Faculty of Engineering
Systems & Biomedical Engineering Department

Computer Vision-Task2

A Third Year Systems & Biomedical Engineering Task Report

Done by **Team 5**

Name	sec	BN
Ibrahim Mohamed	1	2
Omnia Sayed	1	14
Marina Nasser	2	12
Mahmoud Yaser	2	30
Maye Khaled	2	40

Submitted in Partial Fulfillment of the Requirements for
Computer Vision (SBE3230)

Submitted to:
Eng. Peter Emad
Eng. Laila Abbas

March 2023

Hough Transform

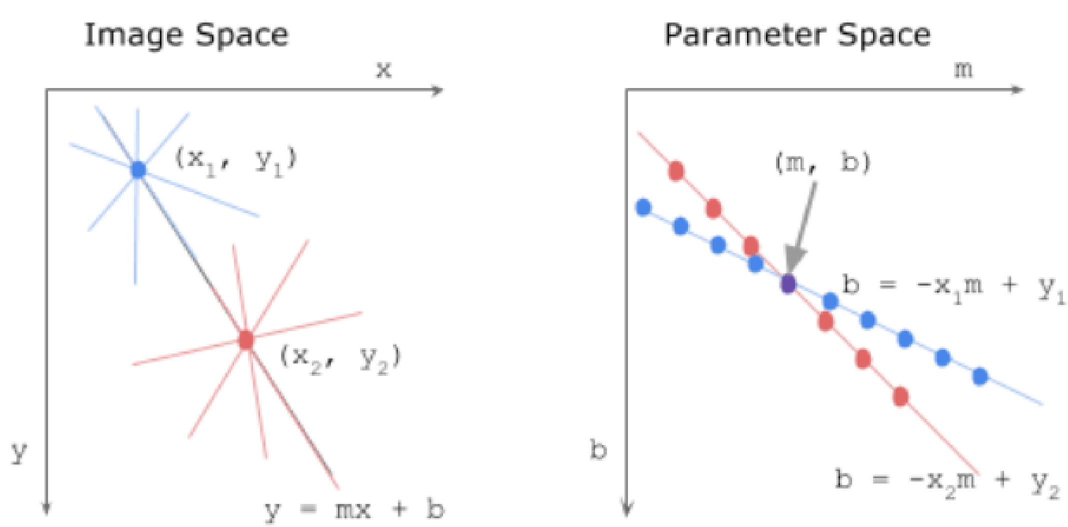
1. Line

- Original Hough transform (Cartesian Coordinates)

In image space line is defined by the slope m and the y-intercept b **:**

$$y=mx+b$$

So to detect the line in the image space we have to define these parameters, which is not applicable in the image domain. In the other domain with m and b coordinates, lines represent a point from the image domain. Points on the same line in the image domain will be mapped to lines in the Hough domain. These lines intersect with each other in a point with specific values m and b.



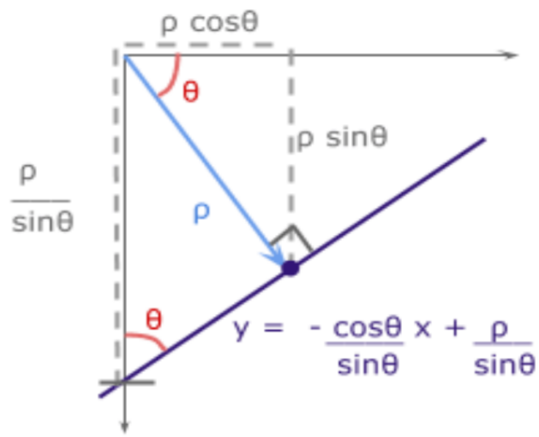
- Alternative Parameter Space (Polar Coordinates)

Due to undefined value of slope for vertical lines in cartesian coordinates, we have to move to polar coordinates. In polar coordinates line is define by ρ and θ where ρ is the norm distance of the line from origin. θ is the angle between the norm and the horizontal x axis.

$$y = \frac{-cos(\theta)}{sin(\theta)}x + \frac{\rho}{sin(\theta)}$$

and

$$\rho = xcos(\theta) + ysin(\theta)$$



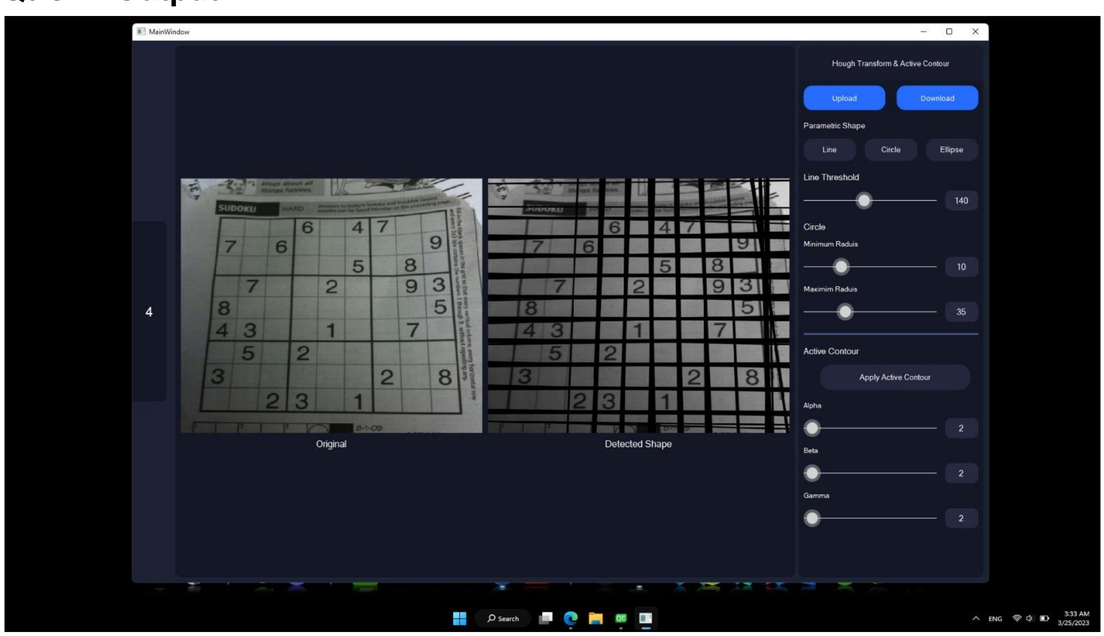
The Range of values of ρ and θ:

- θ in polar coordinate takes value in range of -90 to 90
- The maximum norm distance is given by diagonal distance which is

$$\rho_{max} = \sqrt{x^2 + y^2}$$

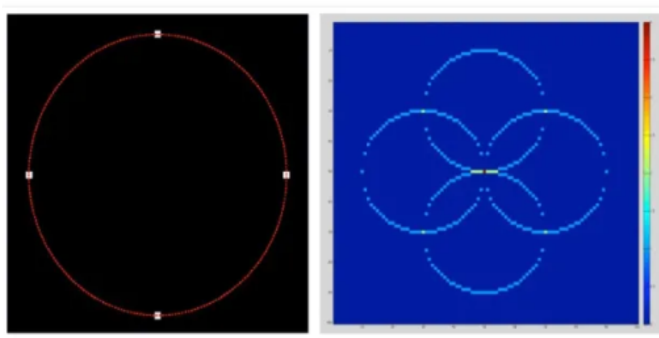
So ρ has values in range from -ρmax to ρmax

Qt C++ Output



2. Hough Circle Transform

- We know that a circle can be represented as (x-a)2 + (y-b)2 = r2 where a, b represents the circle center and r is the radius. So, we require 3 parameters (a,b,r) to completely describe the circle.
- we will first draw the circles in the ab space corresponding to each edge point. Then we will find the point of intersection (actually the local maxima in the accumulator array) which will correspond to the original circle center.

