

Selection Bias & Missing Data Challenge - Part 1

Blue Noise Stippling: Creating Art from Data

Selection Bias & Missing Data Challenge - Part 1

! Challenge Requirements

Your Task: Reproduce the blue noise stippling process demonstrated below to create:

1. A stippled version of your chosen image
2. A progressive stippling GIF animation
3. Post both to a GitHub Pages site with appropriate captions and a brief explanation

Part 2 Preview: On November 18th, we'll tackle Part 2 of this challenge, where you'll create a statistical meme about selection bias and missing data using your stippled images.

The Problem: Can Algorithms Create Art?

Core Question: How can we convert a photograph into an aesthetically pleasing pattern of dots that preserves the visual information of the original image?

The Challenge: Blue noise stippling is a technique that converts images into patterns of dots (stipples) using algorithms that balance visual accuracy with spatial distribution. This challenge asks you to implement a modified “void and cluster” algorithm that combines importance sampling with blue noise distribution properties to create stippling patterns that are both visually accurate and spatially well-distributed.

Our Approach: We'll use a modified void-and-cluster algorithm that:

1. Creates an importance map identifying visually important regions
2. Uses a toroidal (periodic) Gaussian kernel for repulsion to ensure blue noise properties
3. Iteratively selects points with minimum energy
4. Balances image content importance with blue noise spatial distribution



AI Partnership Required

This challenge pushes boundaries intentionally. You'll tackle problems that normally require weeks of study, but with Cursor AI as your partner (and your brain keeping it honest), you can accomplish more than you thought possible.

The new reality: The four stages of competence are Ignorance → Awareness → Learning → Mastery. AI lets us produce Mastery-level work while operating primarily in the Awareness stage. I focus on awareness training, you leverage AI for execution, and together we create outputs that used to require years of dedicated study.

Introduction to Blue Noise Stippling

Blue noise stippling is a technique for converting images into a pattern of dots (stipples) that preserves the visual information of the original image while creating an aesthetically pleasing, evenly distributed pattern. This method follows the approach described by [Bart Wronski](#).

The method uses a modified “void and cluster” algorithm that combines importance sampling with blue noise distribution properties to create stippling patterns that are both visually accurate and spatially well-distributed. This version uses **smooth extreme downweighting** that selectively downweights very dark and very light tones while preserving mid-tones, creating a more balanced distribution of stipples across the image.

Loading the Original Image

First, let's load an image that we'll convert to a blue noise stippling pattern. You can use any image you'd like, but we'll demonstrate with the provided example.

Python

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Load the image
img_path = 'fleischhacker.jpg'
original_img = Image.open(img_path)

# Convert to grayscale if needed
if original_img.mode != 'L':
    original_img = original_img.convert('L')
```

```
# Convert to numpy array and normalize to [0, 1]
img_array = np.array(original_img, dtype=np.float32) / 255.0

# Display the original image
fig, ax = plt.subplots(figsize=(6.5, 5))
ax.imshow(img_array, cmap='gray', vmin=0, vmax=1)
ax.axis('off')

(np.float64(-0.5), np.float64(249.5), np.float64(374.5), np.float64(-0.5))

ax.set_title('Original Image', fontsize=14, fontweight='bold', pad=10)
plt.tight_layout()
plt.show()
```

Original Image



Figure 1: Original image before stippling

```
print(f"Image shape: {img_array.shape}")
```

Image shape: (375, 250)

```
print(f"Image size: {img_array.shape[0]}x{img_array.shape[1]} pixels")
```

Image size: 375x250 pixels

R

```
library(imager)
```

Warning: package 'imager' was built under R version 4.5.2

Loading required package: magrittr

Attaching package: 'imager'

The following object is masked from 'package:magrittr':

add

The following objects are masked from 'package:stats':

convolve, spectrum

The following object is masked from 'package:graphics':

frame

The following object is masked from 'package:base':

save.image

```

library(ggplot2)

# Load the image
img_path <- 'fleischhacker.jpg'
original_img <- load.image(img_path)

# Convert to grayscale if needed
# Check channels (dim[3]), not depth (dim[4])
if(dim(original_img)[3] == 3) {
  original_img <- grayscale(original_img)
}

