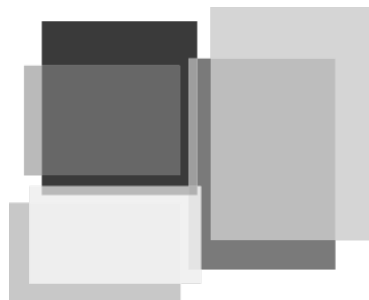


Mini-project: Overlapping Rectangles



How can a software detect overlapping objects? This and other questions are answered in this project. You will work with some simple **geometric concepts and graphics** during the project.

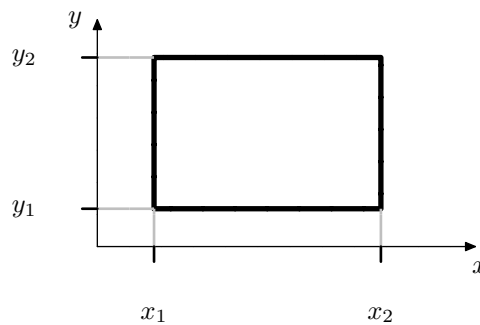
September 15, 2010

1 Assignment

This assignment deals with finding overlap between rectangles. This is a simplified version of the general problem of finding overlap between general shapes, an important component for instance in CAD applications.

1.1 Specification

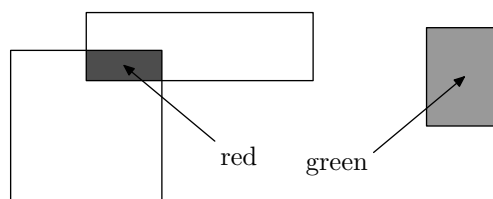
Represent the rectangles using an $n \times 4$ matrix, for n rectangles, where each rectangle is specified using the four values: x_1, y_1, x_2, y_2 as in the figure below. The coordinates satisfy $x_1 < x_2$ and $y_1 < y_2$.



1.2 Examination

- Produce a MATLAB function that finds non-overlapping rectangles and overlapping regions. The rectangles, specified as described above, should be the only argument to the function.

The output from the function should be a plot. In the plot non-overlapping rectangles should be green, and overlapping regions red. Areas of overlapping rectangles that is not part of any overlap should not be filled, as in the example below.



- Write another function to generate hundreds of random rectangles. The function must take at least one argument, how many rectangles to generate, however there are other arguments that you might want to add. The generated rectangles should be interesting, *i.e.*, all types of overlap as well as non-overlapping rectangles must be generated. The result of the function should be a matrix that can be entered directly into the overlap function.

- If you find this interesting, try to figure out how to handle overlap with triangles or circles.

2 Suggested Solution Outline

First read through the problem specification and the suggested solution outline and try to understand how it divides the problem into smaller subproblems, each of which is easier to solve than the original problem.

The steps below provide one way to split the main programming problem into logical subproblems. The subproblems are deliberately chosen so that it is possible to test the code before continuing with the next step. Do take these opportunities! Following the suggestions closely can result in many small functions solving small parts of the problem, however, it may not be the best, most efficient, nor smartest way to call these small functions to solve the whole problem. Some of the smaller functions (all?) are better suited to be pasted into the code where needed.

Note that this solution outline is intended for students with no or little previous programming experience. A skilled programmer would probably, based on previous experience, divide the problem into different and/or fewer parts. If you consider yourself a skilled programmer and think that you have a better solution to the problem you are allowed to solve it your way. However, if you need help, the teaching assistant has more experience of the outlined solution.

2.1 Drawing Rectangles

To enable easy testing of the overlap computations, first write a function to draw rectangles. Let the function have two arguments: the first a list of rectangles to draw the outline of and the second a list of indices of rectangles to paint in green. It is a good idea to use the same rectangle representation as above.

Start with the green rectangles and draw the outlines last. In this way all edges remain visible.

The MATLAB function `hold on` may come handy when drawing many rectangles. If you choose to use it in your drawing function make sure to clear the figure before plotting anything, otherwise you will add to a existing figure. The `close` function (read the online help) ensures that you start fresh each time.

Test your function with some examples before you proceed.

