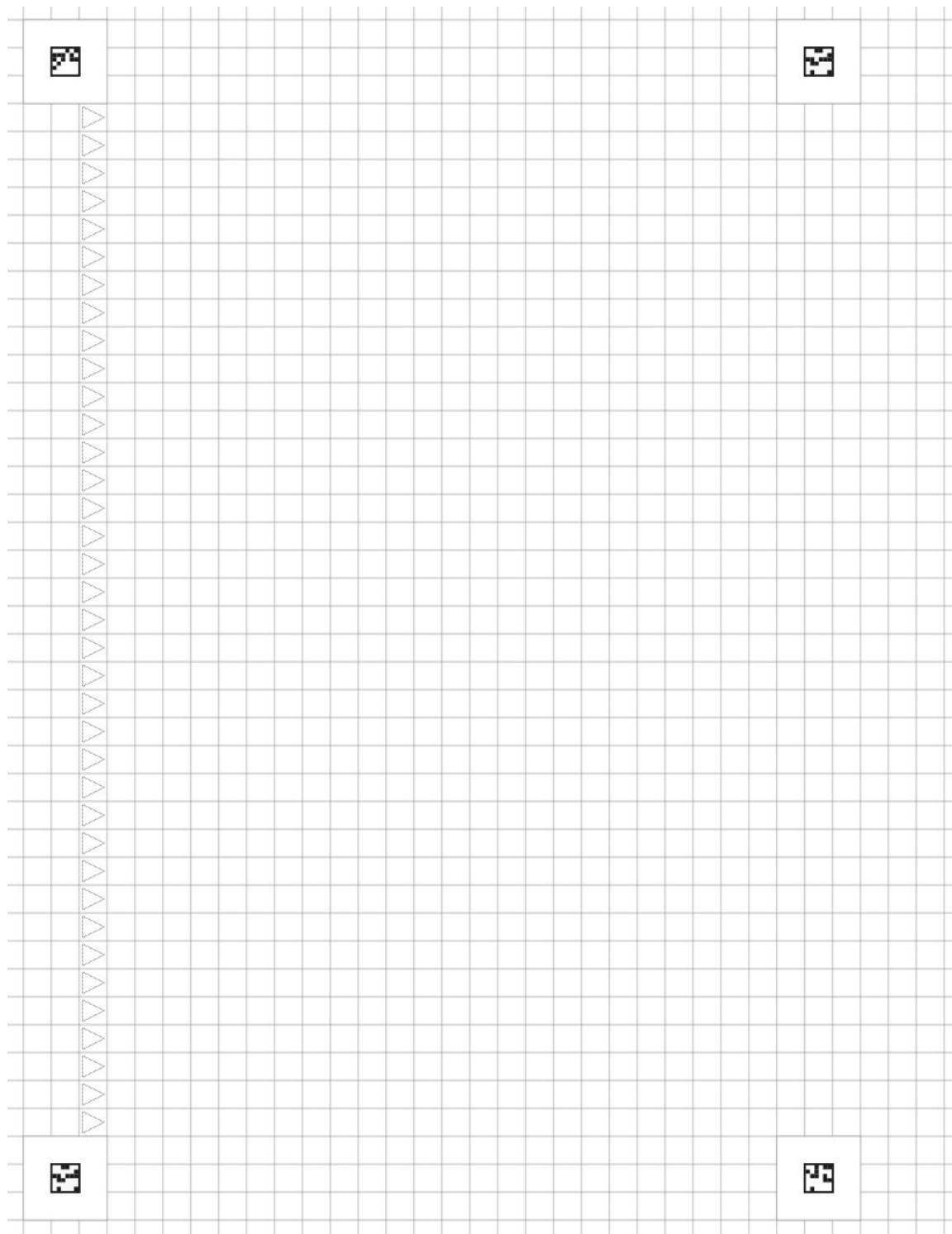


A2 Image Normalization

Each image that is taken with the camera phone will need to be normalized.

```
In [4]: import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns
```

Page Zero Display



Page One Display

[Calibration Page One Display](#)

Other Page Variants

Now that the calibration cubes have been assessed, we turn our attention to another form of

noise.

- Dot Matrix
- Lined
- Blank
- Hex

Thoughts

It is not lost on me that there is an infinite amount of paper types that one could imagine in order to convey ideas. Personally, the type for which I am the most nostalgic is dot matrix. An example below:

[Dot Matrix Page One](#)

Describing the Calibration Features

Quick Calibration Squares: 6 x 6 square in every corner. Each cell in the square is 0.975mm+/- 0.500mm wide. There are 4 distinct styles:

- Top Left and Bottom Right QC square are the same per page (Page 0: Rear; Page 1; Front)
 - see:
 - [Page Zero and One Bottom Right](#)
 - [Page Zero and One Top Left](#)
- The Top Right and Bottom Left of each page is the same
 - see:
 - [Page Zero Top Right and Bottom Left](#)
 - [Page One Top Right and Bottom Left](#)

QC Square Envelope: Each QC square has an envelope around it (as seen in any appended images of the squares). The envelope is 0.75in .

Left Margin Arrows Each page has directional triangles in a straight line along the Left Margin of the drawing area. There are 37 of them pointing with the tips towards the drawing area. The ratio of the triangle is such that the left edge, which is perpendicular to the column direction is 0.75in - 2mm and the two other edges, which are congruent, are also 0.75in - 2mm . The triangles are centered within each cell within each column and row.

[Left Margin](#)

Matrix Representations of QC Squares

In [5]:

```
# 0: black
# 1: white

n = [ # Empty Matrix
    [0, 0, 0, 0, 0, 0], # Row 0
    [0, 0, 0, 0, 0, 0], # Row 1
    [0, 0, 0, 0, 0, 0], # Row 2
    [0, 0, 0, 0, 0, 0], # Row 3
    [0, 0, 0, 0, 0, 0], # Row 4
    [0, 0, 0, 0, 0, 0] # Row 5
]

# Page 0 and 1, Top Left
p01_tl = [
    [1, 1, 1, 0, 1, 0], # Row 0
    [0, 0, 0, 1, 0, 1], # Row 1
    [0, 1, 0, 1, 0, 0], # Row 2
    [1, 0, 1, 1, 1, 1], # Row 3
    [0, 1, 1, 1, 1, 1], # Row 4
    [1, 1, 1, 1, 1, 1] # Row 5
]

# Page 0 and 1, Bottom Right
p01_br = [
    [1, 1, 0, 1, 0, 0], # Row 0
    [1, 1, 0, 1, 1, 1], # Row 1
    [0, 1, 0, 1, 1, 1], # Row 2
    [1, 0, 0, 1, 0, 1], # Row 3
    [1, 1, 1, 1, 1, 1], # Row 4
    [1, 0, 1, 1, 1, 1] # Row 5
]

# Page 0, Top Right and Bottom Left
p0_trbl = [
    [1, 1, 0, 0, 1, 1], # Row 0
    [1, 1, 1, 1, 1, 0], # Row 1
    [0, 0, 1, 0, 0, 0], # Row 2
    [1, 0, 0, 1, 1, 1], # Row 3
    [1, 1, 1, 0, 1, 1], # Row 4
    [1, 0, 1, 1, 1, 0] # Row 5
]

# Page 1, Top Right and Bottom Left
p1_trbl = [
    [0, 0, 0, 1, 0, 1], # Row 0
    [0, 1, 0, 0, 0, 0], # Row 1
    [1, 0, 0, 0, 0, 1], # Row 2
    [1, 1, 0, 1, 0, 1], # Row 3
    [0, 1, 1, 1, 0, 0], # Row 4
    [0, 1, 1, 0, 0, 0] # Row 5
]
```