# Display the original image
plot(original_img, axes = FALSE, main = "Original Image")

```

Original Image



Figure 2: Original image before stippling

```

cat("Image dimensions:", dim(original_img)[1], "x", dim(original_img)[2], "pixels\n")

```

Image dimensions: 250 x 375 pixels

Importance Mapping

Before applying the stippling algorithm, we create an **importance map** that identifies which regions of the image should receive more stipples. The importance map is computed by:

- **Brightness inversion:** The image brightness is inverted so that dark areas receive higher importance and thus more dots, while light areas receive fewer dots
- **Extreme tone downweighting:** Smooth Gaussian functions downweight tones below 0.2 (very dark) and above 0.8 (very light), creating a gradual transition that preserves mid-tones
- **Mid-tone boost:** A smooth Gaussian function centered on mid-tones provides a gradual increase in importance for mid-tone regions, ensuring they receive appropriate stippling density
- **Selective and effective:** This approach ensures that stipples are distributed appropriately (more dots in dark areas and mid-tones, fewer in extreme dark/light areas) while maintaining good spatial distribution

! Key Tuning Point: `compute_importance` Function

This function is all you need to control dot distribution! The `compute_importance` function is the primary mechanism for tuning where more or fewer dots appear in your stippled image. By adjusting its parameters, you can:

- **Control which tones get more dots:** Adjust `extreme_threshold_low` and `extreme_threshold_high` to define what counts as “extreme” tones
- **Reduce dots in extreme regions:** Increase `extreme_downweight` (0.0-1.0) to reduce stipples in very dark or very light areas
- **Boost specific tone ranges:** Adjust `mid_tone_boost` and `mid_tone_center` to emphasize particular brightness ranges (e.g., skin tones around 0.7)
- **Control transition smoothness:** Modify `extreme_sigma` and `mid_tone_sigma` to make transitions sharper or more gradual

No need to modify the stippling algorithm itself—all the tuning happens in this function. Experiment with different parameter values to achieve your desired dot distribution!

Python

```
def compute_importance(  
    gray_img: np.ndarray,  
    extreme_downweight: float = 0.5,  
    extreme_threshold_low: float = 0.4,
```

```

    extreme_threshold_high: float = 0.8,
    extreme_sigma: float = 0.1,
    mid_tone_boost: float = 0.4,
    mid_tone_sigma: float = 0.2,
):
"""
Importance map computation that downweights extreme tones (very dark and very light)
using smooth functions, while boosting mid-tones.

Parameters
-----
gray_img : np.ndarray
    Grayscale image in [0, 1]
extreme_downweight : float
    Strength of downweighting for extreme tones (0.0 = no downweighting, 1.0 = maximum downweighting)
extreme_threshold_low : float
    Threshold below which tones are considered "very dark" and get downweighted
extreme_threshold_high : float
    Threshold above which tones are considered "very light" and get downweighted
extreme_sigma : float
    Width of the smooth transition for extreme downweighting (smaller = sharper transition)
mid_tone_boost : float
    Strength of mid-tone emphasis (0.0 = no boost, 1.0 = strong boost)
mid_tone_sigma : float
    Width of the mid-tone Gaussian bump (smaller = narrower, larger = wider)

Returns
-----
importance : np.ndarray
    Importance map in [0, 1]; higher = more stippling (dark areas and mid-tones get higher importance)
"""

I = np.clip(gray_img, 0.0, 1.0)

# Invert brightness: dark areas should get more dots (higher importance)
I_inverted = 1.0 - I

# Create smooth downweighting mask for extreme tones
# Downweight very dark regions (I < extreme_threshold_low)
dark_mask = np.exp(-((I - 0.0) ** 2) / (2.0 * (extreme_sigma ** 2)))
dark_mask = np.where(I < extreme_threshold_low, dark_mask, 0.0)
if dark_mask.max() > 0:
    dark_mask = dark_mask / dark_mask.max()

```

```

# Downweight very light regions (I > extreme_threshold_high)
light_mask = np.exp(-((I - 1.0) ** 2) / (2.0 * (extreme_sigma ** 2)))
light_mask = np.where(I > extreme_threshold_high, light_mask, 0.0)
if light_mask.max() > 0:
    light_mask = light_mask / light_mask.max()