2.2 Random Rectangles

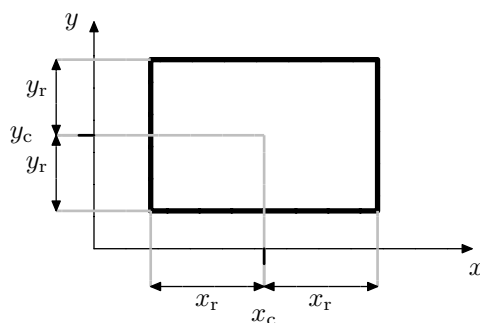
Now is a good time to write the function to generate random rectangles. Make sure it returns interesting rectangle combinations even for two rectangles, *i.e.*,

that it produces different kinds of overlap as well as non-overlapping rectangles. Tip: let the height and width of the rectangles depend on the number of rectangles to generate.

To test the function, plot the rectangles generated with the function using the drawing function, *e.g.*, use an empty vector as second argument to draw the outline of all rectangles.

2.3 Another Rectangle Representation

There are rectangle representations better suited to determine overlap than the one used as input. Now, since each rectangle will be tested against many other rectangles it is a good idea to convert the representation once for all instead of once for each test. Write a function that given the representation in the specification computes and returns a new representation of the same dimension where each row holds the four values: x_c , y_c , x_r , and y_r as depicted in the figure below.



Test the functionality with some examples and make sure it works properly!

2.4 Overlap, one vs. one

Next write a function to test if two rectangles overlap. To simplify testing let the function take only one argument holding both matrices. Let the input be on the rectangle representation used in the specification and convert the representation in your function.

The overlap test, expressed in words, is to test for each axis separately, if the distance between the center of the two rectangles is shorter than the sum of the distances from the centers to the sides. That is, if $|x_{1,c} - x_{2,c}| < x_{1,r} + x_{2,r}$ the rectangles overlap horizontally. If they overlap vertically as well then the rectangles two overlap.

Let the function do two things: return if the rectangles overlap (1 for true or 0 for false in MATLAB), and paint the rectangles green if there is no overlap and otherwise outline them. For now, disregard the overlapping region.

2.5 Overlap, one vs. many

Now extend the previous function so that it tests for overlap between the first rectangle and all the others. Note, this is not a test where all combinations of rectangles are tested. Paint the first rectangle green if it does not overlap with any other rectangle, and paint the other rectangles green if they do not overlap with the first one.

You will have to come up with a way to remember which rectangles to paint green to be able to use your paint function. One way to do this is to put the index of non-overlapping rectangles in a row vector, and then use the vector to decide which rectangles to paint green. However, when considering the many *vs.* many test this turns out not to be the best way to do it.

Introduce an overlap *flag*. Create a row vector with as many elements as there are rectangles. Let the elements in this vector indicate with its value (0 or 1) if rectangle with the same index overlap with any other rectangle. This type of indicators are often called flags. Before any tests have been conducted no rectangles are known to overlap, hence initiate the vector to 0. Once an overlap is found the flags for the overlapping rectangles are set (set to 1). (It is okay to set the flag several times.) Note that you never reset any flags (set to 0) because rectangles 1 and 2 still overlap even though 1 and 3 do not. Once all rectangles have been checked for overlap a call to `find` can be used to determine the indices for the overlapping rectangles, as well as those without known overlap.

Introduce a loop in the previous function to test the first rectangle against all other.

Generate sets of rectangles and test your function. If some rectangles are just outlined then one of them (the first rectangle) cut through all the other. Make sure you get a couple of such test cases before you continue!

2.6 Overlap, many vs. many

Now check all rectangles against each other. Introducing another loop in the code written in the last step does the trick. Let the new loop iterate over all rectangles and test for overlap between the current and all other rectangles, hence introduce a variable that indicates which rectangle should be tested against the other. It is, of course, enough to test each pair of rectangles once. Therefore only test a rectangle against rectangles with higher index.

This function works for many rectangles so test it thoroughly.

2.7 Shape of Overlap

The last step is to compute the shape of the overlaps. This is possible to do by adding a few lines of code where the overlap detected.

The shape of the overlap is best computed using the original rectangle representation. Using min and max to compute the corners of the overlap is prob-

ably the easiest way to go, but other methods are also allowed. If you cannot come up with a solution in few minutes, ask the teaching assistant for help!

It is a good idea to paint the overlaps red as you find them, this way you do not have to store the shape for later. Now, the problem should be solved! (Remember that you probably need to use `hold on` before you start to plot anything.)

Test the final product! The functions you have written are supposed to be used in the manner below (assuming you called the rectangle generating function `randrect` and the plotting function `overlap`):

```
>> P = randrect(100);  
>> overlap(P)
```