Visual Representation

```
In [6]: import matplotlib.pyplot as plt

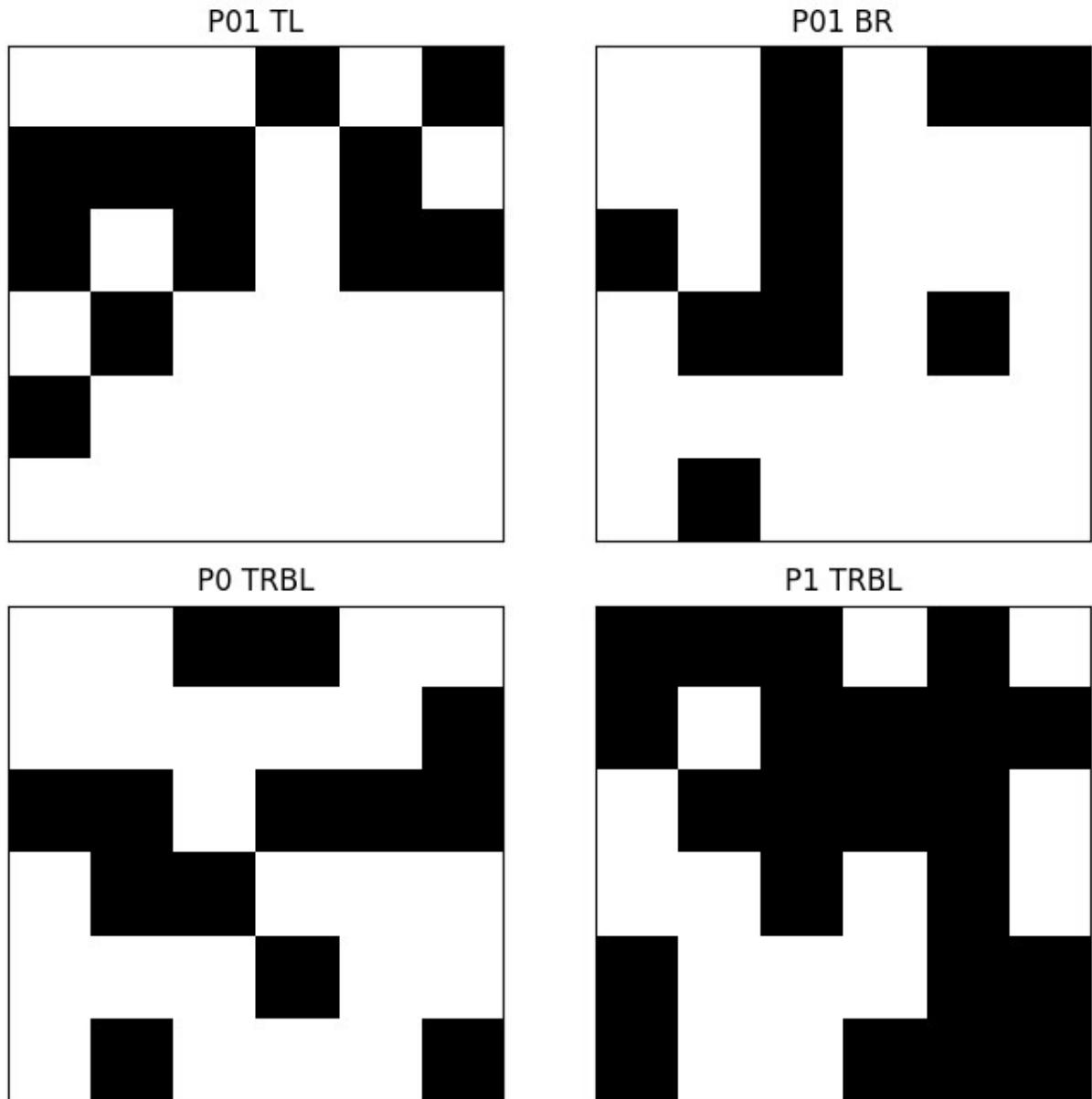
# Show graph of what the QC squares Look like

qc_patterns = {
    'p01_tl': np.array(p01_tl),
    'p01_br': np.array(p01_br),
    'p0_trbl': np.array(p0_trbl),
    'p1_trbl': np.array(p1_trbl)
}

fig, axes = plt.subplots(2, 2, figsize=(7, 7))
for ax, (name, pattern) in zip(axes.flatten(), qc_patterns.items()):
    ax.imshow(pattern, cmap='gray', vmin=0, vmax=1, interpolation='nearest')
    ax.set_title(name.replace('_', ' ').upper())
    ax.set_xticks([])
    ax.set_yticks([])
    ax.grid(False)

plt.suptitle('QC Square Patterns (6x6)', fontsize=14)
plt.tight_layout()
plt.show()
```

QC Square Patterns (6x6)



Thought

I do not approve of the imperial system, but then I didn't make the paper. In future work, I would prefer a metric version of the medium. This is something that I think would take no longer than 30 minutes to design.

Normalization Script

Test Image

Degenerate points

points that do not form a valid geometric shape; they are collinear, overlapping, or not distinct enough to define a proper quadrilateral for transformations.

