Overview of contents

Folders and miscellaneous files

`base_images`

Folder for any starting image, on which one wants to perform some operations.

`data_archive`

Contains test data. There are test folders, which contain 4 folders: `base_images`, `grayscale_images`, `noisy_images`, `result_images`. Additionally, in every test folder there is a summary.txt file which describes given parameters and the outcome.

`grayscale_images`

Typically, images taken from `base_images` are gathered here after they are converted in grayscale format.

`noisy_images`

Images taken from `grayscale_images` are gathered here after artificial noise is added to them.

`result_images`

After applying anisotropic diffusion algorithm on images in folder `noisy_images`, they are stored here.

`readme.md`

Provides usage instructions.

Scripts

`anisotropic_diffusion.py`

The script consists of 2 parts, `numerical_methods` class and `anisotropic_diffusion` function.

`numerical_methods`

This class has 3 static methods: `adams_bashforth_step`, `runge_kutta_step` and `get_numerical_method`.

`get_numerical_method` returns one of other two methods based on its argument `methodNum`.
`adams_bashforth_step` is an implementation of Adams-Bashforth algorithm.
`runge_kutta_step` is an implementation of Explicit Runge-Kutta algorithm.

```
class numerical_methods:

  @staticmethod
  def get_numerical_method(methodNum):

    if (methodNum != 1 and methodNum !=2):
      raise ValueError("1 - Adams-Bashforth, 2 - Runge-Kutta, Other values are
invalid!")
    match methodNum:
      case 1: return numerical_methods.adams_bashforth_step
      case 2: return numerical_methods.runge_kutta_step

  @staticmethod
  def adams_bashforth_step(image, f, gamma):

    return image + gamma * f(image)

  @staticmethod
  def runge_kutta_step(image, f, gamma):

    k1 = gamma * f(image)
    k2 = gamma * f(image + 0.5 * k1)
    k3 = gamma * f(image + 0.5 * k2)
    k4 = gamma * f(image + k3)
    return image + (1 / 6) * (k1 + 2 * k2 + 2 * k3 + k4)
```

`anisotropic_diffusion`

- Arguments
  - `initial_image`
  - `num_iterations`

    - `kappa`
    - `gamma`
    - `methodNum`
- Returns:
    - A diffused image

Performs Anisotropic Diffusion algorithm on image `initial_image` `num_iterations` times, using numerical method which is determined by `methodNum`, with speed coefficient `gamma` and conduction coefficient `kappa`.

```
def anisotropic_diffusion(initial_image, num_iterations=10, kappa=10, gamma=0.1,
methodNum=1):

numerical_method = numerical_methods.get_numerical_method(methodNum)
initial_image = initial_image.astype('float32')
result_image = initial_image.copy()
vertical_gradient = np.zeros_like(result_image)
horizontal_gradient = vertical_gradient.copy()
north_shifted = vertical_gradient.copy()
west_shifted = vertical_gradient.copy()
conduction_vertical = np.ones_like(result_image)
conduction_horizontal = conduction_vertical.copy()

for _ in range(num_iterations):

    vertical_gradient[:-1, :] = np.diff(result_image, axis=0)
    horizontal_gradient[:, :-1] = np.diff(result_image, axis=1)
    conduction_vertical = np.exp(-(vertical_gradient / kappa) ** 2.)
    conduction_horizontal = np.exp(-(horizontal_gradient / kappa) ** 2.)
    f = lambda y: y + gamma * (north_shifted + west_shifted)
    result_image = numerical_method(result_image, f, gamma)
    h = conduction_horizontal * horizontal_gradient
    v = conduction_vertical * vertical_gradient
    north_shifted[:] = v
    west_shifted[:] = h
    north_shifted[1:, :] -= v[:-1,:]
    west_shifted[:, 1:] -= h[:,:-1]

return result_image
```

`kappa` - A low value of kappa allows small intensity gradients to impede conduction, preventing diffusion across step edges. Conversely, a high kappa value diminishes the impact of intensity gradients on conduction.