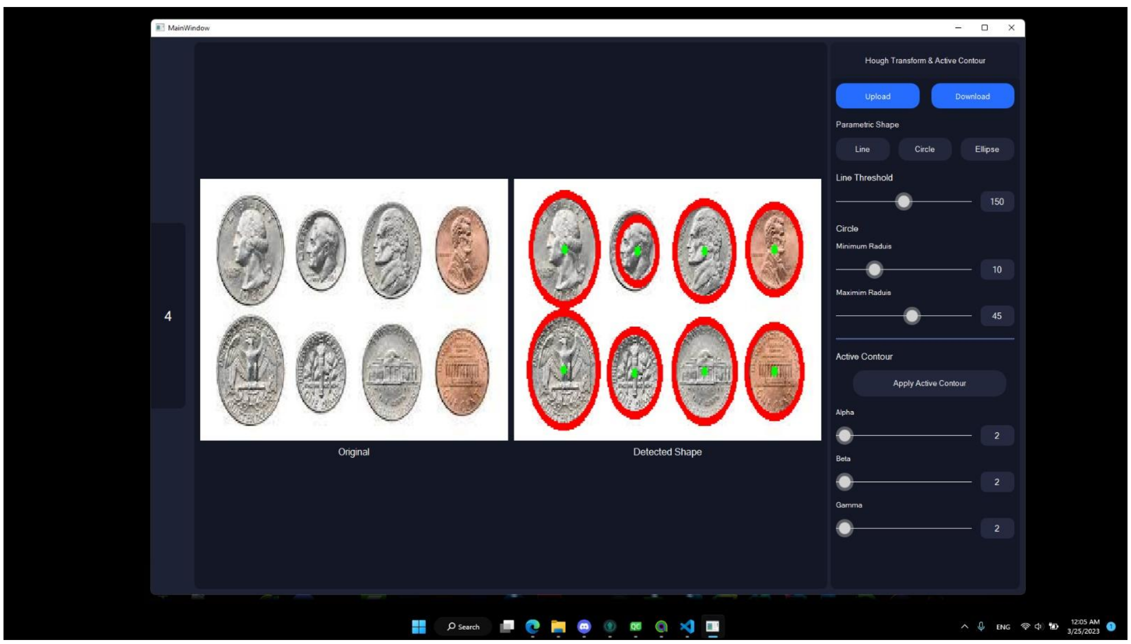


- ❖ The image is converted to grayscale
- ❖ The image is blurred to reduce noise
- ❖ The edge image is calculated using the Canny edge detector.
- ❖ Create the accumulator by looping over the edge image and create center points in the accumulator for candidate for pixels with intestines greater than a specific intensity
- ❖ Looping over the accumulator pixels and storing pixel intensity values in with corresponding pixel values in an array of structs(consists of x-pixel, y-pixel, intensity) ,only if the pixel value is greater than a certain threshold we take it as a candidate center.
- ❖ We call the accumulator through the range of a specific radius values (min & max).
- ❖ For the parameters included in the algorithm

    1. We can roughly estimate minRadius and maxRadius by calculating the number of circles that can fit on the horizontal.
    2. If our circles are next to each other touching each other, we can give minDist twice the value of minRadius.
    3. If our circles are far apart, we need to give higher values.

4. If we detected less circle than we should, we can try to lower the param2.

Qt C++ Output



3. Ellipse

Here are the steps we involved in implementing the Hough transform for detecting ellipses:

1. Load the image & convert it into grayscale format.
2. Applied Canny Edge Detection.
3. Create an accumulator array: The accumulator array is a Five-dimensional array that keeps track of the number of votes for each possible ellipse parameter. For ellipses, we need five parameters: the x and y coordinates of the center, the major and minor axes, and the orientation angle.
4. Loop over all edge points: For each edge point in the Canny edge image, we need to calculate the parameters of all possible ellipses that pass through that point. Then we looped over all possible values of (xc, yc, a, b, theta) and increment the corresponding bin in the accumulator array for each ellipse that passes through the current edge point.
5. Find the maximum votes: After looping over all edge points, we need to find the bin in the accumulator array with the maximum number of votes. This will correspond to the parameters of the ellipse with the highest probability of being present in the image.
6. Draw the ellipse: Finally, we can use the parameters of the detected ellipse to draw it on the original image. This can be done using the cv2.ellipse() function in OpenCV.