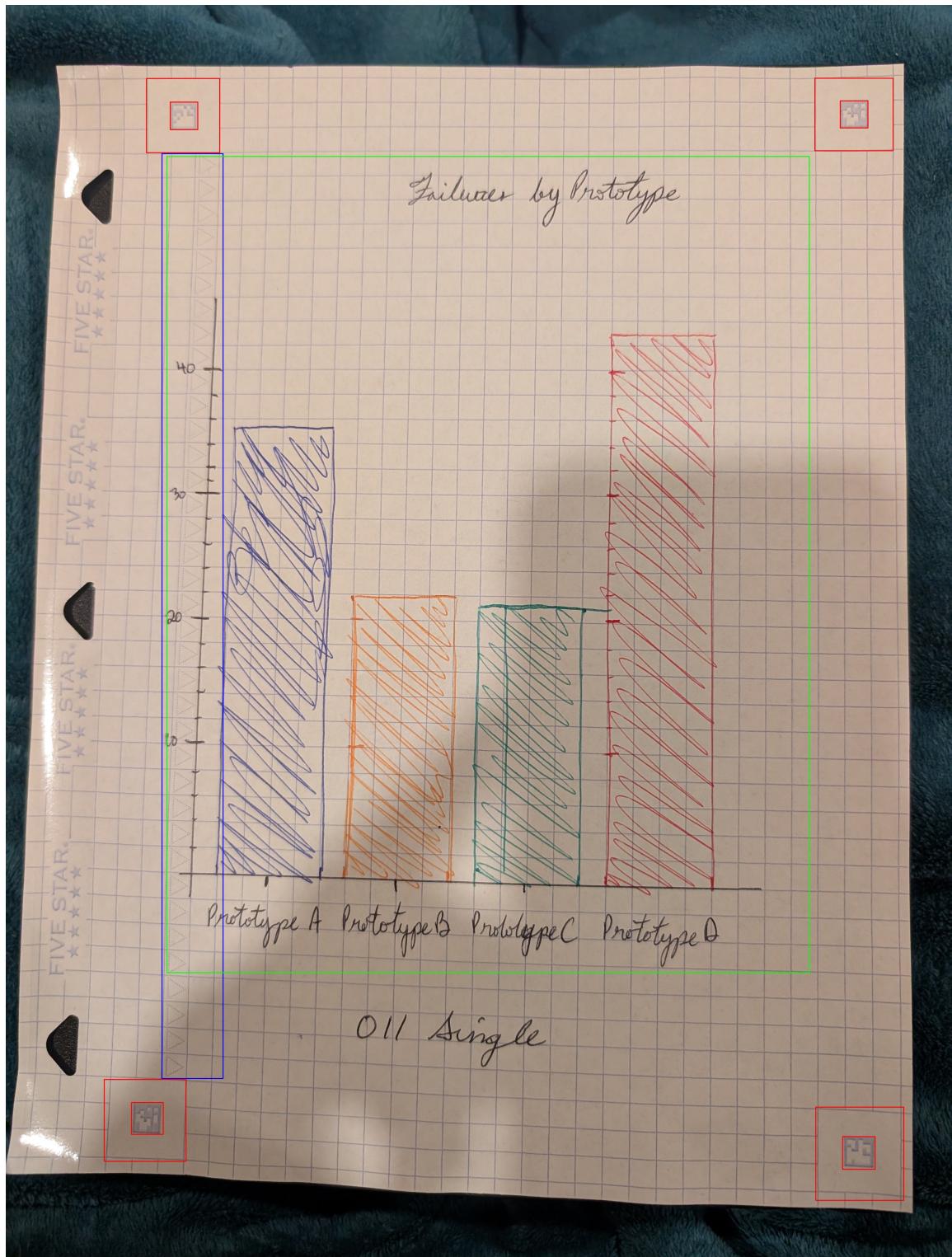
011 Single

Identifying Page Features

Red: QC squares

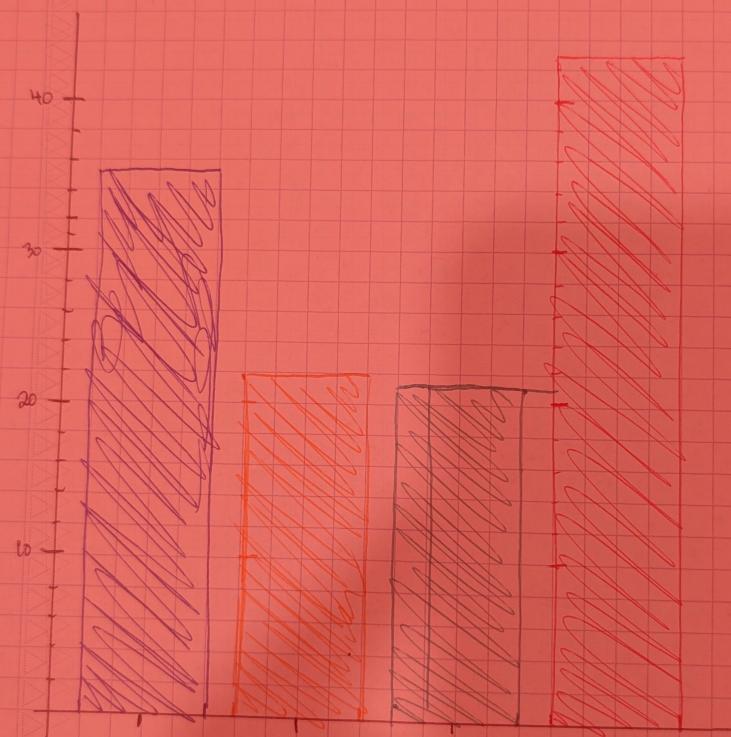
Blue: Triangles

Green: Chart Area



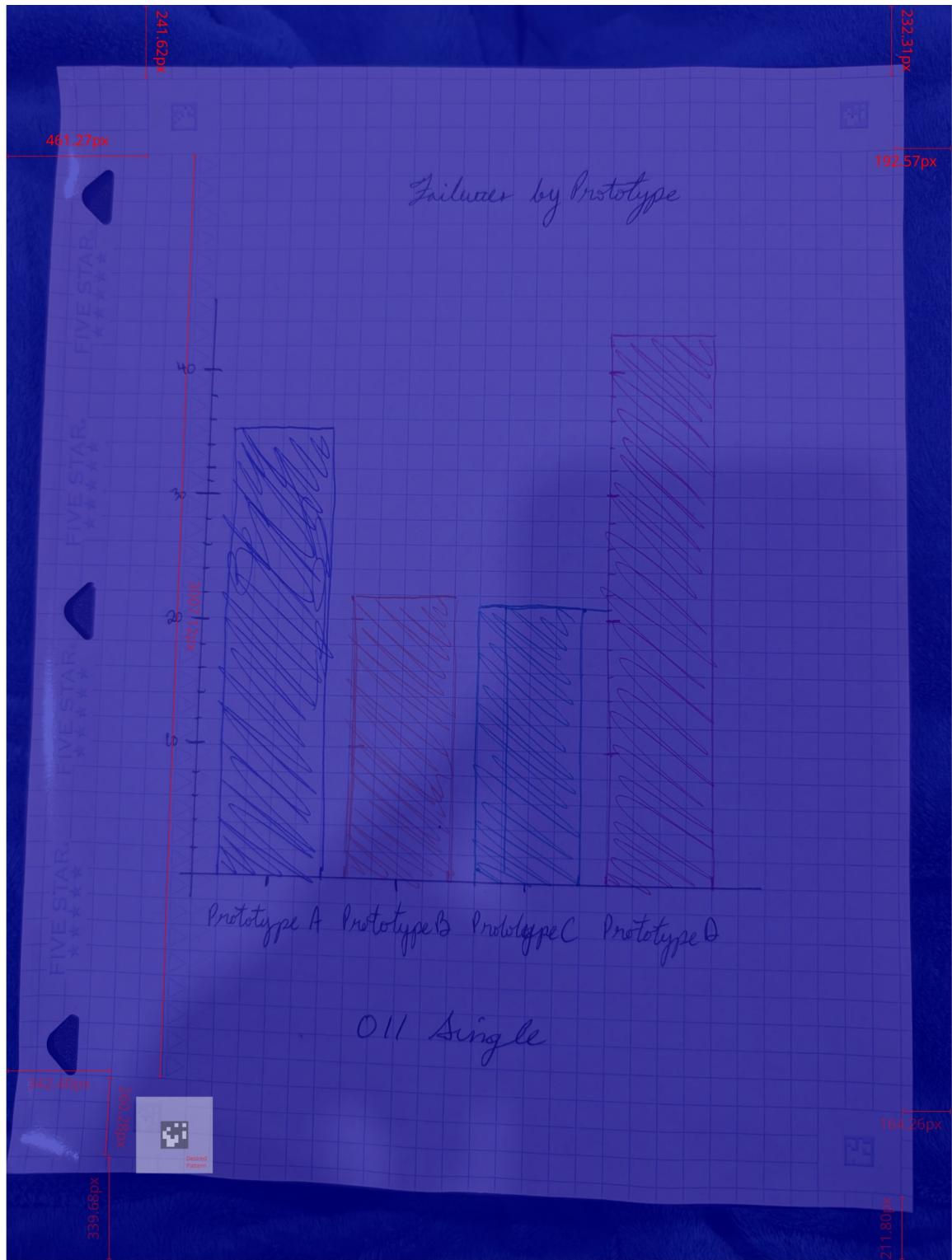
Desired Features

Failures by Prototype



OII Single

Page Dimensions



```
In [7]: ## Test Image
import PIL
import os
import cv2
import numpy as np
```

```
In [8]: def detect_qc_square(image, corner='auto', search_region_size=0.2, min_square_area=
    """
    Detect QC square in the image.