`methodNum` - `anisotropic_diffusion` uses Adams-Bashforth method if `methodNum` = 1 and Explicit Runge-Kutta if `methodNum` = 2.

`helpers.py`

The script contains helper functions, which aid in adding noise to multiple images, saving multiple images, converting them to grayscale or loading them from folder.

`add_spn`

```
def add_spn(image, salt_prob=0.02, pepper_prob=0.02):

    noisy_image = np.copy(image)

    # Add salt noise
    salt = np.random.rand(*image.shape[:2]) < salt_prob
    noisy_image[salt] = 255

    # Add pepper noise
    pepper = np.random.rand(*image.shape[:2]) < pepper_prob
    noisy_image[pepper] = 0

    return noisy_image
```

This function takes an input image and introduces salt-and-pepper noise based on specified probabilities. It returns the resulting noisy image.

`get_images_from_path`

```
def get_images_from_path(path, return_filenames=False):

    images = []

    for filename in os.listdir(path):
        img = cv2.cvtColor(cv2.imread(os.path.join(path, filename)),
cv2.COLOR_BGR2GRAY)
        if img is not None:
            images.append(img)

    if return_filenames:
        return [images, os.listdir(path)]

    return images
```

This function loads grayscale images from a specified folder. If `return_filenames` is True, it returns a tuple containing the images and corresponding filenames.

`convert_to_grayscale`

```
def convert_to_grayscale(source_path, save_path):

    images, filenames = get_images_from_path(source_path, return_filenames=True)
    save_images(images, save_path, filenames)
```

This function converts color images to grayscale and saves them in a specified folder.

`save_images`

```
def save_images(images, folder_path, names=[]):

    if len(names) != len(images):
        raise ValueError("Length of the images must correspond to the length of the
names")

    for i in range(len(images)):
        integer_image = np.uint8(images[i])
        Image.fromarray(integer_image).save(folder_path + "/" + names[i])
```

This function saves a list of images to a specified folder with user-defined filenames.

`generate_noisy_images`

```
def generate_noisy_images(source_path, save_path, salt_min=0.01, salt_max=0.3,
pepper_min=0.01, pepper_max=0.3):

    images = get_images_from_path(source_path)
    noisy_images = []
    salts = []
    peppers = []

    for i in range(len(images)):
        curr_image = images[i]
        curr_salts = []
        curr_peppers = []
        curr_noisy_images = []

        for _ in range(5):
```

```
            salt = round(random.uniform(salt_min, salt_max), 2)
            pepper = round(random.uniform(pepper_min, pepper_max), 2)
            noisy_image = add_spn(curr_image, salt, pepper)
            curr_noisy_images.append(noisy_image)
            curr_salts.append(salt)
            curr_peppers.append(pepper)

        salts.append(curr_salts)
        peppers.append(curr_peppers)
        noisy_images.append(curr_noisy_images)

    for i in range(len(noisy_images)):
        curr_image = noisy_images[i]
        for j in range(len(curr_image)):
            curr_salt = salts[i][j]
            curr_pepper = peppers[i][j]
            noisy_curr = curr_image[j]
            image = Image.fromarray(noisy_curr)
            path = save_path + "/" + str(i) + "_" + "s=" + str(curr_salt) + "_" +
"p=" + str(curr_pepper) + ".jpg"
            image.save(path)
```

This function generates noisy images by adding salt-and-pepper noise to grayscale images from a specified folder. The resulting images are saved in another folder with filenames indicating the applied noise probabilities.