# Combine both masks
extreme_mask = np.maximum(dark_mask, light_mask)

# Apply smooth downweighting
importance = I_inverted * (1.0 - extreme_downweight * extreme_mask)

# Add smooth gradual mid-tone boost (Gaussian centered on 0.65)
mid_tone_center = 0.65
mid_tone_gaussian = np.exp(-((I - mid_tone_center) ** 2) / (2.0 * (mid_tone_sigma ** 2)))
if mid_tone_gaussian.max() > 0:
    mid_tone_gaussian = mid_tone_gaussian / mid_tone_gaussian.max()

# Boost importance in mid-tone regions
importance = importance * (1.0 + mid_tone_boost * mid_tone_gaussian)

# Normalize to [0,1]
m, M = importance.min(), importance.max()
if M > m:
    importance = (importance - m) / (M - m)
return importance

```

R

```

compute_importance <- function(gray_img,
                                extreme_downweight = 0.5,
                                extreme_threshold_low = 0.4,
                                extreme_threshold_high = 0.8,
                                extreme_sigma = 0.1,
                                mid_tone_boost = 0.4,
                                mid_tone_sigma = 0.2) {

# Clip image to [0, 1]
I <- pmax(pmin(gray_img, 1.0), 0.0)

# Invert brightness

```

```

I_inverted <- 1.0 - I

# Dark mask
dark_mask <- exp(-((I - 0.0)^2) / (2.0 * (extreme_sigma^2)))
dark_mask[I >= extreme_threshold_low] <- 0.0
if(max(dark_mask) > 0) {
    dark_mask <- dark_mask / max(dark_mask)
}

# Light mask
light_mask <- exp(-((I - 1.0)^2) / (2.0 * (extreme_sigma^2)))
light_mask[I <= extreme_threshold_high] <- 0.0
if(max(light_mask) > 0) {
    light_mask <- light_mask / max(light_mask)
}

# Combine masks
extreme_mask <- pmax(dark_mask, light_mask)

# Apply downweighting
importance <- I_inverted * (1.0 - extreme_downweight * extreme_mask)

# Mid-tone boost
mid_tone_center <- 0.65
mid_tone_gaussian <- exp(-((I - mid_tone_center)^2) / (2.0 * (mid_tone_sigma^2)))
if(max(mid_tone_gaussian) > 0) {
    mid_tone_gaussian <- mid_tone_gaussian / max(mid_tone_gaussian)
}

importance <- importance * (1.0 + mid_tone_boost * mid_tone_gaussian)

# Normalize
m <- min(importance)
M <- max(importance)
if(M > m) {
    importance <- (importance - m) / (M - m)
}

return(importance)
}

```

Blue Noise Stippling Algorithm

The stippling algorithm uses a modified void-and-cluster approach that:

1. Creates an importance map that identifies visually important regions
2. Initializes an energy field based on the importance map (higher importance → lower energy)
3. Uses a toroidal (periodic) Gaussian kernel for repulsion to ensure blue noise properties
4. Iteratively selects points with minimum energy
5. Adds Gaussian “splats” around selected points to prevent clustering
6. Balances image content importance with blue noise spatial distribution

Python

```
import numpy as np

def toroidal_gaussian_kernel(h: int, w: int, sigma: float):
    """
    Create a periodic (toroidal) 2D Gaussian kernel centered at (0,0).
    The toroidal property means the kernel wraps around at the edges,
    ensuring consistent repulsion behavior regardless of point location.
    """
    y = np.arange(h)
    x = np.arange(w)
    # Compute toroidal distances (minimum distance considering wrapping)
    dy = np.minimum(y, h - y)[:, None]
    dx = np.minimum(x, w - x)[None, :]
    # Compute Gaussian
    kern = np.exp(-(dx**2 + dy**2) / (2.0 * sigma**2))
    s = kern.sum()
    if s > 0:
        kern /= s  # Normalize
    return kern

def void_and_cluster(
    input_img: np.ndarray,
    percentage: float = 0.08,
    sigma: float = 0.9,
    content_bias: float = 0.9,
    importance_img: np.ndarray | None = None,
    noise_scale_factor: float = 0.1,
```

```

):
    """
    Generate blue noise stippling pattern from input image using a modified
    void-and-cluster algorithm with content-weighted importance.
    """
    I = np.clip(input_img, 0.0, 1.0)
    h, w = I.shape

    # Compute or use provided importance map
    if importance_img is None:
        importance = compute_importance(I)
    else:
        importance = np.clip(importance_img, 0.0, 1.0)

    # Create toroidal Gaussian kernel for repulsion
    kernel = toroidal_gaussian_kernel(h, w, sigma)

    # Initialize energy field: lower energy → more likely to be picked
    energy_current = -importance * content_bias

    # Stipple buffer: start with white background; selected points become black dots
    final_stipple = np.ones_like(I)
    samples = []

    # Helper function to roll kernel to an arbitrary position
    def energy_splat(y, x):
        """Get energy contribution by rolling the kernel to position (y, x)."""
        return np.roll(np.roll(kernel, shift=y, axis=0), shift=x, axis=1)

    # Number of points to select
    num_points = int(I.size * percentage)

    # Choose first point near center with minimal energy
    cy, cx = h // 2, w // 2
    r = min(20, h // 10, w // 10)
    ys = slice(max(0, cy - r), min(h, cy + r))
    xs = slice(max(0, cx - r), min(w, cx + r))
    region = energy_current[ys, xs]
    flat = np.argmin(region)
    y0 = flat // (region.shape[1]) + (cy - r)
    x0 = flat % (region.shape[1]) + (cx - r)

```

```

# Place first point
energy_current = energy_current + energy_splat(y0, x0)
energy_current[y0, x0] = np.inf # Prevent reselection
samples.append((y0, x0, I[y0, x0]))
final_stipple[y0, x0] = 0.0 # Black dot

# Iteratively place remaining points
for i in range(1, num_points):
    # Add exploration noise that decreases over time
    exploration = 1.0 - (i / num_points) * 0.5 # Decrease from 1.0 to 0.5
    noise = np.random.normal(0.0, noise_scale_factor * content_bias * exploration, size=(w, w))
    energy_with_noise = energy_current + noise

    # Find position with minimum energy (with noise for exploration)
    pos_flat = np.argmin(energy_with_noise)
    y = pos_flat // w
    x = pos_flat % w

    # Add Gaussian splat to prevent nearby points from being selected
    energy_current = energy_current + energy_splat(y, x)
    energy_current[y, x] = np.inf # Prevent reselection

    # Record the sample
    samples.append((y, x, I[y, x]))
    final_stipple[y, x] = 0.0 # Black dot

return final_stipple, np.array(samples)

```

R

```

# Note: R implementation uses true 2D circular shift (roll2d) to match Python's np.roll behavior

toroidal_gaussian_kernel <- function(h, w, sigma) {
  y <- 0:(h-1)
  x <- 0:(w-1)

  # Compute toroidal distances
  dy <- pmin(y, h - y)
  dx <- pmin(x, w - x)

  # Create distance matrices
  dy_matrix <- outer(dy, dy)
  dx_matrix <- outer(dx, dx)
  distance_matrix <- sqrt(dy_matrix^2 + dx_matrix^2)
  distance_matrix <- distance_matrix / sigma
  distance_matrix <- exp(-distance_matrix^2 / 2)
}
```

```

# each row gets same dy, each col gets same dx
dy_mat <- matrix(rep(dy, each = w), nrow = h, ncol = w)
dx_mat <- matrix(rep(dx, times = h), nrow = h, ncol = w)

# Compute Gaussian
kern <- exp(-(dx_mat^2 + dy_mat^2) / (2.0 * sigma^2))
kern <- kern / sum(kern) # Normalize

return(kern)
}

# Helper function for true 2D circular shift (equivalent to np.roll)
# np.roll equivalent: positive shift moves down/right
roll2d <- function(mat, shift_y = 0, shift_x = 0) {
  h <- nrow(mat)
  w <- ncol(mat)
  sy <- ((shift_y %% h) + h) %% h
  sx <- ((shift_x %% w) + w) %% w
  rows <- if (sy == 0) 1:h else c((h - sy + 1):h, 1:(h - sy))
  cols <- if (sx == 0) 1:w else c((w - sx + 1):w, 1:(w - sx))
  mat[rows, cols, drop = FALSE]
}

void_and_cluster <- function(input_img,
                               percentage = 0.08,
                               sigma = 0.9,
                               content_bias = 0.9,
                               importance_img = NULL,
                               noise_scale_factor = 0.1) {

  # Clip image to [0, 1]
  I <- pmax(pmin(input_img, 1.0), 0.0)
  h <- nrow(I)
  w <- ncol(I)

  # Compute or use provided importance map
  if(is.null(importance_img)) {
    importance <- compute_importance(I)
  } else {
    importance <- pmax(pmin(importance_img, 1.0), 0.0)
  }

  # Create toroidal Gaussian kernel for repulsion
}

```

```

kernel <- toroidal_gaussian_kernel(h, w, sigma)

# Initialize energy field: lower energy → more likely to be picked
energy_current <- -importance * content_bias

# Stipple buffer: start with white background; selected points become black dots
final_stipple <- matrix(1.0, nrow = h, ncol = w)
samples <- vector("list", as.integer(h * w * percentage))

# Helper function to roll kernel to an arbitrary position
# This implements the exact equivalent of Python's np.roll(kernel, shift=y, axis=0)
# followed by np.roll(kernel, shift=x, axis=1)
# Uses true 2D circular shift to preserve directionality and avoid quadrant-mirroring
energy_splat <- function(y, x) {
  # exact np.roll(kernel, y; x)
  roll2d(kernel, shift_y = y - 1, shift_x = x - 1)
}

# Number of points to select
num_points <- as.integer(h * w * percentage)

# Set seed for reproducibility
set.seed(42)

# --- first point: pick min in a center window ---
cy <- as.integer(h / 2)
cx <- as.integer(w / 2)
r <- min(20L, as.integer(h / 10), as.integer(w / 10))
y_start <- max(1L, cy - r)
y_end <- min(h, cy + r)
x_start <- max(1L, cx - r)
x_end <- min(w, cx + r)

region <- energy_current[y_start:y_end, x_start:x_end]
flat_idx <- which.min(region)

# IMPORTANT: R is column-major → (y,x) from flat:
# y = ((idx-1) %% nrow) + 1 ; x = ((idx-1) %/% nrow) + 1
ry <- ((flat_idx - 1) %% nrow(region)) + 1
rx <- ((flat_idx - 1) %/% nrow(region)) + 1
y0 <- y_start + ry - 1
x0 <- x_start + rx - 1

```

```

# Place first point
energy_current <- energy_current + energy_splat(y0, x0)
energy_current[y0, x0] <- Inf # Prevent reselection
samples[[1]] <- c(y0, x0, I[y0, x0])
final_stipple[y0, x0] <- 0.0 # Black dot

# --- iterate ---
for(i in 2:num_points) {
    # Add exploration noise that decreases over time
    exploration <- 1.0 - ((i - 1) / num_points) * 0.5 # Decrease from 1.0 to 0.5
    noise <- matrix(rnorm(h * w, 0, noise_scale_factor * content_bias * exploration),
                     nrow = h, ncol = w)
    energy_with_noise <- energy_current + noise

    # Find position with minimum energy (with noise for exploration)
    pos_flat <- which.min(energy_with_noise)

    # Column-major unflatten:
    # R matrices are column-major, so y = (idx-1) %% h + 1, x = (idx-1) %/% h + 1
    y <- ((pos_flat - 1) %% h) + 1
    x <- ((pos_flat - 1) %/% h) + 1

    # Add Gaussian splat to prevent nearby points from being selected
    energy_current <- energy_current + energy_splat(y, x)
    energy_current[y, x] <- Inf # Prevent reselection

    # Record the sample
    samples[[i]] <- c(y, x, I[y, x])
    final_stipple[y, x] <- 0.0 # Black dot
}

# Convert samples list to matrix
samples_matrix <- do.call(rbind, samples[1:num_points])

return(list(stipple = final_stipple, samples = samples_matrix))
}

```

Preparing the Working Image

Before generating the stippling pattern, we prepare the image by resizing if necessary and computing the importance map.

Python

```
# Resize image if it's too large for faster processing
max_size = 512
if img_array.shape[0] > max_size or img_array.shape[1] > max_size:
    scale = max_size / max(img_array.shape[0], img_array.shape[1])
    new_size = (int(img_array.shape[1] * scale), int(img_array.shape[0] * scale))
    img_resized_pil = original_img.resize(new_size, Image.Resampling.LANCZOS)
    if img_resized_pil.mode != 'L':
        img_resized_pil = img_resized_pil.convert('L')
    img_resized = np.array(img_resized_pil, dtype=np.float32) / 255.0
    print(f"Resized image from {img_array.shape} to {img_resized.shape} for processing")
else:
    img_resized = img_array.copy()

# Ensure img_resized is 2D grayscale
if len(img_resized.shape) > 2:
    img_resized = img_resized[:, :, 0]
elif len(img_resized.shape) == 2:
    pass
else:
    raise ValueError(f"Unexpected image shape: {img_resized.shape}")

print(f"Final image shape: {img_resized.shape} (should be 2D for grayscale)")
```

Final image shape: (375, 250) (should be 2D for grayscale)

```
# Compute importance map using default parameters
importance_map = compute_importance(
    img_resized,
    extreme_downweight=0.5,
    extreme_threshold_low=0.2,
    extreme_threshold_high=0.8,
    extreme_sigma=0.1
)
print("Importance map computed")
```

Importance map computed

R

```
# Resize image if needed
max_size <- 512
img_dims <- dim(original_img)
if(img_dims[1] > max_size || img_dims[2] > max_size) {
  scale <- max_size / max(img_dims[1], img_dims[2])
  new_size <- c(round(img_dims[1] * scale), round(img_dims[2] * scale))
  img_resized <- resize(original_img, new_size[1], new_size[2])
  cat("Resized image to", new_size[1], "x", new_size[2], "for processing\n")
} else {
  img_resized <- original_img
}

# Convert to matrix and normalize
# imager uses (x,y,cc,z) = (width, height, channels, depth)
# but R matrices use (row, col) = (y, x)
# Extract grayscale plane and transpose to get [h, w] orientation
arr <- as.array(img_resized)           # dims: [x, y, cc, z] = [w, h, 1, 1]
img_matrix <- t(arr[, , 1, 1])        # now dims: [h, w]  (row=y, col=x)
img_matrix <- img_matrix / max(img_matrix)

cat("Final image shape:", nrow(img_matrix), "x", ncol(img_matrix), "\n")
```

Final image shape: 375 x 250

```
# Compute importance map
importance_map <- compute_importance(img_matrix)
cat("Importance map computed\n")
```

Importance map computed

Generating the Stippled Image

Now let's apply the stippling algorithm to create the blue noise stippling pattern.

Python

```
# Generate stippling pattern
print("Generating blue noise stippling pattern...")
```

Generating blue noise stippling pattern...

```
stipple_pattern, samples = void_and_cluster(
    img_resized,
    percentage=0.08,
    sigma=0.9,
    content_bias=0.9,
    importance_img=importance_map,
    noise_scale_factor=0.1
)
```

```
print(f"Generated {len(samples)} stipple points")
```

Generated 7500 stipple points

```
print(f"Stipple pattern shape: {stipple_pattern.shape}")
```

Stipple pattern shape: (375, 250)

R

```
# Generate stippling pattern
cat("Generating blue noise stippling pattern...\n")
```

Generating blue noise stippling pattern...

```
stipple_result <- void_and_cluster(
    img_matrix,
    percentage = 0.08,
    sigma = 0.9,
    content_bias = 0.9,
    importance_img = importance_map,
    noise_scale_factor = 0.1
)
```

```

stipple_pattern <- stipple_result$stipple
samples <- stipple_result$samples

cat("Generated", nrow(samples), "stipple points\n")

Generated 7500 stipple points

cat("Stipple pattern shape:", nrow(stipple_pattern), "x", ncol(stipple_pattern), "\n")

```

Stipple pattern shape: 375 x 250

Displaying the Results

Let's visualize the original image, the importance map, and the stippled version side by side for comparison.

Python

```

fig, axes = plt.subplots(1, 3, figsize=(7, 4))

# Display original image
axes[0].imshow(img_resized, cmap='gray', vmin=0, vmax=1)
axes[0].axis('off')

(np.float64(-0.5), np.float64(249.5), np.float64(374.5), np.float64(-0.5))

axes[0].set_title('Original Image', fontsize=14, fontweight='bold', pad=10)

# Display importance map
axes[1].imshow(importance_map, cmap='gray', vmin=0, vmax=1)
axes[1].axis('off')

(np.float64(-0.5), np.float64(249.5), np.float64(374.5), np.float64(-0.5))

```

```

axes[1].set_title('Importance Map', fontsize=14, fontweight='bold', pad=10)

# Display stippled image
axes[2].imshow(stipple_pattern, cmap='gray', vmin=0, vmax=1)
axes[2].axis('off')

(np.float64(-0.5), np.float64(249.5), np.float64(374.5), np.float64(-0.5))

axes[2].set_title('Blue Noise Stippling', fontsize=14, fontweight='bold', pad=10)

plt.tight_layout()
plt.show()

```

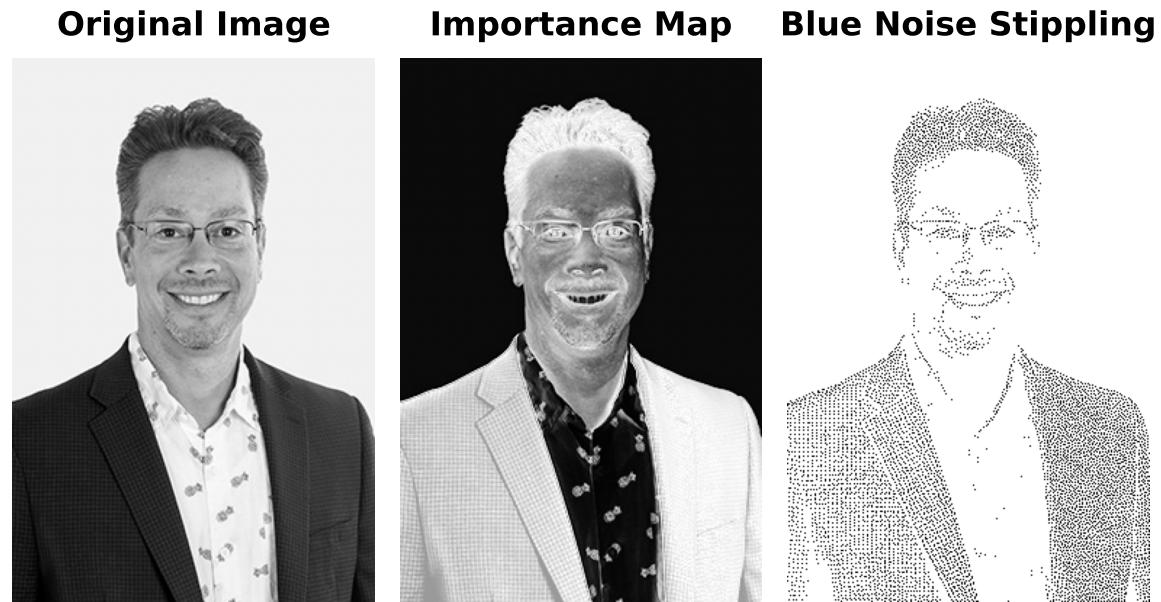


Figure 3: Comparison of original image, importance map, and blue noise stippling

R

```

par(mfrow = c(1, 3), mar = c(2, 2, 2, 2))

# Original

```

```

plot(img_resized, axes = FALSE, main = "Original Image")

# Importance map (matrix [h, w]) -> transpose for imager ([x, y])
plot(as.cimg(t(importance_map), x = ncol(importance_map), y = nrow(importance_map), cc = 1),
     axes = FALSE, main = "Importance Map")

# Stipple pattern (matrix [h, w]) -> transpose for imager ([x, y])
plot(as.cimg(t(stipple_pattern), x = ncol(stipple_pattern), y = nrow(stipple_pattern), cc = 1),
     axes = FALSE, main = "Blue Noise Stippling")

```

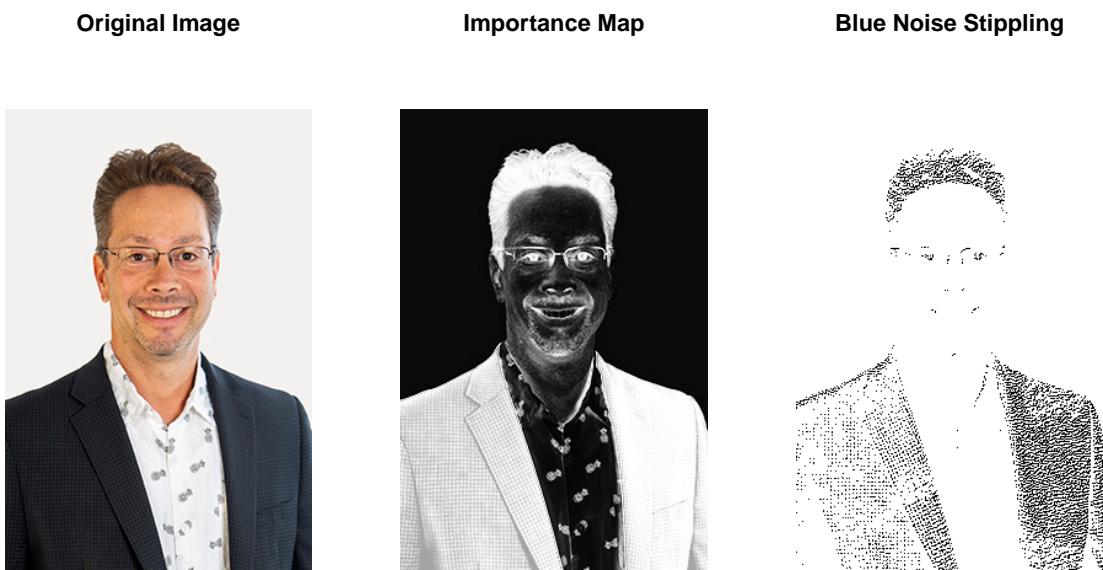


Figure 4: Comparison of original image, importance map, and blue noise stippling

Progressive Stippling Animation

This section creates a GIF showing how the stippled image looks as more points are added sequentially. We'll use the already-computed stippling points to generate frames at increments of 100 points.

Python

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from matplotlib.animation import PillowWriter

# Use the existing samples array from the already-computed stippling
print(f"Using existing stippling with {len(samples)} points")
```

Using existing stippling with 7500 points

```
print(f"Image shape: {img_resized.shape}")
```

Image shape: (375, 250)

```
# Create progressive frames by adding points sequentially
frame_increment = 100
frames = []
point_counts = []

# Start with white background
h, w = img_resized.shape
progressive_stipple = np.ones_like(img_resized)

# Add first point and save initial frame
if len(samples) > 0:
    y0, x0, intensity0 = int(samples[0, 0]), int(samples[0, 1]), samples[0, 2]
    progressive_stipple[y0, x0] = 0.0
    frames.append(progressive_stipple.copy())
    point_counts.append(1)

# Add remaining points sequentially and save frames at increments
for i in range(1, len(samples)):
    y, x = int(samples[i, 0]), int(samples[i, 1])
    progressive_stipple[y, x] = 0.0 # Add black dot

    # Save frame at increments (100, 200, 300, ...) and at the end
    if (i + 1) % frame_increment == 0 or i == len(samples) - 1:
        frames.append(progressive_stipple.copy())
```

```
    point_counts.append(i + 1)

print(f"Generated {len(frames)} frames")
```

Generated 76 frames

```
print(f"Point counts: {point_counts}")
```

Point counts: [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400,

R

```
# Use the existing samples from the already-computed stippling
cat("Using existing stippling with", nrow(samples), "points\n")
```

Using existing stippling with 7500 points

```
cat("Image shape:", nrow(img_matrix), "x", ncol(img_matrix), "\n")
```

Image shape: 375 x 250

```
# Create progressive frames by adding points sequentially
frame_increment <- 100
frames <- list()
point_counts <- c()

# Start with white background
progressive_stipple <- matrix(1.0, nrow = nrow(img_matrix), ncol = ncol(img_matrix))

# Add first point and save initial frame
if(nrow(samples) > 0) {
  y0 <- as.integer(samples[1, 1])
  x0 <- as.integer(samples[1, 2])
  progressive_stipple[y0, x0] <- 0.0
  frames[[1]] <- progressive_stipple
  point_counts <- c(point_counts, 1)
}
```