```

```

Parameters:
-----
image : numpy.ndarray
    RGB image (H, W, 3) or BGR if from cv2.imread
corner : str, optional
    Which corner to search: 'tl', 'tr', 'bl', 'br', or 'auto' (search all)
    Default: 'auto'
search_region_size : float
    Fraction of image dimensions to search in corner (0.0-1.0)
    Default: 0.2 (top/bottom/left/right 20%)
min_square_area : int
    Minimum area for detected square (in pixels)
    Default: 100

Returns:
-----
dict or list of dicts
    If corner='auto': list of dicts, one per detected QC square
    Otherwise: single dict with keys:
        - 'bbox': (x, y, width, height) bounding box
        - 'corner': detected corner position ('tl', 'tr', 'bl', 'br')
        - 'pattern': extracted 6x6 pattern (numpy array)
        - 'pattern_match': matched pattern name (if successful)
        - 'confidence': match confidence (0-1)
"""

# Convert BGR to RGB if needed (cv2 Loads as BGR)
if len(image.shape) == 3:
    # Check if it's likely BGR by comparing first/last channels
    # Or just convert if from cv2.imread
    if image.dtype == np.uint8:
        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    else:
        rgb_image = image.copy()
else:
    raise ValueError("Image must be RGB/BGR (3 channels)")

h, w = rgb_image.shape[:2]

# Define search regions for each corner
search_regions = {
    'tl': (0, 0, int(w * search_region_size), int(h * search_region_size)),
    'tr': (int(w * (1 - search_region_size)), 0, w, int(h * search_region_size)),
    'bl': (0, int(h * (1 - search_region_size)), int(w * search_region_size), h),
    'br': (int(w * (1 - search_region_size)), int(h * (1 - search_region_size)))
}

# Convert to grayscale for detection
gray = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY)

def detect_in_region(region_bbox, corner_name):
    """Detect QC square in a specific region."""
    rx, ry, rw, rh = region_bbox
    region_gray = gray[ry:rh, rx:rw]

    if region_gray.size == 0:

```

```

    return None

    # Enhance contrast for better detection
    clahe = cv2.createCLAHE(clipLimit=2.0, tileSize=(8,8))
    region_enhanced = clahe.apply(region_gray)

    # Adaptive threshold to find edges/boundaries
    binary = cv2.adaptiveThreshold(region_enhanced, 255,
                                   cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                   cv2.THRESH_BINARY_INV, 11, 2)

    # Find contours
    contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL,
                                   cv2.CHAIN_APPROX_SIMPLE)

    # Filter for square-like contours
    square_candidates = []
    region_area = region_gray.shape[0] * region_gray.shape[1]
    min_area_pixels = max(min_square_area, region_area * 0.01) # At Least 1% o.

    for contour in contours:
        area = cv2.contourArea(contour)
        if area < min_area_pixels:
            continue

        # Approximate polygon
        peri = cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, 0.02 * peri, True)

        # Check if roughly square (4 corners)
        if len(approx) >= 4:
            # Get bounding rect
            x_rect, y_rect, w_rect, h_rect = cv2.boundingRect(contour)

            # Check aspect ratio (should be roughly square)
            aspect_ratio = float(w_rect) / h_rect if h_rect > 0 else 0
            if 0.7 < aspect_ratio < 1.3: # Allow some tolerance
                # Check solidity (filled vs outline)
                solidity = area / (w_rect * h_rect) if (w_rect * h_rect) > 0 el

                square_candidates.append({
                    'contour': contour,
                    'area': area,
                    'bbox_local': (x_rect, y_rect, w_rect, h_rect),
                    'solidity': solidity,
                    'aspect_ratio': aspect_ratio
                })

    if not square_candidates:
        return None

    # Sort by area (largest first) and prefer more square-like shapes
    square_candidates.sort(key=lambda x: x['area'] * x['solidity'], reverse=True)
    best_candidate = square_candidates[0]

    # Convert Local coordinates to global image coordinates

```

```

        x_local, y_local, w_local, h_local = best_candidate['bbox_local']
        bbox_global = (rx + x_local, ry + y_local, w_local, h_local)

    # Extract pattern from this region
    try:
        pattern = extract_pattern_from_rgb(rgb_image, bbox_global, envelope_mar

    # Match pattern against known patterns
    pattern_match, confidence = match_pattern(pattern, {
        'p01_tl': np.array(p01_tl),
        'p01_br': np.array(p01_br),
        'p0_trbl': np.array(p0_trbl),
        'p1_trbl': np.array(p1_trbl)
    })

    return {
        'bbox': bbox_global,
        'corner': corner_name,
        'pattern': pattern,
        'pattern_match': pattern_match,
        'confidence': confidence,
        'area': best_candidate['area'],
        'solidity': best_candidate['solidity']
    }
except Exception as e:
    print(f"Error extracting pattern for {corner_name}: {e}")
    return {
        'bbox': bbox_global,
        'corner': corner_name,
        'pattern': None,
        'pattern_match': None,
        'confidence': 0.0,
        'error': str(e)
    }

# Search in specified corner(s)
if corner == 'auto':
    results = []
    for corner_name, region_bbox in search_regions.items():
        result = detect_in_region(region_bbox, corner_name)
        if result:
            results.append(result)
    return results
else:
    if corner not in search_regions:
        raise ValueError(f"Corner must be one of: {list(search_regions.keys())}")
    return detect_in_region(search_regions[corner], corner)

def match_pattern(extracted_pattern, known_patterns):
    """
    Match extracted 6x6 pattern against known patterns.

    Parameters:
    -----
    extracted_pattern : numpy.ndarray
        6x6 binary pattern
    """

```

```

known_patterns : dict
    Dictionary of pattern_name -> pattern_array

Returns:
-----
tuple: (best_match_name, confidence_score)
"""
if extracted_pattern is None:
    return None, 0.0

best_match = None
best_score = 0.0

for name, ref_pattern in known_patterns.items():
    # Calculate similarity (simple matching)
    matches = np.sum(extracted_pattern == ref_pattern)
    similarity = matches / 36.0 # 36 cells total (6x6)

    if similarity > best_score:
        best_score = similarity
        best_match = name

return best_match, best_score

