User Manual (v0.1 - beta)

MarkShark

The fast and accurate open-source bubble sheet scanner

Table of Contents

Table of Contents 1

MarkShark 3

Installing MarkShark 3

MarkShark Overview 4

Quick Start 5

- The MarkShark graphical user interface 5
- Using MarkShark via the command line: 6

Making Your Master BubbleSheet 8

- Rule 1: Bubbles should be evenly spaced in a grid pattern 8
- Rule 2: Make your bubbles medium gray instead of black and keep the letters inside the bubble small (and gray) 8
- Tip: Print your BubbleSheets on white paper. 9
- Tip: Add ArUco markers to your BubbleSheet. 9

Making a BubbleMap File (bubblemap.yaml) 11

- What is YAML format? 11
- A sample bubblemap.yaml file 14

Viewing how your bubblemap.yaml aligns with your master bubblesheet 16

Making Your Test Key 17

Multiple Test Versions 18

Scanning Your Student Sheets 19

Align Your Scanned Student Sheets to the Master Bubblesheet Prior to Scoring 20

- Aligning scans using the graphical user interface 21

Scoring with MarkShark 24

- Scoring via the graphical user interface 24
- Scoring via the command line interface 25
- How the scoring software works 26
- Quick troubleshooting 27

Assessing the quality of your exam 28

The stats module processes the CSV output from grade, computes exam- and item-level metrics, and generates CSV summaries and plots. 28
1. Main CSV – results_with_item_stats.csv 28
2. Exam stats – exam_stats.csv 28
3. Item report – item_analysis.csv 28
4. Plots – folder item_plots/ 28
Interpretation of Statistics 28
Metric 28
Description 28
Ideal / Notes 28
Tips on designing good MCQ tests. 30
What makes a good MC question? 31
Troubleshooting: Under the Hood 33
Understanding how the MarkShark scores tests. ..... **Error! Bookmark not defined.**
Understanding how MarkShark aligns scans 33

MarkShark

MarkShark is an open-source Optical Mark Recognition (OMR) toolkit for educators and researchers. It aligns, scores, and analyzes bubble-sheet exams quickly and reproducibly — without proprietary software. It is written in Python, available on GitHub, and relies on common well-documented and widely used Python modules like OpenCV, numpy, and Streamlit.

MarkShark can be used via the command line or through a graphical interface.

Installing MarkShark

MarkShark requires Python 3.9 or higher.

Installation is simple:

- pip install markshark

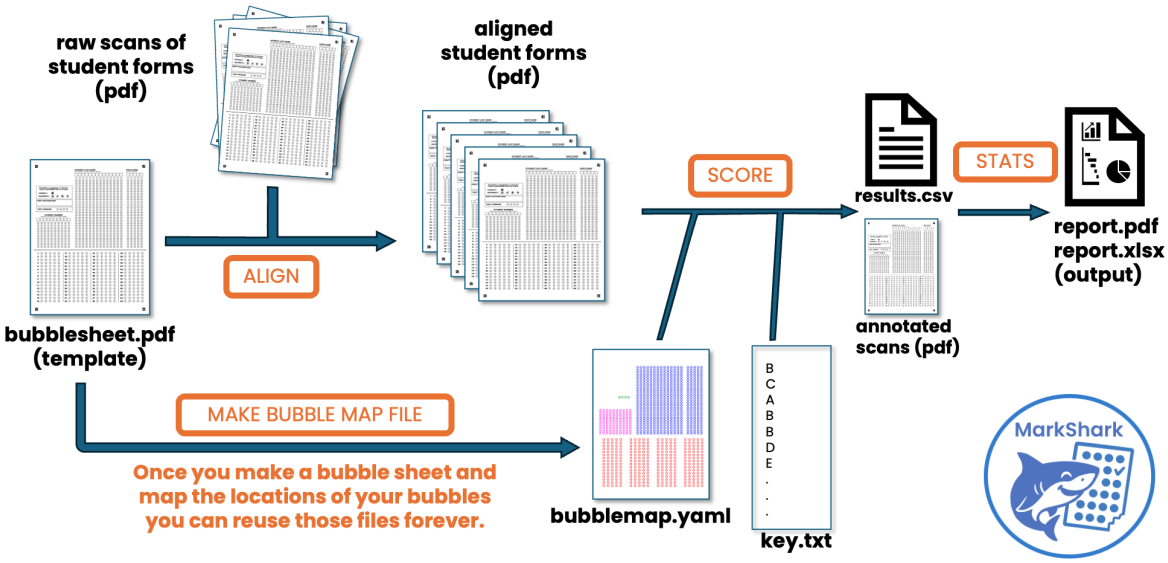
Or, for the latest development version:

- pip install git+https://github.com/navarrew/markshark.git

To verify installation:

- markshark --help

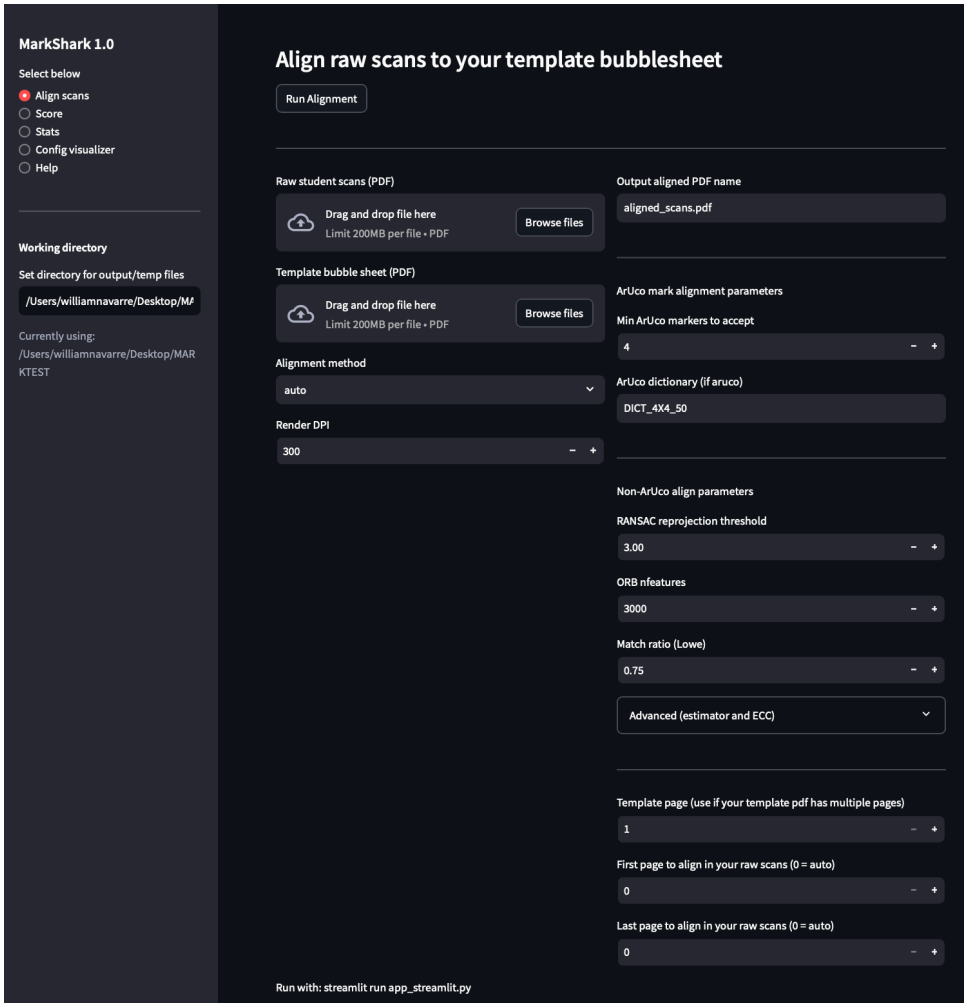
MarkShark Overview



1. **Design your master BubbleSheet** and make a bubble mapping file that tells MarkShark where the bubbles are located and what they represent. *We have several BubbleSheet templates with corresponding mapping files that you can use straight away or modify for your purposes!*
2. **Scan:** After your students have completed their tests/surveys you can scan them using a standard desktop scanner or a photocopier with scanning capabilities.
3. **Align your scans:** Because scans are often a little askew you should align them to your master BubbleSheet. That way the mapping file can accurately find the bubbles.
4. **Score!** You provide a key of correct answers as a text file. Scoring hundreds of sheets only takes a minute or two. You get a spreadsheet of student scores with names, IDs, and their responses to each question. You also get a pdf with each BubbleSheet where answers are circled as correct or incorrect.
5. **Get stats:** MarkShark can generate a more detailed report highlighting questions that are too hard, too easy, or confusing.