Challenges:

1. There were not enough resources for implementation of this algorithms so we conduct it from scratch.
2. We were not able to pass the correct parameters to the function drawing the detected ellipse so it draws a wrong ellipse everytime.

Active Contour Model (Snake)

Algorithm

1. Load the input image

2. **Initialize the contour:** Choose an initial contour that's close to the object boundary you want to detect by selecting a set of points or vertices that lie on the object boundary

3. **Compute the energy function:** Calculate the energy function for the contour by computing the internal energy and external energy terms, which control the smoothness and curvature of the contour, and attract it towards the object boundary.

   In this step, we need to set the energy parameters which are **alpha, beta and gamma**

4. **Optimize the contour:** In this step, we use an iterative optimization process to minimize the energy of the contour and adjust the positions of the contour points. At each iteration, compute the energy of the new contour and check if it has converged

5. **Output the result:** Once the contour has converged, output the final contour positions

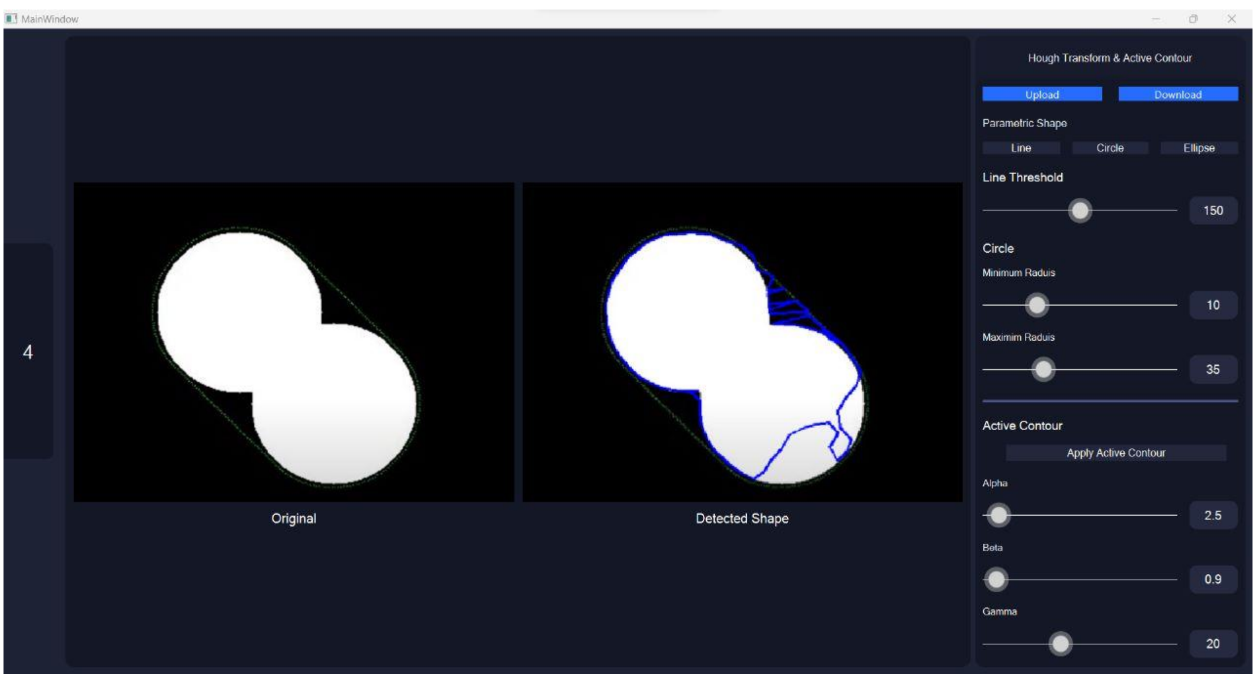
Our Work:

- Despite there were shortage of resources, we began to implement the algorithm in C++ and linked it with our Qt C++ Application
    - The snake did not fit the shapes well
        - We replaced the sobel edge detection with the canny edge detection and the output is enhanced much better but still the snake do not fit with the required shape.
- So, we implemented the algorithm also in python to see if the problem is in our c++ implementation or in tuning the function parameters
    - We conclude that the python and c++ algorithms works fine and the problem is in tuning the function parameters
- After testing with our GUI we found that the alpha, beta, & gamma values (Energy) are robust to most of the images. So, we set them with constants in our program.
- -But for the parameters the make major changes are the init snake parameters (Radius & Center) and also the no of iterations and no points differ according to the image.

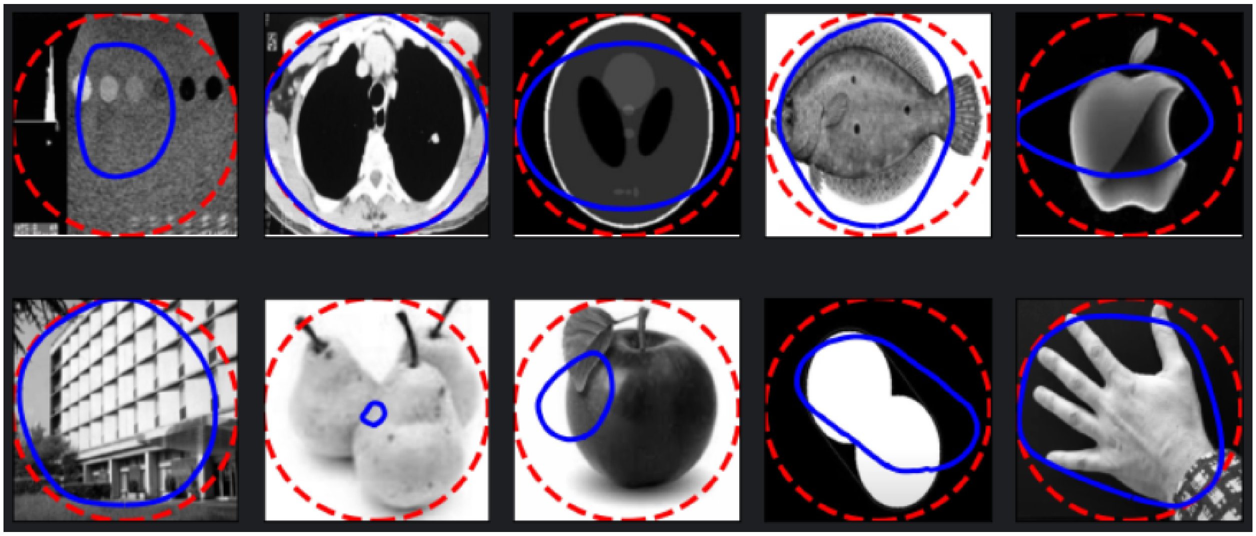
Challenges

- Each photo needs different set of parameters (different internal and external energy) to make the snake fit on it.
    a. We have a combination of different parameters as:
        i. * Alpha
        ii. * Beta
        iii. * Gamma
        iv. * Radius
        v. * Points
        vi. * No of Iterations
- Each photo needs a different non parametric initialized snake for a better results and this would be a computer graphics task for us to implement in this algorithm.
    a. Also, this is from our future plans and we enhanced our C++ Algorithm to take a circular shape with a **variable radius.**

Qt C++ Output



Python Output (Same initialization, Same Parameters)



Python Implementation (Same initialization, Same Parameters)

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
# from .._shared.utils import _supported_float_type
# from ..util import img_as_float
# from ..filters import sobel


def active_contour_implemented(image, snake, alpha=0.01, beta=0.1,
                  w_line=0, w_edge=1, gamma=0.01,
                  max_px_move=1.0,
                  max_num_iter=2500, convergence=0.1,
                  *,
                  boundary_condition='periodic'):
    """Active contour model.

    Active contours by fitting snakes to features of images. Supports single
    and multichannel 2D images. Snakes can be periodic (for segmentation) or
    have fixed and/or free ends.
    The output snake has the same length as the input boundary.
    As the number of points is constant, make sure that the initial snake
    has enough points to capture the details of the final contour.

    Parameters
    ----------
    image : (N, M) or (N, M, 3) ndarray
        Input image.
    snake : (N, 2) ndarray
        Initial snake coordinates. For periodic boundary conditions, endpoints
        must not be duplicated.
```

```
alpha : float, optional
    Snake length shape parameter. Higher values makes snake contract
    faster.
beta : float, optional
    Snake smoothness shape parameter. Higher values makes snake smoother.
w_line : float, optional
    Controls attraction to brightness. Use negative values to attract
    toward dark regions.
w_edge : float, optional
    Controls attraction to edges. Use negative values to repel snake from
    edges.
gamma : float, optional
    Explicit time stepping parameter.
max_px_move : float, optional
    Maximum pixel distance to move per iteration.
max_num_iter : int, optional
    Maximum iterations to optimize snake shape.
convergence : float, optional
    Convergence criteria.
boundary_condition : string, optional
    Boundary conditions for the contour. Can be one of 'periodic',
    'free', 'fixed', 'free-fixed', or 'fixed-free'. 'periodic' attaches
    the two ends of the snake, 'fixed' holds the end-points in place,
    and 'free' allows free movement of the ends. 'fixed' and 'free' can
    be combined by parsing 'fixed-free', 'free-fixed'. Parsing
    'fixed-fixed' or 'free-free' yields same behaviour as 'fixed' and
    'free', respectively.

Returns
-------
snake : (N, 2) ndarray
    Optimised snake, same shape as input parameter.

References
----------
.. [1]  Kass, M.; Witkin, A.; Terzopoulos, D. "Snakes: Active contour
        models". International Journal of Computer Vision 1 (4): 321
        (1988). :DOI:`10.1007/BF00133570`

Examples
--------
>>> from skimage.draw import circle_perimeter
>>> from skimage.filters import gaussian

Create and smooth image:

>>> img = np.zeros((100, 100))
>>> rr, cc = circle_perimeter(35, 45, 25)
>>> img[rr, cc] = 1
>>> img = gaussian(img, 2, preserve_range=False)

Initialize spline:
```

```
    >>> s = np.linspace(0, 2*np.pi, 100)
    >>> init = 50 * np.array([np.sin(s), np.cos(s)]).T + 50

    Fit spline to image:

    >>> snake = active_contour(img, init, w_edge=0, w_line=1, coordinates='rc')
# doctest: +SKIP
    >>> dist = np.sqrt((45-snake[:, 0])**2 + (35-snake[:, 1])**2)  # doctest:
+SKIP
    >>> int(np.mean(dist))  # doctest: +SKIP
    25

    """
    max_num_iter = int(max_num_iter)
    if max_num_iter <= 0:
        raise ValueError("max_num_iter should be >0.")
    convergence_order = 10
    valid_bcs = ['periodic', 'free', 'fixed', 'free-fixed',
                 'fixed-free', 'fixed-fixed', 'free-free']
    if boundary_condition not in valid_bcs:
        raise ValueError("Invalid boundary condition.\n" +
                         "Should be one of: "+", ".join(valid_bcs)+'.')

    img = img_as_float(image)
    float_dtype = _supported_float_type(image.dtype)
    img = img.astype(float_dtype, copy=False)

    RGB = img.ndim == 3

    # Find edges using sobel:
    if w_edge != 0:
        if RGB:
            edge = [sobel(img[:, :, 0]), sobel(img[:, :, 1]),
                    sobel(img[:, :, 2])]
        else:
            edge = [sobel(img)]
    else:
        edge = [0]

    # Superimpose intensity and edge images:
    if RGB:
        img = w_line*np.sum(img, axis=2) \
            + w_edge*sum(edge)
    else:
        img = w_line*img + w_edge*edge[0]

    # Interpolate for smoothness:
    intp = RectBivariateSpline(np.arange(img.shape[1]),
                               np.arange(img.shape[0]),
                               img.T, kx=2, ky=2, s=0)
```

```python
snake_xy = snake[:, ::-1]
x = snake_xy[:, 0].astype(float_dtype)
y = snake_xy[:, 1].astype(float_dtype)
n = len(x)
xsave = np.empty((convergence_order, n), dtype=float_dtype)
ysave = np.empty((convergence_order, n), dtype=float_dtype)

# Build snake shape matrix for Euler equation in double precision
eye_n = np.eye(n, dtype=float)
a = (np.roll(eye_n, -1, axis=0)
     + np.roll(eye_n, -1, axis=1)
     - 2 * eye_n)  # second order derivative, central difference
b = (np.roll(eye_n, -2, axis=0)
     + np.roll(eye_n, -2, axis=1)
     - 4 * np.roll(eye_n, -1, axis=0)
     - 4 * np.roll(eye_n, -1, axis=1)
     + 6 * eye_n)  # fourth order derivative, central difference
A = -alpha * a + beta * b

# Impose boundary conditions different from periodic:
sfixed = False
if boundary_condition.startswith('fixed'):
    A[0, :] = 0
    A[1, :] = 0
    A[1, :3] = [1, -2, 1]
    sfixed = True
efixed = False
if boundary_condition.endswith('fixed'):
    A[-1, :] = 0
    A[-2, :] = 0
    A[-2, -3:] = [1, -2, 1]
    efixed = True
sfree = False
if boundary_condition.startswith('free'):
    A[0, :] = 0
    A[0, :3] = [1, -2, 1]
    A[1, :] = 0
    A[1, :4] = [-1, 3, -3, 1]
    sfree = True
efree = False
if boundary_condition.endswith('free'):
    A[-1, :] = 0
    A[-1, -3:] = [1, -2, 1]
    A[-2, :] = 0
    A[-2, -4:] = [-1, 3, -3, 1]
    efree = True

# Only one inversion is needed for implicit spline energy minimization:
inv = np.linalg.inv(A + gamma * eye_n)
# can use float_dtype once we have computed the inverse in double precision
inv = inv.astype(float_dtype, copy=False)
```

```python
    # Explicit time stepping for image energy minimization:
    for i in range(max_num_iter):
        # RectBivariateSpline always returns float64, so call astype here
        fx = intp(x, y, dx=1, grid=False).astype(float_dtype, copy=False)
        fy = intp(x, y, dy=1, grid=False).astype(float_dtype, copy=False)

        if sfixed:
            fx[0] = 0
            fy[0] = 0
        if efixed:
            fx[-1] = 0
            fy[-1] = 0
        if sfree:
            fx[0] *= 2
            fy[0] *= 2
        if efree:
            fx[-1] *= 2
            fy[-1] *= 2
        xn = inv @ (gamma*x + fx)
        yn = inv @ (gamma*y + fy)

        # Movements are capped to max_px_move per iteration:
        dx = max_px_move * np.tanh(xn - x)
        dy = max_px_move * np.tanh(yn - y)
        if sfixed:
            dx[0] = 0
            dy[0] = 0
        if efixed:
            dx[-1] = 0
            dy[-1] = 0
        x += dx
        y += dy

        # Convergence criteria needs to compare to a number of previous
        # configurations since oscillations can occur.
        j = i % (convergence_order + 1)
        if j < convergence_order:
            xsave[j, :] = x
            ysave[j, :] = y
        else:
            dist = np.min(np.max(np.abs(xsave - x[None, :])
                                 + np.abs(ysave - y[None, :]), 1))
            if dist < convergence:
                break

    return np.stack([y, x], axis=1)
```