```

```

In [9]: def extract_pattern_from_rgb(rgb_image, bbox, envelope_margin=0.1):
    """
    Extract 6x6 binary pattern from RGB image.

    envelope_margin: percentage of bbox to use as margin for envelope
    """
    x, y, w, h = bbox

    # Crop region
    qc_region_rgb = rgb_image[y:y+h, x:x+w]

    # Account for the 0.75in envelope - extract inner square
    margin = int(min(w, h) * envelope_margin)
    inner_region_rgb = qc_region_rgb[margin:h-margin, margin:w-margin]

    # Convert to grayscale
    inner_gray = cv2.cvtColor(inner_region_rgb, cv2.COLOR_RGB2GRAY)

    # Get dimensions
    grid_h, grid_w = inner_gray.shape

    # Divide into 6x6 cells
    cell_h = grid_h // 6
    cell_w = grid_w // 6

    # Extract binary pattern
    pattern_6x6 = np.zeros((6, 6), dtype=int)

    for i in range(6):
        for j in range(6):
            # Sample cell center region (avoid edges between cells)
            y_start = i * cell_h + cell_h // 4

```

```

        y_end = (i + 1) * cell_h - cell_h // 4
        x_start = j * cell_w + cell_w // 4
        x_end = (j + 1) * cell_w - cell_w // 4

        if y_end > y_start and x_end > x_start:
            cell_region = inner_gray[y_start:y_end, x_start:x_end]

            # Threshold: mean value determines if cell is black or white
            # Black cells (filled) have low pixel values
            # White cells (empty) have high pixel values
            mean_value = np.mean(cell_region)

            # Adaptive threshold based on local image statistics
            # If mean is below 128 (or use Otsu's method), it's black (filled)
            if mean_value < 128:
                pattern_6x6[i, j] = 0 # black/filled
            else:
                pattern_6x6[i, j] = 1 # white/empty

    return pattern_6x6

```

Test

In [10]:

```

# Load image
image_path = 'hand_drawn_notes/bc_011_single-000.jpg'
image = cv2.imread(image_path) # Returns BGR

# Detect all QC squares automatically
results = detect_qc_square(image, corner='auto', min_square_area=25)

# Process results
for result in results:
    print(f"Found QC square in {result['corner']} corner")
    print(f" Bounding box: {result['bbox']}")
    if result['pattern_match']:
        print(f" Matched pattern: {result['pattern_match']}")
    print(f" Confidence: {result['confidence']:.2%}")

```

```

Found QC square in tr corner
    Bounding box: (2457, 266, 615, 550)
    Matched pattern: p01_br
    Confidence: 72.22%
Found QC square in bl corner
    Bounding box: (20, 3264, 594, 639)
    Matched pattern: p01_tl
    Confidence: 72.22%
Found QC square in br corner
    Bounding box: (2457, 3264, 615, 685)
    Matched pattern: p01_br
    Confidence: 66.67%

```

Visualisation

In [11]:

```

import matplotlib.pyplot as plt
import matplotlib.patches as patches

```

```

def visualize_qc_squares(image, results, show_labels=True, figsize=(15, 12)):
    """
    Visualize detected QC squares on the original image.

    Parameters:
    -----
    image : numpy.ndarray
        Original RGB image
    results : list of dicts
        List of detection results from detect_qc_square()
    show_labels : bool
        Whether to show labels with corner and pattern info
    figsize : tuple
        Figure size for matplotlib

    Returns:
    -----
    matplotlib.figure.Figure
        The figure object (can be saved or displayed)
    """

# Create a copy of the image to draw on
if len(image.shape) == 3:
    display_image = image.copy()
else:
    display_image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)

# Define colors for each corner
corner_colors = {
    'tl': (255, 0, 0),      # Red for top-left
    'tr': (0, 255, 0),      # Green for top-right
    'bl': (0, 0, 255),      # Blue for bottom-left
    'br': (255, 165, 0),    # Orange for bottom-right
}

corner_names = {
    'tl': 'Top-Left',
    'tr': 'Top-Right',
    'bl': 'Bottom-Left',
    'br': 'Bottom-Right'
}

# Create figure
fig, ax = plt.subplots(1, 1, figsize=figsize)
ax.imshow(display_image)
ax.axis('off')

# Draw bounding boxes for each detected QC square
for result in results:
    if result is None:
        continue

    corner = result.get('corner', 'unknown')
    bbox = result.get('bbox', None)

    if bbox is None:

```

```

    continue

    x, y, w, h = bbox
    color = corner_colors.get(corner, (255, 255, 255)) # Default to white

    # Draw bounding box rectangle
    rect = patches.Rectangle(
        (x, y), w, h,
        linewidth=3,
        edgecolor=[c/255.0 for c in color],
        facecolor='none'
    )
    ax.add_patch(rect)

    # Add Label if requested
    if show_labels:
        label_text = corner_names.get(corner, corner.upper())

        # Add pattern match info if available
        if result.get('pattern_match'):
            pattern_name = result['pattern_match']
            confidence = result.get('confidence', 0)
            label_text += f'\n{pattern_name}\n{confidence:.1%}'

        # Position label at top-left of bounding box
        # Adjust if too close to image edge
        label_x = x
        label_y = y - 10 if y > 30 else y + h + 10

        ax.text(
            label_x, label_y,
            label_text,
            fontsize=10,
            bbox=dict(
                boxstyle='round,pad=0.5',
                facecolor=[c/255.0 for c in color],
                edgecolor='black',
                alpha=0.7
            ),
            color='white' if corner in ['tl', 'br'] else 'black',
            weight='bold'
        )

    plt.tight_layout()
    return fig