Quick Start

The MarkShark graphical user interface



Using MarkShark via the command line:

Alignment of scanned bubble sheets to the bubble sheet template pdf.

```
markshark align raw_scans.pdf --template template.pdf -o aligned_scans.pdf
```

Align only a subset of pages (1-based, inclusive)

```
markshark align raw_scans.pdf --template template.pdf \
  --first-page 1 --last-page 50 \
  -o aligned_scans_p1-50.pdf
```

Force feature-based alignment (useful if ArUco is not present)

```
markshark align raw_scans.pdf --template template.pdf \
  --align-method feature \
  -o aligned_scans.pdf
```

Overlay config geometry on the first page of a PDF and write an image

```
markshark visualize template.pdf -c bubblemap.yaml -o config_overlay.png
```

Minimal scoring (writes results.csv by default)

```
markshark score aligned_scans.pdf -c bubblemap.yaml
```

Score with an answer key, and write annotated pages plus an annotated PDF

```
markshark score aligned_scans.pdf -c bubblemap.yaml -k key.txt \
  -o results.csv \
  --out-annotated-dir annotated_pages \
  --out-pdf scored_scans.pdf
```

If you want to disable annotated PDF output explicitly

```
markshark score aligned_scans.pdf -c bubblemap.yaml \
  --out-pdf ""
```

Getting Help

```
markshark --help
markshark align --help
markshark score --help
markshark stats --help

markshark-gui
```

Launch GUI on default port 8501

```
markshark gui
```

 # Pick a port and do not auto-open a browser window

```
markshark gui --port 8502 --no-open-browser
```

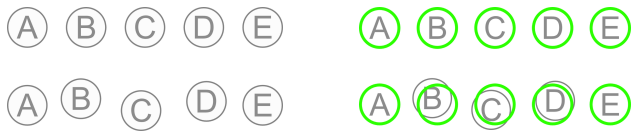
Making Your Master BubbleSheet

Putting a bit of time into making a good bubble sheet will go a long way towards making your life easier later. Fortunately, we provide several master templates ready for you to use right away or modify for your own purposes.

Rule 1: Bubbles should be evenly spaced in a grid pattern

The bubblemaps used by MarkShark are made with the expectation that all bubbles are arranged into a perfect grid – with bubbles placed in evenly spaced columns and rows. If bubbles are unevenly spaced, it will make it harder for the software to identify the bubbles and correctly guess how much of the bubble has been filled in.

An exaggerated version of good vs. bad bubble alignment and spacing is shown below. The Green bubbles are where MarkShark 'expects' the bubbles to be. You can see that the top row is well aligned and evenly spaced. Hence the green circles overlap. The bubbles in the bottom row are misaligned and MarkShark will miss a lot of the area in bubbles B, C, and D as a result.



If you are making your own bubble sheet you should use the 'align' functions in your art program (highlight all bubbles, align them using 'align vertically'). To space them evenly apart use the 'distribute' functions in your graphics program, often found in the same menu as align, and choose 'distribute horizontally'.

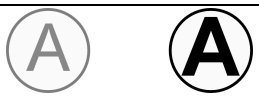
Or…just copy and paste the bubble templates we provide, which are already evenly spaced. You can crop the bubble area to include fewer bubbles.

Rule 2: Make your bubbles medium gray instead of black and keep the letters inside the bubble small (and gray).

During the scoring process, MarkShark tries to distinguish between bubbles that are filled and those that are not…and students aren't always good at filling in

bubbles completely. Therefore, you want to help maximize the differences between blank bubbles and those that students have marked in.

Bubbles with very thick dark text can make it difficult for MarkShark to distinguish between poorly filled bubbles and unfilled bubbles. The letters inside the bubble should take up less than 40% of the space in the bubble. ***So do yourself a favor and make the letters inside your bubbles as small and light as possible while still being readable.***



MarkShark will score the bubble on the right as about 50% 'filled in' even though nobody has written any pencil marks in it yet. To maximize the differences between filled and unfilled bubbles, keep the text inside your bubbles as thin and small as possible while maintaining readability for students.

Tip: Print your BubbleSheets on white paper.

Colored paper will often give a light gray background when the image is converted to a black/white (grayscale) image for processing by MarkShark. This can lessen the differences between the background and the pencil…especially if the student uses a light touch or a hard-leaded pencil.

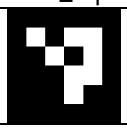
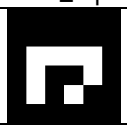
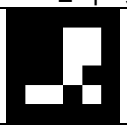
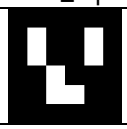
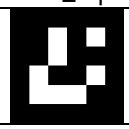
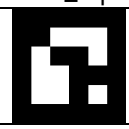
Tip: Add ArUco markers to your BubbleSheet.

If make your own custom bubble sheets you should seriously consider adding unique ArUco tags to each of the corners. This is because scanned bubble sheet images, even when using a high-quality scanner with a good document feeder, are often slightly misaligned with one another and bubbles may fail to align with the zones given in your bubblemap file.

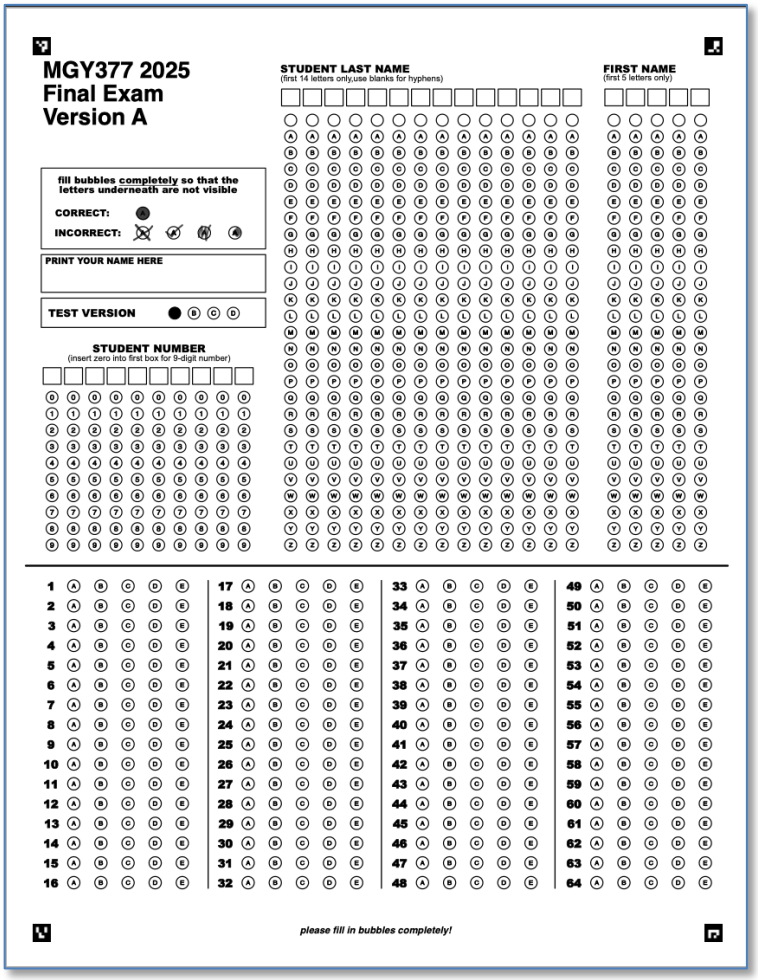
ArUco markers are high-contrast, square fiducial tags that encode unique IDs in a black-and-white pattern, allowing computer vision systems (like Python's OpenCV package, which we use) to precisely detect their position, orientation, and scale in an image.

In image alignment, ArUco markers serve as reliable reference points that can be robustly recognized even under perspective distortion, varying lighting, or partial

occlusion. By placing known ArUco markers at fixed positions on *your* bubble sheet, alignment algorithms can compute the exact geometric transformation—translation, rotation, scaling, and warping—needed to register a scanned or photographed image to the original reference coordinate system. This makes them ideal for tasks like aligning scanned documents.

The first six tags of the ArUco 4x4-50 set					
aruco_0.png	aruco_1 .png	aruco_2.png	aruco_3.png	aruco_4.png	aruco_5.png
					

The BubbleSheet below has ArUco markers at the four corners.



Making a BubbleMap File (bubblemap.yaml)

You only need to do this once for any master bubblesheet you create.

The bubblemap file is the 'map' that tells MarkShark where it can find the bubbles corresponding to the student's first/last name, student ID, test version, and question/answer rows.

What is YAML format?

The bubblemap file is written in 'YAML' format. You can write or modify it in any text editor. YAML (short for "YAML Ain't Markup Language") is a human-readable data format used to store structured information in a simple, text-based form. YAML supports hierarchical data (like dictionaries and lists) and is an easy-to-read alternative to similar data file formats like JSON or XML.

In a bubblemap.yaml file, the various regions of the BubbleSheet (last name, first name, student ID, etc.) are described via blocks of text.

Lines of text starting with # are not read by MarkShark and are used for human readable instructions and comments.

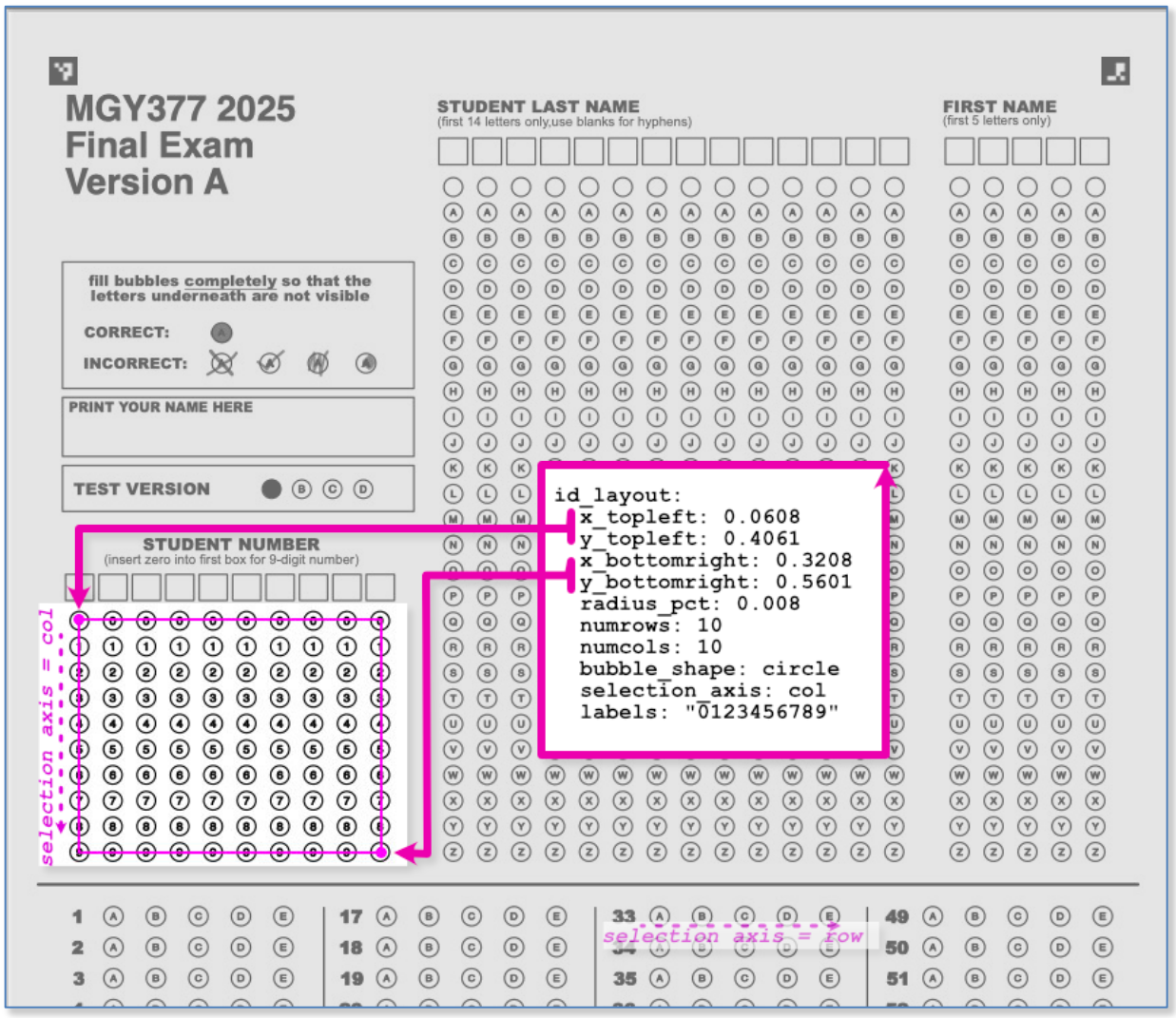
Each block of information starts with an unindented header that tell MarkShark which 'type' of information is being mapped. The indented lines underneath each header indicate the coordinates of that region on the master BubbleSheet template, which axis the bubbles are arranged along (columns or rows), and how many bubbles there are, the bubble shape, and the diameters of the bubbles, and the labels in the bubbles.

These information types include:

- **last_name_layout: CHANGE TO REGION**
- **first_name_layout:**
- **id_layout:**
- **version_layout**
- **answer_layouts**
- **total_questions (maybe put this on top?)**

Underneath each of these headers are the parameters that map the area. Each of these information lines is indented by 2 spaces.

- **x_topleft: 0.###** - gives the x-axis center of the upper-leftmost bubble in that region. This is given as a fractional percentage of the page *width*.
    - **Example:** x_topleft: 0.25 means the center of the top-left bubble sits 25% in from the left edge of the page (*on an 8.5 x 11 inch piece of paper that would mean the center of the bubble sits 2.125 inches in from the left edge*).
- **y_topleft: 0.###** - gives the y-axis center of the upper-leftmost bubble in that region. This is given as a fractional percentage of the page *height*.
    - **Example:** *y_topleft: 0.12* maps the center of the top-left bubble 12% of the way down from the top edge of the page (*on an 8.5 x 11 inch piece of paper that would mean the center of the bubble sits 1.32 inches down from the top edge*).
- **x_bottomright: 0.###** - gives the x-axis center of the bottom-rightmost bubble in that region.
- **y_topleft: 0.###** - gives the x-axis center of the bottom-rightmost bubble in that region.
- **radius_pct: 0.###** The radius of the bubble on the page.
- **numrows: ## -** number of bubble rows in that region
- **numcols: ## -** number of bubble columns in that region
- **bubble_shape:** shape of the bubble (circle or square)
- **selection_axis:** which direction the information is read (col or row)
- **labels:** what is 'in' the bubbles…could be blank (space), digits, or letters



An example showing how the 'id_layout' section of the bubblemap.yaml file corresponds to its master BubbleSheet. The region containing the student ID bubbles is formed by a rectangle with a top-left corner at the x,y coordinates of (0.0608, 0.4061) and a bottom right corner of (0.3208, 0.5601). These coordinates mark the exact centers, not the edges, of the top-left and bottom-right corner bubbles.

