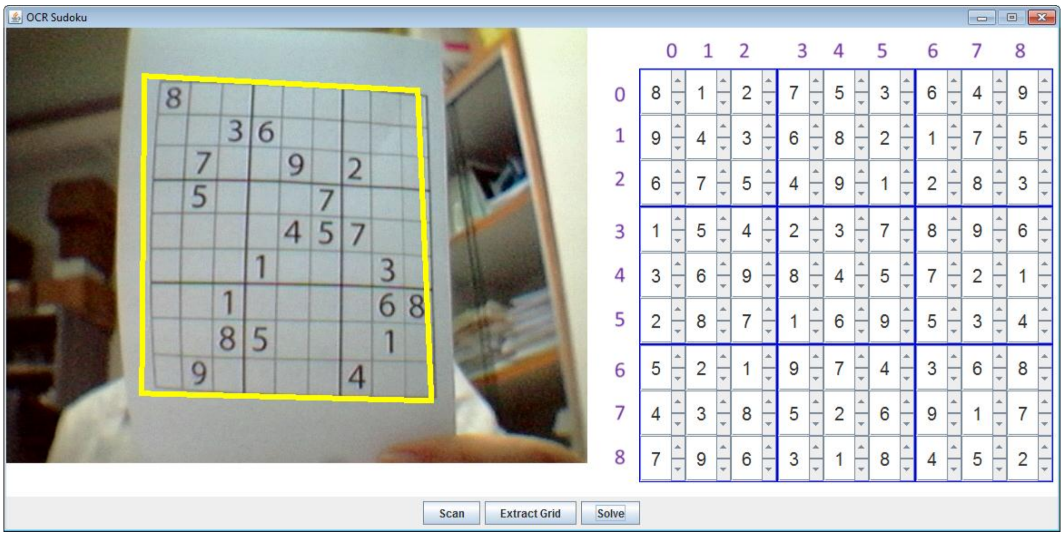
Chapter 12. OCR and Sudoku

This chapter looks at how to use the webcam for optical character recognition (OCR) while solving Sudoku puzzles. The application is shown in Figure 1 after having analyzed the webcam image on the left, and completed the puzzle on the right.



	0	1	2	3	4	5	6	7	8
0	8	1	2	7	5	3	6	4	9
1	9	4	3	6	8	2	1	7	5
2	6	7	5	4	9	1	2	8	3
3	1	5	4	2	3	7	8	9	6
4	3	6	9	8	4	5	7	2	1
5	2	8	7	1	6	9	5	3	4
6	5	2	1	9	7	4	3	6	8
7	4	3	8	5	2	6	9	1	7
8	7	9	6	3	1	8	4	5	2

Figure 1. A Solved Sudoku.

OCR presents two coding problems – it requires a good-quality image (which my old webcam fails to deliver), and OCR functionality isn't part of standard OpenCV.

Tesseract is perhaps the most popular free OCR library, with several Java bindings, including tesjeract (http://code.google.com/p/tesjeract/) and Tess4J (http://tess4j.sourceforge.net/). Alternatively, it's easy to call the tesseract command line tool via Java's Runtime.exec() method. However, Sudoku doesn't require the power, flexibility or accuracy of Tesseract, because the Sudoku format is well-defined (a standard grid) and uses a limited range of characters (the digits, 1 to 9). As a consequence, I'll be using the much simpler gocr OCR command line tool (http://jocr.sourceforge.net/).

JavaCV still plays an important role in the application, to improve the webcam image before passing it over to gocr. The improvements include smoothing, adaptive thresholding, contour finding, perspective warping, and flood-filling, which I'll detail later.

1. An Overview of the Application

The SudokuOCR application requires the user to carry out four tasks before a puzzle can be solved, which are numbered in the flowchart in Figure 2.

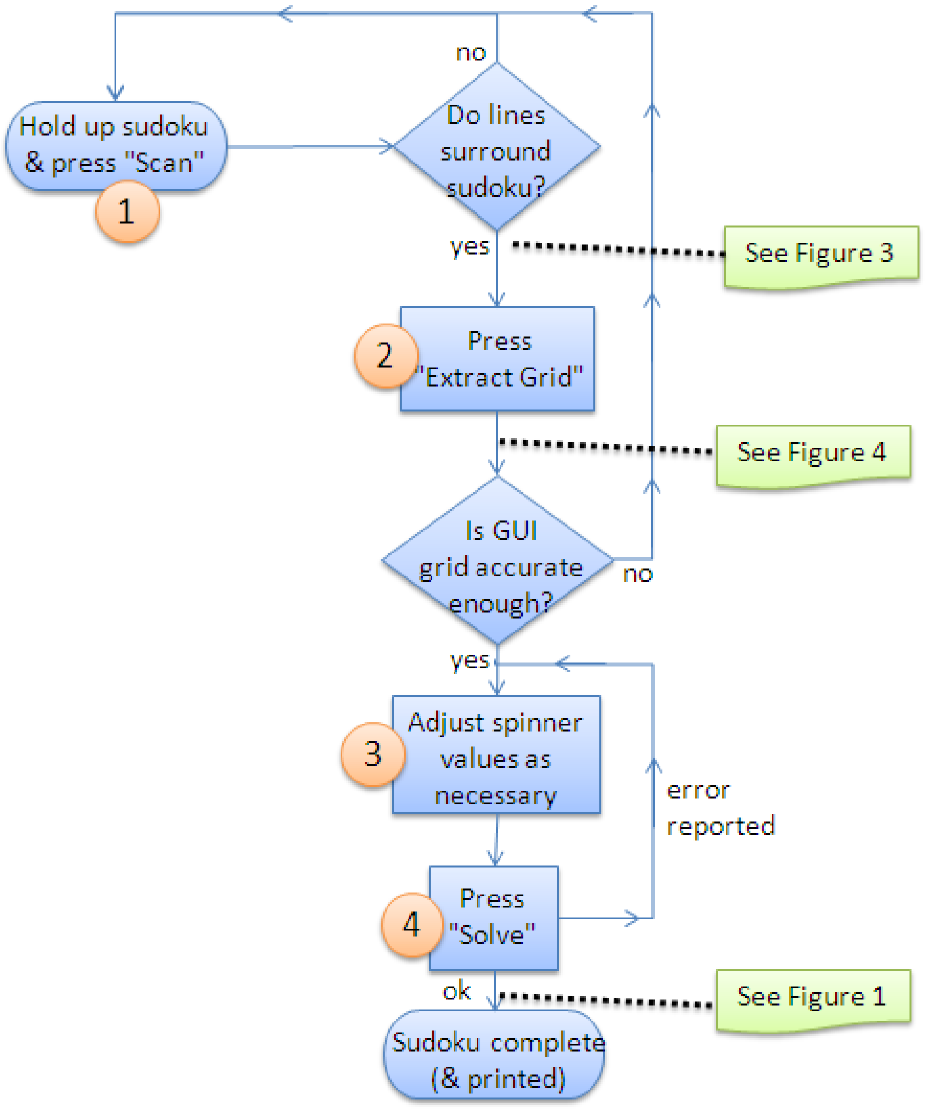

Hold up sudoku & press "Scan"
1
Do lines surround sudoku?
no
yes
See Figure 3
2
Press "Extract Grid"
See Figure 4
Is GUI grid accurate enough?
no
yes
3
Adjust spinner values as necessary
error reported
4
Press "Solve"
ok
See Figure 1
Sudoku complete (& printed)

Figure 2. Steps for Solving a Sudoku.

The user needs to hold the Sudoku puzzle close to the webcam, and press the "Scan" button in the application. A yellow quadrilateral is drawn around the detected puzzle and a copy of the image inside the region is retained for later processing. The puzzle can be held at an angle, and still be detected, as shown in Figure 3.

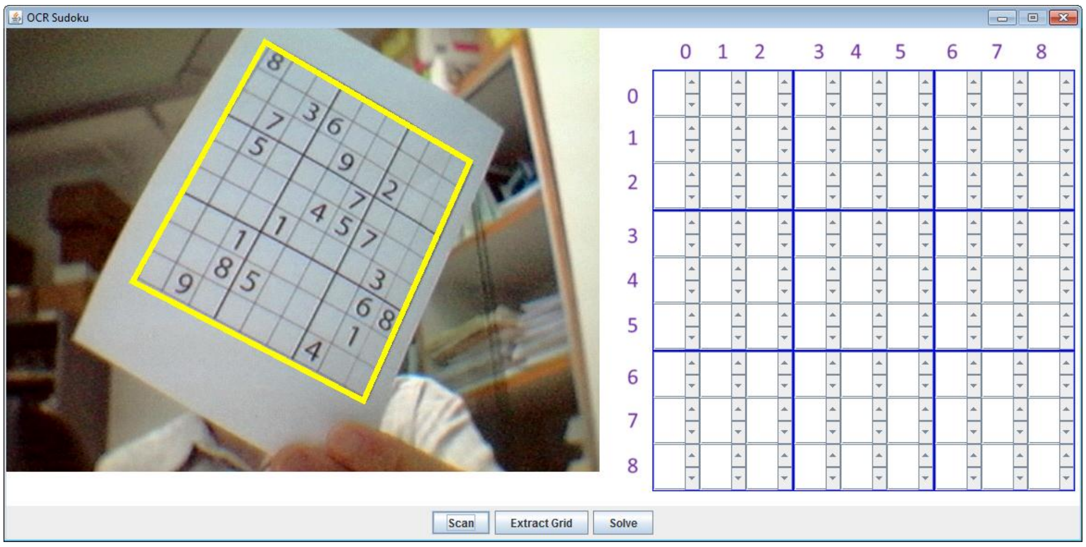
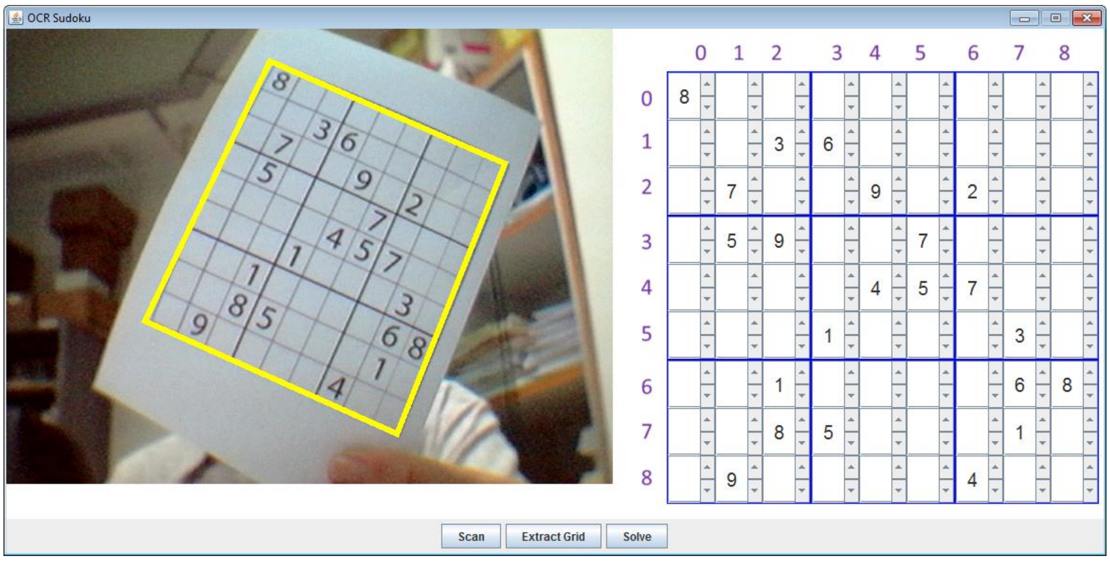


Figure 3. Sudoku at an Angle.

When the user presses the "Extract Grid" button, OpenCV processing is carried out on the stored image, including perspective warping to straighten it, and flood filling to remove some of the Sudoku grid lines. Then OCR processing by the gocr tool extracts integers from the image, which are written into the grid on the right of the GUI, as in Figure 4.



	0	1	2	3	4	5	6	7	8
0	8								
1			3	6					
2		7			9		2		
3		5	9			7			
4					4	5	7		
5				1				3	
6			1					6	8
7			8	5				1	
8		9					4		

Figure 4. OCR Detected Digits.

The OCR results always contain a few errors – typically 3 or 4 numbers out of 20 may be wrong, so the user needs to do a careful comparison of the image and the numbers grid. For example, in Figure 4 there's an extra '9' in the top corner of the left-hand box of the middle row.

The GUI grid is composed out of 81 JSpinner objects (laid out in a 9x9 grid), which the user can adjust by pressing the arrows on each spinner's side.

Once the GUI grid has been adjusted, the "Solve" button is pressed to invoke a recursive backtracking search function that tries different values in the blank Sudoku squares until a complete solution is found.

The Sudoku example in Figure 1 is sometimes claimed to be the World's hardest (e.g. at http://www.telegraph.co.uk/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html). Over several timing tests, the search function took about 1 second to generate an answer, while the computer vision parts took around 1.4 seconds (1100 ms for the OpenCV elements and 300 ms for OCR). For 'easier' puzzles, the search function finishes much faster (e.g. in around 100 ms), but the image processing times remain fairly constant.

2. Overview of the Classes

Figure 5 shows a classes diagram for the SudokuOCR application, with only the names of the classes shown.

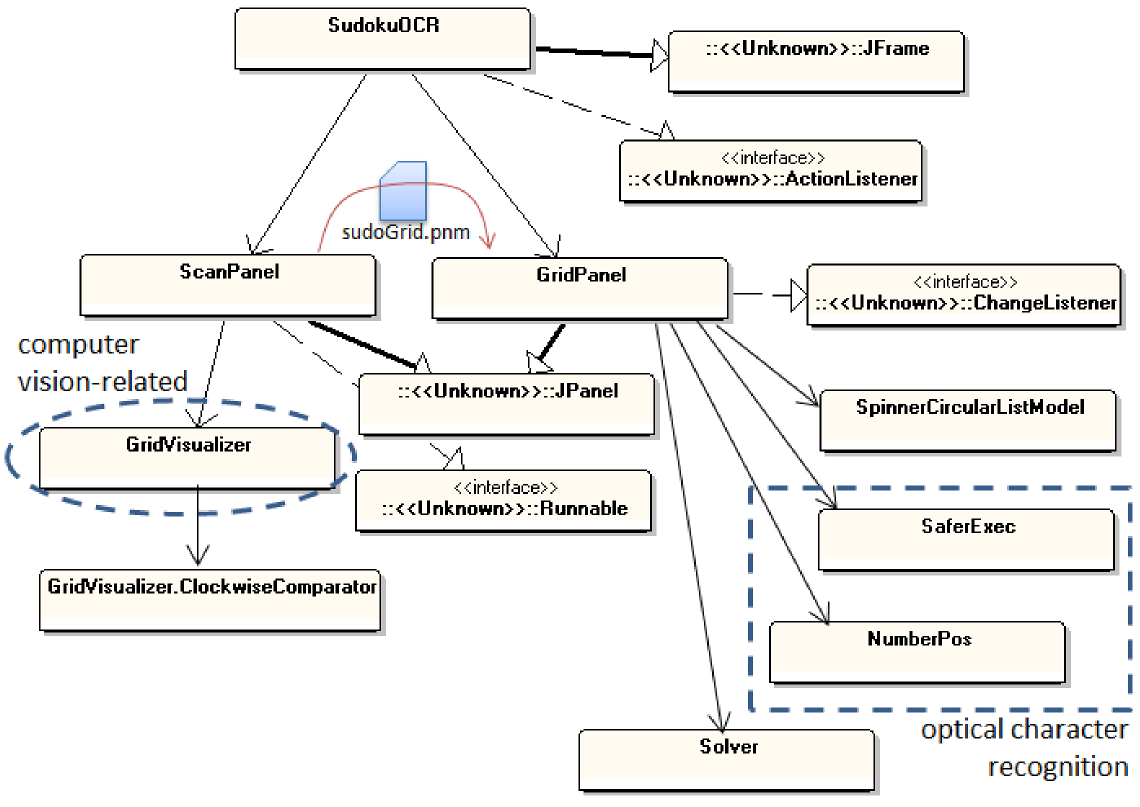


Figure 5. Classes Diagram for SudokuOCR.

The GUI utilizes two JPanel subclasses called ScanPanel and GridPanel, which correspond to the left and right sides of the GUI. ScanPanel displays the current webcam image, and performs computer vision-related tasks such as contour detection and warping by calling the GridVisualizer class. GridPanel manages the grid of spinners, using gocr (the OCR command line tool) to initialize the grid, and calling the search function in the Solver class to complete the puzzle.

I'll only discuss the computer vision techniques employed in GridVisualizer, and the OCR processing called from GridPanel.

One interesting topic I won't cover is the implementation of the recursive backtracking search method in Solver. The code is fully documented, and is based on the "Java Sudoku Solver" post by Bob Carpenter, available at http://www.colloquial.com/games/sudoku/java_sudoku.html. For more background on Sudoku programming, I recommend "Solving Every Sudoku Puzzle" by Peter Norvig (http://norvig.com/sudoku.html). He discusses constraint propagation and backtracking search approaches to the problem.

An important processing link between ScanPanel and GridPanel is a shared image file, called SudoGrid.pnm, an example of which is shown in Figure 6.

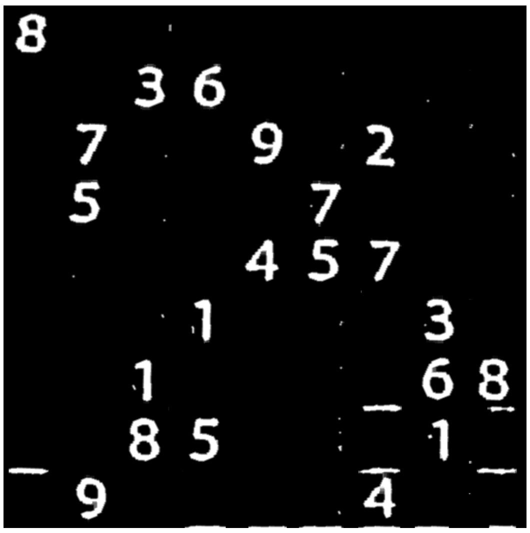


Figure 6. The Processed Sudoku Image.

ScanPanel generates the image using a variety of OpenCV techniques to clean up the original webcam snap. The graphic is passed over to GridPanel via a file because the OCR gocr tool requires a PNM image file as input. gocr is called as an external process using my SaferExec class, and its XML output is converted into a list of NumberPos objects.

3. Visualizing the Sudoku Grid

The OpenCV processing in the GridVisualizer class is carried out in two phases, tied to the "Scan" and "Extract Grid" buttons in the GUI. Pressing "Scan" triggers a call to GridVisualizer.findOutline() which locates the outline of the Sudoku grid. "Extract Grid" is processed by GridVisualizer.extractGrid() which improves character clarity and removes grid lines and noise in the grid image before saving it to the file SudoGrid.pnm.

3.1. Finding the Grid

findOutline() is passed the current webcam image, and returns the grid outline as an array of Point objects. These are used by ScanPanel to draw yellow lines around the image (as in Figure 3).

```
// in the GridVisualizer class
// globals
private CanvasFrame procCanvas;  // a debugging canvas for OpenCV
private IplImage binaryImg;    // copy of grayscale webcam image
private Point[] pts;           // four corners of sudoku grid


public Point[] findOutline(IplImage img)
{
  long startTime = System.currentTimeMillis();
  IplImage im = enhance(img);
  procCanvas.showImage(im);
  binaryImg = cvCloneImage(im);    // save for later

  CvSeq quad = findBiggestQuad(im);
  if (quad == null) {
    // System.out.println("Sudoku grid not found");
    return null;
  }

  pts = clockSort(quad);
  long duration = System.currentTimeMillis() - startTime;
  System.out.println("Grid outline found in " +
                      Math.round(duration) + "ms");
  System.out.println();

  return pts;
}  // end of findOutline()
```

procCanvas is a JavaCV canvas that I use as a 'scratch-pad' for showing the results of various OpenCV operations. The first call to CanvasFrame.showImage() creates a new window showing the image, and subsequent calls update the image. This is a useful debugging tool, which doesn't require any changes/additions to the application's GUI.

enhance() improves the image's quality through smoothing and adaptive thresholding. It returns an inverted binary image, to make the digits in the puzzle stand-out in white against a black background.

```
private IplImage enhance(IplImage img)
{
  // convert to grayscale
  IplImage im = IplImage.create(img.width(), img.height(),
                                            IPL_DEPTH_8U, 1);
  cvCvtColor(img, im, CV_BGR2GRAY);

  // remove image noise
  cvSmooth(im, im, CV_GAUSSIAN, 7, 7, 0, 0);

  /* compensate for glare, and convert image to inverse b&w image
     -- black background, white letters, border, etc.  */
  cvAdaptiveThreshold(im, im, 255,
                CV_ADAPTIVE_THRESH_MEAN_C,
                CV_THRESH_BINARY_INV,
                5, 2);   // block size and offset
  return im;
}  // end of enhance()
```

findBiggestQuad() returns the biggest contour above a minimum size, which must also be convex with NUM_POINTS (4) corners (i.e. it has to be a quadrilateral).

```
// globals
private static final int NUM_POINTS = 4;  // num coords in shape

private static final double SMALLEST_QUAD =  50000.0;
    // ignore contours smaller than SMALLEST_QUAD pixels


private CvSeq findBiggestQuad(IplImage img)
{
  CvMemStorage storage = CvMemStorage.create();
  CvSeq bigQuad = null;

  // generate all the contours in the image as a list
  CvSeq contours = new CvSeq(null);
  cvFindContours(img, storage, contours,
          Loader.sizeof(CvContour.class),
          CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);

  // find the largest quad in the list of contours
  double maxArea = SMALLEST_QUAD;
  while (contours != null && !contours.isNull()) {
    if (contours.elem_size() > 0) {
      CvSeq quad = cvApproxPoly(contours,
         Loader.sizeof(CvContour.class), storage,
         CV_POLY_APPROX_DP, cvContourPerimeter(contours)*0.08, 0);
      CvSeq convexHull = cvConvexHull2(quad, storage,
                                        CV_CLOCKWISE, 1);
      if(convexHull.total() == NUM_POINTS) {
        double area = Math.abs(
                cvContourArea(convexHull, CV_WHOLE_SEQ, 0) );
        if (area > maxArea) {
          maxArea = area;
          bigQuad = convexHull;
        }
      }
    }
    contours = contours.h_next();
  }
  return bigQuad;
}  // end of findBiggestQuad()
```

findBiggestQuad() cycles through the contours, converting each one to a polygon with cvApproxPoly(), and finding its convex hull with cvConvexHull2(). If the hull has four corners, and is bigger than the previous biggest hull, then it's stored in bigQuad.

Back in findOutline(), the call to clockSort() sorts the array of Point objects representing the Sudoku grid. The resulting array has a counter-clockwise order starting from the point nearest the origin, as shown in Figure 7. This order is needed for the subsequent warping of the image.

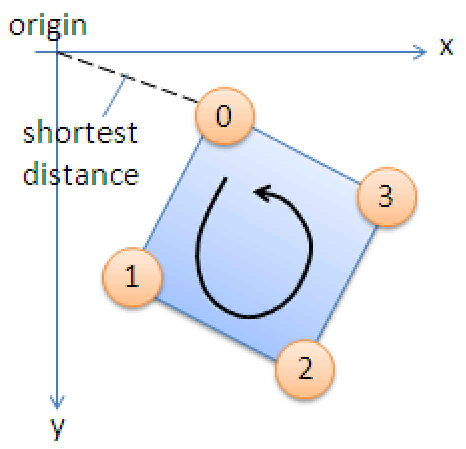


Figure 7. A Clock Sorting for the Grid Points.

The sorting in clockSort() progresses in two steps – first the point nearest the origin is located (it's labeled as point 0 in Figure 7), then the counter-clockwise order is imposed by calling Arrays.sort() with a ClockwiseComparator object.

```
// globals
private static final int NUM_POINTS = 4;  // num of coords


private Point[] clockSort(CvSeq quad)
{
  // copy the CvSeq points into an array
  Point[] pts = new Point[NUM_POINTS];
  CvPoint pt;
  for(int i=0; i < NUM_POINTS; i++) {
    pt = new CvPoint( cvGetSeqElem(quad, i));
    pts[i] = new Point(pt.x(), pt.y());
  }

  // move the point closest to the origin into pts[0]
  int minDist = dist2(pts[0]);
  Point temp;
  for (int i=1; i < pts.length; i++) {
    int d2 = dist2(pts[i]);
    if (d2 < minDist) {
      temp = pts[i];   // swap points
      pts[i] = pts[0];
      pts[0] = temp;
      minDist = d2;
    }
  }

  // sort the array into clockwise order using pts[0]
  ClockwiseComparator pComp = new ClockwiseComparator(pts[0]);
  Arrays.sort(pts, pComp);
  return pts;
 }  // end of clockSort()
```

```
private int dist2(Point pt)
{  return (pt.x*pt.x) + (pt.y*pt.y);  }
```

Although Figure 7 illustrates a counter-clockwise order to the points, this is due to Java's y-axis running down the screen. Taking this into account, the sort order is actually clockwise (still starting from point 0).

The ClockwiseComparator compares two points according to the polar angle they make with point 0, measured relative to the x-axis.

3.2. Improving the Grid Image

When the user is happy that the Sudoku grid has been outlined correctly, he can press the "Extract Grid" button. GridVisualizer.extractGrid() is called to extract the grid from the surrounding webcam image. It also improves character clarity and removes grid lines and noise in the grid image, before saving it to the file SudoGrid.pnm; the end result looks something like Figure 6.

extractGrid() utilizes a number of OpenCV techniques, including perspective image warping (to make the grid square), and flood-filling to erase the vertical grid lines in the puzzle.

```
// globals
private CanvasFrame procCanvas;   // a debugging canvas for OpenCV

private IplImage binaryImg;      // copy of the grayscale webcam image

private Point[] pts;             // four corners of the sudoku grid



public boolean extractGrid(String outFnm)
// extract grid from image, clean it, and save to outFnm
{
  long startTime = System.currentTimeMillis();
  if (pts == null) {
    System.out.println("No grid found");
    return false;
  }

  IplImage squareIm = warp(binaryImg, pts);
          // extract grid image and warp it into a square
  procCanvas.showImage(squareIm);

  IplImage gridIm = cleanGrid(squareIm);

  long duration = System.currentTimeMillis() - startTime;
  System.out.println("Grid extracted in " +
                            Math.round(duration) + "ms");
  System.out.println();
  procCanvas.showImage(gridIm);

  System.out.println("Saving grid image to " + outFnm);
  cvSaveImage(outFnm, gridIm);
  return true;
}  // end of extractGrid()
```

warp() applies perspective warping to the part of the webcam image bordered by the points in pts[] array, reshaping it into a square. The idea is illustrated by Figure 8.

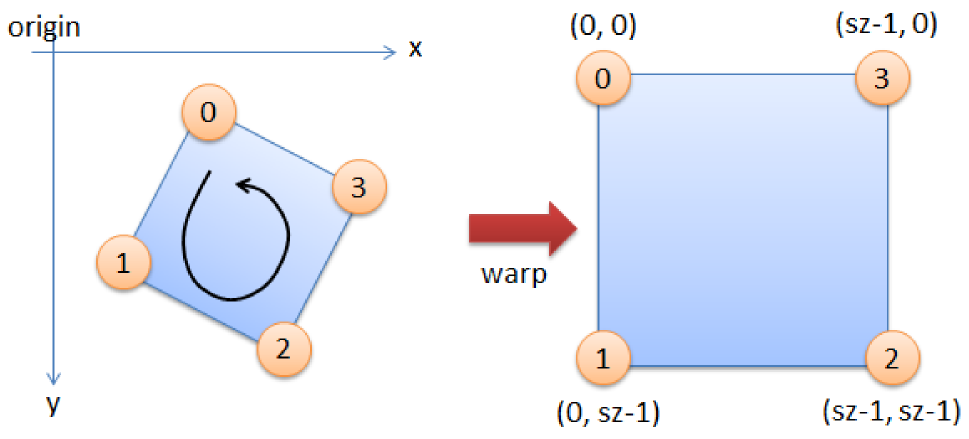


Figure 8. Warping the Sudoku Grid.

The warp() code:

```
// globals
private static final int NUM_POINTS = 4;  // number of coords
private static final int IM_SIZE = 600;  // size of grid square


private IplImage warp(IplImage im, Point[] pts)
{
  CvMat srcPts = CvMat.create(NUM_POINTS, 2);
  for(int i=0; i < NUM_POINTS; i++) {
    srcPts.put(i, 0, pts[i].x);
    srcPts.put(i, 1, pts[i].y);
  }

  CvMat dstPts = CvMat.create(4, 2);
  dstPts.put(0, 0, 0);      // clockwise ordering: point 0 at (0,0)
  dstPts.put(0, 1, 0);

  dstPts.put(1, 0, 0);      // point 1 at (0,sz-1)
  dstPts.put(1, 1, IM_SIZE-1);

  dstPts.put(2, 0, IM_SIZE-1);    // point 2 at (sz-1, sz-1)
  dstPts.put(2, 1, IM_SIZE-1);

  dstPts.put(3, 0, IM_SIZE-1);    // point 3 at (sz-1, 0)
  dstPts.put(3, 1, 0);

  CvMat totalWarp = CvMat.create(3, 3);
  JavaCV.getPerspectiveTransform(
             srcPts.get(), dstPts.get(), totalWarp);

  IplImage warpImg = IplImage.create(IM_SIZE, IM_SIZE,
                                        IPL_DEPTH_8U, 1);
  cvWarpPerspective(im, warpImg, totalWarp);
  return warpImg;
}  // end of warp()
```

A typical image returned by warp() is shown in Figure 9.

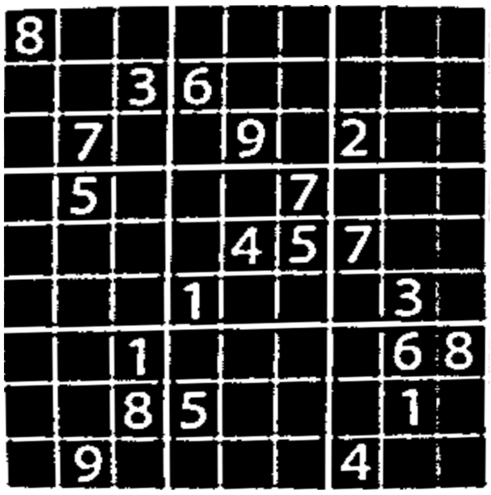


Figure 9. Warped Sudoku Grid.

The grid is now square, with its 9x9 boxes in well-defined locations. However, the picture still includes white borders and vertical and horizontal lines between the numbers.

cleanGrid()'s job is to remove as many of these lines as possible, which is made considerably easier by the fact that they occur in fixed positions inside the Sudoku grid.

cleanGrid() only attempts to remove the vertical lines, for reasons explained shortly:

```
private IplImage cleanGrid(IplImage im)
{
  cvDilate(im, im, null, 2);
          // make white stuff larger (e.g. letters, lines)
  procCanvas.showImage(im);

  highlightVerticals(im);
     // make the vertical grid lines continuous and thicker

  // flood-fill at small intervals across the top of the image
  for(int i=0; i < im.width(); i=i+3)   // across
    cvFloodFill(im, new CvPoint(i, 1), CvScalar.BLACK,
           cvScalarAll(5), cvScalarAll(5), null, 4, null);
     // max lower & upper brightness diffs; 4 is pixel connectivity

  cvErode(im,im, null, 2);   // return white stuff to previous size
  return im;
}  // end of cleanGrid()
```

The initial dilation in cleanGrid() thickens all the white pixels, and hightlightVerticals() further emphasizes the vertical lines by drawing over them with white lines. Then flood-filling in black is carried out at small intervals along the top of the image, which will hopefully fill in all the vertical lines, and any horizontal lines connected to them. A typical outcome is shown in Figure 10.

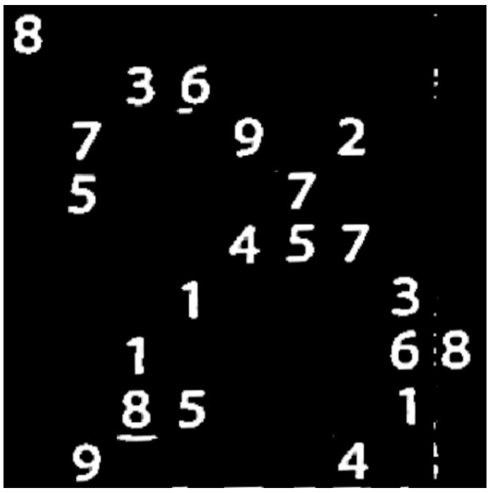


Figure 10. Cleaned Sudoku Grid.

highlightVerticals() assumes that the ten vertical lines in the Sudoku grid are evenly spaced across the image, and draws thick white lines down through those locations.

```
// globals
private static final int IM_SIZE = 600;    // size of grid image
private static final int NUM_BOXES = 9;
private static final int LINE_WIDTH = 6;
                                 // width of lines drawn onto grid


private void highlightVerticals(IplImage im)
{
  double lineStep =  ((double)(IM_SIZE-1))/NUM_BOXES;
  int imHeight = im.height();
  int imWidth = im.width();
  int offset = LINE_WIDTH/2;

  // draw vertical lines
  for(int x=0; x < imWidth; x+= lineStep)
    cvRectangle(im, cvPoint((int)x-offset, 0),
                    cvPoint((int)x+offset, imHeight),
                  CvScalar.WHITE, CV_FILLED, CV_AA, 0);
}  // end of highlightVerticals()
```

The danger with flood-filling is that it may be too effective, and erase some of the Sudoku numbers. This frequently occurred in the first version of cleanGrid() which highlighted both vertical and horizontal lines in the grid. Drawn horizontal lines are closer to the numbers than vertical ones, and so more likely to accidentally intersect with some digits. Flood-filling of the line will then erase those numbers as well.

Another reason for not bothering to highlight horizontal lines is their lack of effect on the OCR command tool, gocr. In tests, gocr ignored horizontal line fragments since they don't correspond to a digit shape. This isn't the case with vertical line fragments which can easily be mistaken for the digit '1'.

A third reason for not flood-filling horizontally (and for not flood-filling vertically in more locations) is computation time. The computer vision processing takes a total of about 1 second, at the upper limit of what a user will accept. Additional flood-filling only makes the processing time longer.

4. OCR with gocr

gocr can recognize without training simply formatted text using sans-serif Latin fonts of 20–60 pixels in height . A noise-less image source is best, but there are various options that allow gocr to deal with poor input.

The tool can be downloaded from http://jocr.sourceforge.net/, and additional documentation and examples are spread around the Web – one benefit of gocr being around for over 10 years. A UNIX-style manpage can be found at http://linux.die.net/man/1/gocr, and slides by Joerg Schulenburg (gocr's developer) giving several examples and details about its internals are at http://www-e.uni-magdeburg.de/jschulen/ocr/linuxtag05/w_lxtg05.pdf.

4.1. Calling gocr

My application calls gocr using an external OS process to invoke it from the command line. Java used to have rather poor support for this kind of programming. Prior to J2SE 1.5, a programmer had to grapple with Runtime.exec(), as described by Michael Daconta in "When Runtime.exec() won't" (http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html). Since v1.5., some of the issues have been fixed with the introduction of the ProcessBuilder class, but issues remain, as detailed by Kyle Cartmell in "Five Common java.lang.Process Pitfalls" (http://kylecartmell.com/?p=9) and Alvin Alexander in "Java exec - execute system processes with Java ProcessBuilder and Process" (Parts 1-3 at http://www.devdaily.com/java/java-exec-processbuilder-process-1).

I 'borrowed' some of their ideas, and added a few of my own to come up with a SaferExec class, which can be downloaded from http://fivedots.coe.psu.ac.th/~ad/SaferExec/. SaferExec tokenizes the OS command string before trying to execute it, merges its output and error streams to simplify IO concerns, and can interrupt blocked commands. The output from the process is read byte-by-byte, to make IO easier, and care is taken to free up resources at the end of the command's execution.

Probably the most comprehensive solution for executing external processes from within Java is Jakarta Commons Exec (http://commons.apache.org/exec/). The website includes a tutorial, FAQ, and source and binary downloads. Commons Exec exposes a lot more of the underlying Java Process to the user than SaferExec, including the IO streams and a timer interface. This allows greater usage flexibility, at the expense of more complex coding.

I decided to utilize my SaferExec class in this application, which is called from applyOCR() in the GridPanel class. SaferExec invokes a Windows batch file, called xmlGocr.bat, which is listed below:

```
@echo off
echo gocr of %1, saving XML to %2

gocr049.exe -C "123456789" -a 30 -f XML -i %1 -o %2
```

The batch file expects two arguments: %1 should be the input image file, while %2 is the output file for the OCR results.

The gocr "-C" option restricts the character set being recognized; in my case, to the digits. A small recognition set increases the chance that a character will be recognized correctly. The "-a" option sets the recognition certainty level to a low 30 (the default is 95, with a maximum of 100). This lets gocr match a character with less certainty, which overcomes some problems with noisy input. The "-f" option changes the output format to XML.

Two other options that I played with, but finally didn't utilize, were "-d" and "-s". "-d" sets a 'dust' size, which specify the size of pixel clusters ignored as noise at recognition time. "-s" sets a space width, which gives gocr hints on the space between characters (or words) in the input.

The code for applyOCR() in GridPanel is:

```
private void applyOCR(String inFnm, String xmlFnm)
{
  System.out.println("Applying OCR to " + inFnm +
                                      " using gocr...");
  System.out.println("Saving XML results to " + xmlFnm);
  long startTime = System.currentTimeMillis();

  SaferExec se = new SaferExec(10);   // timeout is 10 secs
  se.exec("xmlGocr.bat", inFnm, xmlFnm);

  long duration = System.currentTimeMillis() - startTime;
  System.out.println("OCR processing took " +
                              Math.round(duration) + "ms");
  System.out.println();
}  // end of applyOCR()
```

applyOCR() is supplied with arguments for the image input file and the XML output file, which will be "sudoGrid.pnm" and "gocrOut.xml" respectively. The name of the batch file, "xmlGocr.bat", is fixed in the call to SaferExec.exec(). If the call doesn't finish within 10 seconds then it is aborted.

This type of programming can be criticized for limiting the portability of the SudokuOCR application, although gocr is supported on Windows and Linux. One advantage of using a batch file is that the scripting details are hidden from the Java code. Another is testing, since the batch file can be executed stand-alone without the surrounding application.

4.2. Processing gocr's Output

gocr's XML output looks something like:

```
<page x="0" y="0" dx="0" dy="0">
<block x="0" y="0" dx="0" dy="0">
<line x="600" y="0" dx="-599" dy="0" value="0">
  <space x="280" y="273" dx="39" dy="48" />
  <box x="280" y="273" dx="39" dy="48" value="4"
                          numac="1" weights="98" />
  <box x="351" y="272" dx="35" dy="47" value="_" />
  <box x="420" y="272" dx="34" dy="47" value="7"
                          numac="2" weights="98,92" achars="1" />
  <space x="455" y="383" dx="79" dy="3" />
  <box x="535" y="383" dx="3" dy="3" value="_" />
</line>

<line x="13" y="12" dx="525" dy="61" value="1">
  <space x="13" y="12" dx="36" dy="49" />
  <box x="13" y="12" dx="36" dy="49" value="8"
                          numac="1" weights="97" />
  <space x="50" y="11" dx="484" dy="3" />
  <box x="535" y="11" dx="3" dy="3" value="_" />
</line>

<line x="153" y="77" dx="387" dy="58" value="2">
  <space x="153" y="76" dx="35" dy="49" />
  <box x="153" y="76" dx="35" dy="49" value="3"
                         numac="1" weights="100" />
  <space x="189" y="75" dx="31" dy="49" />
  <box x="221" y="75" dx="35" dy="49" value="6"
                         numac="1" weights="100" />
  <space x="257" y="96" dx="277" dy="5" />
  <box x="535" y="96" dx="5" dy="5" value="_" />
  <box x="535" y="79" dx="5" dy="37" value="_" />
</line>
   // 5 more <line> tags not shown ...
   //  :
</block>
</page>
```

The output is divided into lines using the <line>…<//line> tag, with each recognized symbol in its own <box> tag, and spaces inside <space> tags. All tags include x and y attributes which give their location in the image, and dx and dy values for the image area used by the recognized character or line. The x, y, dx, and dy attributes are illustrated in Figure 11, and become important later for determining the position of a number inside the Sudoku grid.

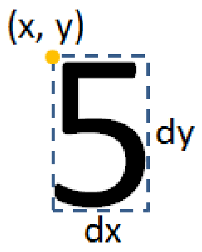


Figure 11. gocr Positional Attributes for a Character.

A <box> tag's recognized character is stored in its value attribute.

A closer look at the XML output given above reveals that line 0 contains four box tags with the values '4', '_', '7', and '_'. Line 1 contains box values '8' and '_', and line 2 reports box values of '3', '6', '_', and '_'.

If you look back at the grid image in Figure 10, the XML line 0 corresponds to the fifth line of the puzzle, line 1 is for the first line, and line 2 for the second line. In other words, the <line> tag ordering need not follow the textual ordering of the lines in the input image.

In fact, a <line> tag needn't correspond to a line in the textual image at all; for example, the puzzle in Figure 10 contains nine lines, but gocr outputs eight <line> tags; one of the <line>s contains characters from two lines in the puzzle.

If a <box> tag contains a '_' value, then gocr was unable to recognize the symbol. All three lines in the XML from above contain '_' boxes, which correspond to noise in the input image. In addition, the order of the boxes in a <line> tag needn't match the left-to-right order of the characters in the image.

The lack of order in the XML isn't too serious, since all the recognized characters come with (x, y) coordinates. These can be combined with knowledge about the image size (a square of 600x600 pixels) and the format of a Sudoku grid (9x9 boxes) to map every character to a grid location.

4.3. Converting XML to NumberPos Objects

GridPanel uses the readNumbersXML() method to read in the gocr XML results from a file, and creates a list of NumberPos objects based on the XML's <box> values. The method starts by building an XPath expression to recognize the <box> tag, then collects all the matching attributes in an XPath NodeList object.

```
// in GridPanel
// globals
private static final String XPATH_EXPR = "//box";


private ArrayList<NumberPos> readNumbersXML(String xmlFnm)
{
  System.out.println("Reading in XML results from " + xmlFnm);
  ArrayList<NumberPos> sudoNumbers = new ArrayList<NumberPos>();

  try {
    DocumentBuilderFactory domFactory =
             DocumentBuilderFactory.newInstance();
    domFactory.setNamespaceAware(true);
    DocumentBuilder builder = domFactory.newDocumentBuilder();
    Document doc = builder.parse(xmlFnm);

    XPathFactory factory = XPathFactory.newInstance();
    XPath xpath = factory.newXPath();
    XPathExpression expr  = xpath.compile(XPATH_EXPR);
          /* all the "box" nodes in the XML are collected,
             since each one represents an OCR result */
    Object result = expr.evaluate(doc, XPathConstants.NODESET);
    NodeList nodes = (NodeList) result;
    NumberPos np;
    for (int i = 0; i < nodes.getLength(); i++) {
      np = new NumberPos( nodes.item(i).getAttributes() );
              // each NumberPos object is built from a box node
      if (np.isValid())
        sudoNumbers.add(np);
    }
  }
  catch(Exception e)
  {  System.out.println(e);  }

  return sudoNumbers;
}  // end of readNumbersXML()
```

A loop iterates through the nodes list, creating a NumberPos object for each <box>. These objects are stored in an ArrayList and later used to initialize the GUI spinners grid with numbers.

4.4. Creating a NumberPos Object

The main tasks of a NumberPos object are to convert a <box>'s value into an integer, and to map a <box>'s x, y, dx, and dy attributes into indices in the Sudoku grid. A grid with x- and y- indices is shown in Figure 12.

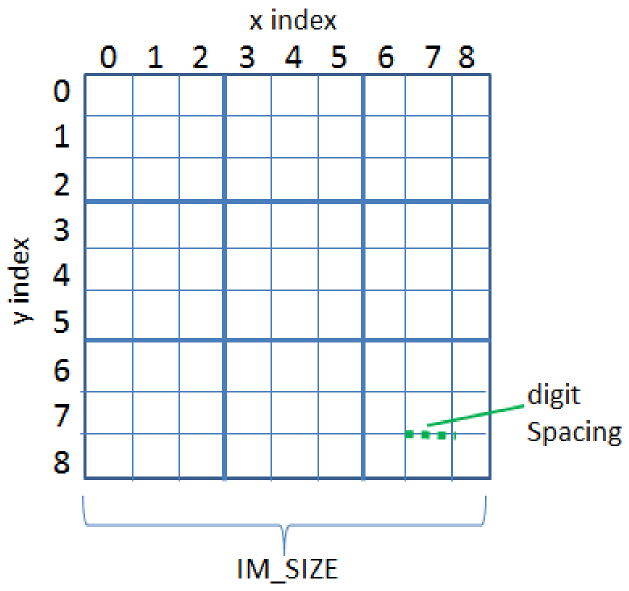


Figure 12. A Sudoku Grid with Indices.

The mapping from image coordinates to indices is fairly simple because we know the size of the image (600 x 600 pixels), and the box layout of the puzzle (9x9). This information allows us to calculate the dimensions of an individual box in the image (called digitSpacing in Figure 12). If we assume that numbers are centered in each box, then the distance between adjacent digits will also be digitSpacing (see Figure 13).

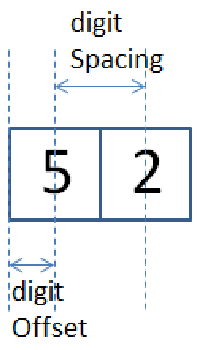


Figure 13. The Spacing Between Grid Digits.

The offset of a digit from the side of the box will be digitSpacing/2 (called digitOffset in Figure 13).

The NumberPos constructor utilizes these image-to-grid correspondences to generate grid indices from the <box> attributes it is passed.

```
public class NumberPos
{
  // 600x600 grid of 9x9 squares
  private static final int IM_SIZE = 600;  // size of image
  private static final int GRID_SIZE = 9;

  private boolean isValid = true;
  private int value;
  private int xCenter, yCenter;
                    // image coords for center of value
  private int xIdx, yIdx;
                   // sudoku indicies for this position


  public NumberPos(NamedNodeMap nodeMap)
  {
     value = getNodeIntValue(nodeMap, "value");
     if ((value < 1) || (value > 9))
       isValid = false;

     int x = getNodeIntValue(nodeMap, "x");     // location of box
     int y = getNodeIntValue(nodeMap, "y");
     int width = getNodeIntValue(nodeMap, "dx");   // size of box
     int height = getNodeIntValue(nodeMap, "dy");

     xCenter = x + width/2;   // center of value in the box
     yCenter = y + height/2;

     double digitSpacing = ((double) IM_SIZE)/GRID_SIZE;
                              // space between digits
     double digitOffset = digitSpacing/2.0;
                            // space from edge to first digit

     xIdx = (int)Math.round((xCenter-digitOffset)/digitSpacing);
     yIdx = (int)Math.round((yCenter-digitOffset)/digitSpacing);
  }  // end of NumberPos()
  private int getNodeIntValue(NamedNodeMap nodeMap, String itemName)
  // extract value of itemName attribute, returning it as an int
  {
    String valStr = null;
    try {
      valStr = nodeMap.getNamedItem(itemName).getNodeValue();
      return Integer.parseInt(valStr);
    }
    catch(DOMException e)
    {  System.out.println("Parsing error");
       isValid = false;
    }
    catch (NumberFormatException ex){
      isValid = false;   // occurs when value is '_'
    }
    return -1;
  }  // end of getNodeIntValue()


  // various getter methods (not shown) ...
  //  :

}  // end of NumberPos class
```

The calculated grid x- and y- indicies are stored in the xIdx and yIdx globals, and are accessible to GridPanel via getter methods. For example, GridPanel.fillTable() uses each NumberPos object to update a table[][] array which stores the Sudoku grid's values.

```
// in GridPanel class
// globals
private static final int GRID_SIZE = 9;

private int[][] table;
    // each box contains a Sudoku number (or 0 meaning no value)


private void fillTable(ArrayList<NumberPos> sudoNumbers)
{
  for (int row = 0; row < GRID_SIZE; row++)
    for (int col = 0; col < GRID_SIZE; col++)
      table[row][col] = 0;      // reset table

  for(NumberPos np : sudoNumbers)
    table[np.getYIndex()][np.getXIndex()] = np.getValue();

}  // end of fillTable()
```

5. Other Approaches to OCR and Sudoku

In this chapter I've concentrated on explaining the computer vision (i.e. OpenCV) operations and OCR (i.e. gocr) capabilities of my application. The OpenCV features include using contours to find the quadrilateral outline of the Sudoku image, perspective warping to reshape the image into a square, and flood-filling to remove Sudoku grid lines.

I benefited from other people's work on OCR and Sudoku, including the "SuDoKu Grabber with OpenCV" blog postings by Utkarsh Sinha (http://www.aishack.in/2010/08/sudoku-grabber-with-opencv/) and a blog entry on how the iPhone Sudoku Grab app was implemented by Chris Greening (http://sudokugrab.blogspot.com/2009/07/how-does-it-all-work.html).

One interesting difference is that they use Hough transforms to identify the location of grid lines, whereas I assume they're at set positions inside the image. Neither of them use gocr for OCR; Sinha utilizes his own digit recognition class based on a trained k-nearest neighbour algorithm (http://www.aishack.in/2010/10/k-nearest-neighbors-in-opencv/), while Greening employs a trained neural network.

The CodeProject article, "Realtime Webcam Sudoku Solver", by Bojan Banko (http://www.codeproject.com/Articles/238114/Realtime-Webcam-Sudoku-Solver) goes into excellent detail about another approach that uses similar computer vision techniques as the other articles, but deals with the OCR part by training zone density features for each of the digits. It also includes a section on implementing the Sudoku solver.