```

```

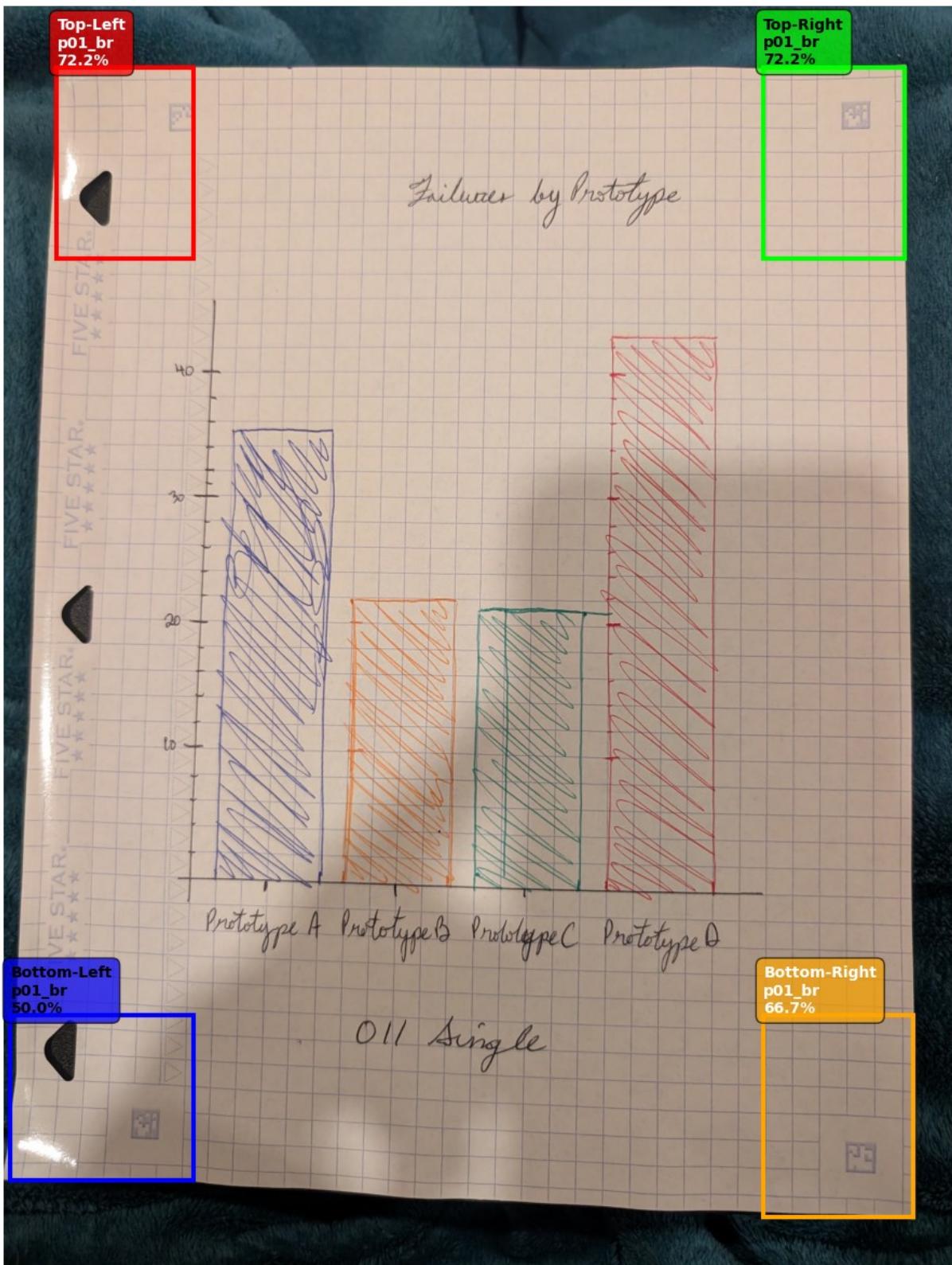
In [12]: # Load image and detect QC squares
image_path = 'hand_drawn_notes/bc_011_single-004.jpg'
image = cv2.imread(image_path)
rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Detect all QC squares
results = detect_qc_square(rgb_image, corner='auto')

# Visualize using matplotlib
fig = visualize_qc_squares(rgb_image, results, show_labels=True)

```

```
plt.show()  
  
# Or save the figure  
# fig.savefig('output_files/qc_detections.png', dpi=150, bbox_inches='tight')
```



Cropping

```
In [13]: def get_page_corners_from_qc(results, allow_diagonalFallback=True):
    """
    Extract page corner coordinates from detected QC squares.

    Parameters:
    -----
    results : list of dicts
        List of detection results from detect_qc_square()
        Should contain 4 results (one for each corner)
    allow_diagonalFallback : bool
        When True (default) the function will synthesize the missing corners
        if only a TL/BR or TR BL diagonal pair is detected.

    Returns:
    -----
    tuple
        (corner dictionary, fallback reason string or None). The dictionary has
        keys 'tl', 'tr', 'bl', 'br'. Returns (None, None) if the required
        corners cannot be computed.
    """
    corners = {}

    for result in results:
        if result is None:
            continue
        corner = result.get('corner')
        bbox = result.get('bbox')

        if corner and bbox:
            x, y, w, h = bbox

            # Use the outer corner of the bounding box (closest to image edge)
            # This represents the corner of the QC square envelope
            if corner == 'tl':
                corners['tl'] = (x, y)
            elif corner == 'tr':
                corners['tr'] = (x + w, y)
            elif corner == 'bl':
                corners['bl'] = (x, y + h)
            elif corner == 'br':
                corners['br'] = (x + w, y + h)

    required_corners = ['tl', 'tr', 'bl', 'br']
    if all(corner in corners for corner in required_corners):
        return corners, None

    def _synthesize_from_tl_br(tl, br):
        return {
            'tl': tl,
            'tr': (br[0], tl[1]),
            'br': br,
            'bl': (tl[0], br[1])
        }

    def _synthesize_from_tr_bl(tr, bl):
        return {
```

```

        'tl': (bl[0], tr[1]),
        'tr': tr,
        'br': (tr[0], bl[1]),
        'bl': bl
    }

    if allow_diagonalFallback:
        if 'tl' in corners and 'br' in corners:
            synthesized = _synthesize_from_tl_br(corners['tl'], corners['br'])
            print("Partial corners detected: using TL/BR diagonal to estimate the missing corners")
            return synthesized, 'diagonal_tl_br'
        if 'tr' in corners and 'bl' in corners:
            synthesized = _synthesize_from_tr_bl(corners['tr'], corners['bl'])
            print("Partial corners detected: using TR BL diagonal to estimate the missing corners")
            return synthesized, 'diagonal_tr_bl'

    missing = [c for c in required_corners if c not in corners]
    print(f"Warning: Missing corners: {missing}")
    return None, None

def crop_image_using_qc_corners(image, results, output_size=None, margin=0, allow_diagonalFallback=False):
    """
    Crop and rectify the image using detected QC square corners.

    Parameters:
    -----
    image : numpy.ndarray
        Original RGB image
    results : list of dicts
        List of detection results from detect_qc_square()
    output_size : tuple, optional
        Desired output size (width, height). If None, calculates from corners.
    margin : int or float
        Margin to add around the page (in pixels or as fraction of page size)
        Default: 0
    allow_diagonalFallback : bool
        When True (default) synthesizes the missing corners when a TL/BR or TR/BL diagonal pair is available so the page can still be rectified.

    Returns:
    -----
    numpy.ndarray
        Cropped and rectified image
    dict
        Metadata including transformation matrix and corner coordinates
    """

    # Get corner coordinates
    corners, fallback_reason = get_page_corners_from_qc(results, allow_diagonalFallback)
    if corners is None:
        raise ValueError("Could not extract all 4 corner points from QC squares")

    # Source points
    src_points = np.array([
        corners['tl'], # Top-left
        corners['tr'], # Top-right

```

```

        corners['br'], # Bottom-right
        corners['bl'], # Bottom-left
    ], dtype=np.float32)

# Calculate destination points
if output_size is None:
    # Calculate output size based on the width and height of the page
    width_top = np.linalg.norm(src_points[1] - src_points[0])
    width_bottom = np.linalg.norm(src_points[2] - src_points[3])
    height_left = np.linalg.norm(src_points[3] - src_points[0])
    height_right = np.linalg.norm(src_points[2] - src_points[1])

    # Use average dimensions
    output_width = int(max(width_top, width_bottom))
    output_height = int(max(height_left, height_right))
else:
    output_width, output_height = output_size

# Apply margin
if isinstance(margin, float):
    margin_x = int(output_width * margin)
    margin_y = int(output_height * margin)
else:
    margin_x = margin_y = margin

output_width += 2 * margin_x
output_height += 2 * margin_y

# Destination points
dst_points = np.array([
    [margin_x, margin_y], # Top-left
    [output_width - margin_x, margin_y], # Top-right
    [output_width - margin_x, output_height - margin_y], # Bottom-right
    [margin_x, output_height - margin_y] # Bottom-left
], dtype=np.float32)

# Get perspective transformation matrix
M = cv2.getPerspectiveTransform(src_points, dst_points)

# Apply perspective transformation
cropped = cv2.warpPerspective(
    image, M,
    (output_width, output_height),
    flags=cv2.INTER_LINEAR,
    borderMode=cv2.BORDER_CONSTANT,
    borderValue=(255, 255, 255) # White background for areas outside page
)

metadata = {
    'transformation_matrix': M,
    'source_corners': corners,
    'output_size': (output_width, output_height),
    'margin': (margin_x, margin_y),
    'corner_source': fallback_reason or 'detected'
}