The zone is 10 bubbles wide by ten bubbles high (numrows = 10 and numcols = 10). The bubbles are shaped in a circle with a radius of 0.008. The 10 bubbles in each column are give labels - 0123456789 (0 through 9).

Each digit of the student ID in this zone is read from the bubbles in an up-down direction, not left-right, so selection_axis: was set to 'col' and not 'row'.

A sample bubblemap.yaml file

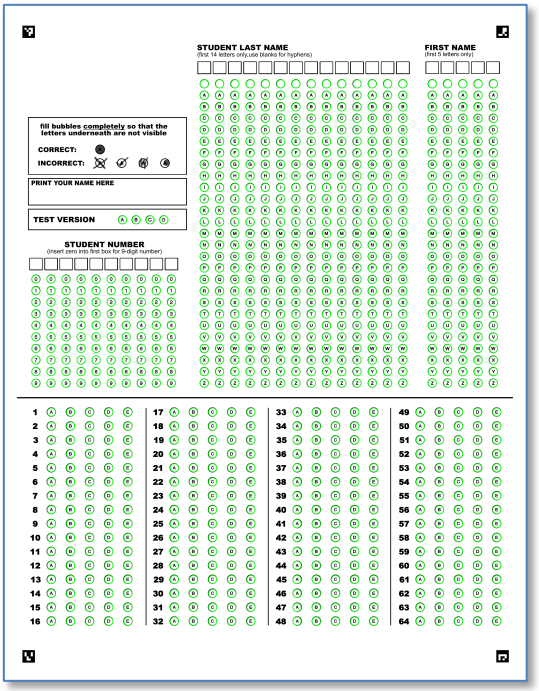
```
# samplebubblemap.yaml
# coordinates are given in percentages
# from top (y values) and left (x values) of page.
# x_topleft: 0.5 -> means a point 50% (middle) of the page from left to right

# ----------------------------
# Name blocks (select one row per column)
# ----------------------------
last_name_layout:
  x_topleft: 0.3837        #the x-center of the top-left bubble in the region
  y_topleft: 0.1167        #the y-center of the top-left bubble in the region
  x_bottomright: 0.7641    #the x-center of the bottom-right bubble in the region
  y_bottomright: 0.5601    #the y-center of the bottom-right bubble in the region
  radius_pct: 0.008        #the diameter of the bubbles in the region
  numrows: 27              #number of bubble rows (26 letters and an empty space)
  numcols: 14              #number of bubble columns
  bubble_shape: circle     #shape of the bubble
  selection_axis: col      #the axis along which information is read (up/down here)
  labels: " ABCDEFGHIJKLMNOPQRSTUVWXYZ" #note the first character is a space

first_name_layout:
  x_topleft: 0.8205
  y_topleft: 0.1167
  x_bottomright: 0.9375
  y_bottomright: 0.5601
  radius_pct: 0.008
  numrows: 27
  numcols: 5
  bubble_shape: circle
  selection_axis: col
  labels: " ABCDEFGHIJKLMNOPQRSTUVWXYZ"

# ----------------------------
# Student ID (select one row per column)
# ----------------------------
id_layout:
  x_topleft: 0.0608
  y_topleft: 0.4061
  x_bottomright: 0.3208
  y_bottomright: 0.5601
  radius_pct: 0.008
  numrows: 10
  numcols: 10
  bubble_shape: circle
  selection_axis: col
  labels: "0123456789"

# ----------------------------
# Test version (select one column in a single row)
# ----------------------------
version_layout:
  x_topleft: 0.2271
  y_topleft: 0.3182
  x_bottomright: 0.3063
  y_bottomright: 0.3185
```

```
  radius_pct: 0.008
  numrows: 1
  numcols: 4
  bubble_shape: circle
  selection_axis: row
  labels: "ABCD"

# ----------------------------
# Answer blocks (select one column per row)
# ----------------------------
answer_layouts:
  - x_topleft: 0.0906
    y_topleft: 0.6030
    x_bottomright: 0.2373
    y_bottomright: 0.9127
    radius_pct: 0.008
    numrows: 16
    numcols: 5
    bubble_shape: circle
    selection_axis: row
    labels: "ABCDE"

  - x_topleft: 0.3262
    y_topleft: 0.6030
    x_bottomright: 0.4728
    y_bottomright: 0.9127
    radius_pct: 0.008
    numrows: 16
    numcols: 5
    bubble_shape: circle
    selection_axis: row
    labels: "ABCDE"

  - x_topleft: 0.5612
    y_topleft: 0.6030
    x_bottomright: 0.7084
    y_bottomright: 0.9127
    radius_pct: 0.008
    numrows: 16
    numcols: 5
    bubble_shape: circle
    selection_axis: row
    labels: "ABCDE"

  - x_topleft: 0.7973
    y_topleft: 0.6030
    x_bottomright: 0.9439
    y_bottomright: 0.9127
    radius_pct: 0.008
    numrows: 16
    numcols: 5
    bubble_shape: circle
    selection_axis: row
    labels: "ABCDE"

total_questions: 64
```

Viewing how your bubblemap.yaml aligns with your master bubblesheet

You will need to spend some time up-front ensuring that your bubblemap faithfully aligns with your master bubblesheet.

Use the ‘zone visualizer’ program in the GUI to get images of your bubblemap overlaid on your master bubblesheet template.



Making Your Test Key

The test key is a simple text file. You can name it whatever you want.

You can put the correct answers for your test simply with one character per line. MarkShark automatically assumes that the letter on the first line is the answer to question 1, the letter on the second line is the answer to question 2…

D
C
D
A
B
.
.
.

Or you can put all the answers in a single line separated by commas. Again, answer for question #1 is the first letter in the list…then proceed from there.

D,C,D,A,B…

Multiple Test Versions

If you have multiple versions of the same test with different answers you will need different keys, one for each version.

Currently MarkShark does not support putting all your scans together into a single PDF and scoring based on version. You must simply score each version separately with its own key and manually put the marks together.

For now, that means manually separating your version A tests from version B, etc… scanning them into separate pdf files (raw_scans_versionA.pdf…). Assuming the master template bubblesheet is the same, you can align each of them with the same master bubblesheet. Then score each version of them simply by changing the key you upload – to avoid confusing yourself make sure your output pdfs are named (example: aligned_scans_versionA.pdf).

A future implementation will allow you to simply put all scanned tests into a single ‘raw’ pdf. The scans would be aligned and then, during scoring, be separated into batches based on the version code filled in by the student on the BubbleSheet. Each student test would be scored with the appropriate key matching the bubble they filled on their test.

Scanning Your Student Sheets

Garbage in = garbage out.

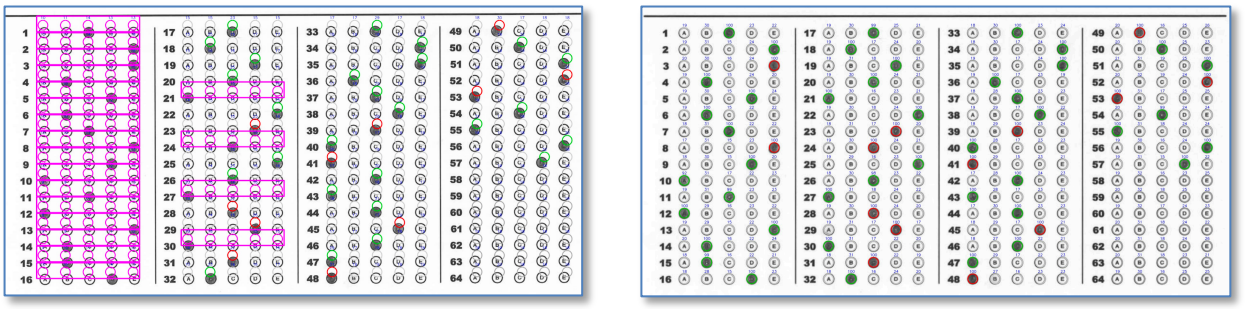
Use a high-quality scanner with a good document feeder. Cheap scanners may not feed paper through at a constant rate, resulting in notable stretching or warping in the scan. This may make the bubbles align poorly with the map of bubble locations in the bubblemap.yaml file. We try to correct for this in the alignment process but if the warping or stretching is severe it will be more likely that a student will get mis-graded.

Don't use a phone camera. It is hard to get even lighting, and you will almost certainly need to do more de-warp the page as the top edge may be shorter or longer than the bottom edge.

Align Your Scanned Student Sheets to the Master Bubblesheet Prior to Scoring

Even high-quality scanners often produce images that are slightly askew and perhaps slightly off size compared to the master bubblesheet you have saved as a pdf. Sizing and positioning issues also happen when the bubblesheets are printed.

If you try to score scans that have not been repositioned to match the bubblemap used by MarkShark you will encounter several issues as shown in the example below.

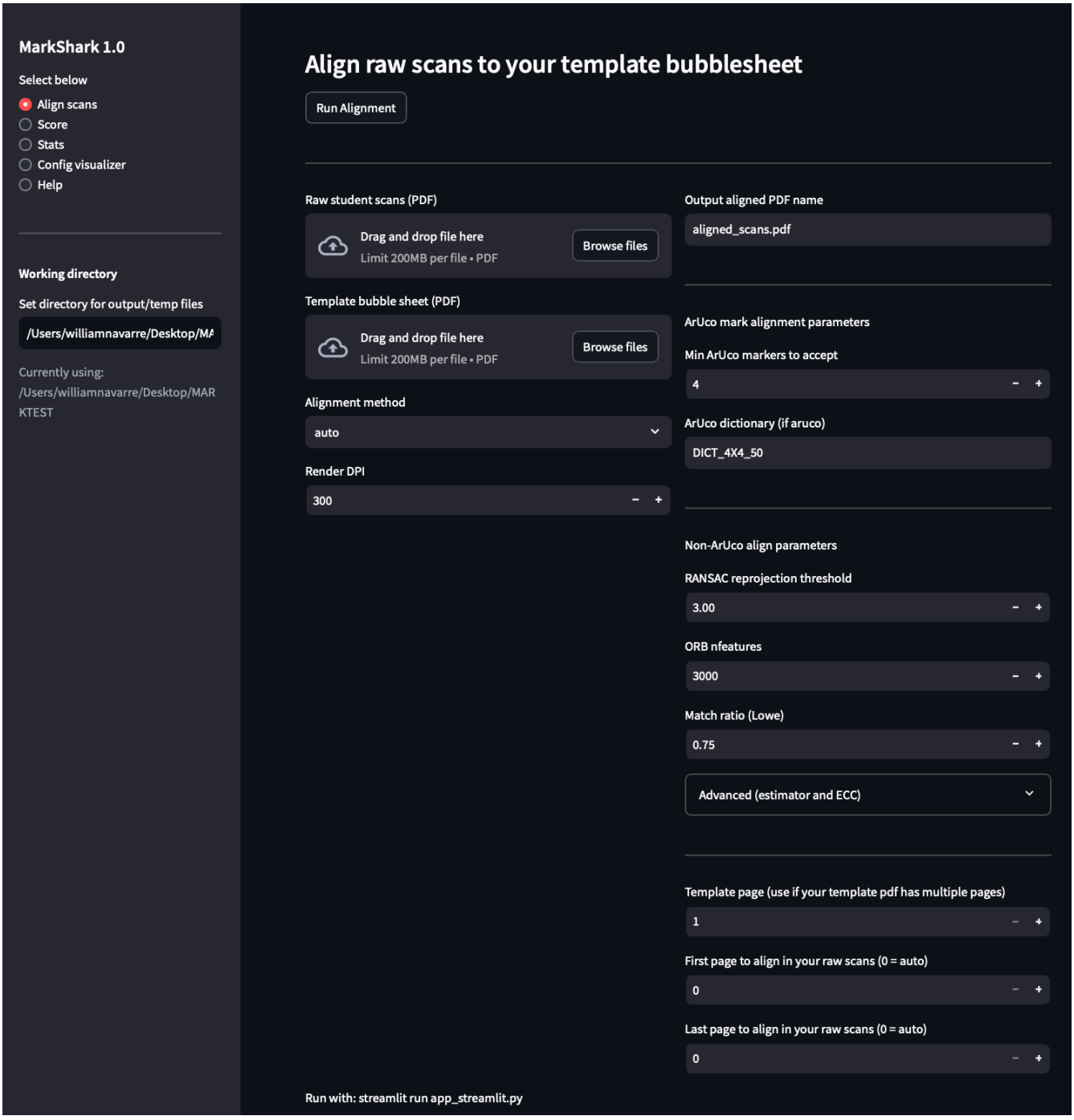


A bubblesheet image that was <u>not</u> aligned before scoring is shown on the left. The same bubblesheet scored after alignment is on the right. The raw, unaligned, scan has the bubbles about 30 pixels lower than where the MarkShark bubblemap.yaml file says they should be. The raw scan is also slightly askew so the bubbles on the left do not align with the map and the bubbles are slightly smaller in the unaligned image. After an alignment step the bubbles overlay nicely with where the MarkShark software expects them to be.

If your printer faithfully produces the correctly sized bubblesheets and you have the perfect scanner that maintains the size of the image faithfully and is not prone to slightly skewing the scans, then it is possible to skip an alignment step.

The rest of us need to align raw scans back to the master template prior to scoring. The MarkShark align program allows you to do this easily.

Aligning scans using the graphical user interface



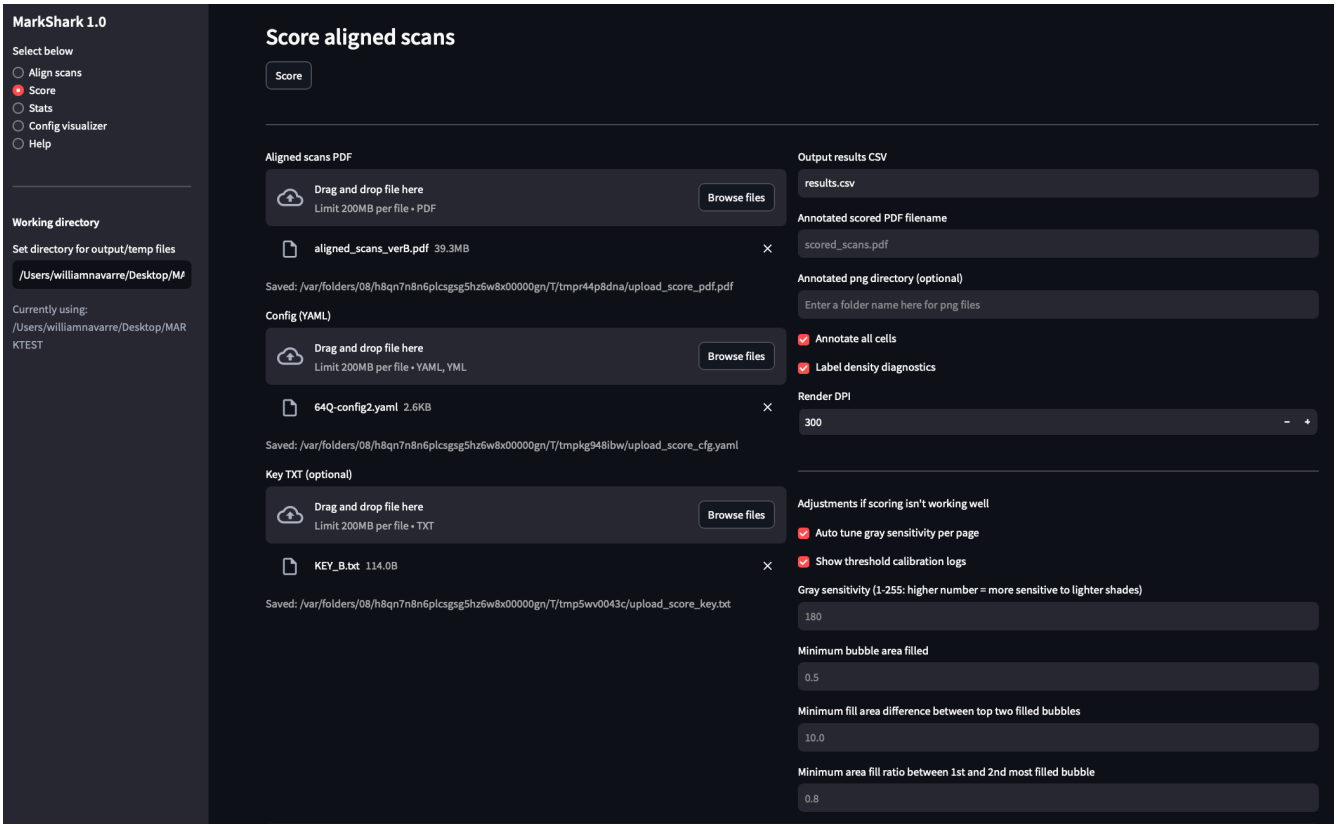
Aligning scans using the command line interface

```
Usage: markshark align [OPTIONS] INPUT_PDF

 Align raw scans to a template PDF.

╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *    input_pdf      TEXT  Raw scans PDF [required]                           │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────╮
│ *  --template          -t                  TEXT     Template PDF to align to │
│                                                     [required]               │
│    --out-pdf           -o                  TEXT     Output aligned PDF       │
│                                                     [default:                │
│                                                     aligned_scans.pdf]       │
│    --dpi                                   INTEGER  Render DPI for alignment │
│                                                     & output                 │
│                                                     [default: 300]           │
│    --template-page                         INTEGER  Template page index to   │
│                                                     use (1-based)            │
│                                                     [default: 1]             │
│    --align-method                          TEXT     Alignment pipeline:      │
│                                                     auto|aruco|feature       │
│                                                     [default: auto]          │
│    --estimator-method                      TEXT     Homography estimator:    │
│                                                     auto|ransac|usac         │
│                                                     [default: auto]          │
│    --min-markers                           INTEGER  Min ArUco markers to     │
│                                                     accept                   │
│                                                     [default: 4]             │
│    --ransac                                FLOAT    RANSAC reprojection      │
│                                                     threshold                │
│                                                     [default: 3.0]           │
│    --use-ecc                --no-use-ecc            Enable ECC refinement    │
│                                                     [default: use-ecc]       │
│    --ecc-max-iters                         INTEGER  ECC iterations           │
│                                                     [default: 50]            │
│    --ecc-eps                               FLOAT    ECC termination epsilon  │
│                                                     [default: 1e-06]         │
│    --orb-nfeatures                         INTEGER  ORB features for         │
│                                                     feature-based align      │
│                                                     [default: 3000]          │
│    --match-ratio                           FLOAT    Lowe's ratio test for    │
│                                                     feature matching         │
│                                                     [default: 0.75]          │
│    --dict-name                             TEXT     ArUco dictionary         │
│                                                     [default: DICT_4X4_50]   │
│    --first-page                            INTEGER  First page index         │
│                                                     (1-based)                │
```

```
    --last-page                              INTEGER  Last page index
                                                      (inclusive, 1-based)
    --help                                            Show this message and
                                                      exit.
```

Scoring with MarkShark

Scoring via the graphical user interface



1. On the left bar select ‘Score’.
2. Set your working directory – this is where the output files will be saved.
3. Load your aligned student bubble sheet scans (as a one multipage pdf), your bubblemap.yaml file, and your key text file. You can do this using the corresponding ‘Browse Files’ buttons or by dragging each file from your desktop or folder to the box corresponding to each type of file.
4. Click the Score button at the top of the page to score the scans using default settings. The output files will appear in a folder within your working directory.
5. However, on the right half of the screen you can set/modify several options including the name of the output files, whether you want to save the annotated student bubble sheets as a set of .png format images, etc.

Scoring via the command line interface

Usage: markshark score [OPTIONS] INPUT_PDF

```
╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *    input_pdf      TEXT  Aligned scans PDF [required]                       │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────╮
│ *  --config            -c                      TEXT     Bubblemap file       │
│                                                         (.yaml/.yml)         │
│                                                         [required]           │
│    --key-txt           -k                      TEXT     Answer key file      │
│                                                         (A/B/C/... one per   │
│                                                         line). If provided,  │
│                                                         only first len(key)  │
│                                                         questions are        │
│                                                         scored.              │
│    --out-csv           -o                      TEXT     Output CSV of        │
│                                                         per-student results  │
│                                                         [default:            │
│                                                         results.csv]         │
│    --out-annotated-d…                          TEXT     Directory to write   │
│                                                         annotated sheets     │
│    --out-pdf                                   TEXT     Annotated PDF output │
│                                                         filename. Default:   │
│                                                         scored_scans.pdf.    │
│                                                         Use """ to disable.  │
│    --annotate-all-ce…                                   Draw every bubble in │
│                                                         each row             │
│    --label-density                                      Overlay % fill text  │
│                                                         at bubble centers    │
│    --dpi                                       INTEGER  Scan/PDF render DPI  │
│                                                         [default: 300]       │
│    --min-fill                                  FLOAT    Minimum fraction of  │
│                                                         the darkest bubble   │
│                                                         required to consider │
│                                                         a mark filled        │
│                                                         (default: 0.25).     │
│                                                         Increase to require  │
│                                                         more completely      │
│                                                         filled bubbles;      │
│                                                         decrease to accept   │
│                                                         lighter or partially │
│                                                         filled marks.        │
│    --top2-ratio                                FLOAT    default 0.8          │
│    --min-score                                 FLOAT    Minimum score        │
│                                                         required to accept a │
│                                                         bubble as filled     │
│                                                         (default: 10.0).     │
│                                                         Increase to require  │
│                                                         higher confidence in │
│                                                         filled bubbles;      │
│                                                         decrease to accept   │
│                                                         lower scores.        │
│    --fixed-thresh                              INTEGER  default 180          │
│    --auto-thresh             --no-auto-thresh          Auto tune            │
│                                                         fixed_thresh per     │
```

```
                                                    page when
                                                    --fixed-thresh is
                                                    omitted
                                                    [default:
                                                    auto-thresh]
    --verbose-thresh                                Print per-page
                                                    threshold
                                                    calibration
                                                    diagnostics
    --help                                          Show this message
                                                    and exit.
```

How the scoring software works

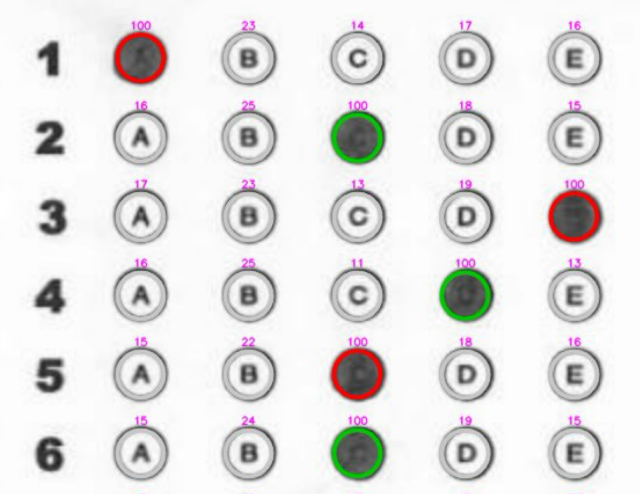
MarkShark scoring works by measuring how much “ink” appears inside each bubble, then picking the most filled bubble in each row as the student’s answer. For every bubble location defined in the bubblemap, the software crops a small region of interest around that bubble and converts it to a grayscale representation where darker pencil marks are easier to detect. It then applies a simple threshold that decides which pixels count as ink, and reports a fill score for the bubble, essentially the fraction of pixels inside the bubble region that look like ink. Unfilled bubbles still get a nonzero score because the printed circle and the letter inside the bubble are real marks on the paper, and those marks contribute a baseline amount of “ink” that is fairly consistent across a page.

Once every bubble has a fill score, MarkShark scores each question row by comparing the bubbles within that row. The bubble with the highest fill score is treated as the candidate selection. A few guardrails prevent false positives: if the best bubble is not sufficiently stronger than the next best bubble, the row is treated as ambiguous or “multiple marks,” and if the best bubble is too close to the baseline unfilled level, the row is treated as blank. This “relative” logic is important because the absolute darkness of pencil marks can vary a lot between students, scanners, and lighting, even when the intended answer is clear to a human.

To handle students who mark lightly, MarkShark can automatically tune its ink sensitivity separately for each scanned page. Before final scoring, it quickly tries a small range of possible gray thresholds and asks, “Which setting best separates the bubbles that look truly filled from the bubbles that look unfilled?” It does this by looking across the page at rows where one bubble clearly stands out, then measuring the gap between the strongest bubble in each row and the other

bubbles in that same row. The threshold that maximizes this separation is chosen for that page. In practice, this makes the system more forgiving for light pencil marks while still resisting background noise, and it avoids forcing a single global threshold that must work for every student and every scan.

After the per page threshold is chosen (or a fixed threshold is used if you specify one), MarkShark runs the normal scoring pass, records the selected letter for each question, and compares it to an optional answer key to compute per student totals. If you request annotated output, it also draws overlays on the page showing detected selections and any flagged rows (blank, multiple marks) so you can visually audit edge cases. The overall approach is simple and transparent: it reduces each bubble to a numeric fill score, uses within row comparisons to decide the answer, and, when needed, adapts ink sensitivity to the specific scan to handle real world variation in how students fill bubbles.



Quick troubleshooting

If you need to troubleshoot you can look at the annotated pdf scans produced after scoring. Above each bubble is the 'fill score' in pink text. You can see that the filled bubbles score 100 while the unfilled bubbles score between 11 and 25.

Assessing the quality of your exam

The stats module processes the CSV output from grade, computes exam- and item-level metrics, and generates CSV summaries and plots.

1. ***Main CSV – results_with_item_stats.csv***

- Original data - adds rows for:
  - Pct correct (0–1): item difficulty
  - Point–biserial: discrimination index

2. ***Exam stats – exam_stats.csv***

- Overall test summary:
  - k_items: number of questions
  - mean_total, sd_total, var_total: score distribution
  - avg_difficulty: mean proportion correct
  - KR-20, KR-21: reliability estimates

3. ***Item report – item_analysis.csv***

- One row per item-option:
  - item, key, option, is_key
  - option_count, option_prop
  - option_biserial, item_difficulty, item_point_biserial

4. ***Plots – folder item_plots/***

- One PNG per item showing Item Characteristic Curves (ICC):
  - X-axis: binned total-minus-item score
  - Y-axis: proportion correct per bin

Interpretation of Statistics

Metric	***Description***	***Ideal / Notes***
Difficulty (Pct correct)	Proportion of students answering correctly.	Ideal range ≈ 0.3–0.8
Point–biserial	Correlation between item correctness and total score (excluding that item).	Higher = better discrimination; negative = problematic

Metric	*Description*	*Ideal / Notes*
KR-20	Reliability coefficient for dichotomous items.	≥ 0.7 = good internal consistency
KR-21	Approximation of KR-20 assuming equal item difficulty.	Use when item data are limited
Option biserial	Correlation between selecting an option and student ability.	Correct = positive; distractors = negative / ~0

Tips on designing good MCQ tests.

Multiple-choice exams often get a bad reputation. It is commonly assumed that multiple-choice questions (MCQs) can only test rote memorization or trivial facts, and that they are poorly suited to assessing conceptual understanding or applied reasoning. This perception is misleading. While poorly designed MCQs can indeed reward memorization, **well-written MCQs can effectively probe higher-order thinking, including the ability to apply principles, evaluate alternatives, and reason through complex scenarios.**

Further, short-answer or essay questions are not inherently superior at assessing learning. These formats can suffer from subjectivity in grading. Teachers and TAs must decide how much to award for partially correct or ambiguously worded answers and may overvalue writing quality or grammar relative to the scientific or conceptual understanding being tested. Such biases can disadvantage students writing in a second language or those with less training in written expression, even when they understand the material well. Moreover, time constraints often limit how effectively students can demonstrate knowledge through extended written responses.

Ultimately, the utility of any assessment method depends on what is being assessed, why it is being assessed, and how the questions are written. MCQs are not suited to measuring writing ability, but they *can* excel at efficiently and objectively assessing conceptual understanding, factual recall, and reasoning across a broad range of topics. With careful design, they can even test creativity and applied logic.

Above all else, a major advantage of multiple-choice exams is that they generate robust quantitative data about question performance. Well-constructed questions discriminate effectively between stronger and weaker students; typically, high-performing students answer them correctly about 90% of the time, while less-prepared students succeed roughly 30–40% of the time (only slightly better than random). A balanced exam includes a mixture of easy and difficult items to span the full range of ability. Items that are too easy offer little discriminatory value, while questions that almost nobody answers correctly often

indicate unclear wording, incomplete instruction, or unrealistic expectations of student ability.

What makes a good MC question?

The benefits of MCQs depend heavily on the quality of the items. Poorly written MCQs can be misleading, ambiguous, or test trivialities, which undermines their advantages. The University of Michigan CRLT "Best Practices" guide explicitly states that MCQ items can measure higher-order skills (application, inference, evaluation) if the question stems are carefully constructed (not just recall) and distractors require reasoning.

Formally, an MCQ has two parts:

1. **Stem** – the question or incomplete statement (e.g., "Which of the following best explains why…?" or "The enzyme catalase functions to…").
2. **Options** – the **key and distractors** – the correct answer (key) and the incorrect but *plausible* alternatives (distractors).

In test design, the **question stem** is the main part of a multiple-choice question. It is the part that poses the **problem or question** that the student must answer. It provides the context or scenario that leads to the choices.

A well-written stem clearly frames the problem so that students must apply knowledge rather than rely on test-taking tricks.

Good stems are also **clear, concise, and self-contained**, avoiding unnecessary wording or clues. For example:

- **Poor stem:** "Catalase is found in peroxisomes. What does it do?"
- **Better stem:** "Which of the following reactions is catalyzed by the enzyme catalase?"

CITATIONS:

1. **Haataja, E. S., Tolvanen, A., Vilppu, H., Kallio, M., Peltonen, J., & Metsäpelto, R. L.** (2023). Measuring higher-order cognitive skills with multiple choice questions–potentials and pitfalls of Finnish teacher education entrance. Teaching and Teacher Education, 122, 103943.
2. **Piontek, M. E.** (2008). Best practices for designing and grading exams. *University of Michigan CRLT Occasional Paper*, *24*, 1-12.
3. **Stanger-Hall, K. F.** (2012). Multiple-choice exams: an obstacle for higher-level thinking in introductory science classes. CBE—Life Sciences Education, 11(3), 294-306.
4. **Zimmaro, D. M.** (2004). Writing good multiple-choice exams. *Measurement and Evaluation Center. University of Texas at Austin. Web site*, *16*.

Troubleshooting: Under the Hood

This section is for people who are interested in modifying or playing with MarkShark to either improve it or troubleshoot issues they may be having.

Understanding how MarkShark aligns scans

Almost everything regarding the alignment of raw scanned student bubble sheets to your template uses functions from the OpenCV Python module.

OpenCV (Open Source Computer Vision Library) is a widely used, open source software library for working with images and video.

What it gives you, in practice:

- **Read, write, and manipulate images/video**: load a PNG/JPEG, resize, rotate, crop, blur, sharpen, convert to grayscale, etc.
- **Image processing**: thresholding, edge detection, filtering, morphology (dilate, erode), histogram operations.
- **Feature detection and matching**: keypoints and descriptors (ORB, SIFT in many builds, AKAZE), matching, and then geometry fitting (homography/affine) often with **RANSAC**.
- **Geometric transforms and alignment**: warpAffine, warpPerspective, image registration, camera calibration.
- **Object detection and tracking**: classical methods and some DNN utilities.

In Python you usually use it as the cv2 package, for example import cv2.

1) Read PDF pages into images

OpenCV does not read PDFs directly. Typical flow is pdf2image, PyMuPDF, or Poppler to rasterize, then OpenCV handles the pixels.

- Once you have a numpy image array, OpenCV starts here:
    - cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) (grayscale)
    - cv2.resize(img, ...) (optional)

2) Preprocess to make matching easier

Common numcols:

- Denoise: cv2.GaussianBlur(gray, (k,k), 0) or cv2.medianBlur(gray, k)
- Contrast normalization: cv2.equalizeHist(gray) or CLAHE: cv2.createCLAHE(...).apply(gray)
- Binarize (sometimes):
    - cv2.threshold(gray, t, 255, cv2.THRESH_BINARY)
    - cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, blockSize, C)

3) Detect "anchors" for alignment (two common approaches)

A) ArUco marker based (very robust when you control the template)

- Create detector + dictionary:
    - cv2.aruco.getPredefinedDictionary(...)
    - cv2.aruco.ArucoDetector(dictionary, parameters) (newer OpenCV) or older cv2.aruco.detectMarkers(...)
- Detect markers:
    - corners, ids, rejected = detector.detectMarkers(gray)
- (Optional) refine corners:
    - cv2.cornerSubPix(...)
- Estimate transform from marker corners:
    - cv2.findHomography(src_pts, dst_pts, method=cv2.RANSAC, ransacReprojThreshold=...)
- Warp:
    - cv2.warpPerspective(img, H, (W,H))

B) Feature based (works without markers, more fragile on low texture)

Keypoint and descriptor extraction:

- ORB (most common in scan tools because it's fast and free):
    - orb = cv2.ORB_create(nfeatures=..., scaleFactor=..., ...)
    - kp1, des1 = orb.detectAndCompute(gray1, None)
    - kp2, des2 = orb.detectAndCompute(gray2, None)

Other options:

- cv2.AKAZE_create()
- cv2.SIFT_create() (available in many builds now, slower but strong)

Match descriptors:

- Brute force matcher (common for ORB):
    - bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
    - matches = bf.knnMatch(des1, des2, k=2)
- Filter matches (ratio test):
    - you implement: keep m if m.distance < ratio * n.distance
- Or FLANN for binary descriptors (less common, more setup)

Estimate transform robustly (this is where RANSAC is used):

- Homography:
    - H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ransacReprojThreshold=...)
- Affine / partial affine:
    - M, inliers = cv2.estimateAffine2D(src_pts, dst_pts, method=cv2.RANSAC, ransacReprojThreshold=...)
    - M, inliers = cv2.estimateAffinePartial2D(...)

Warp into template coordinates:

- For affine:

    - cv2.warpAffine(img, M, (W,H))
- For homography:
    - cv2.warpPerspective(img, H, (W,H))

4) Optional "polish" alignment with intensity-based registration (ECC)

This can correct small residual shifts/scale/shear after feature alignment.

- warp = np.eye(2,3) (affine) or np.eye(3,3) (homography)
- cc, warp = cv2.findTransformECC(template_gray, scan_gray, warp, motionType, criteria, inputMask=None, gaussFiltSize=…)
- Then warp with cv2.warpAffine or cv2.warpPerspective

(You'll often see ECC used after an initial RANSAC warp, because ECC needs a decent starting guess.)

5) Define bubble ROIs (regions of interest)

If you already know bubble centers from a config, OpenCV is mostly just cropping/masking:

- Crop: roi = img[y:y+h, x:x+w]
- Create a circular mask:
    - mask = np.zeros((h,w), np.uint8)

    - cv2.circle(mask, (cx,cy), r, 255, -1)
    - masked = cv2.bitwise_and(roi, roi, mask=mask)

If you detect bubbles automatically (less common if you have a template):

- Edge detection: cv2.Canny(...)
- Find contours: cv2.findContours(...)
- Filter by area/circularity, fit circles:
    - cv2.minEnclosingCircle(contour)
    - cv2.HoughCircles(...) (sometimes)

6) Score "filled-ness" of each bubble

Two standard strategies:

A) Threshold then count ink pixels

- Threshold:
    - _, bw = cv2.threshold(gray, t, 255, cv2.THRESH_BINARY_INV) (invert so ink becomes white)
    - or cv2.adaptiveThreshold(...)
    - or cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
- Count within mask:

    - count = cv2.countNonZero(cv2.bitwise_and(bw, bw, mask=mask))
- Normalize by mask area to get a fill ratio

B) Use mean intensity in the masked ROI

- mean = cv2.mean(gray, mask=mask)[0]

Darker mean usually means more fill.

7) Decide selected option and handle edge cases

This part is usually pure Python logic, but OpenCV sometimes helps with debugging overlays.

8) Draw annotations (debug output)

- Circles/boxes:
    - cv2.circle(img, center, r, color, thickness)
    - cv2.rectangle(img, (x1,y1), (x2,y2), color, thickness)
- Text:
    - cv2.putText(img, "Q1:A", org, cv2.FONT_HERSHEY_SIMPLEX, scale, color, thickness, cv2.LINE_AA)
- Save:
    - cv2.imwrite(path, img)

9) Quality metrics (alignment diagnostics)

You'll often see:

- Difference image:
    - diff = cv2.absdiff(template_gray, aligned_gray)
- Summary stats:
    - cv2.mean(diff) (or numpy stats)
- Keypoint match visualization:
    - cv2.drawMatches(...) or cv2.drawMatchesKnn(...)

If you paste a short snippet from your aligner (the part that finds the transform), I can point to exactly which of these calls you are using and what each parameter is doing in your script.

RANSAC (Random Sample Consensus) is a robust way to estimate the geometric transform between two images (for alignment) when some of your feature matches are wrong.

Here is what it's doing in that context:

- **You start with tentative correspondences**: detect keypoints in both images (ORB/SIFT/etc), compute descriptors, then match them. This match set always contains **outliers** (bad matches) because of repeated patterns, blur, low texture, etc.
- **RANSAC repeatedly fits a transform using a tiny random subset** of matches:

    - For an **affine** transform, it samples the minimum points needed (typically 3 pairs).
    - For a **homography** (projective transform), it samples 4 pairs.
- **Each trial transform is scored by "inliers"**: RANSAC applies the trial transform to all matched points and counts how many land within some distance threshold (the "reprojection error" threshold, often a few pixels). Those are the inliers.
- **It keeps the transform with the most inliers**, then refits the transform using *all* inliers for a cleaner estimate.

Why this helps: without RANSAC, even a modest number of bad matches can make the fitted transform garbage. With RANSAC, the bad matches get rejected automatically because they don't agree with the dominant geometric relationship between the images.

Practical knobs (in most libraries, e.g. OpenCV):

- **Reprojection threshold (pixels)**: smaller is stricter (fewer inliers, maybe fail), larger is more forgiving (more inliers, but can accept wrong geometry if too large).
- **Confidence / max iterations**: higher confidence means more iterations, more compute, and usually more reliable.

Licensing or support:
william.navarre@utoronto.ca