec,[],peb_par);

% Render the image (now automatically logged)
el_par = struct;
IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor,Ea,el_par);

% Add a custom message and a custom variable, both grouped under a single
% structure within GLog.Functions.glog_example
GERT_log = group(GERT_log,'on');
GERT_log = add(GERT_log,'msg','This is an example message');
GERT_log = add(GERT_log,'var',IMG, 'output_image');
GERT_log = group(GERT_log,'off');
GERT_log = add(GERT_log,'msg','This is a separately grouped example message');

% Add a custom file to GLog.Files
GERT_log = add(GERT_log,'file','GERT_RenderDisplay.m','GERT_RenderDisplay');
```

```
% Convert the log to a struct
l = fetch(GERT_log);

% Save this struct to a .mat file
save stimlog.mat l;

% Clear the log. You will notice that this does not affect the
% copied structure 'l' that we fetched. But if we would try fetch the log
% again, GERT would return an error since no log exists anymore.
GERT_log = stop(GERT_log);
```

3.7 Learning more

You are now familiar with the most important aspects of GERT. In the next part, we will discuss more advanced functionality, and things that could in general be making your life together with GERT easier. For a complete overview of all options and argument requirements of GERT functions, we refer to the Full Documentation at the end of this manual.

PART IV

Tips & Tricks

4.1 Speed

Perhaps you found GERT to be a bit slow in some of the examples that were given. Here we will give an overview of the various optional parameters that can help speed up stimulus construction. Even in those cases where you would want the best possible stimulus for the eventual experiment, you could be saving a lot of time when you are merely trying to construct a stimulus that is more or less to your liking.

4.1.1 Describing the contour

As discussed, continuous contours are defined as a discrete set of Cartesian coordinates. The resolution of the discretization will have an effect on generation speed throughout the stimulus construction process. For the GERT_GenerateContour_Ellipse and GERT_GenerateContour_RFP functions, and possibly other future functions starting from a polar definition, the number of contour points is determined by the parameter `th_n`. The default here is 1000. GERT_GenerateContour_FileSVG on the other hand utilizes a function argument `res`, that will determine the number of contour points put on each subcurve of the path definition (default: 100). GERT_GenerateContour_FileTXT will construct the contour at the resolution contained within the original TXT file, which contains already discretized Cartesian coordinates.

One strategy to further simplify an existing contour definition could be to retain only the coordinate points with an odd index. For instance:

```
idx = 1:2:C.n;
```

```
C = GContour(C,idx);
```

Another method could be to eliminate contour points within a certain Euclidean distance of one another:

```
min_dist = 5;
dist = GERT_Aux_EuclDist(C.x,C.y,C.x,C.y);
too_close = logical(dist<min_dist);
too_close = tril(too_close) & ~eye(C.n);

for i = 1:C.n
    if any(too_close(i,:))
        too_close(:,i) = false;
    end
end

idx = find(~any(too_close,2))';
C = GContour(C,idx);
```

Or, lastly, within a certain distance *along the contour*, using an algoritm similar to:

```
min_dist = 5;
dist = diff(C.cdist);
d = 0; keepers = [];

for i = 2:C.n
    if C.cdist(i) > d+min_dist
        keepers = [keepers i];
        d = C.cdist(i);
    end
end
C = GContour(C,keepers);
```

These techniques will in the future be implemented into a single GERT function.

4.1.2 Placing the elements

Contour elements

`'ParallelEquidistant'` is the fastest method within GERT_PlaceElements_Contour. There is not much one can do to speed it up even further. However, an optional parameter `eucl_mindist` is available, specifying the minimal Euclidean distance to keep between elements. Setting this parameter too strict might cause the element placement to fail, after which this method will attempt to place them again `noise_retries_n` times. Ofcourse if there is no randomness to the element placement, such as positional jitter, the procedure will fail each time. But if there is randomness, and one would want to speed up generation for a quick view of the stimulus, it helps to temporarily reduce `eucl_mindist`.

`'SerialEquidistant'` is a slower method to place contour elements. If the contour is simple and smooth enough for parallel placement, it should not be used. Similar to the above method though, `eucl_mindist` can be set to a less strict value to speed up the generation of the stimulus display for a quick view. The `dist_retries_step` parameter is less drastic, and controls the number of positions to try out for the serial element placement (default: `cont_avgdist` divided by 5). As such increasing this value will increase the speed of element placement, but probably also decrease the quality of the stimulus.

`'Random'` uses techniques similar to those used in the background element placement function. Options for tweaking its speed are therefore analogous to that of the next section.

Background elements

Naturally, the number of points to place, set either by `min_dist`, `bg_n` or `border_dist`, are a big factor in the speed of the GERT_PlaceElements_Background function. However these parameters are usually fixed during stimulus design.

We have until now not expanded on the exact methods used in this function. Basically, for optimal speed, we combine two techniques. One way to place elements is to keep track of all remaining possible positions for the next element, that is at a sufficient distance from previously placed elements, and randomly place the element at any of these positions. The advantage of this method is that it allows filling the display completely, and does not slow down towards the end. However, its speed is highly dependent on the number of possible positions, that is, the `resolution` parameter. Another method is to generate a random coordinate within the display, and check its Euclidean distance to all existing elements to see whether a new element can be placed there, or whether a new coordinate pair should be generated. This approach is resolution-free, as keeping track of the available positions is not necessary, but becomes slower as the number of available positions becomes smaller. In GERT, we combine both. We draw a large number of elements at once from the remaining possible positions, determined by the parameter `batch_size`. Then we check within this batch which elements are too close to one another in Euclidean distance, and only place those that are sufficiently far away. The result is that we can generate even high-resolution displays at a great speed. To increase speed, the user can lower the `resolution` (default: the integer part of the size of the longest dimension), or seek the optimal `batch_size` (default: 200). Too small a batch size increases the number of queries to the remaining possible positions, slowing down the function. Too large a batch size prompts the computation of a large Euclidean distance matrix, also slowing down the function.

4.1.3 Local density cues

In GERT_CheckCue_LocalDensity, the relative speed of the density metric methods depends on the number of elements. For large arrays of elements that are all part of the comparison, the default `'Voronoi'` method will be fastest, followed by `'RadCount'` and finally `'AvgDist'`. Performance of `'AvgDist'` with the `avg_n` parameter set to 0 or omitted is however comparable to the `'Voronoi'` method. When the number of elements is large but only a small subset are part of the comparison, `'Voronoi'` is the slowest method, followed by `'AvgDist'`, whether `avg_n` is set or not. `'RadCount'` is then the fastest method. When the total number of elements is small, all methods are fast. The `avg_n` (other than 0) and `rad` values do not affect speed.

The default `'MC'` statistical method to compare both sets of elements, is slower than the `'T'` method. Reducing the `mc_samples_n` parameter will reduce the difference, but also the precision of the test. However when the computation is part of the GERT_MinimizeCue_LocalDensity optimization procedure, this imprecision will likely average out across the generated curve.

Specifically for the GERT_MinimizeCue_LocalDensity function, speed is obviously helped by making the individual calls to GERT_CheckCue_LocalDensity faster, and by reducing the number of parameter values to test. If the parameter to be optimized belongs to the background element placement function, it will also help to provide a fixed set of previously existing elements rather than an element generation function. Note however that this might bias the results whenever the contour element placement routine for the final stimulus image contains a random component.

4.1.4 Rendering the display

The rendering of the display is often the slowest part of stimulus generation. This is a sacrifice to the flexibility of the GERT_RenderDisplay function: Each image patch is generated separately by an element drawing function of choice and pasted into the larger image. However some corners can be cut.

Global rendering

In many stimulus displays, large numbers of identical elements are present. By default, GERT will still generate from scratch each of those identical elements. Large speed gains can in that case be obtained by setting the `global_rendering` option in the `img_params` parameter structure to `true`. GERT will then process the entire element parameter structure into a unique identifier, and recycle a previously generated patch whenever an identical identifier is encountered - even across different calls to the GERT_RenderDisplay function. To flush this global storage of element images, clear the global variables `GERT_glob_el_ids` and `GERT_glob_el_patches`, or use `ccc`. Note that we cannot absolutely guarantee that global rendering will work correctly. As many parameters are being condensed into one (very long) number, it might coincidentally happen that two different element parameter structures result in the same identifier.

The global rendering functionality may be exploited further by discretizing continuous parameter spaces. For instance, consider a display where all Gabor orientations are randomized using the MATLAB `rand` function. No two elements will then be exactly identical, and every element will be generated from scratch. However in practice it could be sufficient to have a resolution of one degree of rotation angle, resulting in at most 360 different image patches. This can be achieved by using the GERT_Aux_RestrictResolution function.

As an example, we re-generate the Mona Lisa display from the previous part of this manual. You may have noticed that its generation took a long time. In the code below, we turn on global rendering and discretize the phase of the Gabors to 20 different values only. On my computer, the rendering time was reduced from 7.5 seconds to just 0.4 seconds, for a display that contained about 3500 elements.

```
ccc;

% Read in image:
im = imread('MonaLisa.jpg');
im = im';
im(:,1:end) = im(:,end:-1:1);
dims = size(im);

% Place background elements:
peb_params.min_dist = 6;
peb_params.dims = [1 dims(1) 1 dims(2)];
E = GERT_PlaceElements_Background([],[],peb_params);

% Gabor patches:
el_params.or = pi/2;
el_params.sigma = 3;
el_params.freq = 0.09;
el_params.size = 12;
el_params.scale = 0.5;
lind = sub2ind(dims,round(E.x),round(E.y));
lums = double(im(lind))/255;
el_params.phase = GERT_Aux_RestrictResolution(pi - (lums*pi),pi/20);

% Image parameters:
img_params.blend_mode = 'MaxDiff';
img_params.global_rendering = true;
```

```
% Render:
IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor,E,el_params,img_params);
figure; imshow(IMG);
```

Note that the effect of `GERT_Aux_RestrictResolution` is in this case rather limited, since the phase values were already discretized to the 256 luminance values present in the original image. When element parameter values are truly continuous, the speed gains are much larger. To see this for yourself, set `el_params.phase = rand(1,E.n);` in the above script. Execution time will then be longer with global rendering enabled than without, because the function is investing in creating unique identifiers and storing element images, but no two parameter structures will ever be the same.

Element parameter checking

GERT contains extensive parameter checking and error handling in all of its functions. While this makes it easy to detect mistakes, it also takes up processing time. This is especially the case for the element drawing functions, which may be called thousands of times for a single display generation. For this reason GERT allows the user to disable the element error checking. This is done through declaring the global variable `GERT_elerrcheck` and setting it to `false`. By default, `GERT_elerrcheck` is set to `true` in the GERT_Init routine.

Anti-aliasing

Geometrically defined elements often have an anti-aliasing parameter, `aa`. This avoids a jagged look on straight edges, and other artefacts caused by the limited resolution of a pixelated image. When in the design stage of a stimulus, you might not care about this, and instead prefer to speed up your display generation. It is then advisable to set the anti-aliasing to a lower level, where '0' will disable this option completely. A full explanation of the anti-aliasing algoritm used in GERT is given below.

Element size

Minor speed gains may be obtained by reducing the element size parameter to what is minimally required to display the entire graphic. Often, this parameter is called simply `size`, and reflects the element patch pixel size minus one, divided by two. When the blend mode is set to `'None'`, i.e. simple pasting of elements on top of one another, reducing element image size will also create much cleaner displays.

4.2 Positional jitter

We have until now strived for contour element placement exactly on the contour, and often even equidistant along the contour. The methods present for introducing random jitter to these positions, are different between the different element placement methods.

4.2.1 'ParallelEquidistant' and 'SerialEquidistant'

When using GERT_PlaceElements_Contour, the methods `ParallelEquidistant` and `SerialEquidistant` have a similar approach to adding jitter. Basically, we identify three sources of position randomization. First, the `cont_startpos` of the first element along the contour can be set. By default, this parameter is set to a random value between 0 and `cont_avgdist`. As an alternative, a specific value can be set, as a proportion of `cont_avgdist`. Second, the relative position of the elements can be jittered along the contour, introducing a deviation from equidistance. For this, we set `noise_oncont`. Third, position jitter can be added perpendicular to the contour, using the local tangent information present in the GContour object provided. Setting the `noise_offcont` parameter will do this for you.

For the position of the first element, the random distribution from which the position jitter is drawn is always uniform. However for the jitter along or perpendicular to the contour, three options are available through the `noise_method` parameter. The default option is `'Uniform'`, in which case a scalar noise value must be provided. The maximal value of 1 results in a maximal range of deviations, between -0.5 and 0.5 times `cont_avgdist`. The second option is `'Gaussian'`; again, the scalar noise value must lie between 0 and 1. In this case, a value of 1 implies that two standard deviations of noise correspond to 0.5 times `cont_avgdist`. Third and finally, a `'Vector'` may be provided, that will serve as a custom distribution from which to draw jitter distances for each element; these noise values are to be understood as a proportion of `cont_avgdist`, and must lie between -0.5 and 0.5.

For both methods, `noise_retries_n` attempts will be undertaken, should the randomization outcome lead to a violation of other requirements, such as `eucl_mindist`. Do note though that the `SerialEquidistant` method was in fact not designed for adding significant amounts of jitter, but for smoothly rendering difficult contours.

4.2.2 'Random'

The third method of GERT_PlaceElements_Contour, `Random`, is completely different from the other two. It will randomly place elements on the GContour until the `el_n` goal is reached, or no more elements can be placed given the `eucl_mindist` requirement. It works similarly to GERT_-PlaceElement_Background, as its parameter fields testify. To place elements away from the actual contour, set the `noise_dilrad` parameter. This will allow random elements to be placed within a given Euclidean distance to the contour definition, instead of necessarily exactly on it.

4.2.3 Snakes

The GERT_PlaceElements_Snake function contains jitter parameters analogous to the equidistant methods discussed above. The reader will remember that snake elements are placed on the midpoint of connected line segments. Logically, the `pt_noise_onseg` parameter will then randomly shift the element along the segment line, whereas `pt_noise_offseg` will move it perpendicular to the segment's orientation. In both cases these noise values correspond to the maximal absolute noise value of a uniform distribution centered on 0.

4.3 Orientation jitter

To systematically control the salience of a contour embedded in a background of randomly oriented Gabor elements, researchers often add orientation jitter to the orientation of the contour elements. GERT contains no special function to do this. Luckily however, it is easy enough to achieve using the standard MATLAB random number functions. For instance, consider this case of uniform orientation jitter, maximally 30 degrees away from perfect collinearity in either direction.

```
% Initiate GERT
ccc; GERT_Init;

% Retrieve the contour from an SVG file, and center it
C = GERT_GenerateContour_FileSVG('tortoise.svg');
C = GERT_Transform_Center(C, [500 500], 'Centroid');

% Place the contour elements equidistant along the contour
pec_params.method = 'ParallelEquidistant';
pec_params.cont_avgdist = 30;
[E ors] = GERT_PlaceElements_Contour(C, pec_params);

% Place the background elements within a 1000x1000 pixel display
peb_params.min_dist = 24.58;
peb_params.dims = [1 1000 1 1000];
Ea = GERT_PlaceElements_Background(E,[], peb_params);
```

```
% Determine which elements are inside, outside, and on the contour
[f_idx b_idx c_idx] = GERT_Aux_InContour(Ea, E);

% Render the display with Gabor elements
% Apply uniform orientation jitter between -30 and +30 degrees
gabel_params.freq = 0.09;
gabel_params.sigma = 3;
gabel_params.or = 2*pi*rand(1,Ea.n);
gabel_params.or(c_idx) = ors + ((rand(1,length(c_idx))-0.5)*(pi/3));
IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor, Ea, gabel_params);
figure; imshow(IMG);
```

To apply normally distributed orientation noise, use the `randn` function instead.

4.4 No background elements between contour elements

We have until now always provided one single `min_dist` value when placing background elements. However, consider a situation where the elements on the contour are placed sparsely. Background elements are allowed to be close to these contour elements, but not to fall inbetween them, so that the perception of a continuous contour is not disturbed. The following example will show how this can be done by supplying both a GContour and a GElements object to GERT_PlaceElements_Background, and setting more than one `min_dist` value.

As explained in 3.3.1, GERT_PlaceElements_Background will accept both GContour and GElements objects, or a combination of these when packed into a cell vector. GElements coordinates will also be placed as elements in the final GElements output object, whereas GContour coordinates will only be used to restrict the placement of background elements. Here, we provide both the GElements object, and the GContour object on which it was based. This will prevent new elements from coming close to the entirety of the contour definition, as in the left panel of Figure 4.1. However, they should not be placed too far away, either. This is where we need to set separate `min_dist` values for the distance to the elements, and the distance to the contour, so that a display similar to the right panel of Figure 4.1 can be obtained.

GERT_PlaceElements_Background will for its `min_dist` parameter accept either a scalar, a value that is then applied to all distances kept, or a vector equal in length to the number of GElements or GContour objects provided as the first argument, plus one. The first value in the vector will then refer to the distance to keep between newly placed elements, the next values to the distances to keep from the elements or contour points within the objects provided. The following example contains the code to create the right-hand panel in Figure 4.1.

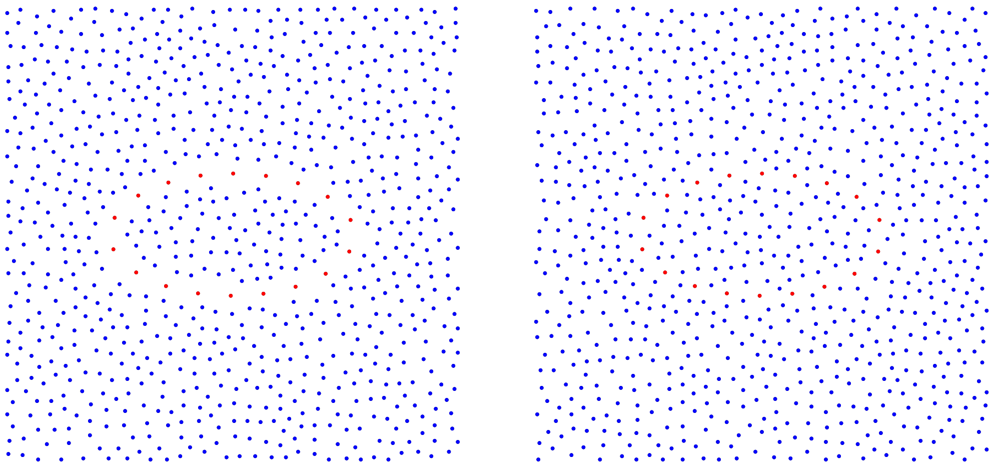

Figure 4.1 – In the left panel, the same minimal distance is kept to the full contour description as is kept to the elements placed on them. In the right panel, the distance to the contour description has been decreased but does not equal 0.

```
ccc;
GERT_Init;

% Generate contour
gce_params.hax = 10;
gce_params.vax = 5;
C = GERT_GenerateContour_Ellipse(gce_params);

% Place elements on contour
pec_params.cont_avgdist = 2.5;
E = GERT_PlaceElements_Contour(C,pec_params);

% Place elements in background
% Staying 1 away from other elements, and 0.2 from the contour description
peb_params.min_dist = [1 1 0.2];
peb_params.dims = [-20 20 -20 20];
peb_params.resolution = 371;
Ea = GERT_PlaceElements_Background({E, C},[],peb_params);
plot(Ea);
```

4.5 Write your own drawing function

Perhaps none of the default GERT drawing functions were to your liking. In that case it is not necessary to go through the trouble of writing your own equivalent of GERT_RenderDisplay, you can just write your own drawing function for a single element. The requirements are that this function can take only one single argument, namely a parameter structure, that all fields in this structure contain scalars (vectors and matrices can be packed as a 1×1 cell), and that the return argument is a rectangular image patch of odd size and type double, with luminance values between 0 and 1. For instance, consider this grayscale cross-drawing function (which is already in your MATLAB path). The user can manipulate the size, the relative point of crossing, and the foreground and background luminance.

```
function IMG = drawcross(params)

IMG = ones((params.size*2)+1,(params.size*2)+1) * params.bg_lum;

horl = 1 + round(params.ver_cross * (params.size*2) );
verl = 1 + round(params.hor_cross * (params.size*2) );

IMG(horl,:) = params.fg_lum;
IMG(:,verl) = params.fg_lum;
```

We now fill a display consisting of random crosses, where the foreground luminance of these crosses changes from left to right. One random target has opposite luminance. We did not need to change anything to the standard GERT calls other than providing a function handle to our custom drawing function, and passing an element parameter structure that was correctly formatted for this function.

```
ccc; GERT_Init;

peb_params.min_dist = 25;
peb_params.dims = [1 500 1 500];
E = GERT_PlaceElements_Background([],[],peb_params);

el_params.size = 5;
el_params.hor_cross = rand(1,E.n);
el_params.ver_cross = rand(1,E.n);
el_params.bg_lum = 0.5;
el_params.fg_lum = E.x/500;
target = ceil(rand*E.n);
el_params.fg_lum(target) = 1 - el_params.fg_lum(target);

IMG = GERT_RenderDisplay(@drawcross,E,el_params);
figure; imshow(IMG);
```

4.6 Dynamic stimuli

To create a dynamic stimulus, it suffices to create a series of normal images and stack them into a 3-D (or 4-D, for color) matrix. The flexibility of GERT_RenderDisplay makes it easy to systematically manipulate element properties over time, as we shall demonstrate in the following example. A regular grid of Gabor elements is created, a square section of which will be regarded as the ‘foreground’. These elements will shift in phase twice as fast as the ‘background’ elements, while rotating twice as slow in the opposite direction. The result is viewed through MATLAB’s default movie player (press the green play button, and turn on the ’repeat’ option).

```
ccc; GERT_Init;

% Define the element positions
[xgr,ygr] = meshgrid(linspace(9,169,16),linspace(9,169,16));
E = GElements([xgr(:)'; ygr(:)'],[1 180 1 180]);

% Indicate the square
idx = sub2ind([16 16],[6:11 6:11 6:11 6:11 6:11 6:11], ...
  [ones(1,6)*6 ones(1,6)*7 ones(1,6)*8 ones(1,6)*9 ones(1,6)*10 ones(1,6)*11]);

% Scale the display up
el_params.scale = 2;
el_params.size = 15;
img_params.dims = [500 500];

% Make the movie
init_ph = zeros(1,E.n);
init_or = rand(1,E.n)*2*pi;
for frame = 1:25
    el_params.phase = init_ph + ((frame-1)*2*pi/25);
    el_params.phase(idx) = init_ph(idx) + 2*((frame-1)*2*pi/25);
    el_params.or = init_or + 2*((frame-1)*2*pi/25);
    el_params.or(idx) = init_or(idx) - ((frame-1)*2*pi/25);
    IMG(:,:,frame) = GERT_RenderDisplay(@GERT_DrawElement_Gabor,E,...
                                        el_params,img_params);
end

% Play the movie
implay(IMG);
```

4.7 Scaling, rotating, shifting and flipping

As we have seen in some of the examples above, it is possible to transform GElements and GContour objects through the scaling, rotating, shifting and flipping functions. Here we will go into a little more detail.

GERT_Transform_Center brings the center of a GElements or GContour object towards a specified coordinate, defined as the first argument, `to_vals`. If this argument is left empty, (0,0) is used. Importantly though, the user should specify how the center of the figure is to be computed, through the `center_to` parameter. By default, the `'BoundingBox'` of the coordinate vectors will be used. Other options are the `'Mean'`, the `'Median'` or the `'GeoMean'` of these vectors. For closed figures, the `'Centroid'` (center of mass) can be also be computed. This generally gives the best results. To specify a custom coordinate, use the `'Custom'` option, and specify the correct value in the third argument, `custom_vals`.

The other transform functions make use of the centering function, since most of them need to define a central, fixed point. GERT_Transform_Rotate rotates the object by a given amount `rot_ang`, around a fixed point `rot_point`. This latter argument has the same options as `center_to` in the GERT_Transform_Center function, including the specification of `custom_vals` in the final argument of the function. The GERT_Transform_Scale function works analogously, scaling up or down a GElements or GContour object by a factor `scale_factor`, as does GERT_Transform_Flip. For the latter function, a `flip_axis` and a central `flip_point` have to be specified.

GERT_Transform_Shift is not as complicated. It allows the translation of an object by a distance `d`, which should be a 1×2 vector (X and Y shift).

4.8 Element tagging

To know which elements belonged to a contour and which belonged to the background, we have until now mostly relied on the knowledge that GERT will place its background elements after previously placed points in the GElements coordinate vectors. However, a more convenient system is in place. You have probably noticed that whenever the `plot` method was called on a GElements object, GERT already knew which elements to draw in a different color. This is because they have been *tagged* within the GElements object when GERT first placed them.

We can access this information manually if we want. For this we use the `gettag` method, which belongs to the GElements class. Tags are specified as a combination of a single letter from a-z, and an integer number of any length. These are passed together as a character array. When GERT functions automatically tag elements, `'c'` is used for a contour, and `'b'` for background. `'c1'` would then be 'contour 1' and `'c2'` would be 'contour 2'. `'b1'` are the first set of background elements, `'b2'` the second set, and so forth. When calling the `gettag` method with such a character string, the indices of the targeted elements are returned. Note that it is not necessary to specify the number in the tag, `gettag('c')` will simply return all elements tagged as a contour.

To set a tag, the `settag` function is used, where both the tag and the indices into the GElements coordinate vectors are provided. Again it is not strictly necessary to provide the number, if none is specified GERT will automatically use the next available number. The example below illustrates the workings of the tagging system. We also illustrate a more advanced use of the `plot` method, where we pass the indices as an argument to plot only a portion of the elements object. Note how the `plot` method recognizes the `'f'` tag, as a foreground element. All element tags other than `'c'`, `'b'` or `'f'` will be plotted in black.

```
ccc;
GERT_Init;
```

```
% Generate two contours
gce_params.hax = 10;
gce_params.vax = 5;
C1 = GERT_GenerateContour_Ellipse(gce_params);
C2 = GERT_Transform_Rotate(C1,pi/2);
C2 = GERT_Transform_Shift(C2,[0 30]);

% Place elements on the contours
pec_params.cont_avgdist = 2;
E1 = GERT_PlaceElements_Contour(C1,pec_params);
E2 = GERT_PlaceElements_Contour(C2,pec_params);

% Place elements around them
peb_params.min_dist = 4;
peb_params.dims = [-50 50 -50 50];
peb_params.in_region = false;
Ea = GERT_PlaceElements_Background({E1 E2},{C1 C2},peb_params);

% Place elements inside them at a higher density, with a different tag
peb_params.min_dist = 2;
peb_params.in_region = true;
Ea = GERT_PlaceElements_Background(Ea,{C1 C2},peb_params);

% Plot the whole display
plot(Ea);

% Plot the first contour only
idx = gettag(Ea,'c1');
plot(Ea,idx);

% Plot both contour, and the elements inside them
idx = [gettag(Ea,'c') gettag(Ea,'b2')];
plot(Ea,idx);

% Set the inside elements to tag 'f' and plot them
Ea = settag(Ea,'f', gettag(Ea,'b2'));
idx = gettag(Ea,'f');
plot(Ea,idx);
```

4.9 Merging elements

GERT allows the merging of several GElements objects, with automatic updating of their status tags. E.g, if there are two objects with a `c1` tag, one will become a `c2` tag. To perform the merging, one should simply call GERT_MergeElements and pass a row vector of GElements objects. A single, merged GElements object will be returned. Whenever different `dims` specifications are present, the `dims` value of the first object in the vector will be retained. Following up on the code in 4.8, one could execute:

```
% As can be seen, both element objects are entirely tagegd as 'c1'
gettag(E1,'c1')
gettag(E2,'c1')
```

```
% Now we merge them
E_merged = GERT_MergeElements({E1 E2});

% They have been updated to 'c1' and 'c2'
gettag(E_merged,'c1')
gettag(E_merged,'c2')
```

4.10 Distinguishing elements inside, outside and on a contour

It is often useful to know whether a given element is situated inside or outside a closed GContour, or a set of GElements placed on this contour. For this we use the GERT_Aux_InContour function. As the first argument a GElements object needs to be specified, namely the set of elements for which the position should be checked. As the second argument `polydef` either a GContour or another GElements can be provided.

If a GContour is passed, the function will determine whether each of the elements of the first argument is situated inside or outside the `polydef`, and return these respective indices as `in_idx` and `out_idx`. Elements exactly on the contour will be regarded as inside the contour. If however another GElements object is passed as the `polydef`, this distinction will be made: Elements that coincide with one another will be returned as `on_idx`. Naturally, the `polydef` elements should be ordered along a continuous, closed contour.

The indices obtained can then for instance be used to set certain rendering properties for the elements. We reiterate an example given in 3.5 to illustrate this:

```
ccc; GERT_Init;

% (1) Read in and position the eagle contour:
C = GERT_GenerateContour_FileTXT('eagle.txt',true,' ');
C = GERT_Transform_Flip(C,0,'Centroid');
C = GERT_Transform_Center(C, [0 0], 'Centroid');

% (2) Place contour elements:
pec_params.method = 'SerialEquidistant';
pec_params.cont_avgdist = 25;
pec_params.eucl_mindist = 20;
[E ors] = GERT_PlaceElements_Contour(C,pec_params);
E.dims = [-250 250 -250 250];

% (3) Place background elements:
peb_params.min_dist = 20;
Ea = GERT_PlaceElements_Background(E,[],peb_params);

% (4) Retrieve which elements inside, outside and on the contour
[in_idx, out_idx, on_idx] = GERT_Aux_InContour(Ea, E);
```

```
% (5) Define the triangle patches:
el_params.width = 10;
el_params.height = 17;
el_params.scale = 1;
el_params.size = 20;
el_params.aa = 8;
el_params.or(in_idx) = repmat(main_axis(C),[1 numel(in_idx)]);
el_params.or(on_idx) = ors;
el_params.or(out_idx) = 2*pi*rand([1 numel(out_idx)]);
el_params.lum_bounds(in_idx) = {[0.4 0; 0.25 0; 0.15 0.6]};
el_params.lum_bounds(on_idx) = {[0.4 .95; 0.25 0; 0.15 0]};
el_params.lum_bounds(out_idx) = {[0.4 .95; 0.25 .95; 0.15 .95]};

% (6) Define the image parameters:
img_params.bg_lum = [0.4 0.25 0.15];
img_params.dims = [1000 1000];

% (7) Render:
IMG = GERT_RenderDisplay(@GERT_DrawElement_Triangle,Ea,el_params,img_params);
figure; imshow(IMG);
```

4.11 Anti-aliasing

The final stimulus image will usually be an undersampled version of the theoretically present stimulus. In some cases, such as Gabor elements, this is not a very severe problem, as its luminances are naturally graded and can be computed exactly for each pixel position. However in the case of for instance a polygon, straight lines can become distorted by the image pixelation, giving it a jagged appearance. This is known as spatial aliasing. The basic solution to this problem is to add in-between luminances to the image to smooth the edges perceptually. Figure 4.2 illustrates this.

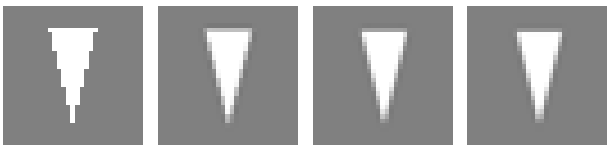

Figure 4.2 – Four levels of anti-aliasing, at a 3x zoom level. The left-most triangle has not been anti-aliased and look jagged. By increasing the antialiasing level to 2x, 4x or 8x, in this figure displayed from left to right, a smoother appearance is attained.

GERT has implemented anti-aliasing for some of its element drawing functions (triangle, polygon, ellipse, ...). It is typically set through the `aa` parameter field, which has a default value of 4. What this means, is that the element will first be created at four times its intended size, aliased as in the leftmost panel of Figure 4.2. We then downscale the image through MATLAB's `imresize` function, using a Lanczos filter to compute the in-between luminances of the smaller image from the larger image. This results in perceptually smooth elements. By setting the anti-aliasing parameter to 0, anti-aliasing is disabled. Rendering will then be faster, but uglier. Should you require a different level of anti-aliasing, qualities 2 and 8 are also available. You may play around with the `aa` parameter to balance the quality/speed ratio of rendering until it meets your expectations.

4.12 Using SVG files to draw elements

One every flexible way to create specific element shapes, is through the SVG vector graphics format. As discussed in 3.1.4, GERT is capable of reading these files and converting them to GContour objects. The missing link to element creation is called GERT_DrawElement_GContour, which enables the definition of element shapes through these objects. Its workings and parameters are quite similar to GERT_DrawElement_Polygon, with a few notable exceptions. First, there is no need to center or scale the SVG contour to the image patch, GERT will do this automatically. Second, when a vector of GContour objects is provided, multiple polygons will be filled at once, such that one polygon inside another will be considered a hole, drawn in the background color. This is quite similar to how Inkscape itself converts a graphical object (e.g., a letter typed in a certain font) to the paths that GERT can read from SVG files.

Below, we illustrate how a letter 'T' can be composed out of little 'T' elements, using the same SVG file and the same resulting GContour object. Contour elements are upright and uniformly yellow, background elements have a random orientation and color. All elements are nicely anti-aliased at the default 4×quality. Open the file `T.svg` (in the 'resources' subfolder) in Inkscape and distort its outline to see the simultaneous effect on the contour and its elements!

```
ccc; GERT_Init;

% Read svg, center, and scale
C = GERT_GenerateContour_FileSVG('T.svg',10);
C = GERT_Transform_Center(C,[0 0],'Centroid');
C = GERT_Transform_Scale(C,3,'Custom',[0 0]);

% Place contour elements on the contour
pecp.cont_avgdist = 50;
pecp.method = 'SerialEquidistant';
pecp.eucl_mindist = 45;
[E ors] = GERT_PlaceElements_Contour(C,pecp);

% And background elemements around it
pebp.min_dist = 50;
pebp.dims = [-500 500 -500 500];
Ea = GERT_PlaceElements_Background(E,[],pebp);
[in out on] = GERT_Aux_InContour(Ea,E);

% Then use the same contour to create the element shapes
elp.contour = {C};
elp.or(1:Ea.n) = rand(1,Ea.n)*pi*2;
elp.or(on) = 0;
elp.size = 10;
for i = 1:Ea.n
```

```
    elp.lum_bounds(i) =  {[0 rand; 0 rand; 0 rand]};
end
elp.lum_bounds(on) =  {[0 1; 0 1; 0 0]};

img_params.bg_lum = [0 0 0];
img_params.dims = [500 500];
img_params.global_rendering = true;
img_params.blend_mode = 'MaxDiff';

IMG = GERT_RenderDisplay(@GERT_DrawElement_GContour,Ea,elp,img_params);
figure; imshow(IMG);
```

4.13 More?

We have in this Tips & Tricks section listed some of the more common specific questions that people have come to us with. But, we welcome any useful additions to this chapter. Just surf to our website, and contact us with your ideas and insights!

PART V

Full Documentation

5.1 GContour

DESCRIPTION:
This class holds information on individual element X and Y positions, the open or CLOSED status of the contour, and the number of contour definition points N. In addition, the distances along the contour CDIST can be present, as well as the local tangents LT and the total contour length CLENGTH.

The methods implemented include a VALIDATE and PLOT function, and generic contour specific computing functions COMPUTE_CDIST and COMPUTE_LT. Often the contour generation function might already have filled in the CDIST and LT values, however, using its own methods. For closed contours, MAIN_AXIS and CENTROID allow the retrieval of the properties to which the function names refer. Finally, RMID allows the removal of identical points from the contour description.

CONSTRUCTORS:

```
Empty:
                                        No requirements

Using a subset of an existing GContour object:
contour                                 required
                                        1x1 GContour

idx                                     required
                                        1xN double, positive integer values

Using an existing GElements object:
els                                     required
                                        1x1 GElements

closed                                  optional, default:  false
                                        1x1 logical

Using existing coordinate vectors:
coordinates                             required
                                        2xN double
                                        finite, real

closed                                  optional, default, false
                                        1x1 logical
```

PROPERTIES:

x	1xN double
y	1xN double
n	1x1 double
closed	1x1 logical
cdist	empty, 1xN or 1xN+1 double
clength	empty or 1x1 double
lt	empty or 1xN double

METHODS:

tf = validate	Validate the object
plot	Plot the contents of the object
compute_cdist	Compute the distances along the contour
compute_lt	Compute the local tangents to the conto ur points
x0,y0 = centroid	Retrieve the centroid of a closed conto ur
ma,pt = main_axis	Retrieve the main axis of a closed cont our, as well as the centroid
rmid	Remove identical points from the contou r description
h = hash	Create a unique identifier out of this object

DETAILS:
None.

EXAMPLE:
None.

5.2 GElements

DESCRIPTION:
This class holds information on individual element X and Y positions, the dimensions DIMS of the display inside which these elements are placed, and their status TAGS. E.g., 'c1' for the first contour. In addition it implements methods to VALIDATE the object, PLOT its contents, and request or change the tags through SETTAG and GETTAG.

CONSTRUCTORS:

Empty:
No requirements

Using a subset of an existing GElements object:

els	required 1x1 GElements

idx	required 1xN double >0, integer value, finite, real

Using an existing GContour object:

contour	required 1x1 GContour
dims	optional, default: [] 1x4 double dims(2)>dims(1) dims(4)>dims(3), finite, real

Using existing coordinate vectors:

coordinates	required 2xN double finite, real
dims	optional, default, [] 1x4 double finite, real

PROPERTIES:

x	1xN double
y	1xN double
n	1x1 double
dims	1x4 double, dims(1)<dims(2),dims(3)<dims(4)
tags	1x2 cell (set access through settag)

METHODS:

validate	Validate the object
plot	Plot the contents of the object
settag(tag,idx)	Set this 1xN char tag for a 1xN vector of indices The first char should be a lowercase letter, the next chars should form an integer value
idx = gettag(tag)	Fetch the indices corresponding to this 1xN char tag
rmid	Remove identical points

DETAILS:

None.

EXAMPLE:

None.

5.3 GLog

DESCRIPTION:

This class contains all logging information. It contains an INFO struct which summarizes basic information on the computer and Matlab installation used. The FUNCTIONS property struct retains all the messages and variables saved, grouped per function and per call to that function. The FILES property struct saves entire plain-text files, (e.g., .m or .svg). The START method enables logging, and will cause many GERT functions to automatically log their input and output variables. STOP disables logging, and clears the contents of the GLog class. EXIST checks whether logging is enabled or disabled. ADD allows the adding of either a variable, a message or a file to the log. FETCH retrieves a copy of the log as a struct, which can be saved to a .mat file. Finally, GROUP allows the user to group several entries under a single function call under the 'Functions' struct property.

CONSTRUCTORS:

Empty:	No requirements

PROPERTIES:

Info	1x1 struct
Functions	1x1 struct
Files	1x1 struct

METHODS:

start	Enable logging
stop	Disable logging, clear the log
add(type,val,name)	Add an entry type 'msg': val is a 1xN char containi ng the message name is ignored type 'var': val is the variable to be stored name is a 1xN char containing the name u nder which to store it type 'file': val is the 1xN char path t o the file name is a 1xN char containing the name u nder which to store it
group(s)	Group the following entries s == 'on': Group the following items wi th the previous one s == 'off': Treat the following items a s separate function calls
l = fetch	Fetch the contents of the log as a stru ct

DETAILS:

None.

EXAMPLE:

None.

5.4 GStructParser

DESCRIPTION:
This class allows the parsing of parameter structures in GERT. It is similar to the inputParser class in Matlab 2007 and later, and is mostly used internally.

CONSTRUCTORS:

None.

PROPERTIES:

fields	1x3 cell
results	1x1 struct
n	1x1 double (dependent on x)

METHODS:

addfield	Add a field to parse
parse	Parse the listed fields

DETAILS:
None.

EXAMPLE:
None.

5.5 ccc

DESCRIPTION:
Clear Close Clear. This removes all variables, both local and global, closes all figures, and clears the command window. The credit for this function goes entirely to Bart Machilsen, who originally conceived and implemented the CCC concept. We also acknowledge the useful feedback of Tom Putzeys.

ARGUMENTS:

None.

RETURNS:

None.

5.6 GERT_Aux_Centroid

DESCRIPTION:
This function computes the centroid of a series of x and y coordinates. They should described a closed contour.

ARGUMENTS:

x	required 1xN double N≥ 0, finite, real
y	required 1xN double N≥ 0, finite, real

RETURNS:

x0	1x1 double
y0	1x1 double

DETAILS:
None.

EXAMPLE:
None.

5.7 GERT_Aux_DrawArc

DESCRIPTION:
This function will output an ellipsoid arc between two points in Cartesian coordinates, at a given resolution. The POINTS matrix further needs to specify the ellipse radii, its rotation, and two flags to determine which of 4 possible solutions needs to be drawn. Multiple arcs can be drawn at once using the third dimension.

ARGUMENTS:

points	required 2x5xN double Finite, real; column 2 >0; column 4 [0 1]
res	required 1x1 double >0, integer value, finite, real

RETURNS:

cart	2xM double, where M = res * N (minus identical points)

DETAILS:
The POINTS input argument contains the same information as described in the specifications of the SVG graphic file format, see:

http://www.w3.org/TR/SVG/paths.html#PathDataEllipticalArcCommands

The first column contains the X and Y coordinates of the start point. The second column contains the X and Y radii of the ellipse The third column contains its rotation (second value is ignored) The fourth column contains the large arc and sweep flags The fifth column contains the X and Y coordinates of the end point

EXAMPLE:
```
points = [5,20,0,1,20; 10,10,0,1,0];
cart = GERT_Aux_DrawArc(points, 100);
```

5.8 GERT_Aux_DrawBezier

DESCRIPTION:
This function will output a cubic Bezier curve in Cartesian coordinates, at a given resolution. The POINTS matrix further specifies the coordinates of both B°ier control points. Multiple curves can be drawn at once using the third dimension.

ARGUMENTS:

points	required 2x4xN double Finite, real
res	required 1x1 double >0, integer value, finite, real

RETURNS:

cart	2xM double, where M = res * N (minus identical points)

DETAILS:
The POINTS input argument is logically structured to contain the start point, the first control point, the second control point, and the end point in its columns, respectively.

EXAMPLE:

```
points = [5,10,15,20; 0,10,10,0];
cart = GERT_Aux_DrawBezier(points, 100);
```

5.9 GERT_Aux_EuclDist

DESCRIPTION:
This function will compute all the pairwise Euclidean distances between the points in row vectors (x1,y1) and (x2,y2).

ARGUMENTS:

x1	required 1xN double Finite, real
y1	required 1xN double Finite, real
x2	required 1xM double Finite, real
y2	required 1xM double Finite, real

RETURNS:

dmat	NxM double

5.10 GERT_Aux_InContour

DESCRIPTION:
This function will determine which ELEMENTS are situated inside a closed GContour POLYDEF. Their indices will be returned as IN_IDX. Note that this will include elements lying exactly on the contour. Elements lying outside the contour can be retrieved as OUT_IDX. If the POLYDEF is defined through a GElements object, a further distinction is made, where elements identical to the POLYDEF members are returned as ON_IDX.

ARGUMENTS:

elements	required 1x1 GElements elements.n>0
polydef	required 1x1 GContour or GElements polydef.n>0, closed

RETURNS:

in_idx	1xL double
out_idx	1xM double
on_idx	1xN double

DETAILS:
None.

EXAMPLE:
None.

5.11 GERT_Aux_MainAxis

DESCRIPTION:
This function computes the main axis of a series of x and y coordinates. They should described a closed contour. It also returns the centroid of these coordinates.

ARGUMENTS:

x	required 1xN double N≥ 0, finite, real
y	required 1xN double N≥ 0, finite, real

RETURNS:

main_axis	1x1 double
pt	1x2 double

DETAILS:
None.

EXAMPLE:
None.

5.12 GERT_Aux_Randi

DESCRIPTION:

This function will return random integers between 1 and MAXI, in a matrix with dimensions DIMS. It was copied from PsychToolbox and given a custom name to avoid interferences with other functions named 'randi'.

ARGUMENTS:

maxi	required 1x1 double >0, integer value, finite, real
dims	optional 1xN double >0, integer value, finite, real

RETURNS:

ri	DIMS double

DETAILS:
None.

EXAMPLE:
None.

5.13 GERT_Aux_RemoveIdenticalPoints

DESCRIPTION:
This function will remove identical points from Cartesian pairs of coordinates, returning both the values and the indices.

ARGUMENTS:

x	required 1xN double N≥ 0, finite, real
y	required 1xN double N≥ 0, finite, real

RETURNS:

xv	1xM double
yv	1xM double
idx	1xM double

DETAILS:
None.

EXAMPLE:
None.

5.14 GERT_Aux_RestrictAngle

DESCRIPTION:
This function will restrict the values in VALS to lie between MIN and MAX, assuming that multiples of MAX minus MIN are equivalent to one another.

ARGUMENTS:

vals	required double finite, real
minv	required 1x1 double finite, real
maxv	required 1x1 double >min, finite, real

RETURNS:

vals	double

DETAILS:
None.

EXAMPLE:
None.

5.15 GERT_Aux_RestrictResolution

DESCRIPTION:
This function will discretize the values in VALS, to a given PRECISION.

ARGUMENTS:

vals	required double finite, real

precision	required 1x1 double >0, finite, real

RETURNS:

vals	double

DETAILS:
None.

EXAMPLE:
None.

5.16 GERT_Aux_UniqueID

DESCRIPTION:
This function takes a double or char matrix, and generates a hash code which can serve as a unique ID of the contents of this matrix.

ARGUMENTS:

cvar	required double or char string finite, real

RETURNS:

id	1xN double

DETAILS:
None.

EXAMPLE:
None.

5.17 GERT_Aux_ValidVec

DESCRIPTION:
This function will check whether VEC is a valid row vector of a given TYPE and length VAL. No error checking, for speed.

ARGUMENTS:

None.

RETURNS:

None.

DETAILS:
None.

EXAMPLE:
None.

5.18 GERT_CheckCue_LocalDensity

DESCRIPTION:
This function checks for an average local density cue in the display, and returns the result of a statistical significance test. The default method is to compute surface areas of the cells in a Voronoi diagram, and perform a Monte Carlo permutation test to determine whether these surface areas differ significantly between contour element and background elements. However other methods are also available, both for measuring local density and for performing the statistical test. If a second output argument is provided, a density variability cue will be computed as well.

ARGUMENTS:

elements	required 1x1 GElements
idx1	required 1xM double M>1, >0, integer value, finite, real
idx2	required 1xM double M>1, >0, integer value, finite, real
params	required 1x1 struct
method_dens	optional, default: 'Voronoi' 1xM char array 'AvgDist', 'RadCount', or 'Voronoi'
method_stat	optional, default: 'MC' 1xM char array 'MC' or 'T'
var_steps	optional, default: 25 1x1 double >0, integer value, finite, real

METHOD_DENS: 'Voronoi'

```
      border_dist                      required
                                       1x1 double
                                       ≥ 0, finite, real

   METHOD_DENS: 'AvgDist'
   (in addition to all 'Voronoi' parameters)

      avg_n                            optional, default:  3
                                       1x1 double
                                       ≥ 0, integer value, finite, real

   METHOD_DENS: 'RadCount'
   (in addition to all 'Voronoi' parameters)

      rad                              required
                                       1xM double
                                       >0, finite, real

   METHOD_STAT: 'T'

      alpha                            optional, default:  0.1
                                       1x1 double
                                       0 ≤ alpha ≤ 1, finite, real

   METHOD_STAT: 'MC'

      alpha                            optional, default:  0.35
                                       1x1 double
                                       0 ≤ alpha ≤ 1, finite, real

      mc_samples_n                     optional, default:  1000
                                       1x1 double
                                       >100, integer value, finite, real
```

RETURNS:

```
res                                    1x1 struct

        pm                             1x1 double

        hm                             1x1 double

        c1                             1xN double

        c2                             1xM double

resv                                   1x1 struct

        pm                             1x1 double

        hm                             1x1 double

        c1                             1xN double

        c2                             1xM double
```

DETAILS:
The exact workings of this function depend on METHOD_DENS and METHOD_STAT. 'Voronoi' is the default method to determine local density. The 'Voronoi' method decomposes the display into polygon cells, each enclosing a portion

of space that is closer to a particular element in the display than to any other element. The surface of these cells is then compared between elements belonging to the contour and elements belonging to the background. The indices of contour and background elements need to be explicitly defined as two separate row vectors IDX1 and IDX2. In the 'AvgDist' option, GERT computes the average distance of each element to its AVG_N nearest neighbours. The observed difference in mean is then compared between the series of contour and background elements. If avg_n is set to 0, a Delaunay triangulation will be performed to determine the number of natural neighbours for each element. The 'RadCount' method counts the number of elements within a certain radius RAD of each point included in the element list. When a vector is provided, the total difference between two curves is computed. For all 3 density measures, a BORDER_DIST needs to be defined, indicating how far an element should be away from the edge of the display to be included in the list of contour or background elements. The nearest neighbours involved in the computation do not get selected based on the distance from the border. 'MC' is the default method to test whether the observed difference in means is significant or not. A Monte Carlo permutation test is used: the labels (idx1 or idx2) are randomly shuffled, MC_SAMPLES_N times, to generate a random distribution of differences in mean between both series of elements. Within this distribution the actually observed difference can be located. The resulting proportion p of random sample differences that are smaller than the observed difference can be compared to a value ALPHA. 'T' is a faster alternate method, using a T test with Satterthwaite's compensation for unequal variances. However a normal distribution is assumed here, whereas the Monte Carlo test is distribution free. In the output structure RES, PM contains the proportion of Monte Carlo samples smaller than the observed difference, or the result of a one-sided T-test. HM is the binary evaluation of this value against the criterion level ALPHA. C1 and C2 contain the full distributions of all retained density measurements on which this test was performed. The output structure RESV contains analogous results for a comparison of a more complete density distribution, including differences in variance (see manual).

EXAMPLE:

```
idx1 = 1:fg_n;
idx2 = fg_n+1:elements.n;
cld_params.avg_n = 4;
cld_params.border_dist = 10;
cld_params.method_dens = 'AvgDist';
res = GERT_Check_LocalDensity(elements,idx1,idx2,cld_params);
```

5.19 GERT_Demo

DESCRIPTION:
Generates nine demo figures and benchmarks the speed.

ARGUMENTS:

None.

RETURNS:

None.

DETAILS:
None.

EXAMPLE:
None.

5.20 GERT_Dependencies

DESCRIPTION:
This function will quickly check whether all necessary components are present in your Matlab installation.

ARGUMENTS:

None.

RETURNS:

None.

DETAILS:
None.

EXAMPLE:
None.

5.21 GERT_DrawElement_Ellipse

DESCRIPTION:
This function will draw a single ellipse patch, according to the parameters defined. Important parameters are the WIDTH and the HEIGHT defining the ellipse its OR, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
size	optional, default: 10 1x1 double >0, integer value, finite, real
width	optional, default: 5 1x1 double >0, < size*2, finite, real
height	optional, default: width 1x1 double >0, < size*2, finite, real
or	optional, default: 0 1x1 double finite, real
scale	optional, default: 1 1x1 double >0, finite, real
aa	optional, default: 4 1x1 double 0,2,4,8, finite, real
lum_bounds	optional, default: [0.5 1] 1x1 cell, containing 1x2 or 3x2 double ≥ 0 ≤ 1, finite, real

RETURNS:

IMG	MxM double, where M=(size*2)+1

DETAILS:
Anti aliasing is by default enabled, to disable it change AA to 0. Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:
```
circle_params.width = 6;
circle_params.scale = 3;
circle_params.size = 20;
circle_params.lum_bounds = {[1 0; 0 0; 0 1]};

IMG = GERT_DrawElement_Ellipse(circle_params);
```

5.22 GERT_DrawElement_Gabor

DESCRIPTION:
This function will draw a single Gabor stimulus onto a rectangular image patch, according to the parameters defined. The parameters typically manipulated are the OR of the grating, the SIGMA of the Gaussian components, the FREQ of the Sinusoidal component, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
or	optional, default: 0 1x1 double finite, real
sigma	optional, default: 2.5 1x1 double >0, finite, real
freq	optional, default = 0.1 1x1 double >0, finite, real
size	optional, default: 10 1x1 double >0, integer value, finite, real
phase	optional, default: 0 1x1 double finite, real
amp	optional, default: 1 1x1 double ≥ 0, finite, real
scale	optional, default: 1 1x1 double >0, finite, real
lum_bounds	optional, default: [0 0.5 1] 1x1 cell, containing 1x3 or 3x3 double ≥ 0 ≤ 1finite, real
lum_scale	optional, default: 'None' 1x1 cell, containing char vector ≥ 0 ≤ 1, finite, real

RETURNS:

IMG	MxM double, where M=(size*2)+1

DETAILS:
Setting LUM_BOUNDS allows the user to change the maximal, minimal and background luminance values independently. For instance, [0 0.2 1] will create a

high-contrast Gabor against a dark gray background. Note that assymmetrical lum_bounds will create non-Gabor luminance profiles. To return a color image, pass a 3x3 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable. LUM_SCALE allows further luminance rescaling operations, and must be a 1x1 cell containing a char array. The default 'None' mode disables rescaling. 'SymmMax' will attempt to create a maximal contrast through symmetrical rescaling around the background value. For instance, this comes in handy when using a phase of pi/2, since in that case often no pixel will have a 0 or 1 luminance even though the amplitude is maximal (due to the Gaussian convolution). Applying 'SymmMax' will then return a Gabor of maximal contrast. 'AsymmMax' is similar, but allows the rescaling to be assymmetrical. A possible application is when the background luminance has to be dark (e.g., during eye tracking the pupils have to remain large), yet high contrast is needed. 'AsymmMax' will then maximize the contrast such that the positive part of the Gabor is very clear, whereas the negative part remains subtle.

EXAMPLE:

Create a large, high contrast odd Gabor against a dark gray background

```
gabel_params.or = 0;
gabel_params.sigma = 2.5;
gabel_params.size = 200;
gabel_params.freq = 0.1071;
gabel_params.phase = pi/2;
gabel_params.scale = 20;
gabel_params.lum_bounds = {[0 0.2 1]};
gabel_params.lum_scale = {'SymmMax'};

IMG = GERT_DrawElement_Gabor(gabel_params);
```

5.23 GERT_DrawElement_Gaussian

DESCRIPTION:

This function will draw a single Gaussian stimulus onto a rectangular image patch, according to the parameters defined. Important parameters are the OR of the grating, the SIGMA of the Gaussian components, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
or	optional, default: 0 1x1 double finite, real
sigmax	optional, default: 2 1x1 double >0, finite, real

```
sigmay                            optional, default:  2
                                  1x1 double
                                  >0, finite, real

size                              optional, default:  10
                                  1x1 double
                                  >0, integer value, finite, real

amp                               optional, default:  1
                                  1x1 double
                                  ≥ 0, finite, real

scale                             optional, default:  1
                                  1x1 double
                                  >0, finite, real

lum_bounds                        optional, default:  [0.5 1]
                                  1x1 cell, containing 1x2 or 3x2 double
                                  ≥ 0 ≤ 1, finite, real
```

RETURNS:

```
IMG                               MxM double, where M=(size*2)+1
```

DETAILS:
Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:
Create an elongated blue Gaussian against a red background

```
gausel_params.or = 0;
gausel_params.sigmax = 2.5;
gausel_params.sigmay = 5.5;
gausel_params.size = 50;
gausel_params.scale = 5;
gausel_params.lum_bounds = {[1 0; 0 0; 0 1]};

IMG = GERT_DrawElement_Gaussian(gausel_params);
```

5.24 GERT_DrawElement_GContour

DESCRIPTION:
This function will draw an element based on one or several GContour objects, specified as CONTOUR, packed in a cell scalar. The contour will be scaled automatically so that its bounding box is at a distance BORDER_DIST (default: 0.1 times the SIZE parameter, defined as the total width divided by two, minus one) from the image patch border. Multiple polygons are drawn such that an overlap with the previously combined polygons becomes a hole, unless the UNION parameter is set to true.

ARGUMENTS:

```
params                                  required
                                        1x1 struct

        contour                         required
                                        1x1 cell
                                        Closed contours

        border_dist                     optional, default:  0.1 * size
                                        1x1 double
                                        >0, finite, real

        union                           optional, default:  false
                                        1x1 logical
                                        true or false

        center                          optional, default:  middle of bounding b
                                        ox
                                        1x2 double
                                        finite, real

        or                              optional, default:  0
                                        1x1 double
                                        finite, real

        size                            optional, default:  10
                                        1x1 double
                                        >0, integer value, finite, real

        scale                           optional, default:  1
                                        1x1 double
                                        >0, finite, real

        aa                              optional, default:  4
                                        1x1 double
                                        0,2,4,8, finite, real

        lum_bounds                      optional, default:  [0.5 1]
                                        1x1 cell, containing 1x2 or 3x2 double
                                        ≥ 0 ≤ 1, finite, real
```

RETURNS:

```
IMG                                     MxM double, where M=(size*2)+1
```

DETAILS:

The GContour will automatically be centered such that the middle of its bounding box becomes the middle of the element. To adjust this, set the optional CENTER parameter, expressed in the units of the original GContour definition. Rotation through the OR parameter will also occur around this middle point. Anti-aliasing is by default enabled, to disable it change AA to 0. Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:
(none)

5.25 GERT_DrawElement_Image

DESCRIPTION:
This function will load an image stored in file FNAME using "imread", either as a COLOR or a grayscale matrix. No other parameters can be provided at the moment. The image must be square, and of an odd size.

ARGUMENTS:

params	required 1x1 struct
fname	required 1x1 cell containing 1xN char
scale	optional, default: 1 1x1 double >0
color	optional, default: false 1x1 logical, finite, real

RETURNS:

IMG	MxM double

DETAILS:
The current implementation is a simple paste-job of an existing image file. Rotation or scaling could in principle be implemented easily. However, in many cases the image quality will suffer from the pixel interpolation algoritm that will have to be applied. For instance, what can be displayed perfectly as an straight line when upright, cannot be rendered perfectly straight when at an angle of 45°degrees, due to the pixelation of the image. Therefore rotated images can differ on more properties than just rotation, making them unsuitable for scientific research. When we have figured out a way around this kind of quality loss, we will implement it. Suggestions are of course also welcome.

EXAMPLE:
```
params.fname = {'E.png'};
params.color = false;
IMG = GERT_DrawElement_Image(params);
```

5.26 GERT_DrawElement_Polygon

DESCRIPTION:
This function will draw a single ploygon patch, according to the parameters defined. Important parameters are the POINTS defining the polygon, the OR of the polygon, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
points	required 2xN double N>2 abs < size
or	optional, default: 0 1x1 double finite, real
size	optional, default: 10 1x1 double >0, integer value, finite, real
scale	optional, default: 1 1x1 double >0, finite, real
aa	optional, default: 4 1x1 double 0,2,4,8, finite, real
lum_bounds	optional, default: [0.5 1] 1x1 cell, containing 1x2 or 3x2 double ≥ 0 ≤ 1, finite, real

RETURNS:

IMG	MxM double, where M=(size*2)+1

DETAILS:
The POINTS definition of the polygon must be centered around 0; the function will then automatically center it around SIZE+1. Anti-aliasing is by default enabled, to disable it change AA to 0. Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:
```
poly_params.points = [1 2 1 -1 -2 -1; 2 0 -2 -2 0 2];
poly_params.or = 0;
poly_params.scale = 5;
poly_params.size = 20;
```

```
poly_params.lum_bounds = {[1 0; 0 0; 0 1]};

IMG = GERT_DrawElement_Polygon(poly_params);
```

5.27 GERT_DrawElement_RadialGabor

DESCRIPTION:
This function will draw a radial Gabor stimulus onto a rectangular image patch, according to the parameters defined. The parameters typically manipulated are the SIGMA of the Gaussian components, the FREQ of the Sinusoidal component, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

```
params                                   required
                                         1x1 struct

        sigma                            optional, default:  2.5
                                         1x1 double
                                         >0, finite, real

        freq                             optional, default = 0.1
                                         1x1 double
                                         >0, finite, real

        size                             optional, default:  10
                                         1x1 double
                                         >0, integer value, finite, real

        phase                            optional, default:  0
                                         1x1 double
                                         finite, real

        amp                              optional, default:  1
                                         1x1 double
                                         ≥ 0, finite, real

        scale                            optional, default:  1
                                         1x1 double
                                         >0, finite, real

        lum_bounds                       optional, default:  [0 0.5 1]
                                         1x1 cell, containing 1x3 or 3x3 double
                                         ≥ 0 ≤ 1, finite, real
```

RETURNS:

```
IMG                                      MxM double, where M=(size*2)+1
```

DETAILS:
The PHASE and AMPlitude of the sinusoidal component can also be set. Setting LUM_BOUNDS allows the user to change the maximal, minimal and background luminance values independently. For instance, [0 0.2 1] will create a radial

Gabor against a dark gray background. To return a color image, pass a 3x3 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:

```
Draw a large blue Radial Gabor
gabel_params.sigma = 2.5;
gabel_params.size = 200;
gabel_params.freq = 0.12;
gabel_params.phase = 0;
gabel_params.scale = 20;
gabel_params.lum_bounds = {[0 0.5 1; 0 0.3 0.8; 1 1 1]};

IMG = GERT_DrawElement_RadialGabor(gabel_params);
```

5.28 GERT_DrawElement_Rectangle

DESCRIPTION:
This function will draw a single rectangle patch, according to the parameters defined. Important parameters are the WIDTH and the HEIGHT defining the rectangle, its OR, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
width	optional, default: 5 1x1 double >0, < size*2, finite, real
height	optional, default: 5 1x1 double >0, < size*2, finite, real
or	optional, default: 0 1x1 double finite, real
size	optional, default: 10 1x1 double >0, integer value, finite, real
scale	optional, default: 1 1x1 double >0, finite, real
aa	optional, default: 4 1x1 double 0,2,4,8, finite, real
lum_bounds	optional, default: [0.5 1] 1x1 cell, containing 1x2 or 3x2 double ≥ 0 ≤ 1, finite, real

RETURNS:

IMG	MxM double, where M=(size*2)+1

DETAILS:
Anti-aliasing is by default enabled, to disable it change AA to 0. Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:

```
rect_params.width = 5;
rect_params.height = 7;
rect_params.or = pi/4;
rect_params.scale = 3;
rect_params.size = 20;
rect_params.lum_bounds = {[1 0; 0 0; 0 1]};

IMG = GERT_DrawElement_Rectangle(rect_params);
```

5.29 GERT_DrawElement_Triangle

DESCRIPTION:
This function will draw a single triangle patch, according to the parameters defined. Required parameters are the WIDTH and the HEIGHT defining the triangle, its OR, and the SIZE of the image patch, defined as the total width divided by two, minus one. All these parameters are to be passed in pixel units.

ARGUMENTS:

params	required 1x1 struct
width	optional, default: 5 1x1 double >0, < size*2, finite, real
height	optional, default: 5 1x1 double >0, < size*2, finite, real
or	optional, default: 0 1x1 double finite, real
size	optional, default: 10 1x1 double

```
                                      >0, integer value, finite, real

        scale                         optional, default:  1
                                      1x1 double
                                      >0, finite, real

        aa                            optional, default:  4
                                      1x1 double
                                      0,2,4,8, finite, real

        lum_bounds                    optional, default:  [0.5 1]
                                      1x1 cell, containing 1x2 or 3x2 double
                                      ≥ 0 ≤ 1, finite, real
```

RETURNS:

```
IMG                                   MxM double, where M=(size*2)+1
```

DETAILS:
Anti-aliasing is by default enabled, to disable it change AA to 0. Setting LUM_BOUNDS allows the user to change the peak and background luminances separately. The first value is always the background luminance, even if the second is lower. To return a color image, pass a 3x2 matrix, with each row vector containing the luminance bounds for that RGB layer. Luminance bounds must always be passed as a 1x1 cell variable.

EXAMPLE:

```
triangle_params.width = 5;
triangle_params.height = 7;
triangle_params.or = pi/4;
triangle_params.scale = 3;
triangle_params.size = 20;
triangle_params.lum_bounds = {[1 0; 0 0; 0 1]};

IMG = GERT_DrawElement_Triangle(triangle_params);
```

5.30 GERT_GenerateContour_Ellipse

DESCRIPTION:
This function generates an ellipse contour description, from the length of the horizontal and vertical semi-axes (HAX and VAX) and clockwise rotation (ROT) defined. It returns a Cartesian contour description.

ARGUMENTS:

```
params                                required
                                      1x1 struct

        hax                           required
                                      1x1 double
                                      >0, finite, real
```

vax		required 1x1 double >0, finite, real
rot		optional, default: 0 1x1 double finite, real
th_range		optional, default: [0 2*pi] 1x2 double finite, real
th_n		optional, default: 1000 1x1 double >1, integer value, finite, real
scale		optional, default: 1 1x1 double >0, finite, real

RETURNS:

contour	1x1 GContour

DETAILS:
The resolution of the contour description is chosen through setting TH_-N (default: 1000 points). To generate an open contour, change the beginning and ending values in TH_RANGE. The function will return the contour segment that is always enclosed between the lowest and the highest value, but in reverse if the first value is the highest. The segment selected does not depend on the rotation of the shape. Note that in the final stimulus images, the segment selection goes counter-clockwise from 0 to 2*pi. Rotation is similarly counter-clockwise. An ellipse where shax<lax will then be vertically oriented, with point 0 at its lowest point in the image. The SCALE parameter allows rescaling of the entire contour (default: 1).

EXAMPLE:
```
params.hax = 50;
params.vax = 100;
params.rot = pi/4;
params.th_range = [0 pi/2];

contour = GERT_GenerateContour_Ellipse(params);
```

5.31 GERT_GenerateContour_FileSVG

DESCRIPTION:
This function reads in a file FNAME containing a plain SVG file, and render these vector graphics at a resolution RES. RES is equal to the number of points placed on each subcurve of the path. In practice, only one <path> definition

should be present in the file, although this path may be discontinuous. This wil result in an array of multiple GContours.

ARGUMENTS:

fname	required 1xN char Valid file in the Matlab path
res	optional, default: 100 1x1 double >0, integer value, finite, real

RETURNS:

contour	1xN GContour

DETAILS:
Supported path commands: MmHhVvLlCcSsQqTtAaZz. Transform commands (translation, rotation, skewing...) will be ignored. The recommended program for creating these files is the free Inkscape; you may save the file either as an Inkscape SVG or a plain SVG.

EXAMPLE:
```
contour = GERT_GenerateContour_FileSVG('R.svg');
```

5.32 GERT_GenerateContour_FileTXT

DESCRIPTION:
This function reads in a file FNAME consisting of XY coordinates, one pair per line, delimited by a DELIMITER (default: space). The coordinates need to be continuous along the contour, and the user should specify whether the contour is to be regarded as CLOSED or not. The XY coordinates should be in the first two columns, other columns will be ignored.

ARGUMENTS:

fname	required 1xN char Valid file in the Matlab path
closed	optional, default: false 1x1 logical
delimiter	optional, default: ' ' 1xN char Valid delimiter to the 'textread' functi on

RETURNS:

contour	1x1 GContour

DETAILS:
None.

EXAMPLE:
```
contour = GERT_GenerateContour_FileTXT('bear.txt',true,' ');
```

5.33 GERT_GenerateContour_RFP

DESCRIPTION:
This function generates an Radial Frequency Pattern contour description (Wilkinson, Wilson, & Habak, 1998). RFP's arise through sinusoidal deformation of a number of base circles, that are then summed to create a single contour. Each base circle has an identical base radius BASER, and a sinus deformation of amplitude AMP, frequency FREQ, and phase PH. The function returns a Cartesian contour description.

ARGUMENTS:

params	required 1x1 struct
baser	required 1x1 double >0, finite, real
amp	required 1xN double $\geq$ 0, finite, real
freq	required 1xN double >0, integer value, finite, real
ph	required 1xN double finite, real
th_range	optional, default: [0 2*pi] 1x2 double finite, real
th_n	optional, default: 1000 1x1 double >1, integer value, finite, real
rot	optional, default: 0 1x1 double finite, real
scale	optional, default: 1 1x1 double >0, finite, real

RETURNS:

contour 1x1 GContour

DETAILS:
The resolution of the contour description is chosen through setting TH_-N (default: 1000 points). To generate an open contour, change the beginning and ending values in TH_RANGE. The function will return the contour segment that is always enclosed between the lowest and the highest value, but in reverse if the first value is the highest. The segment selected does not depend on the rotation ROT of the shape. Note that in the final stimulus images, the segment selection goes counter- clockwise from 0 to 2*pi. Rotation is similarly counter-clockwise. Point 0 is at the lowest point in the image. The SCALE parameter allows rescaling of the entire contour (default: 1).

EXAMPLE:
```
params.baser = 10;
params.freq = [2 3 4];
params.amp = [1 0.5 2];
params.ph = [rand rand rand]*2*pi;
params.th_range = [0 pi];

contour = GERT_GenerateContour_RFP(params);
```

5.34 GERT_Init

DESCRIPTION:
Initialization routine. Global variables are initialized, and the necessary dependencies are checked.

ARGUMENTS:

None.

RETURNS:

None.

DETAILS:
None.

EXAMPLE:
None.

5.35 GERT_MergeElements

DESCRIPTION:
This function will join a vector of GElement objects into a single GElements object.

ARGUMENTS:

elements	required 1xN GElements or cell N>1

RETURNS:

merged_elements	1x1 GElements

DETAILS:
None.

EXAMPLE:
```
els = GERT_PlaceElements_Snake(params);
els1 = GERT_Transform_Shift(els,[-100 0]);
els2 = GERT_Transform_Shift(els,[100 0]);
all_els = GERT_MergeElements([els1 els2]);
all_els.dims = [-200 200 -200 200];
```

5.36 GERT_MinimizeCue_LocalDensity

DESCRIPTION:
This function will minimize the local density cue in the display. FNC1 specifies the function call string for the placement of foreground elements, as well as its arguments vector in ARGS1. Background elements are then placed around these points using GERT_PlaceElements_Background, according to the ARGS2 specified. OPTFIELD specifies whether the parameter to be manipulated pertains to the first or the second element placement function, as well as the name of the parameter and the range to be tested. CLD_PARAMS contains the parameters into the GERT_CheckCue_LocalDensity function. Optionally, a GElements object might be passed as FNC1, leaving the ARGS1 argument empty. A fixed set of foreground elements will then be used, and the user is required to manipulate an ARGS2 parameter instead.

ARGUMENTS:

fnc1	required 1xN char OR 1xN GElements/GContour/cell
args1	optional, pass [] to skip 1xM cell

types must match parameter types required by FNC1

args2	required 1x3 cell

types must match parameter types required by GERT_PlaceElements_Background

cld_params	required 1x1 struct (see GERT_CheckCue_LocalDensity)
optfield	required 1x3 cell
{1}	1x1 double (1 or 2)
{2}	1xN char, contains argument name
{3}	MxN, type must match argument

RETURNS:

opt_p	1x1 double

DETAILS:
The parameter names in FNC1 do not necessarily have to match the variables names in the ARGS1 cells. However, if the optimized parameter belongs to the first function, at least one argument name in FNC1 must be named 'params', and this must in ARGS1 be struct containing the parameter as a field.

EXAMPLE:
```
(1) fnc1 = 'GERT_PlaceElements_Contour(contour,params)'
args1 = [{contour},{params}]
args2 = [{[]},{peb_params}]
optfield = [{1},{'cont_avgdist'},{1:50}];
ansf = GERT_MinimizeCue_LocalDensity(fnc1,args1,args2,cld_params,optfield);

(2) elements = GERT_PlaceElements_Contour(contour,pec_params);
optfield = [{2},{'min_dist'},{10:20}];
ansf = GERT_MinimizeCue_LocalDensity(fnc1,args1,args2,cld_params,optfield);
```

5.37 GERT_PlaceElements_Background

DESCRIPTION:
This function will randomly place background elements around a series of existing FIXED_POINTS, respecting a MIN_DIST from other elements. If it is a GElements object, these elements will be added to ALL_ELEMENTS; if it is a Gcontour object, they will not. A REGION defined as one or more closed GContour objects can also be passed, to restrict the region inside which elements should be placed.

ARGUMENTS:

fixed_points	pass [] to skip, default: empty fields 1xN GElements, GContour or cell
region	pass [] to skip, default: empty fields 1xM GContour or cell
params	required 1x1 struct
min_dist	required 1x1 or 1xN+1 double ≥ 0, finite, real
dims	optional, can be taken from fixed_poin ts 1x4 double (1)<(3)&&(2)<(4), finite, real
bg_n	optional, default: minus 1 1x1 double >0 or minus 1, integer value, finite, re al
timeout	optional, default: 60 1x1 double >0, finite, real
batch_size	optional, default: 200 1x1 double >0, integer value, finite, real
border_dist	optional, default: min_dist+1 1x1 double ≥ 0 <dims/2, finite, real
in_region	optional, default: true 1x1 logical true or false
resolution	optional, default: 500 x1 double 1, integer value, finite, real

RETURNS:

all_elements	1x1 GElements

DETAILS:
By default, element placement will continue until the display is filled with elements. A hard limit to the number of elements added can be set using the BG_N parameter. The function will also stop when a TIMEOUT is reached (default: 60s). To change the distance kept from the border of the display, set BORDER_-DIST to a value other than the default MIN_DIST+1. If more than one MIN_DIST value is provided, these will refer to the various distances kept from the FIXED_POINTS positions. If only one value is provided, the same distance will be kept from other background points as is kept from the FIXED_POINTS. The DIMS variable uses arbitrary units; however whenever feasible, the most accurate results will be obtained when the canvas dimensions used here correspond to

the final pixel dimensions, due to possible rounding errors. Performance is negatively affected by the RESOLUTION at which element placement operates, i.e. the number of possible positions on the largest dimension. This parameter defaults to 500. Up to a point, increasing the BATCH_SIZE of the number of elements placed at once will increase performance. The ideal value depends on the other parameters, but values between 50 and 250 are good for most purposes (default: 200).

EXAMPLE:

```
1) With fixed elements
contour = GERT_GenerateContour_RFP(params);
cont_els = GERT_PlaceElements_Contour(contour, pec_params);
peb_params.dims = [-45 45 -45 45];
peb_params.min_dist = 1.5;
all_els = GERT_PlaceElements_Background(cont_els,[],peb_params);

2) Without fixed elements, and a bg_n limit
peb_params.dims = [-45 45 -45 45];
peb_params.min_dist = 1.5;
peb_params.bg_n = 500;
all_els = GERT_PlaceElements_Background([],[],peb_params);
```

5.38 GERT_PlaceElements_Contour

DESCRIPTION:

This function will place elements on a contour, defined as a vector of Cartesian coordinates. Three methods are available: Parallel equidistant placement, serial equidistant placement, and entirely random placement.

ARGUMENTS:

contour	required 1x1 GContour
params	required 1x1 struct
method	optional, default: 'ParallelEquidistant ' 1xM char array 'ParallelEquidistant', 'SerialEquidistan t', or 'Random'
METHOD: 'ParallelEquidistant'	
cont_avgdist	optional, default minus 1 (=determined by el_n) 1x1 double >0, finite, real
el_n	optional, default: minus 1 (=determined by cont_avgdist) 1x1 double >0 or minus 1, integer value, finite, re

```
                                 al

      eucl_mindist               optional, default:  0
                                 1x1 double
                                 ≥ 0, finite, real

      cont_startpos              optional, default:  minus 1 (=random)
                                 1x1 double
                                 0≤ x≤ 1 or minus 1, finite, re
                                 al

      noise_method               optional, default:  'Uniform'
                                 1xM char
                                 'Uniform', 'Gaussian', or 'Vector'

      noise_oncont               optional, default:  0
                                 1xI double
                                 0≤ x≤ 1 or 0.5≤ x≤ 0
                                 .5, finite, real

      noise_offcont              optional, default:0
                                 1xJ double
                                 0≤ x≤ 1 or 0.5≤ x≤ 0
                                 .5, finite, real

      noise_retries_n            optional, default:  0
                                 1x1 double
                                 ≥ 0, integer value, finite, real

      timeout                    optional, default:  60
                                 1x1 double
                                 >0, finite, real

   METHOD: 'SerialEquidistant'
   (where different from 'ParallelEquidistant' parameters)

      eucl_mindist               required
                                 1x1 double
                                 ≥ 0, finite, real

      dist_retries_step          optional, default:  cont_avgdist/5
                                 1x1 double
                                 >0, finite, real

   METHOD: 'Random'

      eucl_mindist               required
                                 1x1 double
                                 ≥ 0, finite, real

      el_n                       optional, default:  minus 1 (= until ful
                                 l)
                                 1x1 double
                                 >0 or minus 1, integer value, finite, re
                                 al

      resolution                 optional, default:  500
                                 1x1 double
                                 >1, integer value, finite, real

      noise_dilrad               optional, default:  0
                                 1x1 double
                                 ≥ 0, finite, real

      timeout                    optional, default:  60
```

	1x1 double >0, finite, real
batch_size	optional, default: 100 1x1 double >0, integer value, finite, real

RETURNS:

elements	1x1 GElements
ors	1xn double
actual_vals	1x1 struct

DETAILS:
'ParallelEquidistant' is the default method. Here, the length of the contour will be subdivided in equal segment, respecting the CONT_AVGDIST between elements as closely as possible. Alternatively, you may provide the EL_N parameter to automatically determine this value based on the number of elements. Position noise can be applied either along the contour through NOISE_ONCONT, or perpendicular to it, through NOISE_OFFCONT. These values reflect the limits of the noise offset, where 0 is no noise and 1 is from -0.5*cont_avgdist to 0.5*cont_avgdist. For the Gaussian case, this corresponds to 2SD. Note that the Gaussian distribution is therefore cut off at 2SD, and not truly Gaussian. For 'Vector' noise, a sufficiently large vector should be passed, from which the position deviations will be sampled randomly. Noise values are to be understood as a proportion of cont_avgdist. CONT_STARTPOS can be set to a fixed value; if not specified, the position of the first element will be random. The value is to lie between 0 and 1, where 0.5 is placement on the middle of the segment. Should the EUCL_MINDIST be violated by the noise displacements, the function will attempt NOISE_RETRIES_N new placements of all points. This method is especially useful for simple, smooth contours, where different parts of the contour description do not typically come within a distance of EUCL_MINDIST of one another.

'SerialEquidistant' uses similar methods, but places the points one by one. If even after a number of noise retries no suitable solution is found, the average distance is increased by DIST_RETRIES_STEP, until a point along the contour is found where no conflicts exist with previous points. No position noise is applied to this point. For the next point, the normal procedure is continued. This method is especially useful for complex contours with difficult parts, where contour segments come too close to one another.

'Random' uses methods similar to the GERT_PlaceElements_Background function: Elements are placed on the remaining possible positions (that is, at a distance of minimally EUCL_MINDIST from other points) until the contour is filled, or the EL_N limit to the number of elements is reached. Setting RESOLUTION and BATCH_SIZE to different values may affect precision and performance. Position noise can be applied through setting the NOISE_DILRAD parameter, which applies 'blurring' to the contour description that limits the element placement. This method is useful for any contour where equidistance is not necessarily required, and allows element placement even on non-continuous contours.

In addition to the 'elements' structure containing the Cartesian coordinates of the contour points placed, their orientation along the contour is returned, if the original contour contained both cdist and lt fields. As the third return argument, the actual values used in a method may be found, since they might deviate from those passed as a parameter. For instance, cont_avgdist in the ParallelEquidistant case can only take certain discrete values.

EXAMPLE:

(1) Equidistant placement with slight jitter on the contour

```
contour = GERT_GenerateContour_RFP(params);
pec_params.eucl_mindist = 1;
pec_params.cont_avgdist = 5;
pec_params.noise_method = 'Gaussian';
pec_params.noise_oncont = 0.2;
els = GERT_PlaceElements_Contour(contour, pec_params);
```

(2) Random placement of exactly 15 elements

```
contour = GERT_GenerateContour_RFP(params);
pec_params.method = 'Random';
pec_params.eucl_mindist = 1;
pec_params.el_n = 15;
els = GERT_PlaceElements_Contour(contour, pec_params);
```

5.39 GERT_PlaceElements_Snake

DESCRIPTION:

This function generates directly generates 'snake' grouping elements, without needing to define an underlying continuous contour description. Similar to the methods of Hess & Dakin (1999) it will construct a snake as a series of SEG_N connected line segments, each of length SEG_LEN. The average angle between each successive segment equals SEG_OR_AVGANG. The snake elements are then placed on the midpoints of these segments. The function returns both the position of the snake elements, and the orientation of the segments on which they were placed.

ARGUMENTS:

params	required 1x1 struct
seg_n	required 1x1 double >0, integer value, finite, real
seg_len	required 1x1 double >0, finite, real
seg_or_avgang	required 1x1 double

	≥ 0 ≤ pi, finite, real
seg_or_jitang	optional, default: 0 1x1 double ≥ 0 ≤ seg_or_avgang, finite, real
seg_or_bias	optional, default: 0.5 1x1 double ≥ 0 ≤ 1, finite, real
pt_noise_onseg	optional, default: 0 1x1 double ≥ 0, finite, real
pt_noise_offseg	optional, default: 0 1x1 double ≥ 0, finite, real
rot	optional, default: rand*2*pi 1x1 double ≥ 0 ≤ 2*pi, finite, real

RETURNS:

elements	1x1 GElements
ors	1xn double

DETAILS:
The shape of the snake can be manipulated further. SEG_OR_JITANG controls the amount of uniform orientation jitter, relative to the SEG_OR_AVGANG average angle between successive line segments. The default segment orientation jitter is 0. Whereas the chance of this segment angle pointing to the left or the right is by default 0.5, SEG_OR_BIAS allows the introduction of bias, between 0 and 1. Position noise can be added to the element placement, either along the segment (PT_NOISE_ONSEG) or perpendicular to it (PT_NOISE_OFFSEG). By default both are equal to 0, meaning no noise. As this value nears 1, a uniform noise equal to the length of the segment can be applied. Position noise exceeding the segment length is not possible. An overall rotation ROT can also be specified. By default the rotation equals rand*2*pi, that is, completely random.

EXAMPLE:
```
params.seg_n = 7;
params.seg_len = 5;
params.seg_or_avgang = pi/6;
params.seg_or_jitang = pi/10;
params.pt_noise_onseg = 0.2;
snake = GERT_PlaceElements_Snake(params);
```

5.40 GERT_RenderDisplay

DESCRIPTION:
This function renders the stimulus image, using the EL_FNC function to draw each element provided in the ELEMENTS object, according to a fitting set of EL_-PARAMS.

ARGUMENTS:

elements	required 1x1 GElements
el_fnc	required 1x1 function handle must be compatible with el_params
el_params	required 1x1 struct of 1x1 or 1xN fields el_fnc will check the fields
img_params	optional, skip to use default values 1x1 struct
dims	optional, default: elements.dims(2) and (4) 1x2 double >1, integer value, finite, real
bg_lum	optional, default: 0.5 1x1 or 1x3 double ≥ 0 ≤ 1, finite, real
blend_mode	optional, default: 'none' 1xM char array 'MaxDiff' or 'None'
global_rendering	optional, default: 'false' 1x1 logical

RETURNS:

IMG	dims(2) x dims(4) double
pospix	1x1 struct
x	1xelements.n double
y	1xelements.n double

DETAILS:
The EL_PARAMS structure must consist of either scalars, or vectors with the length equal to the number of elements. In the first case, all elements will receive the same constant value for that parameter, in the second case each element will receive its own parameter value. These parameters must be compatible with the EL_FNC drawing function, and will not be checked by this function. DIMS are the pixel dimensions of the image. The dimensions of the elements struct can be in any arbitrary units, and will automatically be

rescaled to these values. If DIMS is omitted, the elements struct dimensions will be taken as pixel dimensions (if possible). By default, BG_LUM will be set to 0.5, i.e. gray. Only one image layer will then be created. If a 1x3 vector is used, a three-layered RGB image will be created. However, EL_FNC must then also return three-layered RGB image patches, through setting its parameters correctly BLEND_MODE pertains to the method used to paste image patches into the overall image. The default mode is 'None', meaning a simple paste on top of what has already been generated before that element. 'MaxDiff' is convenient when the image patches overlap slightly; when pasting, it will retain the pixel value that is maximally different from BG_LUM. If the image patches overlap by too much, this might result in strange effects, however. When GLOBAL_RENDERING is enabled, the function will store all patches rendered in global space, to re-use should any next image have an identical element.

EXAMPLE:

```
(1) Grayscale image of randomly placed and oriented Gabors
peb_params.dims = [1 500 1 500];
peb_params.min_dist = 25;
all_els = GERT_PlaceElements_Background([],[],peb_params);

gabel_params.sigma = 2.5;
gabel_params.amp = 1;
gabel_params.size = 10;
gabel_params.freq = 0.1071;
gabel_params.phase = 0;
gabel_params.scale = 1;
gabel_params.or = pi*rand(1,all_els.n);

IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor,all_els,gabel_params);
imshow(IMG);

(2) Color image of orthogonally oriented elements
img_params.bg_lum = [0.5 0.5 0.5];
idx1 = 1:100; idx2 = 100:all_els.n;
gabel_params.or(idx1) = 0;
gabel_params.or(idx2) = pi/2;
gabel_params.lum_bounds = {[0.5 0 0; 0.5 0.5 0.5; 1 0.5 0.5]'};

IMG = GERT_RenderDisplay(@GERT_DrawElement_Gabor,all_els,gabel_params,img_-
params);
imshow(IMG);
```

5.41 GERT_ShowError

DESCRIPTION:
Handles the different types of errors thrown by other functions.

ARGUMENTS:

fnc	required 1xN char
msg	required 1xM char
lvl	required 1x1 double 1,2,3,4
E	optional 1x1 MException

DETAILS:
Level 1 is a command window warning, level 2 a pop-up dialog without halting, level 3 a pop-up dialog with halting, and level 4 an interpretation of a thrown exception. An MException object then needs to be passed. Warning messages can be suppressed for each GERT function separately. E.g., warning('off','GERT:GERT_RenderDisplay')

EXAMPLE:
```
GERT_ShowError('GERT_Init','Could not initialize',3);
```

5.42 GERT_Transform_Center

DESCRIPTION:
This function will put the center of a GContour or GElements collection of Cartesian coordinates to a specified point TO_VALS. CENTER_TO describes the method used for determining which point should be moved there. By default, the bounding box is used.

ARGUMENTS:

cart	required 1x1 GContour or GElements
to_vals	optional, pass [] to skip, default: [0 0] 1x2 double finite, real
center_to	optional, omit or pass '' to skip, default: 'BoundingBox' 1xM char 'Median','Mean','Centroid','GeoMean','Custom','DispCenter', 'BoundingBox'
custom_vals	optional, omit to skip, default: [0 0] 1x2 or 1x4 double finite, real

RETURNS:

d_cart	1x1 GContour or GElements
d	1x2 double
ct	1x2 double

DETAILS:
The 'Mean' method will move the mean of all Cartesian coordinates in CART to the point specified in TO_VALS. 'Median' analogously uses the Median of those points. 'Centroid' will first compute the centroid using GERT_Aux_Centroid. The points should then describe a continuous, closed contour. 'GeoMean' uses the geometrical mean. 'Custom' will take the value in CUSTOM_VALS and move it to TO_VALS. If omitted, (0,0) will be used. 'DispCenter' will use the center of the display. In case of a GContour object, or when GElements.dims is empty, the dimensions can be specified in custom_vals. 'BoundingBox' will center the bounding box. The displacement itself is returned as D. Please note that the display dimensions will remain unaffected by this function. Through adding D to them, they can be shifted as well. The point from which the displacement was done (e.g., the centroid) is returned as CT

EXAMPLE
```
(1) Move the median of this contour to point (300,300)
cart = GERT_GenerateContour_RFP(params);
cart = GERT_Transform_Center(cart,[300 300],'Median');

(2) Move the display center to (0,0), and shift the dims too
params.dims = [1 500 1 500]; params.min_dist = 10;
cart = GERT_PlaceElements_Background([],[],params);
[cart,d] = GERT_Transform_Center(cart,[0,0],'DispCenter');
cart.dims = [cart.dims(1)+d(1) cart.dims(2)+d(1) cart.dims(3)+d(2)
cart.dims(4)+d(2)];
```

5.43 GERT_Transform_Flip

DESCRIPTION:
This function will flip a GContour or GElements collection of Cartesian coordinates around a specified flip point SCALE_POINT, and a flip axis FLIP_-AXIS. FLIP_POINT uses the same strings as GERT_Transform_Center, and CUSTOM_VALS serves the same function as FROM_VALS in that function.

ARGUMENTS:

cart	required 1x1 GContour or GElements
flip_axis	required 1x1 double, finite, real
flip_point	optional, default: 'BoundingBox' 1xM char 'Median','Mean','Centroid','GeoMean','Cu

	stom','DispCenter', 'BoundingBox'
custom_vals	optional 1x2 or 1x4 double, finite, real

RETURNS:

d_cart	1x1 GContour or GElements

DETAILS:
None.

EXAMPLE:
```
(1) Flip horizontally around the centroid
cart = GERT_GenerateContour_RFP(params);
cart = GERT_Transform_Flip(cart,0,'Centroid');

(2) Flip around the main axis of the object
[ma,pt] = main_axis(cart);
cart = GERT_Transform_Flip(cart,ma,'Custom',pt);
```

5.44 GERT_Transform_Rotate

DESCRIPTION:
This function will rotate a GContour or GElements collection of Cartesian coordinates around a specified point ROT_POINT, by an amount ROT_ANG. ROT_POINT uses the same strings as GERT_Transform_Center, and CUSTOM_VALS serves the same function as FROM_VALS in that function.

ARGUMENTS:

cart	required 1x1 GContour or GElements
rot_ang	required 1x1 double, finite, real
rot_point	optional, default: 'BoundingBox' 1xM char 'Median','Mean','Centroid','GeoMean','Custom','DispCenter','BoundingBox'
custom_vals	optional 1x2 double, finite, real

RETURNS:

d_cart	1x1 GContour or GElements

DETAILS:
None.

EXAMPLE:
```
(1) Rotate around the centroid by 90 degrees
cart = GERT_GenerateContour_RFP(params);
cart = GERT_Transform_Rotate(cart,pi/2,'Centroid');

(2) Rotate by 45 degrees around (20,20)
cart = GERT_PlaceElements_Background([],[],params);
cart = GERT_Transform_Rotate(cart,pi/4,'Custom',[20 20]);
```

5.45 GERT_Transform_Scale

DESCRIPTION:
This function will scale a GContour or GElements collection of Cartesian coordinates around a specified point SCALE_POINT, by an amount SCALE_FACTOR. SCALE_POINT uses the same strings as GERT_Transform_Center, and CUSTOM_VALS serves the same function as FROM_VALS in that function.

ARGUMENTS:

cart	required 1x1 GContour or GElements
scale_factor	required 1x1 double finite, real
scale_point	optional, default: 'BoundingBox' 1xM char 'Median','Mean','Centroid','GeoMean','Cu stom','DispCenter', 'BoundingBox'
custom_vals	optional 1x2 or 1x4 double, finite, real

RETURNS:

d_cart	1x1 GContour or GElements

DETAILS:
None.

EXAMPLE
```
(1) Scale down by a factor 2, keeping the centroid constant
cart = GERT_GenerateContour_RFP(params);
cart = GERT_Transform_Scale(cart,0.5,'Centroid');

(2) Scale up by a factor 2, keeping (20,20) constant
```

```
cart = GERT_PlaceElements_Background([],[],params);
cart = GERT_Transform_Scale(cart,2,'Custom',[20 20]);
```

5.46 GERT_Transform_Shift

DESCRIPTION:
This function will translate a GContour or GElements collection of Cartesian coordinates along a specified distance D.

ARGUMENTS:

cart	required 1x1 GContour or GElements
d	required 1x2 double, finite, real

RETURNS:

d_cart	1x1 GContour or GElements

DETAILS:
None.

EXAMPLE:
```
cart = GERT_GenerateContour_RFP(params);
cart = GERT_Transform_Shift(cart,[100 0]);
```

5.47 GERT_Version

DESCRIPTION:
Outputs the current version of GERT.

ARGUMENTS:

None.

DETAILS:
None.

EXAMPLES:
None.

PART VI

An introduction to Object-Oriented Programming

MATLAB's default data types, such as `double`, `struct`, `cell` or `uint8`, are probably familiar to you. However, its object-oriented programming (OOP) framework also allows the definition of custom data types, usually called *classes*. Classes can then be instantiated into *objects*. Exactly like one can create a double variable named 'a' and another double variable named 'b', we have defined a custom class GContour that allows you to create separate GContour objects named, for instance, 'C1' and 'C2'. The Matlab syntax for creating a GContour object is: `C1 = GContour;`

Classes contain data fields or *properties*, much like a struct. For instance, each GContour object has a vector `x` and a vector `y` to store the contour coordinates. However, the definition of the GContour class also puts restrictions on which fields can be present exactly, and what they should look like. As an example, you will not be able to create a new field `contour_name` in a GContour object, nor will you be able to change the property `closed` to anything else than `true` or `false`. In addition to data, classes also contain *methods* - simply put, their very own functions. An intuitive example here is the 'plot' method. By executing `plot(C)` (or `C.plot`), where C is a valid GContour object, a graphical display of the contour will automatically be generated, using the data present in the GContour object.

One final point of interest is the concept of a *constructor*. To save time and coding space, a newly generated object does not have to be empty. By passing arguments to GContour when creating the object, some of the fields can automatically be set to certain values. For instance, if you have a separate x and y vector available already, the command `C1 = GContour([x;y]);` will automatically fill the GContour `x` and `y` fields with these data upon creation of the object.

The advantages of this approach over using structures to hold contour data should be clear: Increased control over the data defining a contour, and automatic coupling of these data with functions that can use them. Moreover, there is the advantage of sheer clarity. When we speak in this manual of a GContour object, the MATLAB workspace overview will display it as such, and you as a user can be certain that it will automatically contain all the relevant data fields (although possibly, not with valid values for a given function).

PART VII

A note for Octave users

Octave users should experience few problems, provided they have downloaded the correct file, for Octave and older MATLAB versions. The only difference between this version and the standard GERT installation is the implementation of the classes. From MATLAB 2007b onward, a new and much easier OOP framework became available, in which we opted to write GERT, not in the least because it allows users to browse normally through objects as through a struct. However older MATLAB versions and Octave do not support this new style of class implementation. We have therefore also written alternate, old-style class implementations for those users.

The primary issue for Octave users will be speed: Where MATLAB finishes the demo script in a few seconds, Octave can take 15-30 seconds even on a fast computer. Optimizing the code for Octave is not a priority, although we will of course be happy to receive your suggestions for improvement. Some further differences at this moment:

- The `imread` function requires that the `IMAGE_PATH` function is used to define its path. Inspect `GERT_Demo.m` for an example.

- Concatenation of GERT objects into an array of the actual object type does not work. Instead, a cell array must be passed. This also implies that no discontinuous paths can be read from SVG files.

- The Voronoi tesselation and the T-test used in the local density check do not work. Use `'AvgDist'` and `'MC'` instead. `avg_n` cannot be set to 0.

- No curve fit is performed in the local density minimization routine. However the figure is available for visual inspection.

- Lanczos anti-aliasing is replaced by a linear interpolation for element drawing.

- No display methods have been implemented for the GERT classes, objects may therefore appear empty in the browser or the command window even when they are not.

- Text files containing xy contour description points must consist of exactly two columns.

- The `implay` function, used to display movies in some examples, doesn ot exist in Octave.

Bibliography

Braun, J. (1999). On the detection of salient contours. *Spatial Vision*, *12(2)*, 211-25.

Demeyer, M., & Machilsen, B. (2012). The construction of perceptual grouping displays using GERT. *Behavior Research Methods*, *44*(2), 439-446.

Hess, R. F., & Dakin, S. C. (1999). Contour integration in the peripheral field. *Vision Research*, *39*, 947-959.

Kubovy, M., & Wagemans, J. (1995). Grouping by proximity and multistability in dot lattices: A quantitative Gestalt theory. *Psychological Science*, *6*(4), 225-234.

Snodgrass, J. G., & Vanderwart, M. (1980). A standardized set of 260 pictures: Norms for name agreement, image agreement, familiarity, and visual complexity. *Journal of Experimental Psychology: Human Learning & Memory*, *6*(2), 174-215.

Wilkinson, F., Wilson, H., & Habak, C. (1998). Detection and recognition of radial frequency patterns. *Vision Research*, *38*, 3555-3568.

Acknowledgments

We thank Lee de-Wit, Naoki Kogo, Krista Overvliet, Tom Putzeys, Michaël Sassi, Kathleen Vancleef, Nathalie Van Humbeeck, and Sander Van de Cruys for suggestions and testing of the GERT toolbox. We also thank Geir Nygård and Lizzy Bleumers for laying out the foundations from which we started. We like to acknowledge Rudy Dekeerschieter for maintenance of the GERT website. The development of GERT toolbox was supported by a Methusalem grant from the Flemish Government awarded to Johan Wagemans (METH/08/02) and a PDMK grant awarded to Maarten Demeyer (PDMK/10/061). Bart Machilsen is a postdoctoral research fellow of the Research Fund K.U. Leuven (PDMK/11/055). Maarten Demeyer and Bart Machilsen currently each hold a postdoctoral research fellowship from the Research Foundation-Flanders (FWO).