```

```

    return cropped, metadata

# Alternative: Use center of QC squares instead of corners
def get_page_corners_from_qc_centers(results):
    """
    Extract page corners using the center of each QC square.
    This is useful if you want the page boundary to be at the center of the QC square.
    """
    corners = {}

    for result in results:
        if result is None:
            continue
        corner = result.get('corner')
        bbox = result.get('bbox')

        if corner and bbox:
            x, y, w, h = bbox
            # Use center of bounding box
            center_x = x + w / 2
            center_y = y + h / 2
            corners[corner] = (center_x, center_y)

    required_corners = ['tl', 'tr', 'bl', 'br']
    if all(corner in corners for corner in required_corners):
        return corners
    else:
        missing = [c for c in required_corners if c not in corners]
        print(f"Warning: Missing corners: {missing}")
        return None

# Visualization function to show the crop region
def visualize_crop_region(image, results, show_corners=True):
    """
    Visualize the crop region defined by QC squares.
    """
    corners, _ = get_page_corners_from_qc(results)
    if corners is None:
        print("Cannot visualize: missing corners")
        return None

    # Create a copy to draw on
    display_image = image.copy()

    # Draw Lines connecting corners
    corner_order = ['tl', 'tr', 'br', 'bl', 'tl'] # Close the polygon
    points = [corners[corner] for corner in corner_order]
    points = np.array(points, dtype=np.int32)

    # Draw polygon outline
    cv2.polylines(display_image, [points], isClosed=True,
                  color=(0, 255, 0), thickness=3)

    # Draw corner points
    if show_corners:
        for corner_name, (x, y) in corners.items():

```

```

        cv2.circle(display_image, (int(x), int(y)), 10, (255, 0, 0), -1)
        cv2.putText(display_image, corner_name.upper(),
                    (int(x) + 15, int(y)),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)

    return display_image

```

Test

```
In [14]: def visualize_cropped_result(cropped_image, title="Cropped and Rectified Page", figsize=(12, 15)):
    """
    Visualize the cropped/rectified image result.

    Parameters:
    -----
    cropped_image : numpy.ndarray
        The cropped image result
    title : str
        Title for the plot
    figsize : tuple
        Figure size for matplotlib
    """

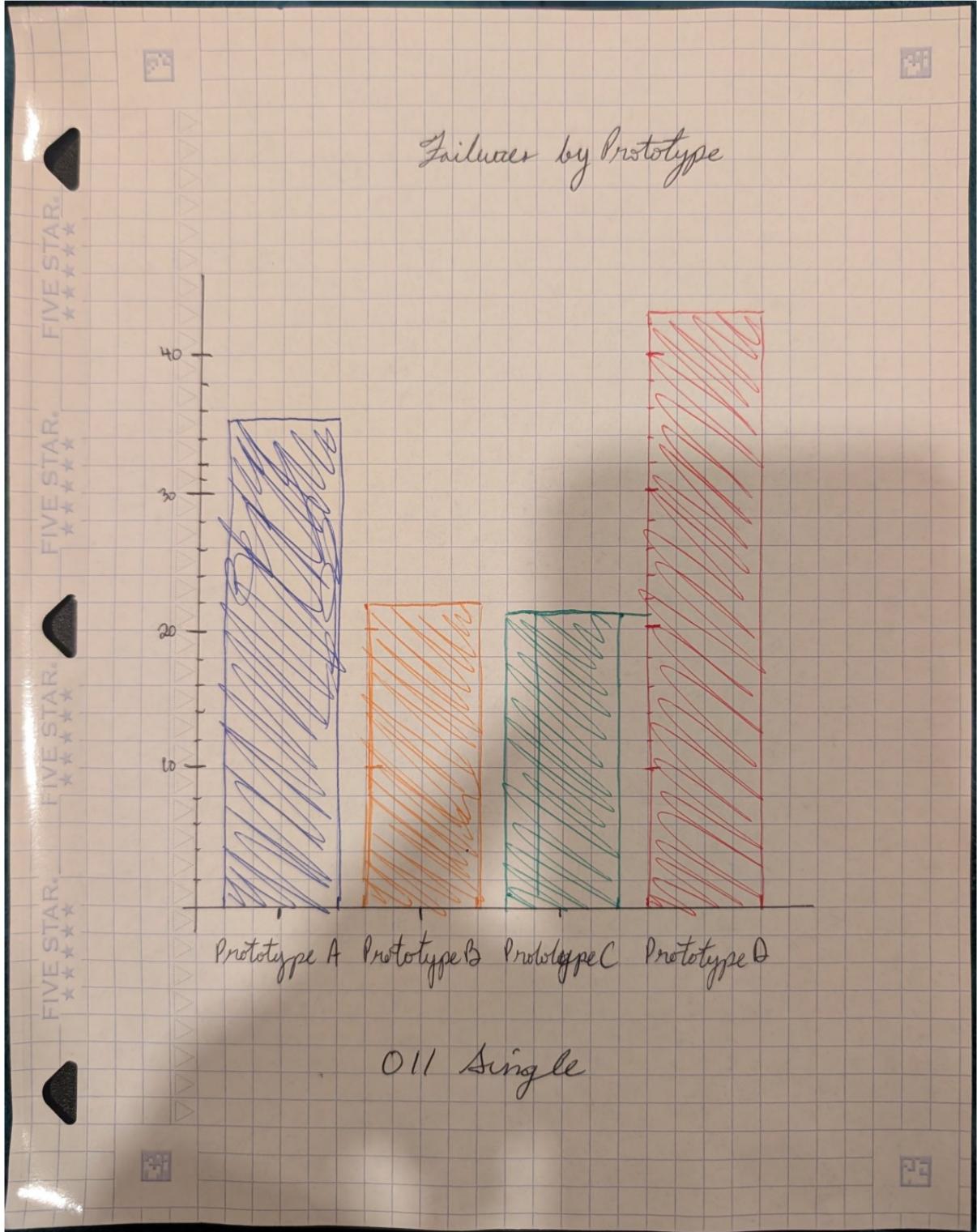
    fig, ax = plt.subplots(1, 1, figsize=figsize)
    ax.imshow(cropped_image)
    ax.set_title(title, fontsize=14)
    ax.axis('off')
    plt.tight_layout()
    return fig
```

```
In [15]: # Load and detect
image_path = 'hand_drawn_notes/bc_011_single-004.jpg'
image = cv2.imread(image_path)
rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
results = detect_qc_square(rgb_image, corner='auto')

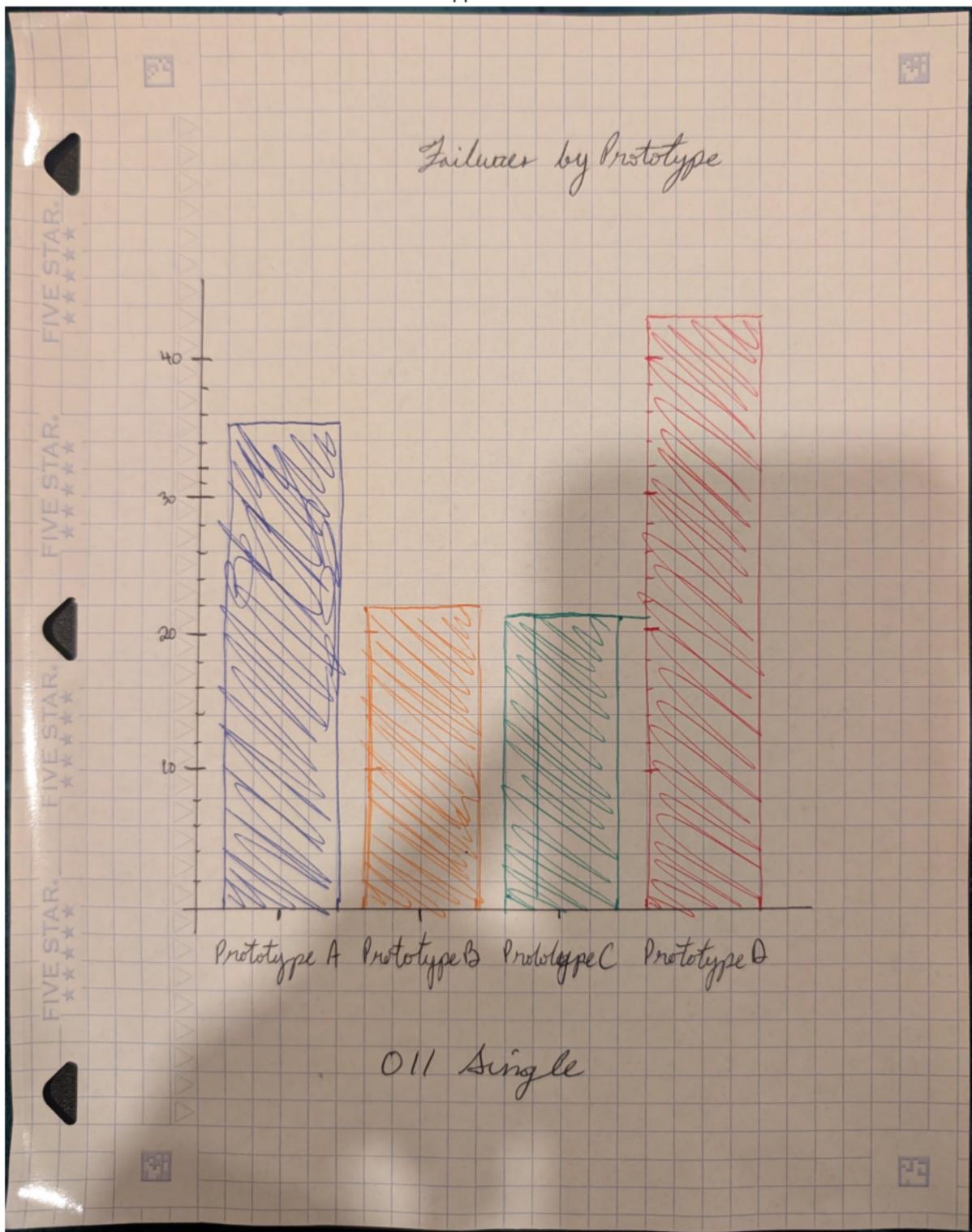
# Crop the image
cropped_image, metadata = crop_image_using_qc_corners(
    rgb_image, results, margin=20
)

# Show the cropped result
fig = visualize_cropped_result(cropped_image)
plt.figure(figsize=(12, 15))
plt.imshow(cropped_image)
plt.title('Cropped Result')
plt.axis('off')
plt.show()

fig.savefig('output_files/bc_011_single_cropped.png', dpi=150, bbox_inches='tight')
```