`index.py`

This is the main script, which uses the functions above to perform Anisotropic diffusion

`exec`

```
from anisotropic_diffusion import anisotropic_diffusion as ad
from helpers import *
from tqdm import tqdm

def exec(base_path, grayscale_path, noisy_path, result_path, noisy_generated=True,
grayscale_generated=True):
    """
    Execute image processing and anisotropic diffusion.

    Parameters:
    - base_path (str): Path to the folder containing base images.
```

```
    - grayscale_path (str): Path to the folder where grayscale images will be
saved.
    - noisy_path (str): Path to the folder where noisy images will be saved.
    - result_path (str): Path to the folder where the result images will be saved.
    - noisy_generated (bool): If True, generate noisy images. Default is True.
    - grayscale_generated (bool): If True, convert color images to grayscale.
Default is True.

    Returns:
    None
    """
    if not grayscale_generated:
        convert_to_grayscale(base_path, grayscale_path)
    if not noisy_generated:
        generate_noisy_images(grayscale_path, noisy_path)

    images, file_names = get_images_from_path(noisy_path, return_filenames=True)
    diffused_images = [ad(image, kappa=100, num_iterations=0) for image in images]
    save_images(diffused_images, result_path, file_names)

exec(
  "base_images",
  "grayscale_images",
  "noisy_images",
  "result_images",
  noisy_generated=True,
  grayscale_generated=True
)
```

This function performs image processing and anisotropic diffusion on a set of base images. It first checks whether grayscale and noisy images need to be generated. If not, it loads existing images, applies anisotropic diffusion, and saves the resulting images.

`exec_for_analysis`

```
def exec_for_analysis(base_path, grayscale_path, noisy_path, result_path,
noisy_generated=True, grayscale_generated=True):
    """
    Execute image analysis using anisotropic diffusion with varied parameters.

    Parameters:
    - base_path (str): Path to the folder containing base images.
    - grayscale_path (str): Path to the folder where grayscale images will be
```

```
saved.
    - noisy_path (str): Path to the folder where noisy images will be saved.
    - result_path (str): Path to the folder where the result images will be saved.
    - noisy_generated (bool): If True, generate noisy images. Default is True.
    - grayscale_generated (bool): If True, convert color images to grayscale.
Default is True.

    Returns:
    None
    """
    if not grayscale_generated:
        convert_to_grayscale(base_path, grayscale_path)
    if not noisy_generated:
        generate_noisy_images(grayscale_path, noisy_path)

    images, file_names = get_images_from_path(noisy_path, return_filenames=True)
    diffused_images = []
    names = []

    for i in range(len(images)):
        for kappa in tqdm(range(1, 100, 5)):
            for num_iterations in tqdm(range(1, 5, 1)):
                for method in range(1):
                    res = ad(images[i], kappa=kappa, num_iterations=num_iterations,
methodNum=method + 1)
                    diffused_images.append(res)
                    print(file_names[i])
                    new_filename = file_names[i]
[:-4]+"_k="+str(kappa)+"_i="+str(num_iterations)+"_m="+str(method)+".jpg"
                    names.append(new_filename)

    save_images(diffused_images, result_path, names)

# Uncomment the following lines to execute the analysis
# exec_for_analysis(
#   "base_images",
#   "grayscale_images",
#   "noisy_images",
#   "analysis_images",
#   noisy_generated=True,
#   grayscale_generated=True
# )
```

This function executes image analysis using anisotropic diffusion with varied parameters. It generates a set of noisy images, applies anisotropic diffusion with different parameters, and

saves the resulting images with informative filenames. Note that the function is currently commented out, and you can uncomment the last section to execute the analysis.

`data_archive` and image naming

Inside `data_archive` folder, one can find 2 test sets of images. Each image has a name which looks something like this: `0_s=0.1_p=0.07_k=1_i=1_m=0`.

- The first digit is image number.
- s=0.1 signifies that when the noise was artificially added on original image, the probability of salt(white) noise was 0.1.
- p=0.07 signifies the same as above but for pepper(black) noise.
- k=1 signifies the value of kappa.
- i=1 signifies the number of iterations.
- m=0 signifies which method was used. 0 - Adams-Bashfourth, 1- Explicit Runge-Kutta

Conclusion

One can see that the best values for kappa are values between 20-80. As of number of iterations, 1-2 iterations are sufficient. Method used has no determining factor on resulting image.