Cropped Result



Rotated Photo

In [16]: `NotImplementedError`

Out[16]: `NotImplementedError`

Gallery of Processed Images

In [21]: # Gallery of Processed Images

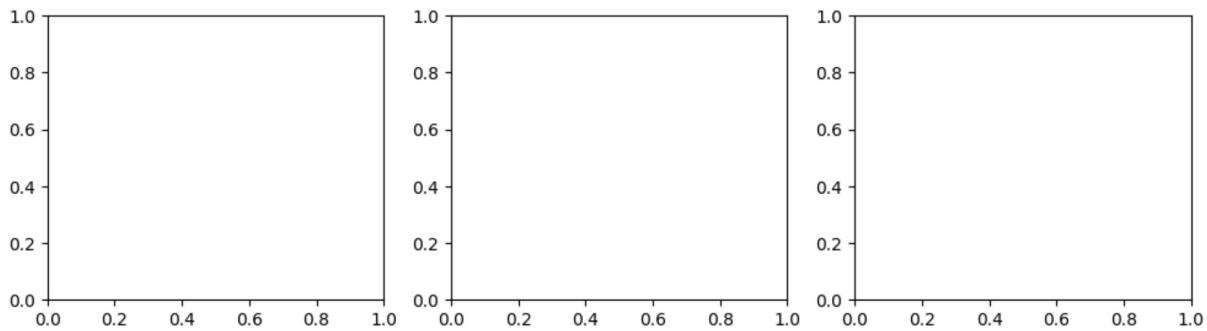
```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
# from normalize_image import detect_qc_square # adjust import path
import cv2
import math

image_paths = [
    ...
] # Selection
ncols = 3
nrows = math.ceil(len(image_paths) / ncols)
fig, axes = plt.subplots(nrows, ncols, figsize=(ncols * 4, nrows * 3))

for ax, path in zip(axes.flat, image_paths):
    image = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)
    results = detect_qc_square(image, corner='auto')
    ax.imshow(image)
    ax.axis('off')
    for result in results or []:
        x, y, w, h = result['bbox']
        color = {'tl': 'red', 'tr': 'green', 'bl': 'blue', 'br': 'orange'}.get(result['corner'])
        rect = patches.Rectangle((x, y), w, h, edgecolor=color, facecolor='none', lw=2)
        ax.add_patch(rect)
        ax.text(x, y - 5, f'{result["corner"]} {result.get("confidence", 0):.0%}', color=color,
                bbox=dict(facecolor=color, alpha=0.6, pad=1), fontsize=8)

for ax in axes.flat[nr]: # len(image_paths):]:
    ax.remove()
plt.tight_layout()
plt.savefig("output_files/qc_corner_gallery.png", dpi=150)
```

```
-----  
error                                                 Traceback (most recent call last)  
Cell In[21], line 19  
     16 fig, axes = plt.subplots(nrows, ncols, figsize=(ncols * 4, nrows * 3))  
     17  
--> 18 for ax, path in zip(axes.flat, image_paths):  
     19     image = cv2.cvtColor(cv2.imread(path), cv2.COLOR_BGR2RGB)  
     20     results = detect_qc_square(image, corner='auto')  
     21     ax.imshow(image)  
  
error: OpenCV(4.12.0) :-1: error: (-5:Bad argument) in function 'imread'  
> Overload resolution failed:  
> - Expected 'filename' to be a str or path-like object  
> - Expected 'filename' to be a str or path-like object  
> - Expected 'filename' to be a str or path-like object
```



In []: