1. Business Understanding

- Malaria is a life-threatening disease spread to humans by mosquitoes, and is considered to be a severe public health issue. According to the World Health Organization, globally in 2022, there were an estimated 249 million malaria cases and 608,000 malaria deaths in 85 countries (source: https://www.who.int/news-room/fact-sheets/detail/malaria). The WHO African Region carries a disproportionate amount of the global malaria burden at 94% of malaria cases (233 million) and 95% (580,000) of malaria deaths. The infection caused by a parasite from the mosquito is preventable and curable. One method of preventing malaria is through thorough surveillance of it.

- The `Malaria Cell Images Dataset` contains over 27,000 images of cells that are labeled as either `parasitized` or `uninfected`. Each image will be converted to greyscale and resized to a 64x64 pixel image for data processing. The dataset was collected to improve the detection of malaria via image classification, in order to reduce the burden of malaria worldwide. According to the National Library of Medicine, existing drugs make malaria a curable disease, but inadequate diagnostics and emerging drug resistances are major barriers to successful mortality reduction. Therefore, the development of a fast and reliable diagnostic method is one of the most promising methods to fight malaria (source: https://lhncbc.nlm.nih.gov/LHC-research/LHC-projects/image-processing/malaria-project.html).

- Accurate parasite counts are essential to diagnosing malaria correctly. However, performing microscopic diagnostics is not standardized in every region and depends on the skills and experience of the microscopist. In areas of lower resources, diagnostic quality can be negatively impacted, leading to incorrect diagnostics. In instances where false negative cases arise, this means unnecessary use of antibiotics, a second consultation, or even severe progression in malaria. In instances with false positive cases, a misdiagnosis means unnecessary use of anti-malaria treatments and resource depletion for those who might actually be suffering from malaria. The prediction algorithm for our case would need to be highly accurate, and avoid both false negatives and false positives altogether for it to be successful. We would measure the part of the success of our algorithm using metrics like `precision` and `recall`. High precision would ensure fewer false positives, while high recall ensures fewer false negatives. An `F1 score` that balances both metrics would provide a comprehensive assessment of the model's performance. It is impractical to believe that our prediction algorithm could reach 100% accuracy in classifying whether a cell is infected or not because some who may have been infected may not experience severe symptoms due to prior past infections or immunities. When considering our measure for a successful algorithm, we considered third parties that would be interested in the results.

- Accurate and precise malaria diagnostics would benefit `public health organizations`, `disease researchers`, and `hospitals`. Public health organizations, such as WHO, would benefit from our binary classification analysis because they can use our data to accurately diagnose someone with malaria, and provide appropriate treatment options to reduce the severity of the disease and prevent mortality. In areas that lack adequate and appropriate resources, having a tool that accurately and precisely detects malaria could save the lives of millions, especially in the WHO African Region. Due to its disproportionate amount of global malaria burden, the WHO African Region cannot afford to have false negatives or false positives, since that would deplete their malaria treatment resources without adequately addressing the issue. We aim for our prediction algorithm to achieve 95-97% accuracy in classifying whether a cell is infected with malaria. It is equally important that the precision of the algorithm aligns with this accuracy to minimize false negatives and false positives, which we seek to avoid. Our target of 95-97% accuracy is based on the performance of current diagnostic tools, such as EasyScan Go, where human microscopists typically achieve accuracy rates of 85-90% (source: https://malariajournal.biomedcentral.com/articles/10.1186/s12936-022-04146-1). Achieving a 5-10% improvement from EasyScan Go is doable, as high quality smears and stainings are paramount in allowing a machine-learning algorithm to analyze infected cells, which is something EasyScan Go's current technology is unable to achieve at the moment. For example, incorporating noise reduction parameters can aid the screening tool in more accurately identifying and diagnosing an infected cell.

- Another stakeholder that would appreciate our data are disease researchers. By studying the patterns that are exhibited from infected and uninfected cells, researchers can develop treatment plans that can aid in preventing or even treating the infections. According to WHO, partial resistance in antimalarial drugs has emerged in the WHO African Region (source: https://www.who.int/news-room/fact-sheets/detail/malaria). Accurate and precise classification of malaria cells, along with the discovery of patterns and trends, can help disease researchers develop new antimalarial drugs that are less prone to resistance. Faster diagnosis enables a quicker response, which can delay or prevent the onset of drug resistance.

- Lastly, hospitals can benefit from this screening tool as it can allocate appropriate resources to those who are infected or uninfected. By having a prediction algorithm of 95-97% accuracy and precision, hospitals can more confidently start treatment plans for those infected. If a false positive arises, the worst case is using malaria treatment on someone who does not need it. The only issue with this occurs when the hospital is in an area with limited supplies, or a public health emergency emerges. In this case, the accuracy and precision of our algorithm needs to be 97% and higher, in order to conserve as much resources as possible. However, in general, hospitals would benefit from this malaria screening tool because they can plan ahead for the resources they'll need by geographic location.

- The prediction task for this dataset is a binary classification to detect whether a cell is parasitized or not with malaria, essentially developing an accurate and precise screening tool for malaria. Key stakeholders that would be interested in our results are public health organizations, disease researchers, and hospitals. This data is important because early and accurate detection of malaria can improve patient outcomes and reduce the burden of the disease. This data could also facilitate faster, more scalable diagnostics in essential regions where traditional microscopy is either too slow or resource-intensive. Achieving higher accuracy and precision with our algorithm would reduce the strain on healthcare workers and improve diagnostic precision in regions with limited medical expertise, ultimately to reduce the burden of malaria worldwide. Lastly, using metrics like precision and recall, and an F1 score can provide an assessment of our model's overall performance. This dataset can have important global implications, offering scalable solutions for regions that face sporadic and disproportionate malaria outbreaks.

- Dataset: https://www.kaggle.com/datasets/iarunava/cell-images-for-detecting-malaria

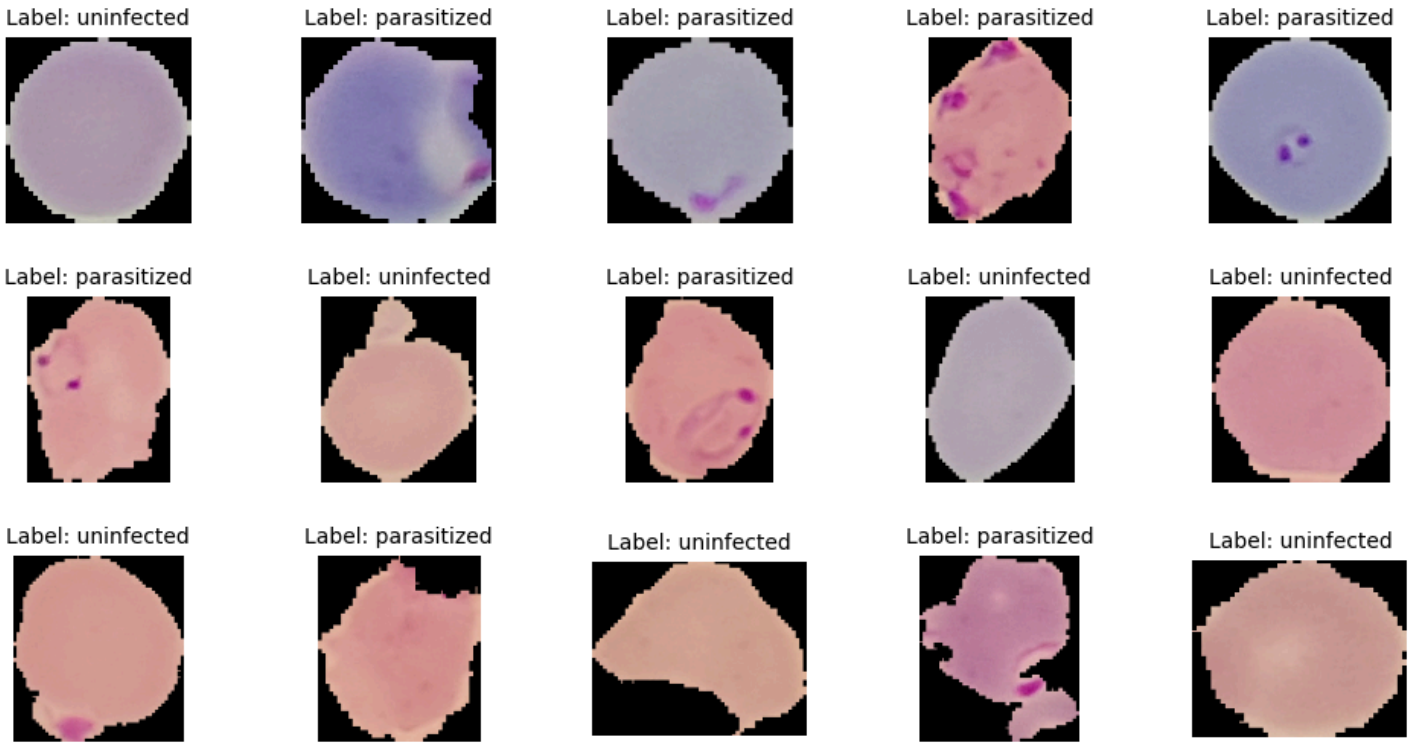
2. Data Preparation

In [1]:
```
# Modules and Libraries
import os
import random
import time
from collections import Counter

# Numerical and data manipulation libraries
import numpy as np
import cv2
import matplotlib.pyplot as plt

# Machine learning and data processing libraries
from sklearn.utils import shuffle
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from skimage.feature import daisy
from skimage.color import rgb2gray
from skimage.io import imshow
from sklearn.metrics.pairwise import pairwise_distances
from sklearn.metrics import classification_report, confusion_matrix

# Interactive widgets and visualization libraries
from ipywidgets import widgets, interactive
import plotly
```

```
from plotly.graph_objs import Bar, Scatter, Layout
from plotly.graph_objs.layout import XAxis, YAxis
import seaborn as sns
```

In [2]:

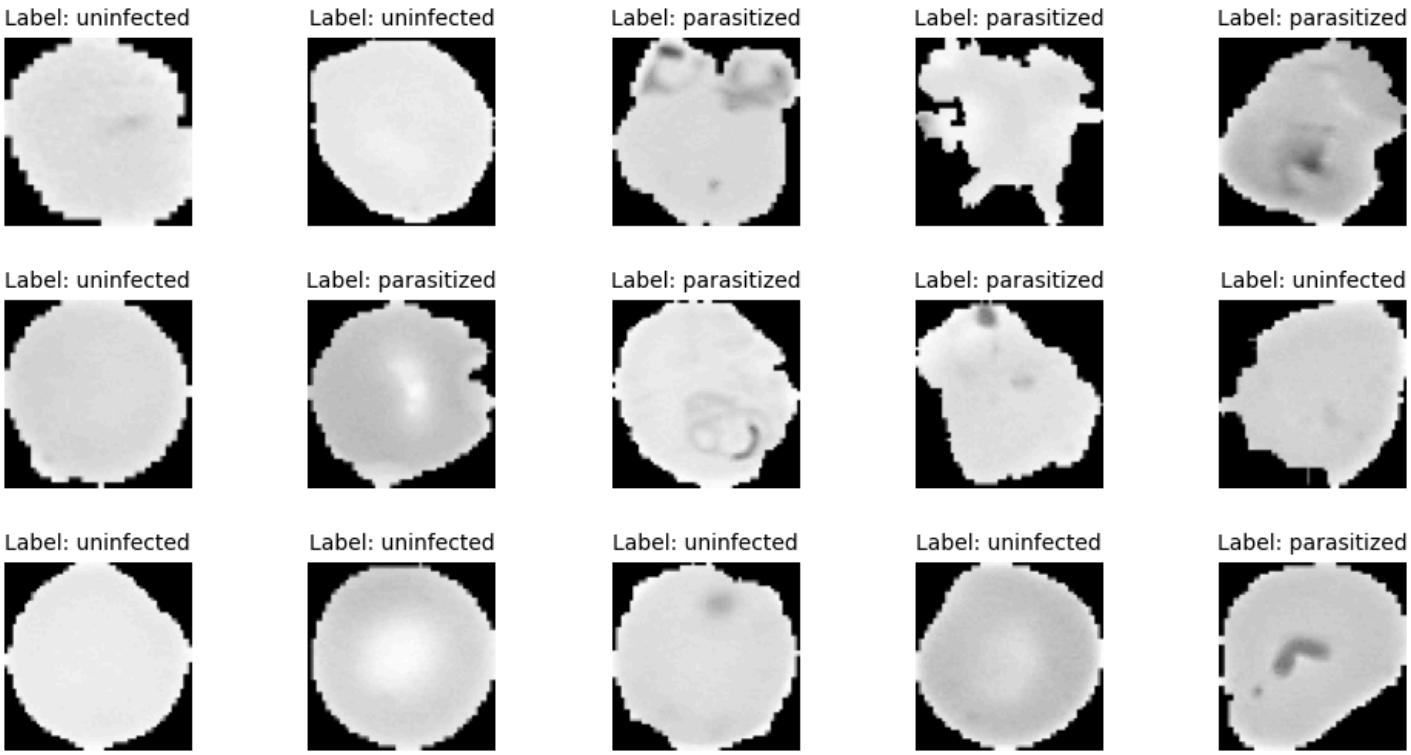
```
# Directory path
image_dir = 'C:/Users/Juan Dominguez/Desktop/Malaria_Dataset/malaria_ds/cell_images'

# Lists to store original imgs & labels
original_images = []
original_labels = []

# Iterating through subdirectories to load images
for category in ['parasitized', 'uninfected']:
    category_dir = os.path.join(image_dir, category)
    image_files = [f for f in os.listdir(category_dir) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

    for img_file in image_files:
        img_path = os.path.join(category_dir, img_file)
        img = cv2.imread(img_path)

        if img is None:
            print(f"Failed to load image: {img_path}")
            continue

        # Stores the original image and label for images
        original_images.append(img)
        original_labels.append(category)

# Displaying 15 random original images
num_images_to_show = 15
rows, cols = 3, 5
plt.figure(figsize=(12, 6))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

# Random sample, select based on sampled indices
indices = random.sample(range(len(original_images)), num_images_to_show)
selected_images = [original_images[i] for i in indices]
selected_labels = [original_labels[i] for i in indices]

# Loop through selected images for display
for i in range(num_images_to_show):
    plt.subplot(rows, cols, i + 1)

    # Convert image from BGR(OpenCV format) to RGB
    plt.imshow(cv2.cvtColor(selected_images[i], cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.title(f"Label: {selected_labels[i]}", fontsize=10)

# Display
plt.show()
```



In [3]:

```
# Lists to store processed imgs and labels
images = []
labels = []

# Iterating through subdirectories to process imgs
for category in ['parasitized', 'uninfected']:
    category_dir = os.path.join(image_dir, category)
    image_files = [f for f in os.listdir(category_dir) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

    for img_file in image_files:
        img_path = os.path.join(category_dir, img_file)
        img = cv2.imread(img_path)

        if img is None:
            print(f"Failed to load image: {img_path}")
            continue

        # Resizing and converting to grayscale
        img_resized = cv2.resize(img, (64, 64))
        img_gray = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)

        # Stores the processed image and label for image
        images.append(img_gray)
        labels.append(category)

# Display 15 random processed grayscale images
plt.figure(figsize=(12, 6))
plt.subplots_adjust(wspace=0.4, hspace=0.4)  # Adjust space between subplots

# Checks if the number of images is sufficient
num_images_to_show = min(num_images_to_show, len(images))

# Random sample indices
indices = random.sample(range(len(images)), num_images_to_show)

# Loops through selected indices for display
for i in range(num_images_to_show):
    plt.subplot(3, 5, i + 1)
    plt.imshow(images[indices[i]], cmap='gray')  # Displays the grayscale image
    plt.axis('off')  # Hide axes
    plt.title(f"Label: {labels[indices[i]]}", fontsize=10)  # Uses labels for titles
```

```
# Display
plt.show()
```



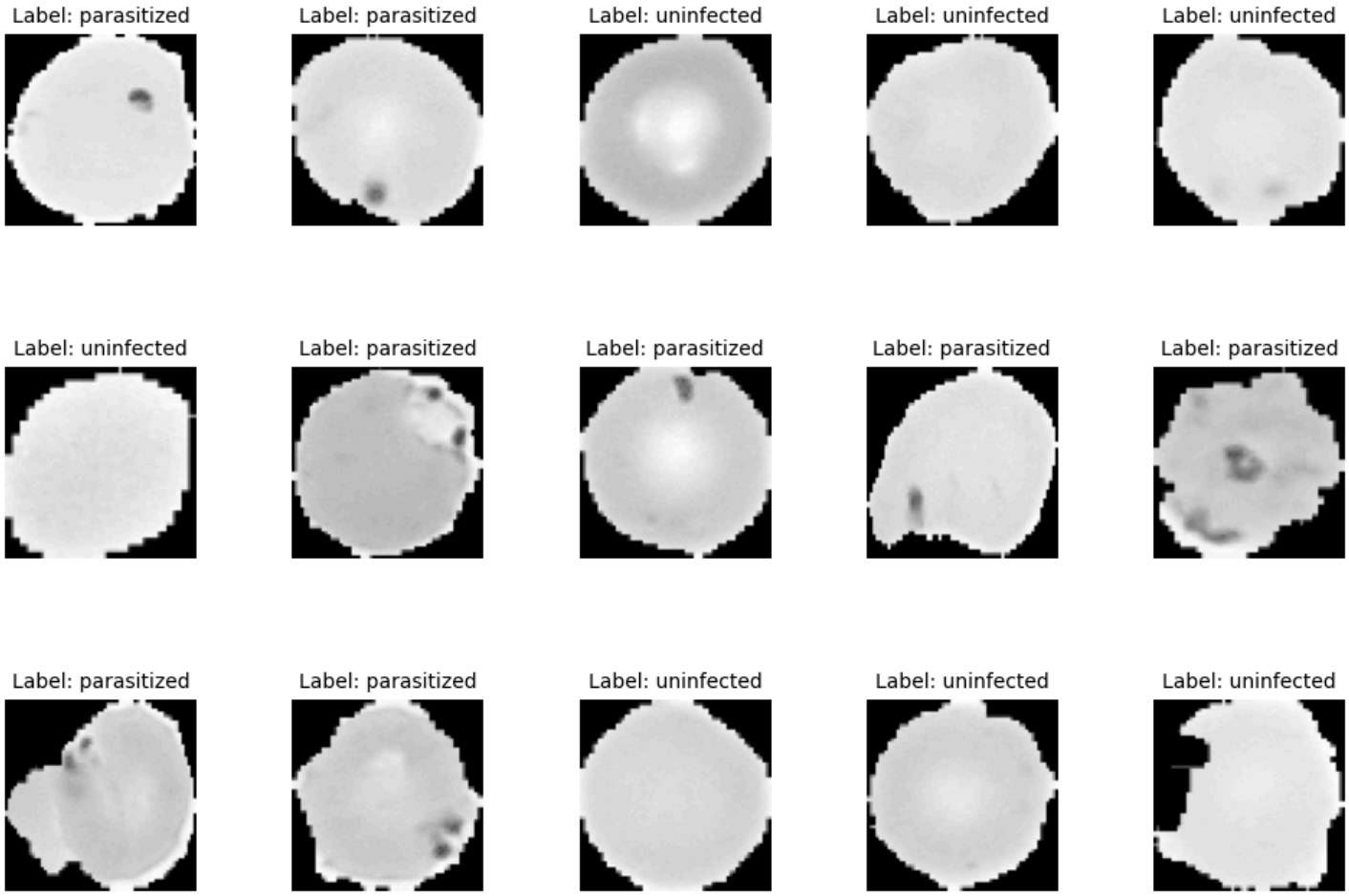
In [4]:
```
# Converting list of imgs to numpy array
images_np = np.array(images)
print(f"Shape of images_np: {images_np.shape}")
```

```
Shape of images_np: (27558, 64, 64)
```

In [5]:
```
# Linearizing the images, flatten each image into a 1-D array
# Unpacking 3D shape to later reshape
n_images, height, width = images_np.shape

# Converting 2D image into 1D arr & Normalizing the pixels in range [0, 1]
images_flattened = images_np.reshape(n_images, height * width) / 255.0

# Display info about flattened images
print(f"Number of images: {n_images}")
print(f"Flattened image shape: {images_flattened.shape}")
```

```
Number of images: 27558
Flattened image shape: (27558, 4096)
```

In [6]:
```
# Visualizing Several Images
# Shuffling images and labels together
images_np, labels = shuffle(images_np, labels, random_state=42)

# Visualizes 15 random imgs 3x5 grid
num_images_to_show = 15
plt.figure(figsize=(12, 8))
rows, cols = 3, 5

# Loops through number of images for display
for i in range(num_images_to_show):
    plt.subplot(rows, cols, i + 1)
    plt.imshow(images_np[i], cmap='gray')
    plt.axis('off')
    plt.title(f"Label: {labels[i]}", fontsize=10)

# Adding space, display
plt.subplots_adjust(wspace=0.5, hspace=0.5)
plt.show()
```



In [7]:
```
# Counting occurrences of each label
label_counts = Counter(labels)
print(f"Number of parasitized images: {label_counts['parasitized']}")
print(f"Number of uninfected images: {label_counts['uninfected']}")
```

```
Number of parasitized images: 13779
Number of uninfected images: 13779
```

3. Data Reduction

PCA

In [8]:
```
# Function to plot the explained variance
def plot_explained_variance(pca):
    plotly.offline.init_notebook_mode()

    # Extract explained variance ratio and cumulative variance
    explained_var = pca.explained_variance_ratio_
    cum_var_exp = np.cumsum(explained_var)

    # Plot individual and cumulative explained variance
    plotly.offline.iplot({
        "data": [Bar(y=explained_var, name='individual explained variance'),
                 Scatter(y=cum_var_exp, name='cumulative explained variance')
            ],
        "layout": Layout(
            xaxis=XAxis(title='Principal components'),
            yaxis=YAxis(title='Explained variance ratio')
        )
    })
```

In [9]:
```
# Performing Linear Dimensionality Reduction Using PCA
# Number of components to keep
n_components = 800
pca = PCA(n_components=n_components)

# Fit PCA of flattened img data, copy to not alter original
X_pca = pca.fit(images_flattened.copy())

# Plot explained variance
plot_explained_variance(pca)

# Display
plt.show()
```

In [10]:
```
# Compute cumulative explained variance
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)

# Variance thresholds to check
thresholds = [0.85, 0.90, 0.95]

# Dictionary for threshold
components_for_thresholds = {}

# Iterate through cumulative explained variance to find the number of components for each threshold
for i, cumulative in enumerate(cumulative_variance):
    for threshold in thresholds[:]:
        if cumulative >= threshold:
            components_for_thresholds[threshold] = i + 1
            print(f"Number of components for {int(threshold * 100)}% variance: {components_for_thresholds[th

            # Removes satified threshold avoids rechecking
            thresholds.remove(threshold)
```

```
Number of components for 85% variance: 172
Number of components for 90% variance: 336
Number of components for 95% variance: 751
```

Analysis:

- **85% variance with 172 components**:
  This level of dimensionality reduction significantly reduces the number of features while retaining the majority of the dataset's information. It offers a balance between efficiency and data representation. This is particularly useful for tasks where computational resources are limited or where simpler models are preferred. However, it may come at the cost of missing out on some important but subtle patterns in the data.

- **90% variance with 336 components**:
  Capturing 90% of the variance offers a stronger representation of the data while still reducing dimensionality substantially. This strikes a good balance between maintaining sufficient information and optimizing computational efficiency. It is often used in real-world applications where retaining more variance leads to better model performance without excessive computational burden.

- **95% variance with 751 components**:
  Including 95% of the variance can help preserve almost all the information, which is crucial for accurately predicing malaria in the cell images. While this approach carries the risk of diminishing returns—where additional components provide marginal increases in variance explained—it ensures the model captures small but critical variations in the data. This is especially important in sensitive tasks like medical image analysis. However, we remain cautious of overfitting and the increase in computational complexity, balancing the need for accuracy with efficient resource usage.

In [11]:
```python
# Get the individual explained variance (variance for each component)
individual_explained_variance = pca.explained_variance_ratio_

# Prints the individual explained variance for the first 25 components
print("Individual explained variance for each component:")
for i, var in enumerate(individual_explained_variance[:25]):
    print(f"Component {i+1}: {var:.5f}")
```

```
Individual explained variance for each component:
Component 1: 0.12736
Component 2: 0.10900
Component 3: 0.05632
Component 4: 0.05523
Component 5: 0.03711
Component 6: 0.02693
Component 7: 0.02486
Component 8: 0.02445
Component 9: 0.01892
Component 10: 0.01515
Component 11: 0.01281
Component 12: 0.01234
Component 13: 0.01215
Component 14: 0.01065
Component 15: 0.01060
Component 16: 0.01055
Component 17: 0.01021
Component 18: 0.00998
Component 19: 0.00828
Component 20: 0.00815
Component 21: 0.00764
Component 22: 0.00601
Component 23: 0.00569
Component 24: 0.00560
Component 25: 0.00557
```

PCA Analysis:

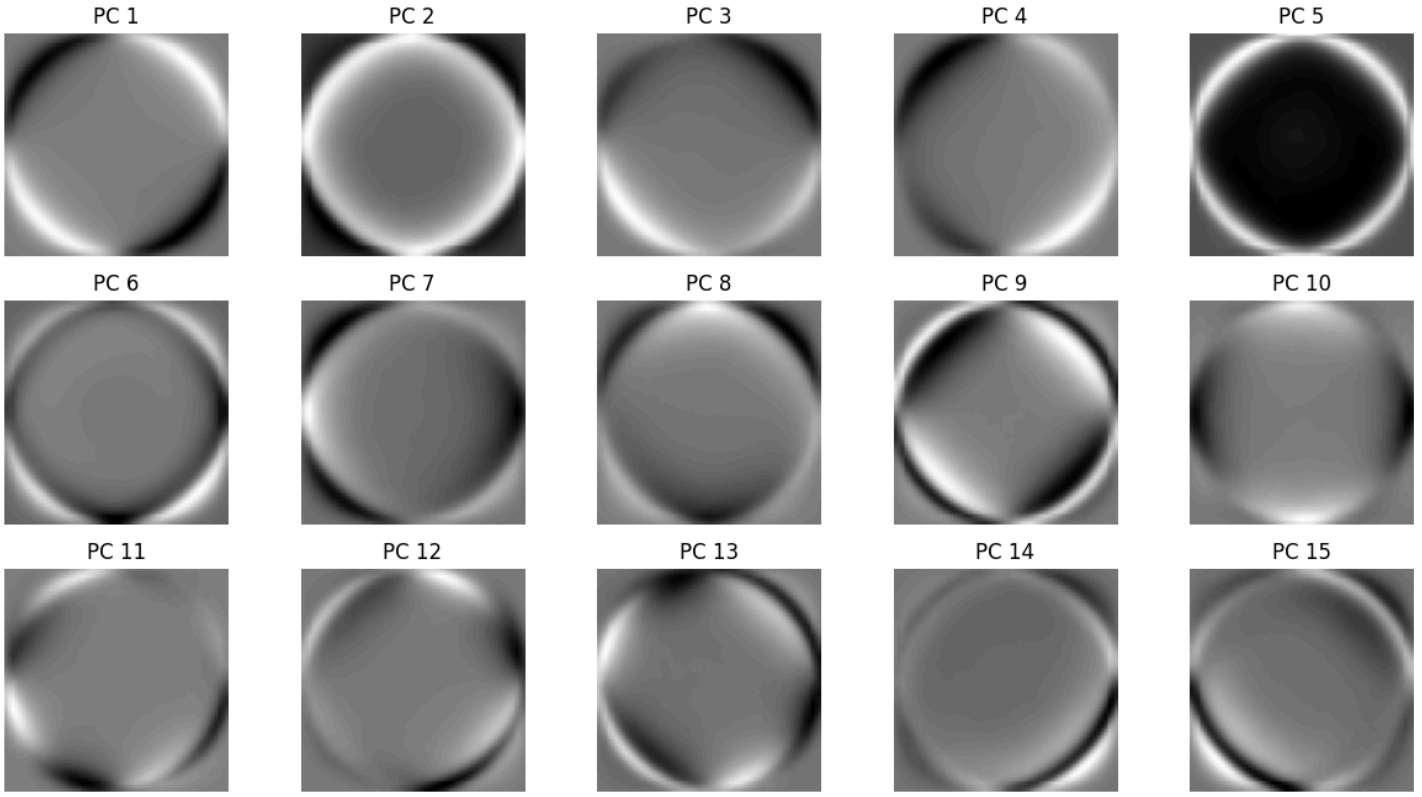
Individual Explained Variance

- The first principal component accounts for 12.74% of the total variance, while the second component accounts for 10.90%. After the initial components, diminishing returns occur as seen in components 6 to 10, where each explain less than 3% of the variance. This pattern s typical in PCA where the first few components capture the most significant variance and subsequent components contribute progressively less.

Dimensions Required

- Despite the diminishing returns after the first few components, we chose to retain 800 principal components for the following reasons:
    - 1. To cover 95% of the variance, 751 principal components are required. We opted for 800 components to ensure a more comprehensive representation of the dataset. This ensures that the dataset is well-represented, preserving as much information as possible, which is crucial for an accurate classification of the cell images.
    - 2. Since we are working on classifying whether a cell image is infected or not, maintaining a high level of variance is essential to minimize false positives and negatives. Retaining 800 components, allows us to capture more detailed patterns in the data potentially improving the performance of our classification model.
    - 3. If we were satisfied with 85% variance, we would require 172 dimensions, nearly half of what is required for 90%. However given the importance of retaining variance for detecting infections, we chose to cover 95% of the variance. This trade-off reflects a balance between dimensionality reduction and preserving the nuances in the data.

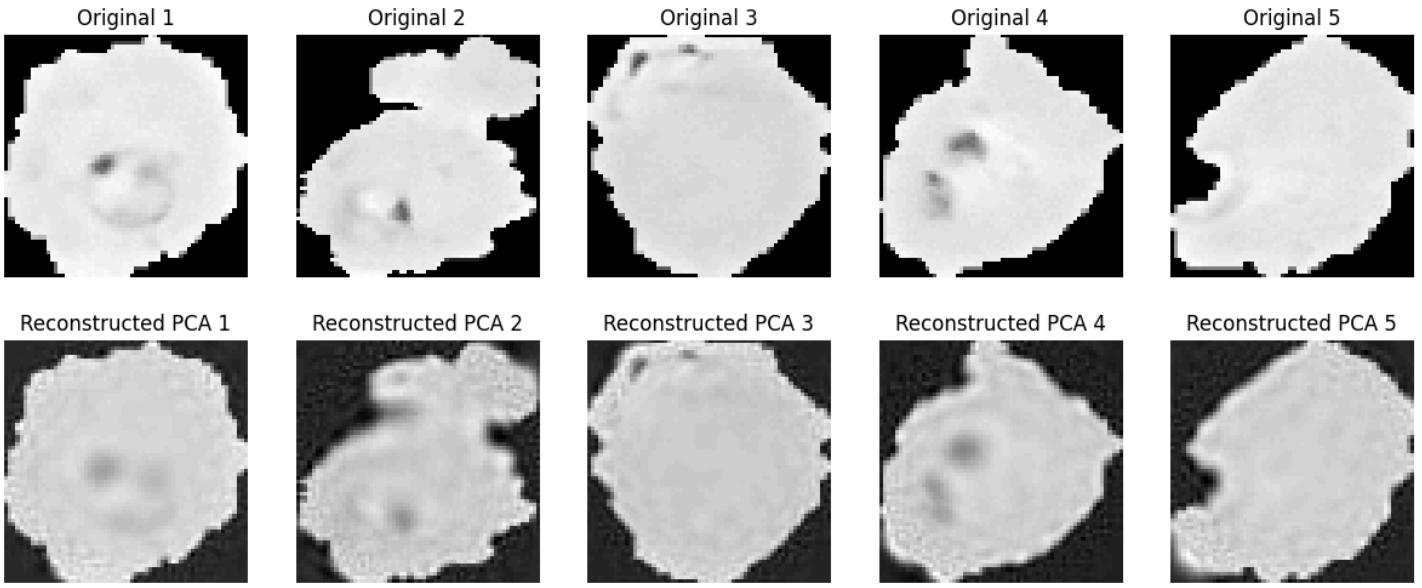
In [12]:
```
# Getting the principal components (eigenfaces)
eigenfaces = pca.components_.reshape((n_components, 64, 64))

# Visualizing first 15 principal components (eigenfaces)
plt.figure(figsize=(15, 8))

# Display the first 15 eigenfaces
for i in range(num_images_to_show):
    plt.subplot(3, 5, i + 1)
    plt.imshow(eigenfaces[i], cmap='gray')
    plt.title(f"PC {i+1}")
    plt.axis('off')

# Display
plt.show()
```



In [13]:
```
# Visualize explained variance
plot_explained_variance(pca)
```

In [14]:
```
def reconstruct_image(trans_obj,org_features):
    # Project original image lower dimensional
    low_rep = trans_obj.transform(org_features)

    # Reconstruct image
    rec_image = trans_obj.inverse_transform(low_rep)
```

```
    return low_rep, rec_image

# Examples to visualize
num_examples = 5

# Plot original and reconstructed images
plt.figure(figsize=(15,6))

for i in range(num_examples):
    # Index of image to reconstruct
    idx_to_reconstruct = i
    X_idx = images_flattened[idx_to_reconstruct]

    # Get low dimensional representation and reconstruct
    low_dimensional_representation, reconstructed_image = reconstruct_image(pca, X_idx.reshape(1, -1))

    # Original image
    plt.subplot(2, num_examples, i + 1)
    plt.imshow(X_idx.reshape((64, 64)), cmap=plt.cm.gray)
    plt.title(f'Original {i+1}')
    plt.axis('off')

    # Reconstruct image from PCA
    plt.subplot(2, num_examples, i + 1 + num_examples)
    plt.imshow(reconstructed_image.reshape((64,64)), cmap=plt.cm.gray)
    plt.title(f'Reconstructed PCA {i+1}')
    plt.axis('off')

# Display
plt.show()
```



Original vs Reconstructed from Full PCA

- The reconstruction from Full PCA demonstrates that the original image is fairly well preserved, with key features such as the shape of the cell and some finer details still recognizable. This confirms that PCA can serve as a meaningful tool for dimensionality reduction while maintaining the integrity of the original data.
- In our case, we chose to retain 800 principal components, which captures 95% of the variance in the dataset. This decision was made to ensure that we are accounting for both the major structural elements and more subtle patterns present in the cell images. By capturing this amount of variance, we strike a balance between dimensionality reduction and retaining sufficient detail, which is critical for the accuracy of subsequent analysis or classification tasks.

RPCA

In [15]:
```
# Performing Linear Dimensionality Reduction Using Randomized PCA
rpca = PCA(n_components=n_components, svd_solver='randomized')
X_pca = rpca.fit(images_flattened.copy())

# Function to plot the explained variance
def plot_explained_variance(rpca):
    explained_var = rpca.explained_variance_ratio_
    cum_var_exp = np.cumsum(explained_var)

    # Plot individual and explained variance
    plotly.offline.iplot({
        "data": [Bar(y=explained_var, name='Individual Explained Variance'),
                 Scatter(y=cum_var_exp, name='Cumulative Explained Variance')
             ],
        "layout": Layout(xaxis=XAxis(title='Principal Components'), yaxis=YAxis(title='Explained Variance Ra
    })

# Calls function to plot
plot_explained_variance(rpca)
```

In [16]:
```
# Compute cumulative explained variance
rpca = PCA(n_components, svd_solver='randomized')
X_pca = rpca.fit(images_flattened.copy())

cumulative_variance = np.cumsum(rpca.explained_variance_ratio_)

# Initialize variables to store the number of components for 85%, 90%, and 95% variance
components_85 = None
components_90 = None
components_95 = None

# Iterate through cumulative explained variance to find the number of components for each threshold
```

```
for i, cumulative in enumerate(cumulative_variance):
    if cumulative >= 0.85 and components_85 is None:
        components_85 = i + 1
        print(f"Number of components for 85% variance: {components_85}")

    if cumulative >= 0.90 and components_90 is None:
        components_90 = i + 1
        print(f"Number of components for 90% variance: {components_90}")

    if cumulative >= 0.95 and components_95 is None:
        components_95 = i + 1
        print(f"Number of components for 95% variance: {components_95}")

    # Break the loop once all thresholds are met
    if components_85 and components_90 and components_95:
        break
```

```
Number of components for 85% variance: 172
Number of components for 90% variance: 336
Number of components for 95% variance: 751
```

In [17]:
```
# Get the individual explained variance (variance for each component)
individual_explained_variance = rpca.explained_variance_ratio_

# Prints the individual explained variance for the first 25 components
print("Individual explained variance for each component:")
for i, var in enumerate(individual_explained_variance[:25]):
    print(f"Component {i+1}: {var:.5f}")
```

```
Individual explained variance for each component:
Component 1: 0.12736
Component 2: 0.10900
Component 3: 0.05632
Component 4: 0.05523
Component 5: 0.03711
Component 6: 0.02693
Component 7: 0.02486
Component 8: 0.02445
Component 9: 0.01892
Component 10: 0.01515
Component 11: 0.01281
Component 12: 0.01234
Component 13: 0.01215
Component 14: 0.01065
Component 15: 0.01060
Component 16: 0.01055
Component 17: 0.01021
Component 18: 0.00998
Component 19: 0.00828
Component 20: 0.00815
Component 21: 0.00764
Component 22: 0.00601
Component 23: 0.00569
Component 24: 0.00560
Component 25: 0.00557
```

Analysis for Randomized PCA

Individual Explained Variance

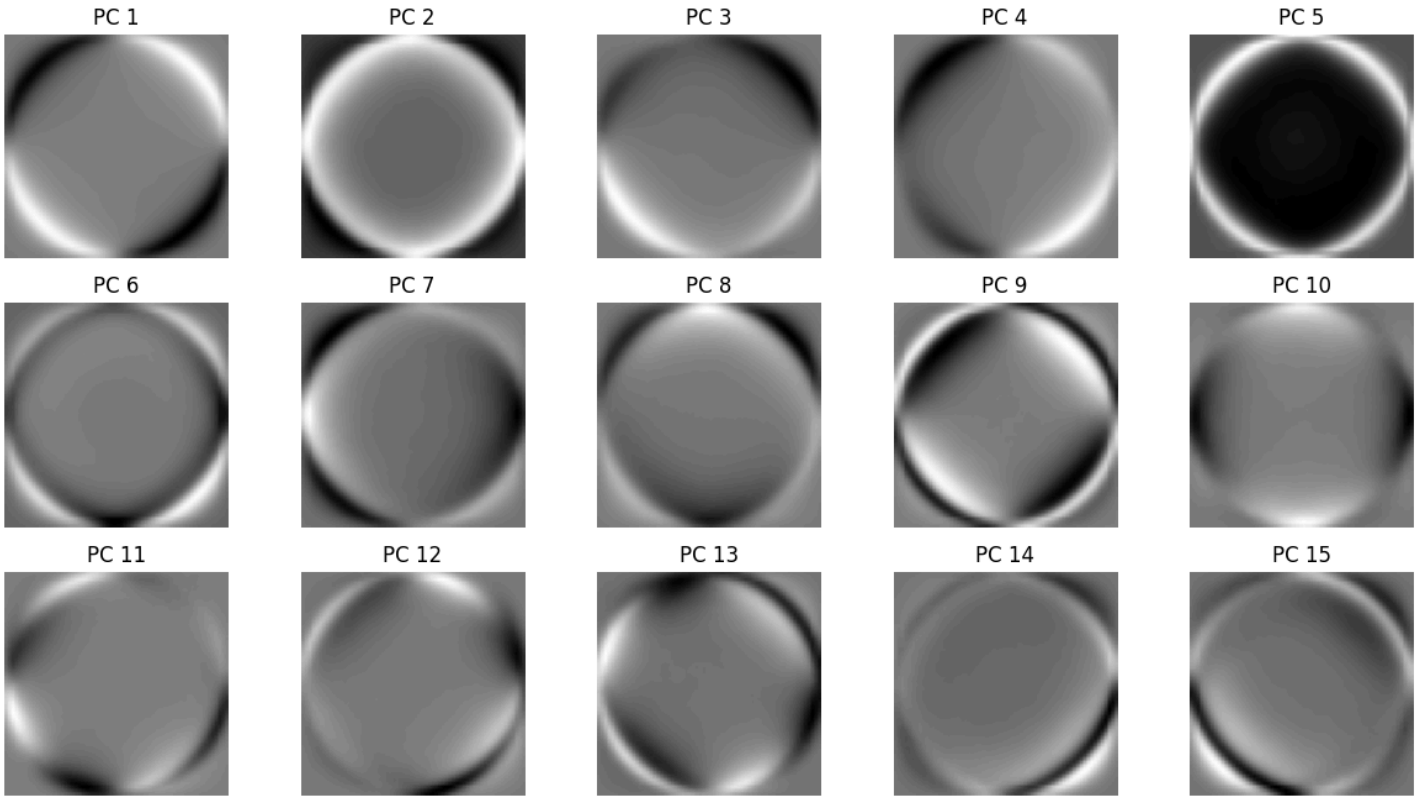
- The individual explained variance for the first principal component accounts for 12.74% of the total variance, while the second component explains 10.90%. These values are comparable to the results obtained using traditional full PCA. This similarity is expected because both full PCA and randomized PCA are designed to efficiently capture the most significant directions of variance within the data. As in full PCA, the explained variance decreases as the number of components increases, with each subsequent component contributing less to the total variance. This decreasing

trend is consistent across both methods, demonstrating that randomized PCA effectively mirrors the patterns observed in full PCA, but with a computational advantage when dealing with large datasets.

Dimensions Required

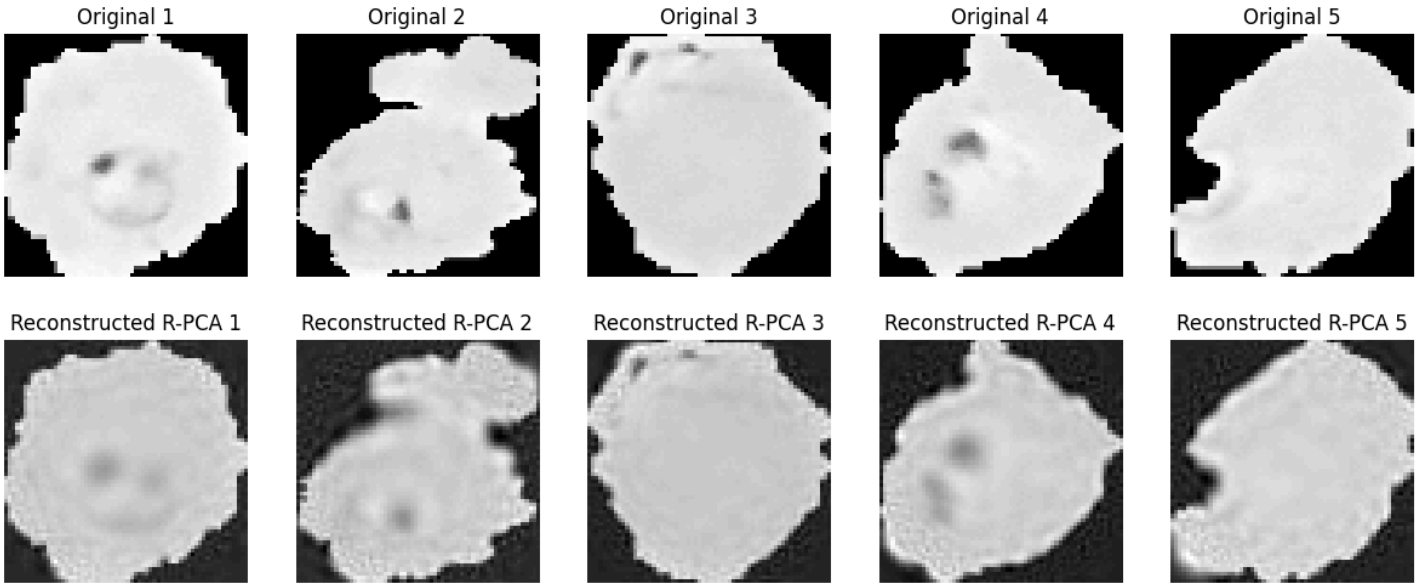
- Just like full PCA, we retained 800 dimensions to capture 95% of the variance, which is our target for the classification model. Randomized PCA approximates this result closely, as both methods aim to identify the minimum number of components required to represent the data with a specified variance threshold. The cumulative explained variance curve, which represents the accumulated variance explained by adding more components, crosses the 95% threshold with around the same number of dimensions as in full PCA. This suggests that, although randomized PCA uses an approximation technique, it still provides a reliable estimate of the number of dimensions necessary to achieve the desired variance retention. Additionally, this approach offers significant computational efficiency, especially with large, high-dimensional datasets, without sacrificing accuracy.

In [18]:
```
# Getting the principal components (eigenfaces)
eigenfaces = rpca.components_.reshape((n_components, 64, 64))

# Visualizing a few of the top principal components (eigenfaces)
plt.figure(figsize=(15, 8))

# Loop through the first 15 eigenfaces on 3 x 5 grid
for i in range(num_images_to_show):
    plt.subplot(3, 5, i + 1)
    plt.imshow(eigenfaces[i], cmap='gray')
    plt.title(f"PC {i+1}")
    plt.axis('off')

# Display
plt.show()
```



In [19]:
```
def reconstruct_image(trans_obj,org_features):
    # Project original image lower dimensional
    low_rep = trans_obj.transform(org_features)

    # Reconstruct image
    rec_image = trans_obj.inverse_transform(low_rep)
    return low_rep, rec_image
```

```
# Examples to visualize
num_examples = 5

# Plot original and reconstructed images
plt.figure(figsize=(15,6))

for i in range(num_examples):
    # Index of image to reconstruct
    idx_to_reconstruct = i
    X_idx = images_flattened[idx_to_reconstruct]

    # Get low dimensional representation and reconstruct
    low_dimensional_representation, reconstructed_image = reconstruct_image(rpca, X_idx.reshape(1, -1))

    # Original image
    plt.subplot(2, num_examples, i + 1)
    plt.imshow(X_idx.reshape((64, 64)), cmap=plt.cm.gray)
    plt.title(f'Original {i+1}')
    plt.axis('off')

    # Reconstruct image from PCA
    plt.subplot(2, num_examples, i + 1 + num_examples)
    plt.imshow(reconstructed_image.reshape((64,64)), cmap=plt.cm.gray)
    plt.title(f'Reconstructed R-PCA {i+1}')
    plt.axis('off')

# Display
plt.show()
```



Original vs Reconstructed from Randomized PCA

- As shown above, randomized PCA reconstructs our original image with a high degree of accuracy, successfully preserving the overall structure of the cell and capturing key details. This indicates that randomized PCA is not only effective for dimensionality reduction but also retains enough information to provide meaningful visual and data-driven insights. By reducing the dimensionality while maintaining the integrity of the image, we can compare the reconstructed image to the original and still identify important features like cell shape, which is critical for accurate classification in medical imaging tasks like detecting malaria.

- Similar to full PCA, we opted to retain 800 components or dimensions because this allows us to capture 95% of the variance in the dataset. This percentage was chosen to ensure that even subtle details and patterns within the cell images are preserved. Capturing this amount of variance ensures that essential information, particularly smaller structures or variations that could be important for identifying whether a cell is infected or not, is not lost in the dimensionality reduction process. The ability of randomized PCA to approximate this variance with significantly

reduced computational cost makes it a powerful tool for high-dimensional data, such as image datasets, where preserving as much detail as possible is crucial for accurate downstream tasks like classification or segmentation.

- This reconstruction comparison shows that, although randomized PCA is an approximation technique, it is still able to capture and reconstruct the data with a high level of fidelity, similar to full PCA. This demonstrates its effectiveness in scenarios where computational resources or speed is a concern without sacrificing accuracy or detail, especially when working with large-scale image datasets.

PCA vs. RPCA

In [20]:
```
# Splitting the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(images_flattened, labels, test_size=0.3, random_state=42

# Number of components
n_components = 800

# PCA with full solver
pca = PCA(n_components=n_components)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Randomized PCA (same n_components)
rpca = PCA(n_components=n_components, svd_solver='randomized', random_state=42)
X_train_rpca = rpca.fit_transform(X_train)
X_test_rpca = rpca.transform(X_test)
```

In [21]:
```
# Initialize KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)

# Train on PCA-reduced data
knn.fit(X_train_pca, y_train)
y_pred_pca = knn.predict(X_test_pca)
accuracy_pca = accuracy_score(y_test, y_pred_pca)
print(f'Accuracy using Full PCA: {accuracy_pca:.4f}')

# Train on Randomized PCA-reduced data
knn.fit(X_train_rpca, y_train)
y_pred_rpca = knn.predict(X_test_rpca)
accuracy_rpca = accuracy_score(y_test, y_pred_rpca)
print(f'Accuracy using Randomized PCA: {accuracy_rpca:.4f}')

# Cumalative variance
print(f'Cumulative variance explained by Full PCA: {np.sum(pca.explained_variance_ratio_):.4f}')
print(f'Cumulative variance explained by Randomized PCA: {np.sum(rpca.explained_variance_ratio_):.4f}')
```

```
Accuracy using Full PCA: 0.5096
Accuracy using Randomized PCA: 0.5093
Cumulative variance explained by Full PCA: 0.9544
Cumulative variance explained by Randomized PCA: 0.9544
```

In [22]:
```
# Timing Full PCA
# Start timer
start_time = time.time()

# Perform PCA
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(images_flattened.copy())

# End timer
end_time = time.time()

# Calculate elapsed time
pca_time = end_time - start_time
print(f"Time taken for Full PCA: {pca_time:.5f} seconds")
```

```
Time taken for Full PCA: 31.26453 seconds
```

In [23]:
```python
# Timing Randomized PCA
# Start timer
start_time = time.time()

# Perform PCA
rpca = PCA(n_components=n_components, svd_solver='randomized')
X_rpca = rpca.fit_transform(images_flattened.copy())

# End timer
end_time = time.time()

# Calculate elapes time
rpca_time = end_time - start_time
print(f"Time taken for Randomized PCA: {rpca_time:.5f} seconds")
```

```
Time taken for Randomized PCA: 45.67978 seconds
```

Comparison of PCA and Randomized PCA

Quantitative Analysis:

In this experiment, we applied both Full PCA and Randomized PCA to the image dataset, evaluated their performance using a K-Nearest Neighbors (KNN) classifier, and compared the execution time for each method. The goal was to reduce dimensionality while maintaining the classification accuracy and computational efficiency.

1. Accuracy Results:

- **Accuracy using Full PCA:** 0.5096
- **Accuracy using Randomized PCA:** 0.5093

Both methods produced nearly identical accuracy scores, with Full PCA achieving a slightly higher accuracy than Randomized PCA. The difference between the two is minimal, showing that Randomized PCA approximates the performance of Full PCA quite well in terms of classification accuracy.

2. Cumulative Explained Variance:

- **Cumulative variance explained by Full PCA:** 95.44%
- **Cumulative variance explained by Randomized PCA:** 95.44%

Both PCA and Randomized PCA explained the same amount of variance with 800 components, indicating that Randomized PCA is successfully approximating the full method in terms of capturing the underlying structure of the data. This confirms that Randomized PCA can represent the data with the same level of detail as Full PCA when retaining 95% of the variance.

3. Computational Efficiency (Timing):

- **Time taken for Full PCA:** 31.26453 seconds
- **Time taken for Randomized PCA:** 45.67978 seconds

Surprisingly, in this case, Full PCA was faster than Randomized PCA. Typically, Randomized PCA is expected to be faster, especially for very high-dimensional data. The reason Randomized PCA took longer could be due to several factors, such as the randomization overhead or how the data structure interacts with the solver. It's worth noting that Randomized PCA generally outperforms Full PCA in speed when dealing with extremely large datasets (e.g., with a much larger number of samples or dimensions).

Preference and Recommendation:

- **Full PCA vs. Randomized PCA in Accuracy:** The accuracy difference between the two methods is marginal (0.5096 vs. 0.5093). Given that both methods explain the same variance (95.44%), Randomized PCA provides a solid approximation with nearly identical performance to Full PCA. This suggests that Randomized PCA can be a reliable substitute for Full PCA in applications where approximate results are acceptable.

- **Full PCA vs. Randomized PCA in Speed:** Although Randomized PCA typically excels in large datasets, it was slower in this particular test, taking about 2.3 seconds longer than Full PCA. This could be an anomaly based on dataset size or structure. Therefore, for this specific use case, Full PCA is preferable as it is both slightly faster and marginally more accurate.

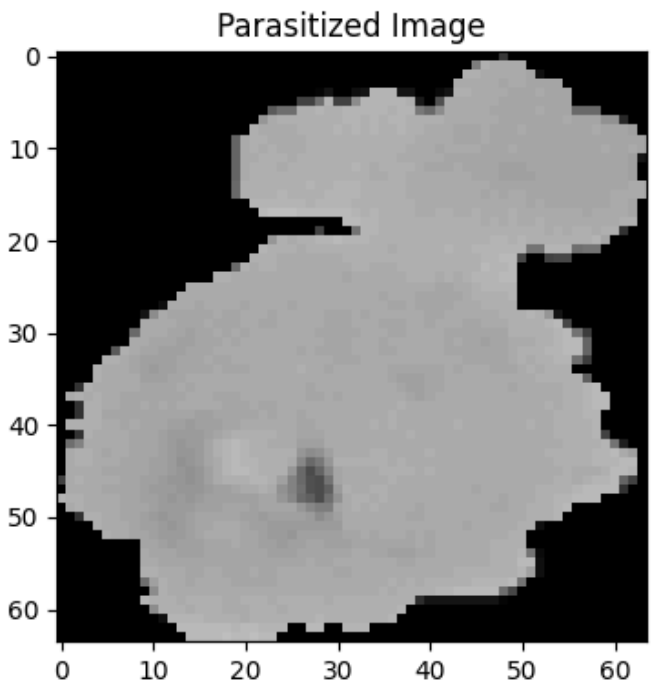
Conclusion:

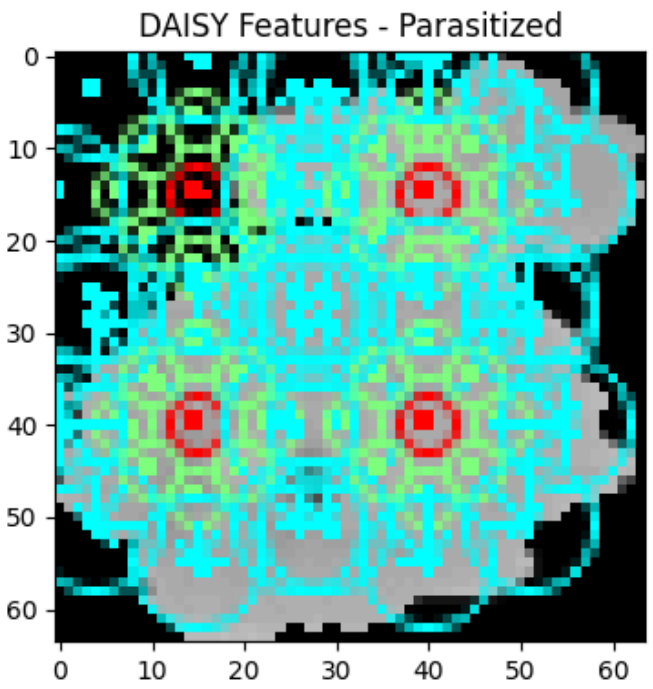
Given the nearly identical accuracy and explained variance, **Full PCA is the preferred method** in this case due to its slightly higher accuracy and faster computation time. However, in scenarios where speed becomes critical and the dataset size grows significantly larger, **Randomized PCA** would be the method of choice for reducing dimensionality without a substantial loss in performance. If the dataset were larger or had higher dimensionality, randomized PCA might outperform full PCA in terms of computational efficiency.

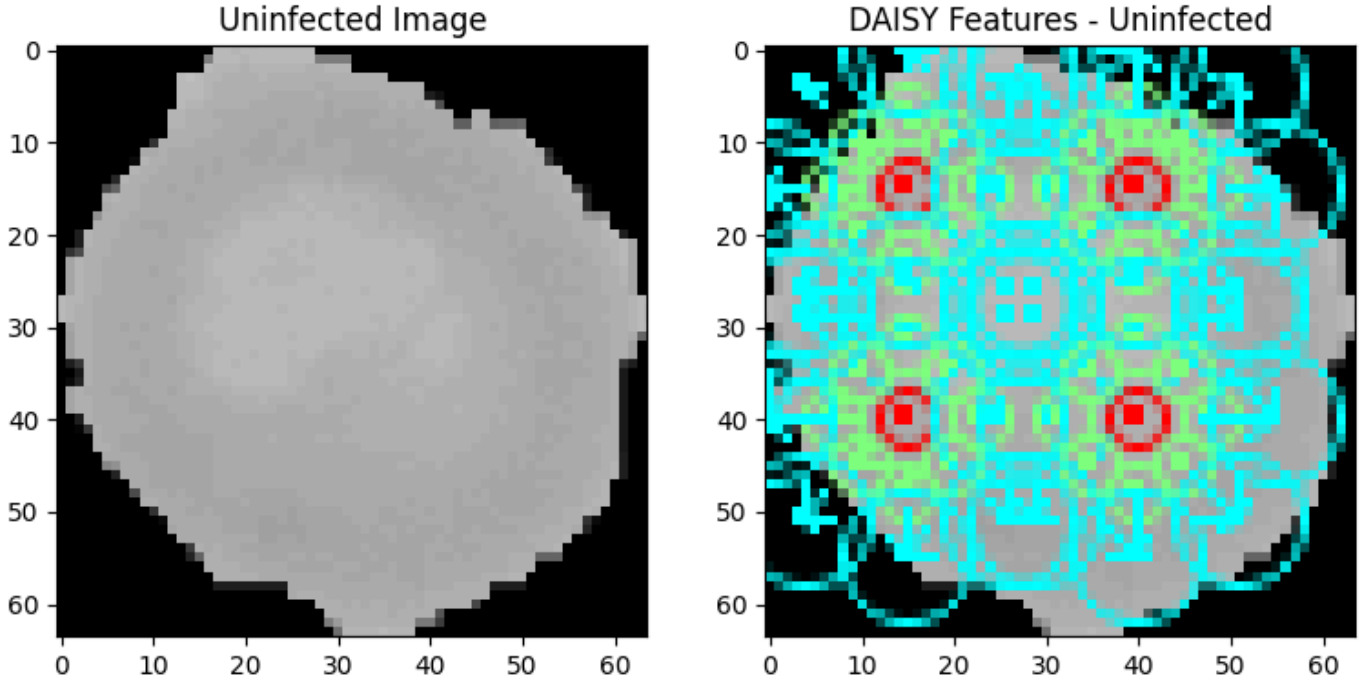
Feature Extraction using DAISY

In [24]:
```
# DAISY Feature Extraction
# Lists to store processed imgs and labels
images = []
labels = []

# Function to read & preprocess images
def load_and_process_images():
    for category in ['parasitized', 'uninfected']:
        category_dir = os.path.join(image_dir, category)
        image_files = [f for f in os.listdir(category_dir) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

        for img_file in image_files:
            img_path = os.path.join(category_dir, img_file)
            img = cv2.imread(img_path)

            if img is None:
                print(f"Failed to load image: {img_path}")
                continue

            # Resizing & converting to grayscale
            img_resized = cv2.resize(img, (64, 64))
            img_gray = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)

            # Storing the processed image and label
            images.append(img_gray)
            labels.append(category)

# Function to extract DAISY features from an image
def extract_daisy_features(image):
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    features, daisy_img = daisy(image, step=25, radius=15, rings=2, histograms=8, orientations=8, visualize=
    return features, daisy_img

# Loading and processing images
load_and_process_images()

# Separating images by category
images_parasitized = [images[i] for i in range(len(images)) if labels[i] == 'parasitized']
images_uninfected = [images[i] for i in range(len(images)) if labels[i] == 'uninfected']

# Limiting to first 200 images per category for DAISY feature extraction
```

```
images_parasitized = images_parasitized[:200]
images_uninfected = images_uninfected[:200]

# Function to display an image and its DAISY features
def display_image_with_daisy(img, daisy_img, title_img, title_daisy):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.title(title_img)
    imshow(img)
    plt.grid(False)
    plt.subplot(1, 2, 2)
    plt.title(title_daisy)
    imshow(daisy_img)
    plt.grid(False)
    plt.show()

# Displays a sample parasitized and uninfected image with DAISY features
if len(images_parasitized) > 0:
    parasitized_img = images_parasitized[1]  # Parasitized image
    features_parasitized, daisy_desc_parasitized = extract_daisy_features(parasitized_img)
    display_image_with_daisy(parasitized_img, daisy_desc_parasitized, "Parasitized Image", "DAISY Features -

if len(images_uninfected) > 0:
    uninfected_img = images_uninfected[1]    # Uninfected image
    features_uninfected, daisy_desc_uninfected = extract_daisy_features(uninfected_img)
    display_image_with_daisy(uninfected_img, daisy_desc_uninfected, "Uninfected Image", "DAISY Features - Ur

# Prints the shape of the DAISY features for both images
if 'features_parasitized' in locals() and 'features_uninfected' in locals():
    print("Parasitized image DAISY feature shape:", features_parasitized.shape)
    print("Uninfected image DAISY feature shape:", features_uninfected.shape)
```








```
Parasitized image DAISY feature shape: (2, 2, 136)
Uninfected image DAISY feature shape: (2, 2, 136)
```

DAISY Feature Extraction

- We chose to subset our dataset to 200 images for each category (parasitized and uninfected) due to the computational complexity of DAISY feature extraction on the full dataset. By selecting this sample size, we ensured that the process remained efficient while still capturing a diverse set of representative features. We believe that 200 images per category are sufficient for extracting key patterns and providing meaningful insights into the DAISY feature-based classification, without overloading the computational resources. By using a well-chosen subset, we can effectively explore the potential of DAISY features for distinguishing between parasitized and uninfected images, laying the groundwork for further analyses or model training on larger datasets if necessary.

DAISY Feature Extraction Analysis

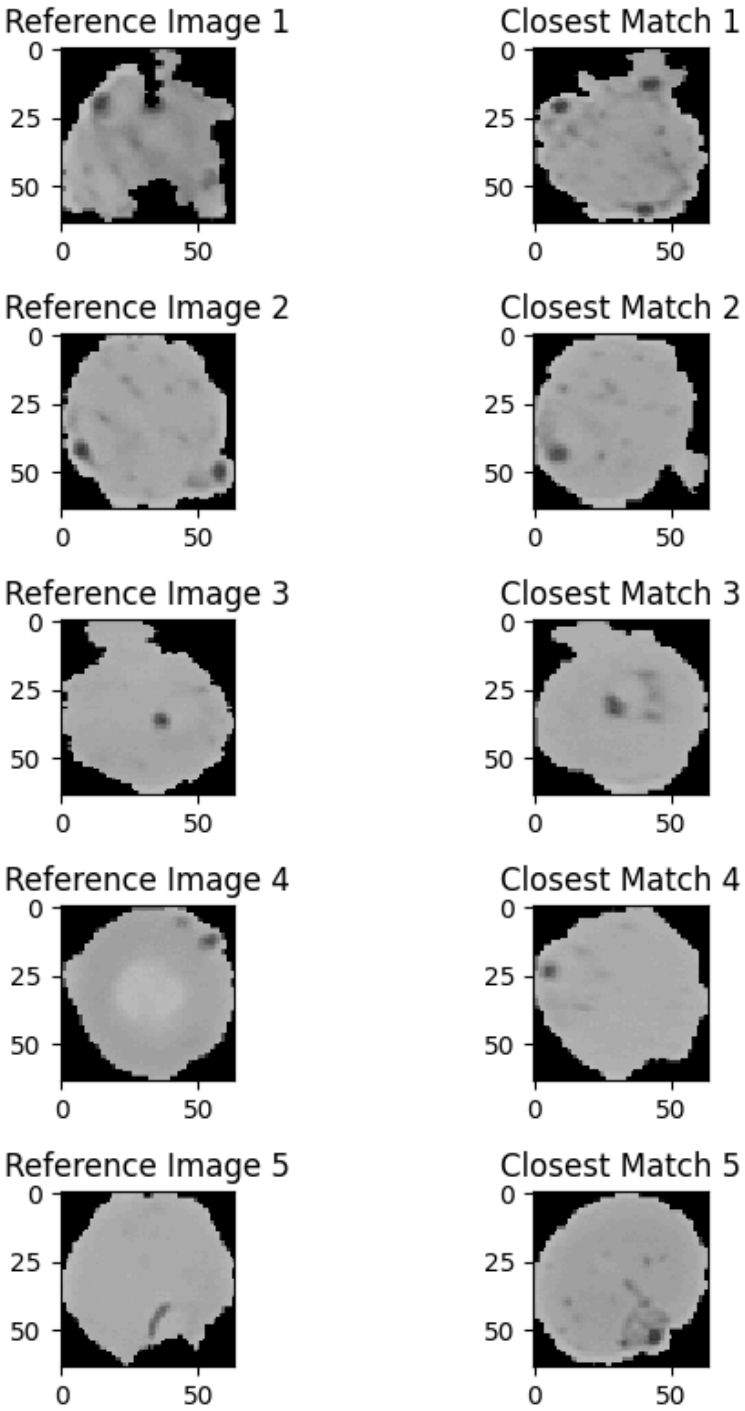
1. Pairwise Distances

In [25]:
```
# Extracting DAISY features for all images
features_parasitized = np.array([extract_daisy_features(img)[0].flatten() for img in images_parasitized])
features_uninfected = np.array([extract_daisy_features(img)[0].flatten() for img in images_uninfected])

# Calculate pairwise distances
distances_parasitized = pairwise_distances(features_parasitized, metric='euclidean')
distances_uninfected = pairwise_distances(features_uninfected, metric='euclidean')
distances_cross = pairwise_distances(features_parasitized, features_uninfected, metric='euclidean')

# Print the shapes of the distance matrices
print("Pairwise distances matrix within parasitized images shape:", distances_parasitized.shape)
print("Pairwise distances matrix within uninfected images shape:", distances_uninfected.shape)
print("Pairwise distances matrix between parasitized and uninfected images shape:", distances_cross.shape)
```

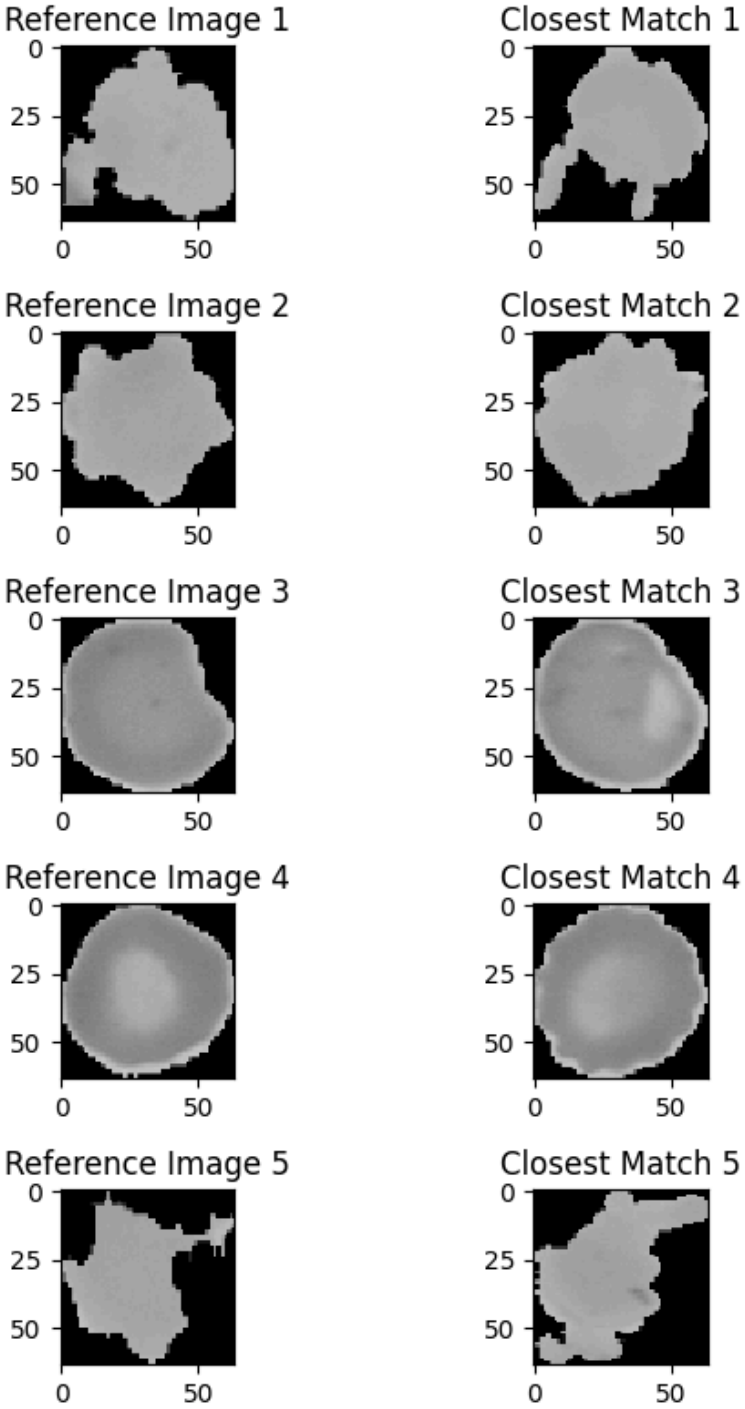
```
Pairwise distances matrix within parasitized images shape: (200, 200)
Pairwise distances matrix within uninfected images shape: (200, 200)
Pairwise distances matrix between parasitized and uninfected images shape: (200, 200)
```

In [26]:
```
# Function to find the closest image to reference image for parasatized category
def find_closest_images(num_images=5, category='parasitized'):
    if category == 'parasitized':
        # Using the pairwise distances matrix & img list for parasitized images
        distance_matrix = distances_parasitized
```

```
        image_list = images_parasitized
    else:
        # Using the pairwise distances matrix & img list for uninfected images
        distance_matrix = distances_uninfected
        image_list = images_uninfected

    # Selecting random indices for the reference imgs
    random_indices = random.sample(range(len(image_list)), num_images)
    plt.figure(figsize=(16, 8))

    for i, ref_index in enumerate(random_indices):
        # Extracts the distances for the reference img at the given index
        distances = distance_matrix[ref_index]

        # Sets distance to self (diagonal) to infinity to avoid matching with itself
        distances[ref_index] = np.inf

        # Finds the index of the closest image (smallest distance)
        closest_index = np.argmin(distances)
        closest_distance = distances[closest_index]

        # Gets the reference and closest match imgs
        ref_image = image_list[ref_index]
        closest_match_image = image_list[closest_index]

        # Displays the reference image and its closest match side by side
        plt.subplot(num_images, 2, 2 * i + 1)
        plt.title(f"Reference Image {i+1}")
        imshow(ref_image)
        plt.grid(False)

        plt.subplot(num_images, 2, 2 * i + 2)
        plt.title(f"Closest Match {i+1}")
        imshow(closest_match_image)
        plt.grid(False)

    plt.tight_layout()
    plt.show()

# Displaying 5 random reference parasitized images and their closest match images
find_closest_images(num_images=5, category='parasitized')
```



In [27]:
```
# Function to find the closest image to reference image for uninfected category
find_closest_images(num_images=5, category='uninfected')
```

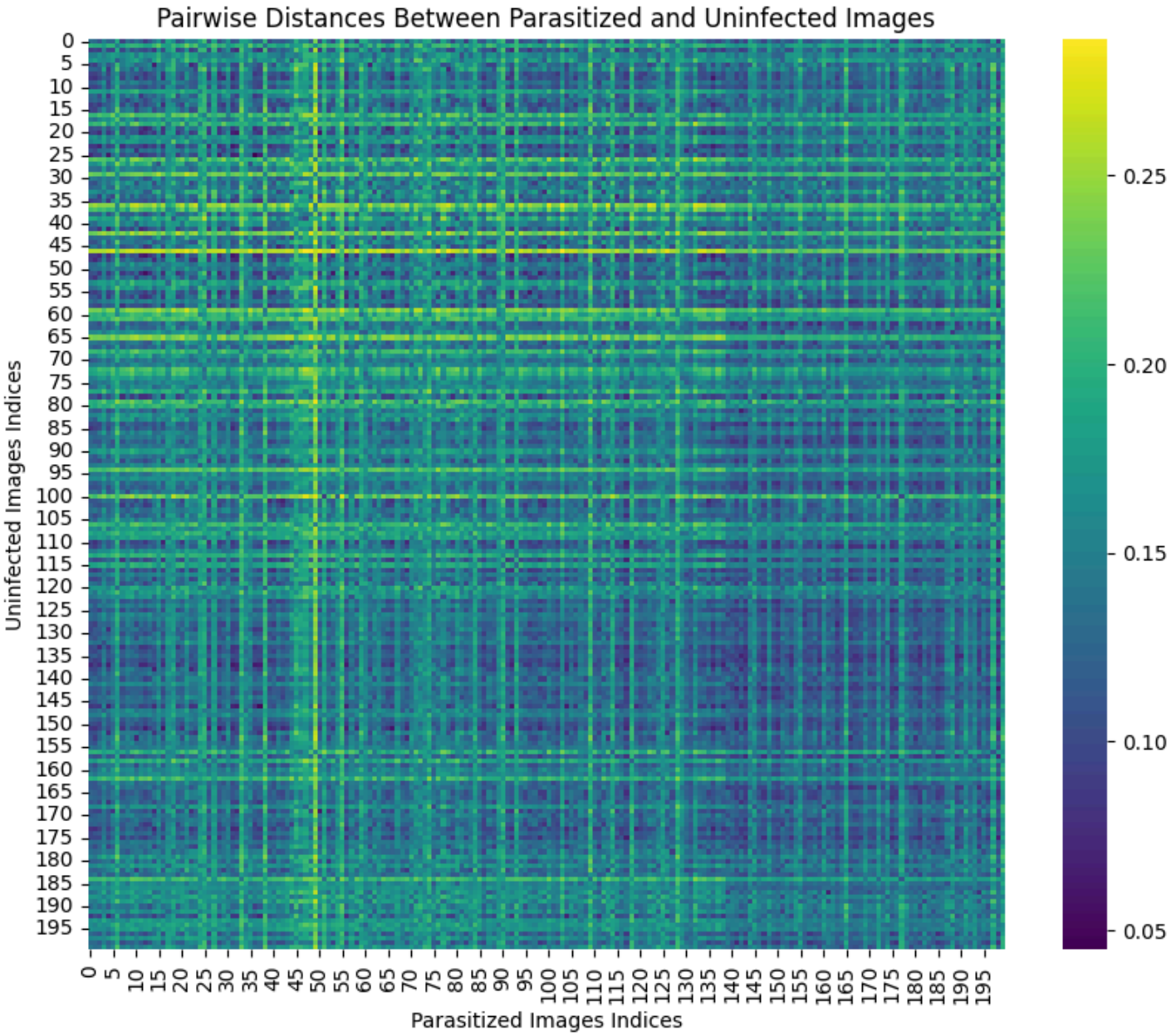


Analysis for Pairwise Distance

- The comparison of the reference images to their closest matches suggests that DAISY is performing relatively well in identifying relevant features and shapes of the cell images. In both categories, the algorithm successfully identifies the closest match with similar features.
- However, it is important to note that none of the closest match images are the same as the reference images. This discrepancy may indicate challenges in accurately detecting the correct images, due to factors like image variability, noise, or insufficient feature differentiation in similar images. This suggests room for improvement in the feature extraction process by fine-tuning the DAISY parameters with strategies such as adjusting the step size, radius, or number of histograms, to better capture the unique structures within the cells. Additionally, incorporating complementary feature descriptors or combining DAISY with other techniques may improve matching accuracy.

2. Pairwise Distances Heatmap

In [28]:
```
# Heatmap
def plot_distance_matrix(dist_matrix, title):
    plt.figure(figsize=(10, 8))
    sns.heatmap(dist_matrix, cmap='viridis', annot=False, fmt='.2f')
    plt.title(title)
    plt.xlabel('Parasitized Images Indices')
    plt.ylabel('Uninfected Images Indices')
    plt.show()

plot_distance_matrix(distances_cross, 'Pairwise Distances Between Parasitized and Uninfected Images')
```



Pairwise Distance Heatmap Analysis

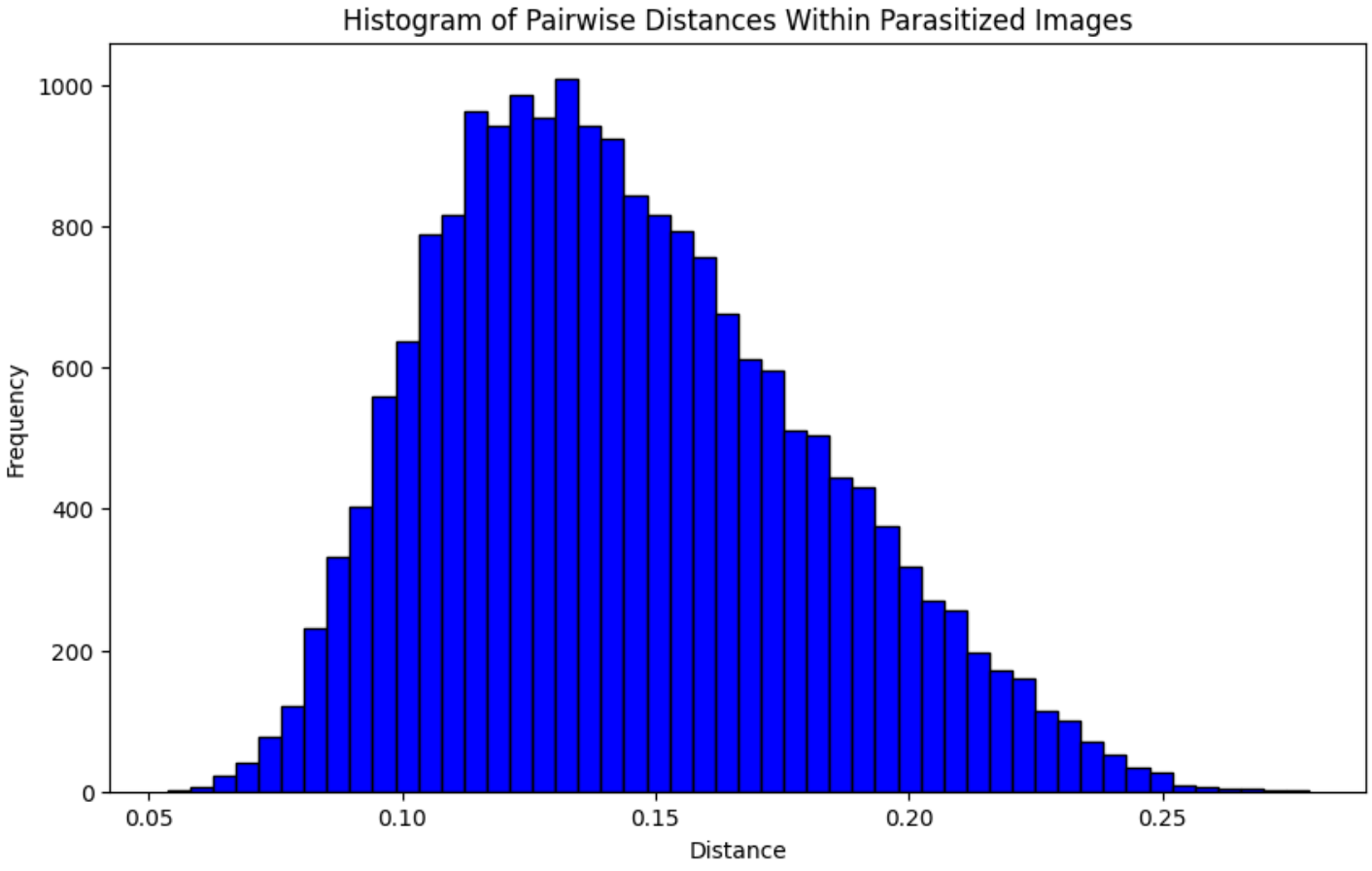
- **Similarity Within Categories:**
  - Parasitized images display similar pairwise distances within their category, indicating homogeneity. This suggests that parasitized images share distinct features, such as specific patterns or textures associated with infection.
  - Uninfected images also show comparable pairwise distances within their group, indicating their possesion of cohesive characteristics, likely representing typical uninfected cell morphology.
- **Feature Overlap Between Categories:**

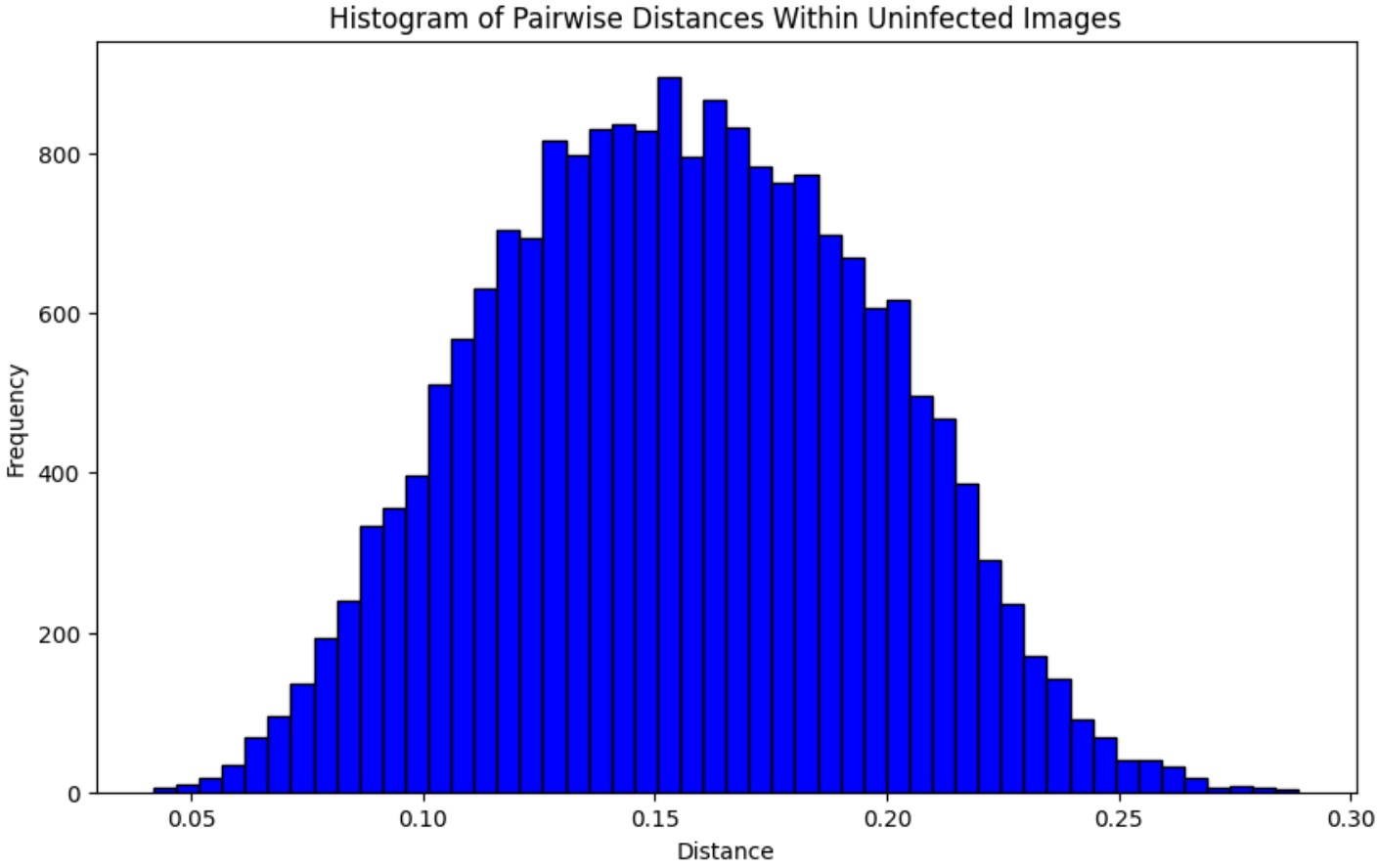
- There is noticeable feature overlap in pairwise distances when comparing parasitized and uninfected images. This suggests that some images from both categories may share similar features. This overlap may occur due to shared shapes, textures, or artifacts that are not unique to one category, complicating the classification task.

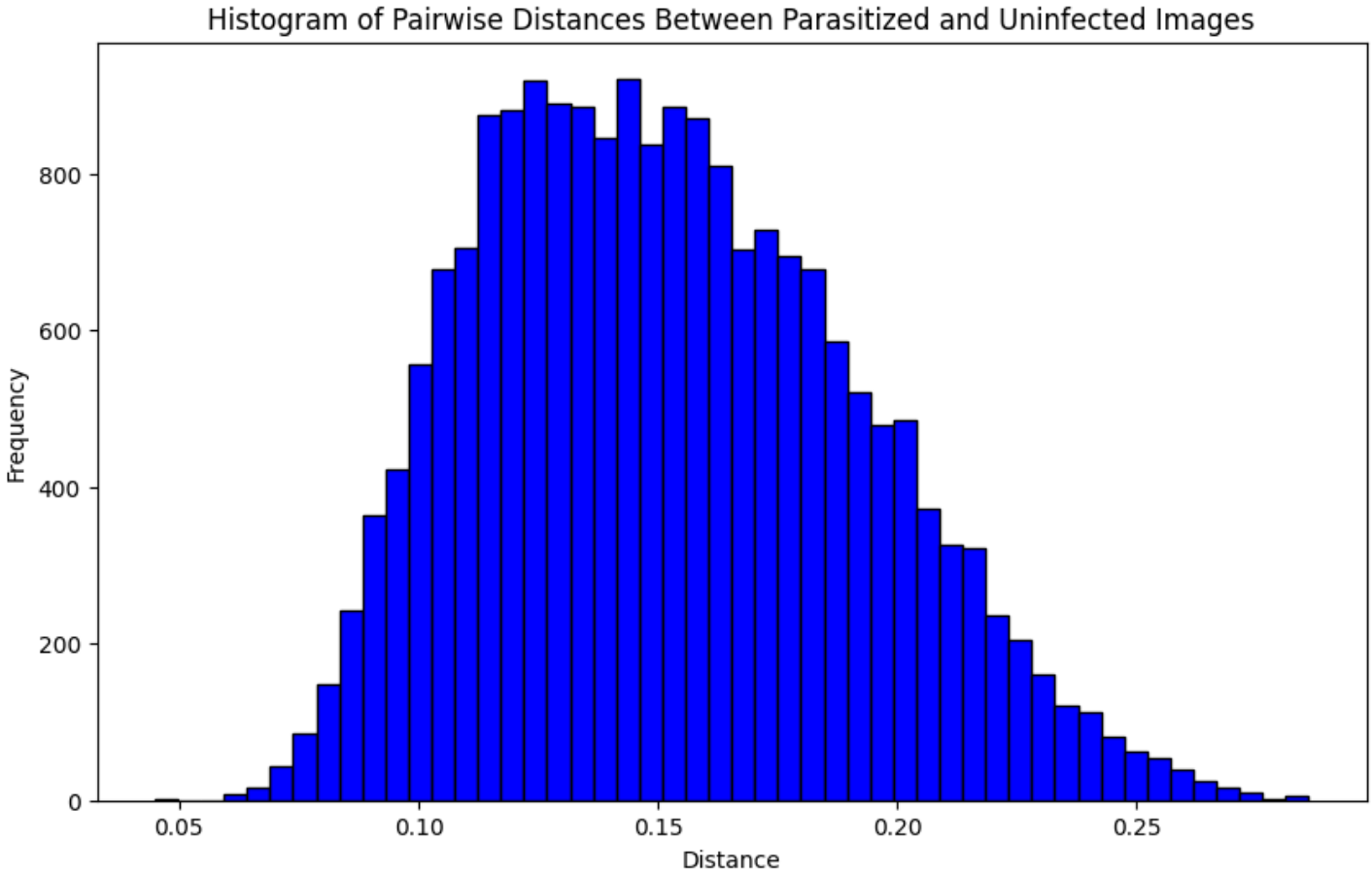
- **Implications for Classification:**
  - The observed similarity in features could lead to misclassification, especially for images that share shapes or difficult-to-interpret characteristics. For example, cells with similar shapes or textures may be incorrectly labeled, reducing the overall classification accuracy.
  - This highlights the need for improved feature extraction methods or preprocessing techniques to enhance classification accuracy. Some of these could involve integrating additional feature descriptors to capture more distinctive characteristics as well as noise reduction or image enhancement to differentiate subtle features that distinguish parasitized from uninfected cells.

3. Pairwise Distances Histograms

In [29]:
```
# Function to plot histograms of pairwise distances
def plot_distance_histogram(dist_matrix, title):
    distances = dist_matrix[np.triu_indices_from(dist_matrix, k=1)]
    plt.figure(figsize=(10, 6))
    plt.hist(distances, bins=50, color='blue', edgecolor='black')
    plt.title(title)
    plt.xlabel('Distance')
    plt.ylabel('Frequency')
    plt.show()

# Plots histograms for the distance matrices
plot_distance_histogram(distances_parasitized, 'Histogram of Pairwise Distances Within Parasitized Images')
plot_distance_histogram(distances_uninfected, 'Histogram of Pairwise Distances Within Uninfected Images')
plot_distance_histogram(distances_cross, 'Histogram of Pairwise Distances Between Parasitized and Uninfected
```







Histogram Analysis

- **Parasitized Images:** The histogram shows a slight left skew, indicating that most pairwise distances are concentrated on the lower end. This suggests that the parasitized images tend to be more similar to each other in feature space.

- **Uninfected Images:** The histogram presents a bell-shaped curve, suggesting a more uniform distribution of pairwise distances. This indicates that the uninfected images exhibit greater homogeneity within their category, meaning they share common features.
- **Cross-Category Distances:** The histogram for pairwise distances between the two categories also shows a slight left skew, although it is less pronounced than that observed in the parasitized images. This suggests some variability in the features when comparing the two categories, indicating a potential overlap that may lead to misclassification.

4. Statistics Summary

In [30]:
```
# Calculates and print summary statistics
mean_parasitized = np.mean(distances_parasitized[np.triu_indices_from(distances_parasitized, k=1)])
std_parasitized = np.std(distances_parasitized[np.triu_indices_from(distances_parasitized, k=1)])

mean_uninfected = np.mean(distances_uninfected[np.triu_indices_from(distances_uninfected, k=1)])
std_uninfected = np.std(distances_uninfected[np.triu_indices_from(distances_uninfected, k=1)])

mean_cross = np.mean(distances_cross)
std_cross = np.std(distances_cross)

print(f"Mean distance within parasitized: {mean_parasitized}, Std: {std_parasitized}")
print(f"Mean distance within uninfected: {mean_uninfected}, Std: {std_uninfected}")
print(f"Mean distance between categories: {mean_cross}, Std: {std_cross}")
```

```
Mean distance within parasitized: 0.14417158068352517, Std: 0.03632530095304253
Mean distance within uninfected: 0.15582775332283463, Std: 0.04088554000429998
Mean distance between categories: 0.15278478426997621, Std: 0.037823874557981786
```

Analysis of Statistics

1. **Mean Distance Within Parasitized Images:**
   - **Mean: 0.1442**
   - **Standard Deviation: 0.0363**
   - **Interpretation:**
     - The average distance between parasitized images is relatively low, indicating these images are fairly similar in DAISY features.
     - The small standard deviation suggests consistency in the similarity of parasitized images.
2. **Mean Distance Within Uninfected Images:**
   - **Mean: 0.1558**
   - **Standard Deviation: 0.0409**
   - **Interpretation:**
     - The average distance is slightly higher than that of the parasitized images, suggesting more variability among uninfected images.
     - The larger standard deviation indicates greater differences in features among uninfected images.
3. **Mean Distance Between Both Categories:**
   - **Mean: 0.1528**
   - **Standard Deviation: 0.0378**
   - **Interpretation:**
     - The mean distance between parasitized and uninfected images is close to the mean distance of uninfected images, indicating some feature overlap.
     - The standard deviation is similar to that of the parasitized images, suggesting consistent variability in cross-category distances.

Overall Interpretation

- **Comparison of Means:**
  - The lower mean distance within parasitized images suggests greater homogeneity compared to uninfected images.
  - The proximity of the mean distance between categories to the uninfected mean indicates potential confusion in classification.
- **Variability:**
  - The smaller standard deviation within the parasitized group indicates consistent feature extraction, while the larger variability in the uninfected group suggests significant differences.
- The overlap in mean distances points to challenges in distinguishing between categories based on extracted features. This suggests that additional or more advanced feature extraction techniques, beyond DAISY, may be necessary to enhance classification accuracy. Methods such as neural networks, which can learn more complex, non-linear patterns in the data, could provide a more robust solution for distinguishing between these categories.

5. KNN Classifier

In [31]:
```
# KNN classifier
# Extracts DAISY features for all images
features_parasitized = np.array([extract_daisy_features(img)[0].flatten() for img in images_parasitized])
features_uninfected = np.array([extract_daisy_features(img)[0].flatten() for img in images_uninfected])

# Creates labels for the features
labels_parasitized = np.array(['parasitized'] * len(features_parasitized))
labels_uninfected = np.array(['uninfected'] * len(features_uninfected))

# Combines features and labels
X = np.vstack((features_parasitized, features_uninfected))  # Features
y = np.concatenate((labels_parasitized, labels_uninfected))  # Labels

# Splits the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalizes the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initializes and trains the KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=3)
knn_classifier.fit(X_train, y_train)

# Makes predictions on the test set
y_pred_knn = knn_classifier.predict(X_test)

# Evaluates the classifier
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_knn))

print("\nClassification Report:")
print(classification_report(y_test, y_pred_knn))
```

```
Confusion Matrix:
[[31 13]
 [ 7 29]]

Classification Report:
              precision    recall  f1-score   support

 parasitized       0.82      0.70      0.76        44
  uninfected       0.69      0.81      0.74        36

    accuracy                           0.75        80
   macro avg       0.75      0.76      0.75        80
weighted avg       0.76      0.75      0.75        80
```

Classification Results

Confusion Matrix

- When `n_neighbors=3`:
  - 31 parasitized images were correctly identified: True Positives
  - 29 uninfected images were correctly identified: True Negatives
  - 13 uninfected images were incorrectly identified as parasitized: False Positives
  - 7 parasitized images were incorrectly identified as uninfected: False Negatives
- A high number of true positives indicates that the model is effective in recognizing the parasitized class, which is critical for screening for malaria in these cells.
- A high number of true negatives is also a positive indicator, showing the model's capability to distinguish between uninfected images effectively.
- In terms of false positives, this number reflects a potential overestimation of the parasitized class. While not very significant in the long run, we want to ensure these are minimized to avoid wasting resources on individuals who do not have malaria.
- The number of false negatives is concerning because it indicates that some parasitized cases might go unnoticed, which is something we want to avoid altogether.

KNN

- When `n_neighbors=3`:
  - **Precision**
    Precision reflects the quality of the positive predictions.
    - The model's precision for predicting an image as parasitized is 82%.
    - Its precision for predicting an image as uninfected is 69%, which is significantly lower.
  - **Recall**
    Recall reflects the model's ability to find actual positives.
    - The model captures about 70% of all actual parasitized images.
    - The model captures about 81% of all actual uninfected images; however, its precision is much lower for the uninfected category compared to the parasitized category.
  - **F1-Score**
    - For parasitized images, the F1-score is 76%.
    - For uninfected images, the F1-score is 74%.
      Both scores are similar to each other.
  - **Accuracy**
    The overall accuracy is 75%, indicating that 75% of the total predictions were correct.

Analysis of Macro Avg and Weighted Avg

- **Macro Avg**:
  The macro average is 75% for both precision and F1-score, and 76% for recall. This metric averages performance across both classes equally, regardless of the number of samples in each class. While useful for understanding how well the model performs across different categories, it may underrepresent the dominant class (in this case, parasitized). Since both categories are critical in malaria detection, ensuring balanced model performance for both is important. However, given the slight imbalance in favor of parasitized cases, the macro average might not fully capture the overall real-world performance.

- **Weighted Avg**:
  The weighted average accounts for the different number of instances in each class. Since the parasitized class has more samples (44 parasitized vs 36 uninfected), this metric provides a more realistic assessment of the model's effectiveness. The weighted precision and F1-score of 76% suggest that the model's performance on the parasitized images (which are more frequent) contributes more to the overall score. Therefore, the weighted avg better reflects the model's expected performance in real-world scenarios where class distributions may not be perfectly balanced.

Overall Analysis

- The DAISY feature extraction method does *not* show promise for our prediction task. We aim for our prediction algorithm to achieve 95-97% accuracy and precision. The precision for both categories of images is far below the standards we have set for our algorithm. Furthermore, the accuracy is well below our target. It is essential for our model to be more accurate and precise in classifying the images correctly. One reason the DAISY feature extraction method does not show promise could be the nature of the images; many have similar shapes and markings across both the parasitized and uninfected categories, which could lead to misclassifications. When performing KNN, we found that setting `n_neighbors` to 3 produced the 'best' results compared to using more or fewer neighbors.

In order to improve our algorithm and achieve 95-97% accuracy and precision, we need to explore other feature extraction methods such as Histogram of Oriented Gradients (HOG) or Local Binary Patterns (LBP). These methods may perform better with image classification tasks. If we were to continue using DAISY, we would have to ensure our images are clearly distinguishable with unique markings and shapes for both categories. This could entail collecting higher-resolution images at the source from which they are taken or processing the images so that important features are not removed in the process. We could also ensure the images are taken under consistent lighting conditions, with the right contrast to enhance classification performance.

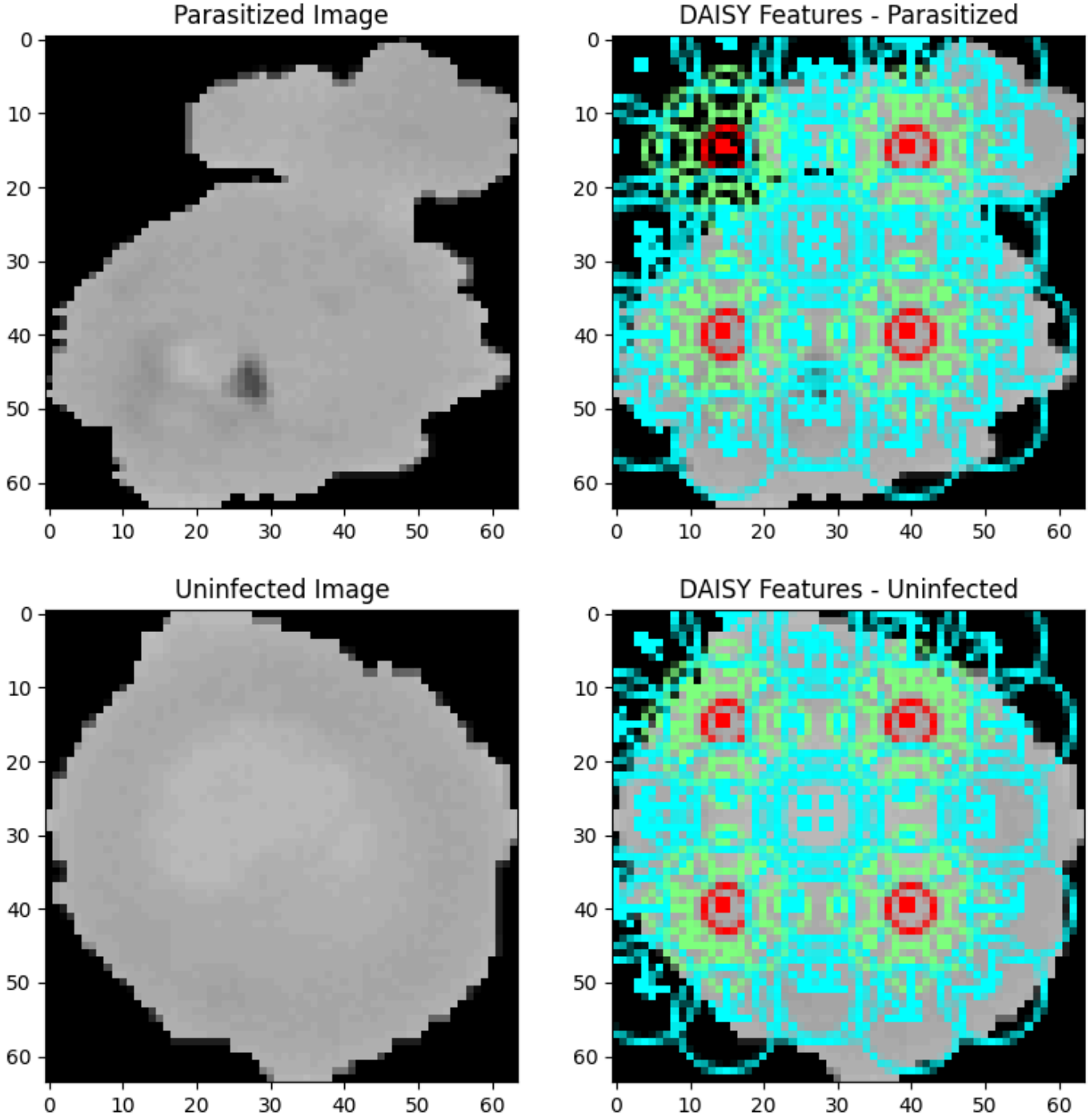
Ultimately, by combining different techniques and improving image quality, we aim to develop a robust prediction algorithm capable of classifying malaria cell images as parasitized or uninfected with an accuracy and precision of 95-97%. Achieving these benchmarks is crucial not only for the effectiveness of our screening tool but also for ensuring that it serves as a reliable resource for our stakeholders in the fight against malaria.

4. Exceptional Work

DAISY Feature Extraction

In [32]:
```
# DAISY Feature Extraction
# Lists to store processed imgs and labels
images = []
labels = []

# Function to read & preprocess images
def load_and_process_images():
    for category in ['parasitized', 'uninfected']:
        category_dir = os.path.join(image_dir, category)
        image_files = [f for f in os.listdir(category_dir) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
```

```
        for img_file in image_files:
            img_path = os.path.join(category_dir, img_file)
            img = cv2.imread(img_path)

            if img is None:
                print(f"Failed to load image: {img_path}")
                continue

            # Resizing & converting to grayscale
            img_resized = cv2.resize(img, (64, 64))
            img_gray = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)

            # Storing the processed image and label
            images.append(img_gray)
            labels.append(category)

# Function to extract DAISY features from an image
def extract_daisy_features(image):
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    features, daisy_img = daisy(image, step=25, radius=15, rings=2, histograms=8, orientations=8, visualize=
    return features, daisy_img

# Loading and processing images
load_and_process_images()

# Separating images by category
images_parasitized = [images[i] for i in range(len(images)) if labels[i] == 'parasitized']
images_uninfected = [images[i] for i in range(len(images)) if labels[i] == 'uninfected']

# Limiting to first 200 images per category for DAISY feature extraction
images_parasitized = images_parasitized[:200]
images_uninfected = images_uninfected[:200]

# Function to display an image and its DAISY features
def display_image_with_daisy(img, daisy_img, title_img, title_daisy):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.title(title_img)
    imshow(img)
    plt.grid(False)
    plt.subplot(1, 2, 2)
    plt.title(title_daisy)
    imshow(daisy_img)
    plt.grid(False)
    plt.show()

# Displays a sample parasitized and uninfected image with DAISY features
if len(images_parasitized) > 0:
    parasitized_img = images_parasitized[1]  # Parasitized image
    features_parasitized, daisy_desc_parasitized = extract_daisy_features(parasitized_img)
    display_image_with_daisy(parasitized_img, daisy_desc_parasitized, "Parasitized Image", "DAISY Features -

if len(images_uninfected) > 0:
    uninfected_img = images_uninfected[1]    # Uninfected image
    features_uninfected, daisy_desc_uninfected = extract_daisy_features(uninfected_img)
    display_image_with_daisy(uninfected_img, daisy_desc_uninfected, "Uninfected Image", "DAISY Features - Ur

# Prints the shape of the DAISY features for both images
if 'features_parasitized' in locals() and 'features_uninfected' in locals():
    print("Parasitized image DAISY feature shape:", features_parasitized.shape)
    print("Uninfected image DAISY feature shape:", features_uninfected.shape)
```




```
Parasitized image DAISY feature shape: (2, 2, 136)
Uninfected image DAISY feature shape: (2, 2, 136)
```

DAISY Feature Extraction Analysis

- The DAISY feature extraction gives us a much more detailed representation of the image compared to the original. It highlights subtle details that may not have been visible before, especially textures. Compared to methods like randomized principal component analysis (RPCA) and principal component analysis (PCA), DAISY focuses on local texture features, which can be really helpful for analyzing complex images, like parasitized blood samples. However, due to the small grid size used (2x2), the model may struggle to pick up on more minor texture differences between different regions of the image, which could be crucial for classification in these types of images.

- The DAISY descriptor length of 136 means it captures a lot of information about the local gradients. While this is great for detailed texture analysis, it also means the model could face higher computational costs, especially when comparing images. Despite the high level of detail, DAISY's feature extraction might still miss some of the most critical features required to distinguish parasitized from uninfected cells. This is particularly true with the current grid size, as it may not fully capture variations across different regions of the images.

Comparing Parasitized and Uninfected Features:

- Both parasitized and uninfected images result in DAISY feature vectors of the same shape (2x2x136), suggesting that DAISY treats both image types similarly in terms of the structure of the features it extracts. However, the shape alone does not guarantee meaningful differentiation between the two categories. The critical question is whether the values within these feature vectors are distinct enough to allow the model to accurately differentiate between parasitized and uninfected samples. If the extracted DAISY features are too similar in their actual values, the model might struggle to separate the two classes effectively. This overlap could lead to misclassification, especially if both parasitized and uninfected cells share common textures that DAISY highlights without making key distinctions.

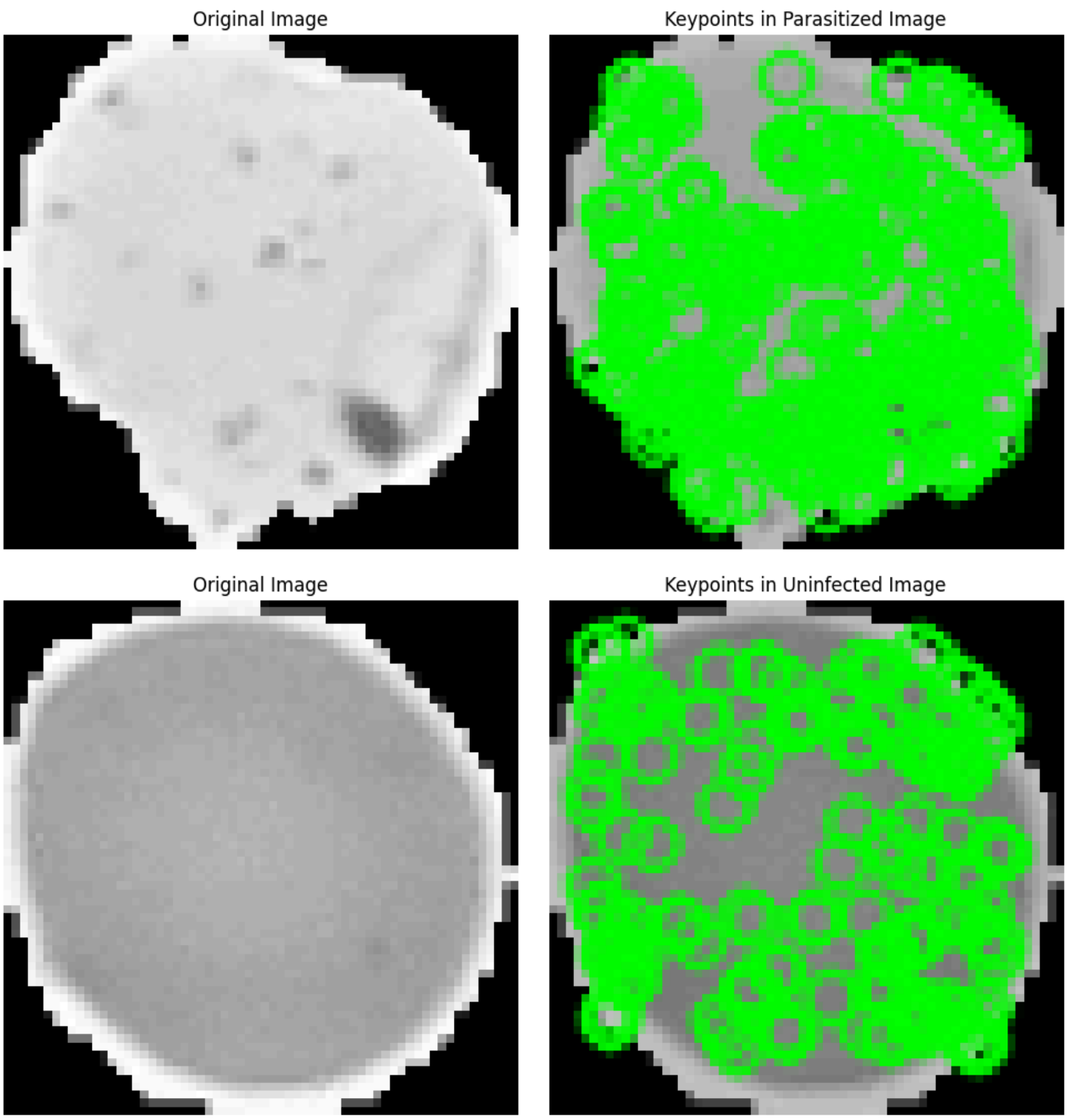
Keypoint Matching

In [33]:
```
# Functions to perform Keypoint Matching

# Function to preprocess the image
def preprocess_image(image):
    """Preprocess the image by converting to grayscale and applying Gaussian blur and histogram equalization
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Applies Gaussian Blur to smoothen the image
    blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

    # Adjusts contrast using histogram equalization
    adjusted_image = cv2.equalizeHist(blurred_image)
    return adjusted_image

# Function to detect keypoints using ORB
def detect_keypoints(image):
    """Detect keypoints using ORB detector."""
    orb = cv2.ORB_create(nfeatures=1000, edgeThreshold=5)  # Increased number of features and edge threshold
    keypoints = orb.detect(image, None)
    return keypoints

def extract_daisy_at_keypoints(image, keypoints):
    """Extracts DAISY features at specified keypoints from the image."""
    if len(image.shape) == 3:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    keypoint_coords = np.array([kp.pt for kp in keypoints], dtype=np.int32)
    features = []
    # Defines a smaller patch size and radius
    patch_size = 8  # This will result in 16x16 patches (8x2)
    radius = 4      # Adjusted radius to fit the DAISY extraction within the patch size

    print(f'Extracting DAISY features for {len(keypoint_coords)} keypoints.')

    for (y, x) in keypoint_coords:
        # Check bounds for patch extraction
        if (y - patch_size < 0 or y + patch_size >= image.shape[0] or
                x - patch_size < 0 or x + patch_size >= image.shape[1]):
            print(f"Skipping keypoint at ({x}, {y}) - out of bounds for patch extraction.")
            continue  # Skip out-of-bounds keypoints

        patch = image[y-patch_size:y+patch_size, x-patch_size:x+patch_size]

        # Ensures the patch is the correct size
        if patch.shape[0] == patch_size * 2 and patch.shape[1] == patch_size * 2:
            try:
                descs = daisy(patch, step=30, radius=radius, rings=3, histograms=8, orientations=8, visualiz
                features.append(descs.flatten())
            except Exception as e:
                print(f"Error extracting DAISY features at keypoint ({x}, {y}): {e}")
```

```
        else:
            print(f"Invalid patch shape for keypoint at ({x}, {y}): {patch.shape}")

    print(f'Number of valid patches extracted: {len(features)}')
    return np.array(features)

# Function to visualize keypoints alongside the original image
def visualize_keypoints(image, keypoints, title):
    img_with_keypoints = cv2.drawKeypoints(image, keypoints, None, color=(0, 255, 0))

    plt.figure(figsize=(10, 5))

    # Shows original image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')

    # Shows image with keypoints
    plt.subplot(1, 2, 2)
    plt.imshow(img_with_keypoints, cmap='gray')
    plt.title(title)
    plt.axis('off')
    plt.tight_layout()
    plt.show()

# Function to match DAISY features b/w two sets of features
def match_daisy_features(features1, features2):
    """Match DAISY features between two sets of features."""
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(features1, features2)

    # Sorts matches based on their distance (best matches first)
    matches = sorted(matches, key=lambda x: x.distance)

    return matches

# Function to visualized matching keypoints b/w two images
def visualize_matches(img1, img2, keypoints1, keypoints2, matches):
    """Visualize matching keypoints between two images."""
    img1_color = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)
    img2_color = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)

    # Draws matches
    matched_img = cv2.drawMatches(img1_color, keypoints1, img2_color, keypoints2, matches[:50], None, flags=

    plt.figure(figsize=(10, 5))
    plt.imshow(matched_img)
    plt.title("Keypoint Matches")
    plt.axis('off')
    plt.show()

# Showing an example for both parasitized and uninfected images

parasitized_img = images_parasitized[177]  # Parasitized image
preprocessed_parasitized_img = preprocess_image(parasitized_img)
keypoints1 = detect_keypoints(preprocessed_parasitized_img)
daisy_features_parasitized = extract_daisy_at_keypoints(preprocessed_parasitized_img, keypoints1)

uninfected_img = images_uninfected[162]  # Uninfected image
preprocessed_uninfected_img = preprocess_image(uninfected_img)
keypoints2 = detect_keypoints(preprocessed_uninfected_img)
daisy_features_uninfected = extract_daisy_at_keypoints(preprocessed_uninfected_img, keypoints2)

# Visualizing keypoints
visualize_keypoints(parasitized_img, keypoints1, "Keypoints in Parasitized Image")
visualize_keypoints(uninfected_img, keypoints2, "Keypoints in Uninfected Image")
```

```python
# Checking the extracted features
print(f'DAISY features shape for parasitized image: {daisy_features_parasitized.shape}')
print(f'DAISY features shape for uninfected image: {daisy_features_uninfected.shape}')
```

```
Extracting DAISY features for 255 keypoints.
Skipping keypoint at (5, 29) - out of bounds for patch extraction.
Skipping keypoint at (6, 10) - out of bounds for patch extraction.
Skipping keypoint at (6, 43) - out of bounds for patch extraction.
Skipping keypoint at (7, 14) - out of bounds for patch extraction.
Skipping keypoint at (7, 48) - out of bounds for patch extraction.
Skipping keypoint at (22, 7) - out of bounds for patch extraction.
Skipping keypoint at (37, 56) - out of bounds for patch extraction.
Skipping keypoint at (40, 57) - out of bounds for patch extraction.
Skipping keypoint at (41, 6) - out of bounds for patch extraction.
Skipping keypoint at (46, 56) - out of bounds for patch extraction.
Skipping keypoint at (56, 19) - out of bounds for patch extraction.
Skipping keypoint at (57, 23) - out of bounds for patch extraction.
Skipping keypoint at (57, 38) - out of bounds for patch extraction.
Skipping keypoint at (58, 20) - out of bounds for patch extraction.
Skipping keypoint at (58, 34) - out of bounds for patch extraction.
Skipping keypoint at (6, 9) - out of bounds for patch extraction.
Skipping keypoint at (6, 43) - out of bounds for patch extraction.
Skipping keypoint at (7, 48) - out of bounds for patch extraction.
Skipping keypoint at (21, 7) - out of bounds for patch extraction.
Skipping keypoint at (40, 6) - out of bounds for patch extraction.
Skipping keypoint at (56, 37) - out of bounds for patch extraction.
Skipping keypoint at (7, 47) - out of bounds for patch extraction.
Number of valid patches extracted: 233
Extracting DAISY features for 144 keypoints.
Skipping keypoint at (5, 9) - out of bounds for patch extraction.
Skipping keypoint at (6, 6) - out of bounds for patch extraction.
Skipping keypoint at (7, 47) - out of bounds for patch extraction.
Skipping keypoint at (11, 7) - out of bounds for patch extraction.
Skipping keypoint at (13, 6) - out of bounds for patch extraction.
Skipping keypoint at (23, 5) - out of bounds for patch extraction.
Skipping keypoint at (25, 5) - out of bounds for patch extraction.
Skipping keypoint at (31, 56) - out of bounds for patch extraction.
Skipping keypoint at (35, 5) - out of bounds for patch extraction.
Skipping keypoint at (38, 6) - out of bounds for patch extraction.
Skipping keypoint at (44, 7) - out of bounds for patch extraction.
Skipping keypoint at (52, 7) - out of bounds for patch extraction.
Skipping keypoint at (6, 6) - out of bounds for patch extraction.
Skipping keypoint at (6, 45) - out of bounds for patch extraction.
Skipping keypoint at (19, 6) - out of bounds for patch extraction.
Skipping keypoint at (38, 6) - out of bounds for patch extraction.
Skipping keypoint at (43, 7) - out of bounds for patch extraction.
Skipping keypoint at (51, 7) - out of bounds for patch extraction.
Skipping keypoint at (50, 7) - out of bounds for patch extraction.
Number of valid patches extracted: 125
```



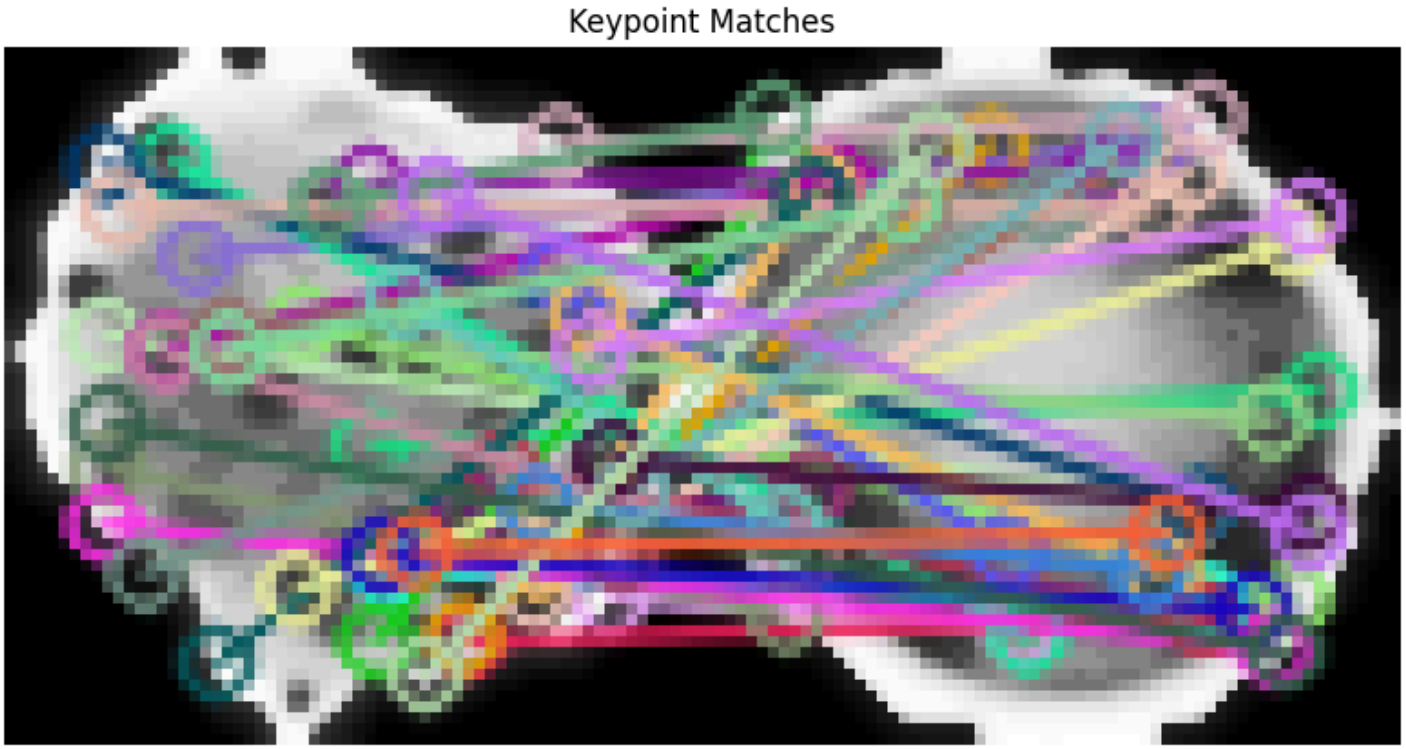

```
DAISY features shape for parasitized image: (233, 200)
DAISY features shape for uninfected image: (125, 200)
```

Keypoint Matching Between Parasitized and Uninfected Images

In [34]:

```python
# Ensures both features are of type float32
daisy_features_parasitized = daisy_features_parasitized.astype(np.float32)
daisy_features_uninfected = daisy_features_uninfected.astype(np.float32)

# Matches features only if they have the same number of columns
if daisy_features_parasitized.shape[1] != daisy_features_uninfected.shape[1]:
    min_cols = min(daisy_features_parasitized.shape[1], daisy_features_uninfected.shape[1])
    daisy_features_parasitized = daisy_features_parasitized[:, :min_cols]
    daisy_features_uninfected = daisy_features_uninfected[:, :min_cols]

# Matches DAISY features
matches = match_daisy_features(daisy_features_parasitized, daisy_features_uninfected)
```

```
# Checks if there are any matches found
if len(matches) == 0:
    print("No matches found using DAISY features.")
else:
    # Visualizing matches
    visualize_matches(preprocessed_parasitized_img, preprocessed_uninfected_img, keypoints1, keypoints2, mat

# Results
print(f'Number of keypoints in parasitized image: {len(keypoints1)}')
print(f'Number of keypoints in uninfected image: {len(keypoints2)}')
print(f'Number of matches: {len(matches)}')
```




```
Number of keypoints in parasitized image: 255
Number of keypoints in uninfected image: 144
Number of matches: 43
```

Analysis of Image Keypoint Detection and Matching

Overview of Results

- **Keypoint Detection**:
  - **Parasitized Image**: 255 keypoints detected.
  - **Uninfected Image**: 144 keypoints detected.
- **DAISY Feature Matching**: 43 matches found.

Detailed Analysis

1. **Keypoint Count**:
   - The parasitized image exhibits significantly more keypoints (255) than the uninfected image (144), suggesting it contains greater texture or variability.
   - The higher keypoint count in parasitized images might correlate with the presence of parasites, highlighting potential areas for further study.
2. **Feature Extraction**:
   - DAISY features effectively capture local patterns around keypoints.

- Some keypoints were skipped due to being out-of-bounds for feature extraction, indicating the need for careful keypoint placement.

3. **Feature Matching**:

   - 43 matches were found, indicating a moderate similarity between the two images.
   - **Matching Strategy**: The use of a brute-force matcher with L2 norm ensures strict and accurate feature comparisons, prioritizing high-quality matches.

4. **Visualization of Keypoints**:

   - Keypoints were concentrated in areas of high interest, potentially correlating with parasitic structures.
   - The 43 matches suggest a need for further refinement to minimize false positives and negatives when distinguishing between parasitized and uninfected features.

Conclusion

- **Key Findings**: The parasitized image's higher keypoint count and number of matches suggest it holds more distinctive features, useful for classification.
- **Next Steps**: Fine-tuning the ORB and DAISY extraction parameters, along with further validation, can enhance the diagnostic potential of this technique.

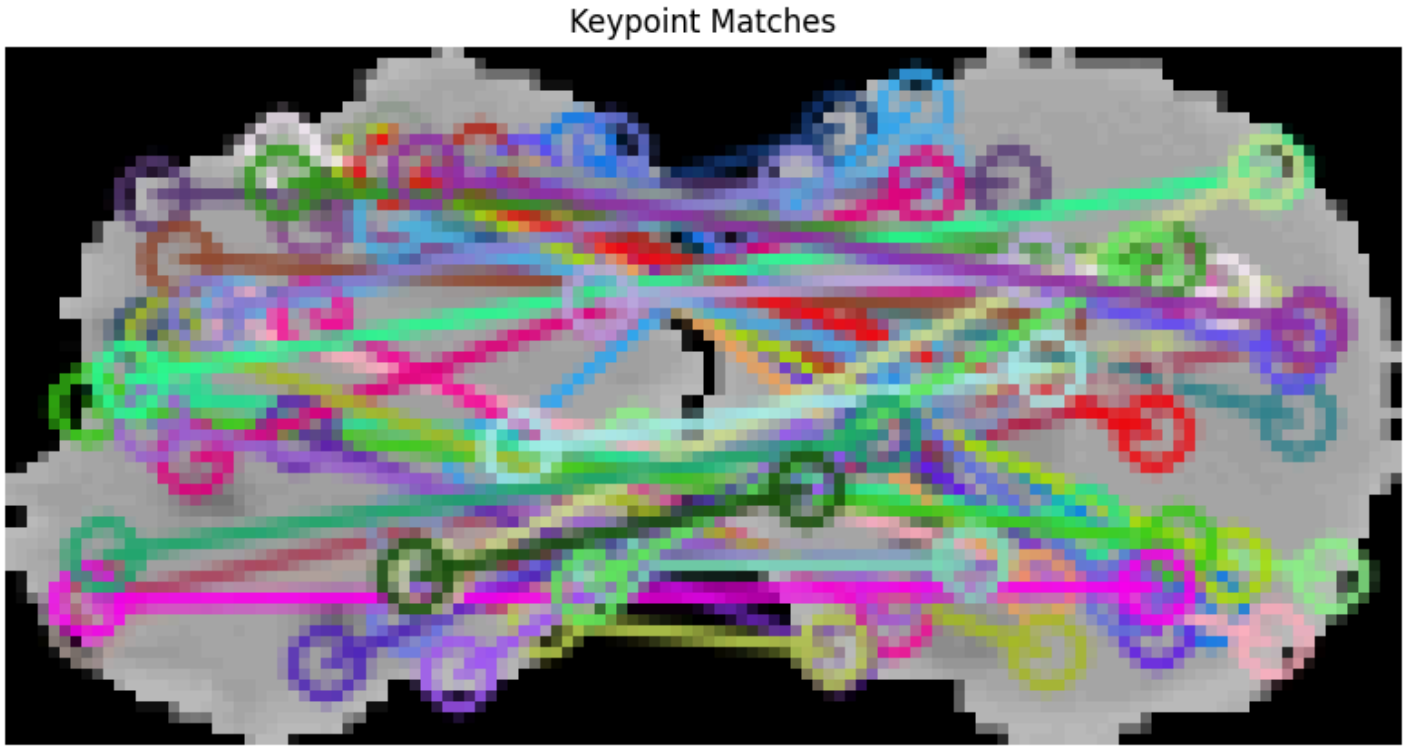
Keypoint Matching Between Parasitized Images

In [35]:
```
# Initializing lists to store results
keypoints_list = []
daisy_features_list = []
images_to_match = images_parasitized[66:69]  # Selects five parasitized images

# Preprocess images, detect keypoints, and extract DAISY features
for img in images_to_match:
    preprocessed_img = preprocess_image(img)
    keypoints = detect_keypoints(preprocessed_img)
    daisy_features = extract_daisy_at_keypoints(preprocessed_img, keypoints)

    keypoints_list.append(keypoints)
    daisy_features_list.append(daisy_features)

# Loops through each pair of images to match features
for i in range(len(daisy_features_list)):
    for j in range(i + 1, len(daisy_features_list)):
        # Ensures both features are of type float32
        features1 = daisy_features_list[i].astype(np.float32)
        features2 = daisy_features_list[j].astype(np.float32)

        # Matches features only if they have the same number of columns
        if features1.shape[1] != features2.shape[1]:
            min_cols = min(features1.shape[1], features2.shape[1])
            features1 = features1[:, :min_cols]
            features2 = features2[:, :min_cols]

        # Matches DAISY features
        matches = match_daisy_features(features1, features2)

        # Checks if there are any matches found
        if len(matches) == 0:
            print(f"No matches found between image {i} and image {j}.")
        else:
            # Visualizing matches
            visualize_matches(images_to_match[i], images_to_match[j], keypoints_list[i], keypoints_list[j],

        # Prints results for each pair
        print(f'Image {i} and Image {j}:')
```

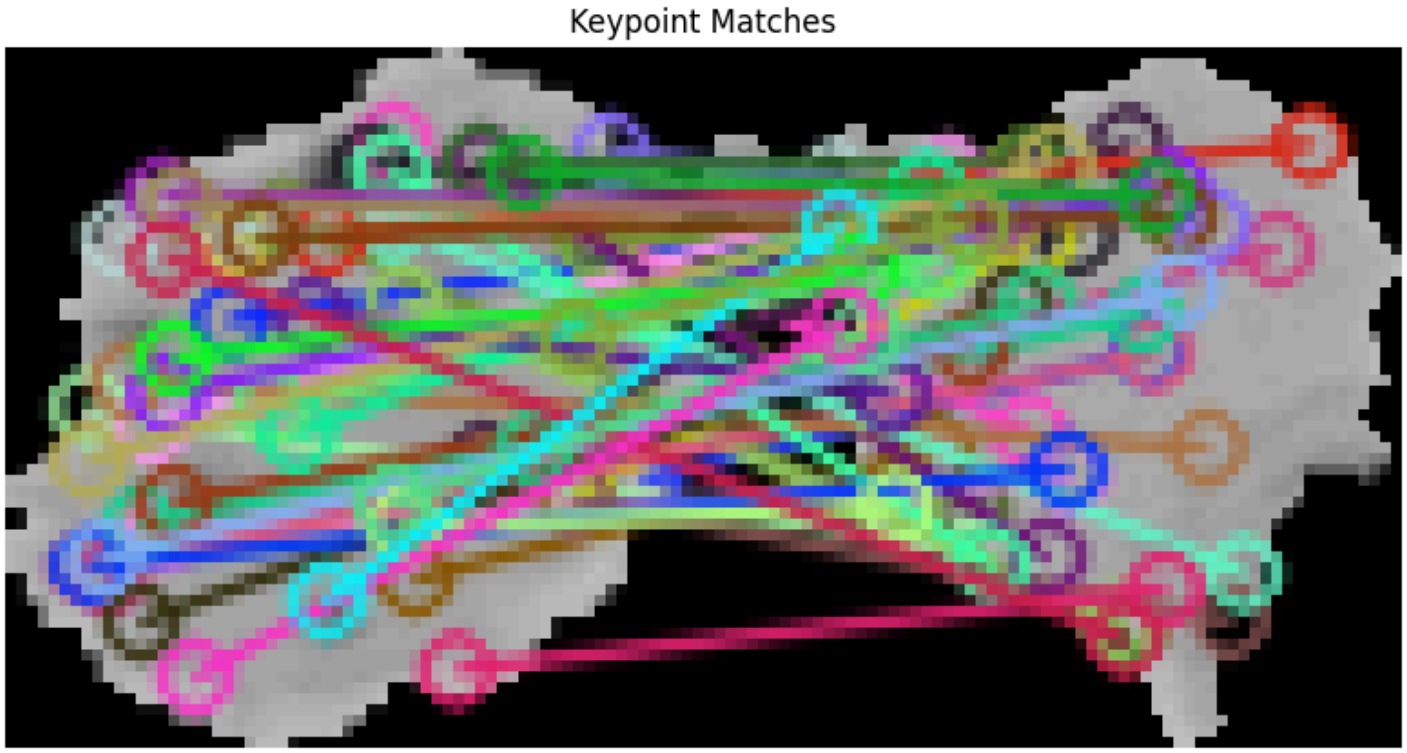
```
        print(f'Number of keypoints in image {i}: {len(keypoints_list[i])}')
        print(f'Number of keypoints in image {j}: {len(keypoints_list[j])}')
        print(f'Number of matches: {len(matches)}\n')
```

```
Extracting DAISY features for 245 keypoints.
Skipping keypoint at (5, 34) - out of bounds for patch extraction.
Skipping keypoint at (24, 58) - out of bounds for patch extraction.
Skipping keypoint at (36, 7) - out of bounds for patch extraction.
Skipping keypoint at (36, 57) - out of bounds for patch extraction.
Skipping keypoint at (38, 6) - out of bounds for patch extraction.
Skipping keypoint at (39, 58) - out of bounds for patch extraction.
Skipping keypoint at (43, 56) - out of bounds for patch extraction.
Skipping keypoint at (47, 7) - out of bounds for patch extraction.
Skipping keypoint at (51, 5) - out of bounds for patch extraction.
Skipping keypoint at (56, 29) - out of bounds for patch extraction.
Skipping keypoint at (57, 11) - out of bounds for patch extraction.
Skipping keypoint at (57, 17) - out of bounds for patch extraction.
Skipping keypoint at (57, 36) - out of bounds for patch extraction.
Skipping keypoint at (57, 39) - out of bounds for patch extraction.
Skipping keypoint at (57, 41) - out of bounds for patch extraction.
Skipping keypoint at (57, 43) - out of bounds for patch extraction.
Skipping keypoint at (58, 32) - out of bounds for patch extraction.
Skipping keypoint at (32, 7) - out of bounds for patch extraction.
Skipping keypoint at (36, 56) - out of bounds for patch extraction.
Skipping keypoint at (37, 7) - out of bounds for patch extraction.
Skipping keypoint at (46, 7) - out of bounds for patch extraction.
Skipping keypoint at (52, 7) - out of bounds for patch extraction.
Skipping keypoint at (56, 10) - out of bounds for patch extraction.
Skipping keypoint at (56, 16) - out of bounds for patch extraction.
Skipping keypoint at (56, 40) - out of bounds for patch extraction.
Skipping keypoint at (56, 43) - out of bounds for patch extraction.
Skipping keypoint at (7, 50) - out of bounds for patch extraction.
Skipping keypoint at (31, 7) - out of bounds for patch extraction.
Skipping keypoint at (34, 7) - out of bounds for patch extraction.
Skipping keypoint at (50, 7) - out of bounds for patch extraction.
Skipping keypoint at (53, 7) - out of bounds for patch extraction.
Number of valid patches extracted: 214
Extracting DAISY features for 256 keypoints.
Skipping keypoint at (5, 15) - out of bounds for patch extraction.
Skipping keypoint at (5, 19) - out of bounds for patch extraction.
Skipping keypoint at (5, 43) - out of bounds for patch extraction.
Skipping keypoint at (6, 45) - out of bounds for patch extraction.
Skipping keypoint at (7, 12) - out of bounds for patch extraction.
Skipping keypoint at (17, 7) - out of bounds for patch extraction.
Skipping keypoint at (18, 58) - out of bounds for patch extraction.
Skipping keypoint at (27, 7) - out of bounds for patch extraction.
Skipping keypoint at (29, 56) - out of bounds for patch extraction.
Skipping keypoint at (45, 58) - out of bounds for patch extraction.
Skipping keypoint at (46, 5) - out of bounds for patch extraction.
Skipping keypoint at (48, 57) - out of bounds for patch extraction.
Skipping keypoint at (56, 12) - out of bounds for patch extraction.
Skipping keypoint at (56, 28) - out of bounds for patch extraction.
Skipping keypoint at (57, 50) - out of bounds for patch extraction.
Skipping keypoint at (58, 43) - out of bounds for patch extraction.
Skipping keypoint at (15, 7) - out of bounds for patch extraction.
Skipping keypoint at (30, 7) - out of bounds for patch extraction.
Skipping keypoint at (48, 7) - out of bounds for patch extraction.
Skipping keypoint at (48, 56) - out of bounds for patch extraction.
Skipping keypoint at (56, 33) - out of bounds for patch extraction.
Skipping keypoint at (7, 47) - out of bounds for patch extraction.
Skipping keypoint at (47, 7) - out of bounds for patch extraction.
Number of valid patches extracted: 233
Extracting DAISY features for 231 keypoints.
Skipping keypoint at (6, 35) - out of bounds for patch extraction.
Skipping keypoint at (6, 52) - out of bounds for patch extraction.
Skipping keypoint at (7, 13) - out of bounds for patch extraction.
Skipping keypoint at (7, 40) - out of bounds for patch extraction.
Skipping keypoint at (9, 56) - out of bounds for patch extraction.
Skipping keypoint at (12, 7) - out of bounds for patch extraction.
Skipping keypoint at (13, 58) - out of bounds for patch extraction.
Skipping keypoint at (14, 5) - out of bounds for patch extraction.
Skipping keypoint at (30, 56) - out of bounds for patch extraction.
```

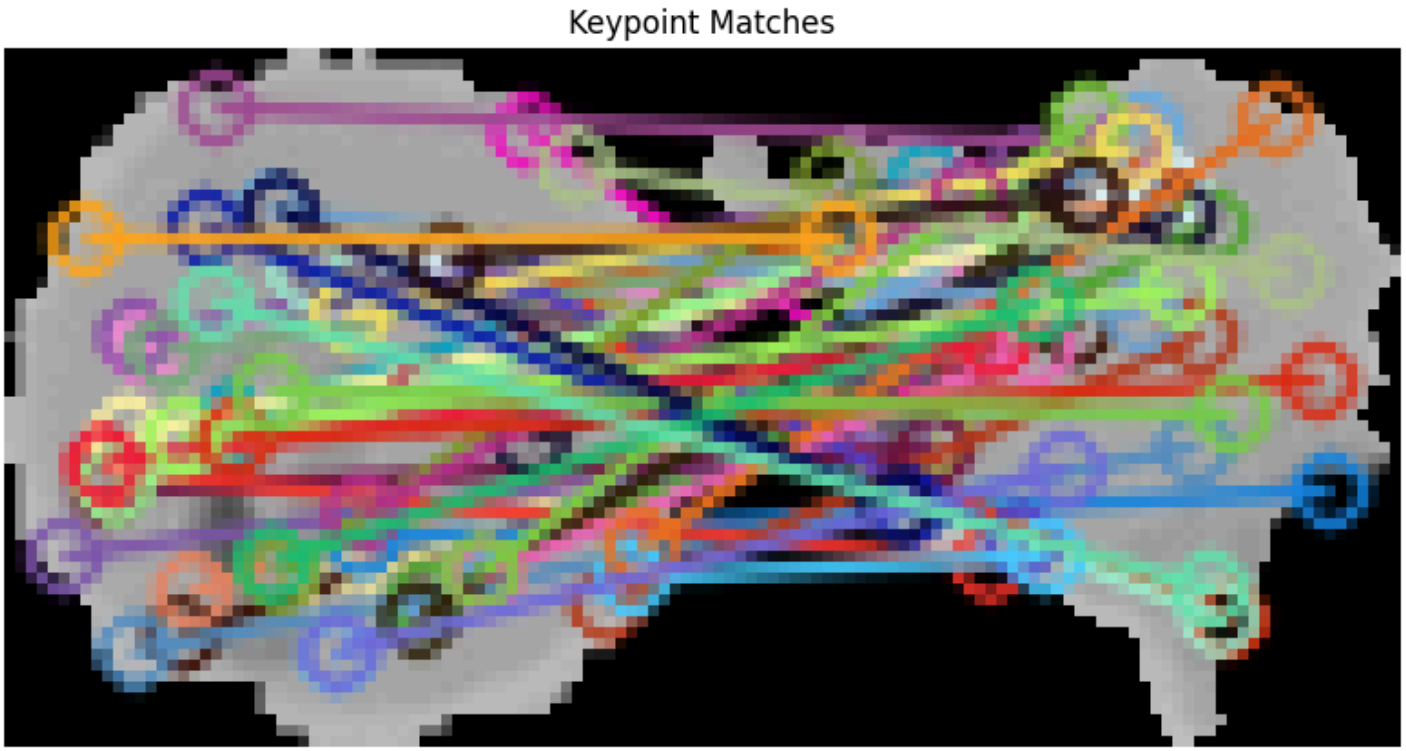
```
Skipping keypoint at (34, 57) - out of bounds for patch extraction.
Skipping keypoint at (40, 57) - out of bounds for patch extraction.
Skipping keypoint at (57, 44) - out of bounds for patch extraction.
Skipping keypoint at (6, 34) - out of bounds for patch extraction.
Skipping keypoint at (7, 13) - out of bounds for patch extraction.
Skipping keypoint at (7, 43) - out of bounds for patch extraction.
Skipping keypoint at (39, 56) - out of bounds for patch extraction.
Skipping keypoint at (7, 12) - out of bounds for patch extraction.
Number of valid patches extracted: 214
```




```
Image 0 and Image 1:
Number of keypoints in image 0: 245
Number of keypoints in image 1: 256
Number of matches: 61
```



```
Image 0 and Image 2:
Number of keypoints in image 0: 245
Number of keypoints in image 2: 231
Number of matches: 55
```




```
Image 1 and Image 2:
Number of keypoints in image 1: 256
Number of keypoints in image 2: 231
Number of matches: 52
```

Analysis for Keypoint Matching between Parasitized Images

Keypoint Matching Between Parasitized Images:

1. **Image 0 and Image 1**:
   - Keypoints in Image 0: 245
   - Keypoints in Image 1: 256
   - Matches: 61
2. **Image 0 and Image 2**:
   - Keypoints in Image 0: 245
   - Keypoints in Image 2: 231
   - Matches: 55
3. **Image 1 and Image 2**:
   - Keypoints in Image 1: 256
   - Keypoints in Image 2: 231
   - Matches: 52

Keypoint Matching Between Parasitized and Uninfected Images:

- **Parasitized Image vs. Uninfected Image**:
  - Keypoints in Parasitized Image: 255
  - Keypoints in Uninfected Image: 144
  - Matches: 43

Comparison:

1. **Keypoint Detection**:
    - **Parasitized Images**: The number of keypoints in parasitized images ranged from 231 to 256, indicating a similar level of feature-rich content across these images.
    - **Parasitized vs. Uninfected**: In contrast, the uninfected image had significantly fewer keypoints (144), suggesting that uninfected images have less feature variation or texture complexity for keypoint detection compared to parasitized images.
2. **Keypoint Matching**:
    - **Parasitized Images**: The number of matches between parasitized images ranged from **52 to 61**, indicating a high degree of similarity and shared features between different parasitized images.
    - **Parasitized vs. Uninfected**: Only **43 matches** were found between the parasitized and uninfected images, which is notably lower than the matches between parasitized images. This indicates a lower degree of shared features between parasitized and uninfected images.
3. **Key Observations**:
    - **Consistency in Parasitized Images**: The higher number of matches between parasitized images suggests that they share more consistent patterns or structures, likely related to parasitic features, making keypoint matching between these images more reliable.
    - **Contrast with Uninfected Images**: The lower number of matches between parasitized and uninfected images reflects the distinct differences in texture or structure between the two types, making them less similar in terms of keypoint features.

Conclusion:

- The higher number of keypoints and matches between parasitized images reflects the presence of complex and distinguishable features, which are crucial for identifying cells with malaria. This suggests that parasitized images have more texture variation and are more consistent in keypoint detection and matching.
- The lower number of keypoints and matches between parasitized and uninfected images indicates distinct feature differences. This reinforces the potential of using keypoint detection and matching techniques to differentiate between infected and uninfected samples, which is promising for further diagnostic development.

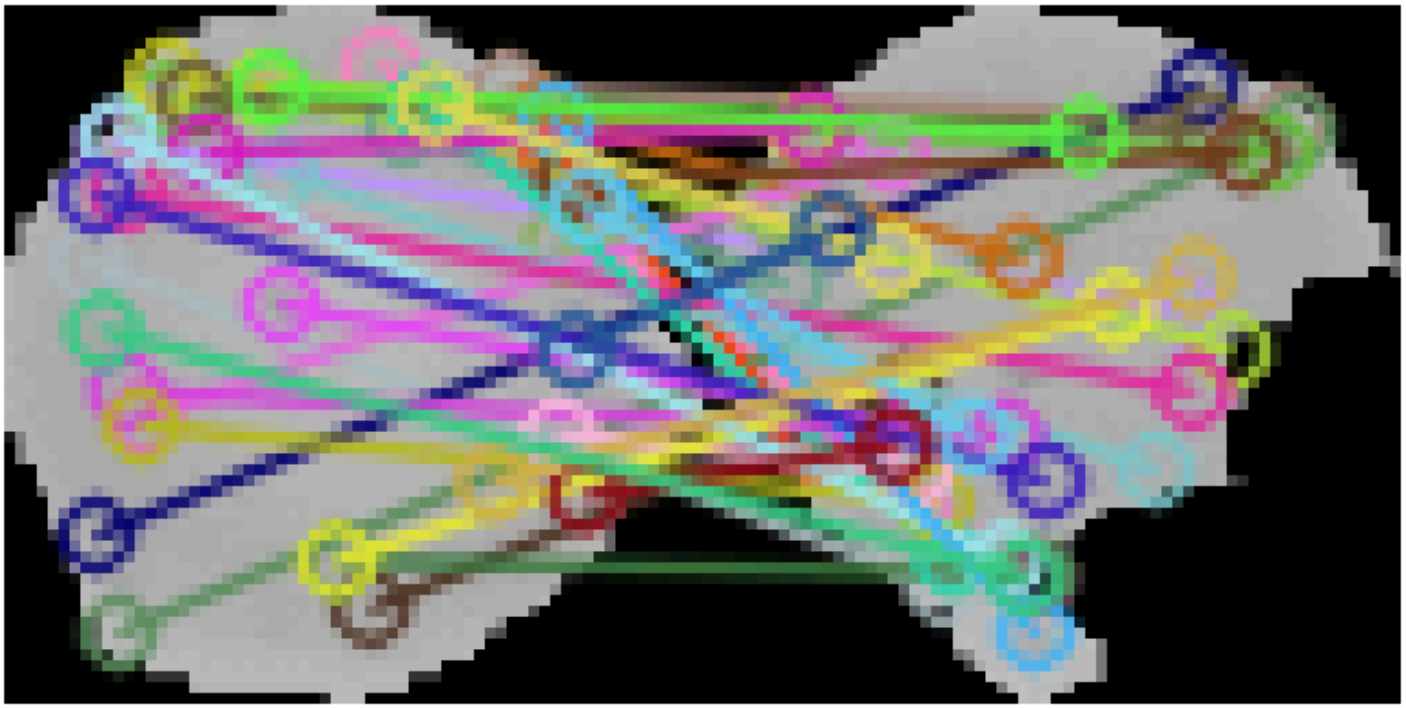
Keypoint Matching Between Uninfected Images

In [36]:
```
# Initializes lists to store results
keypoints_list = []
daisy_features_list = []
images_to_match = images_uninfected[20:23]  # Select five uninfected images

# Preprocess images, detect keypoints, and extract DAISY features
for img in images_to_match:
    preprocessed_img = preprocess_image(img)
    keypoints = detect_keypoints(preprocessed_img)
    daisy_features = extract_daisy_at_keypoints(preprocessed_img, keypoints)

    keypoints_list.append(keypoints)
    daisy_features_list.append(daisy_features)

# Loops through each pair of images to match features
for i in range(len(daisy_features_list)):
    for j in range(i + 1, len(daisy_features_list)):
        # Ensures both features are of type float32
        features1 = daisy_features_list[i].astype(np.float32)
        features2 = daisy_features_list[j].astype(np.float32)

        # Matches features only if they have the same number of columns
        if features1.shape[1] != features2.shape[1]:
            min_cols = min(features1.shape[1], features2.shape[1])
```

```
            features1 = features1[:, :min_cols]
            features2 = features2[:, :min_cols]

        # Matches DAISY features
        matches = match_daisy_features(features1, features2)

        # Checks if there are any matches found
        if len(matches) == 0:
            print(f"No matches found between image {i} and image {j}.")
        else:
            # Visualizing matches
            visualize_matches(images_to_match[i], images_to_match[j], keypoints_list[i], keypoints_list[j],

        # Prints results for each pair
        print(f'Image {i} and Image {j}:')
        print(f'Number of keypoints in image {i}: {len(keypoints_list[i])}')
        print(f'Number of keypoints in image {j}: {len(keypoints_list[j])}')
        print(f'Number of matches: {len(matches)}\n')
```

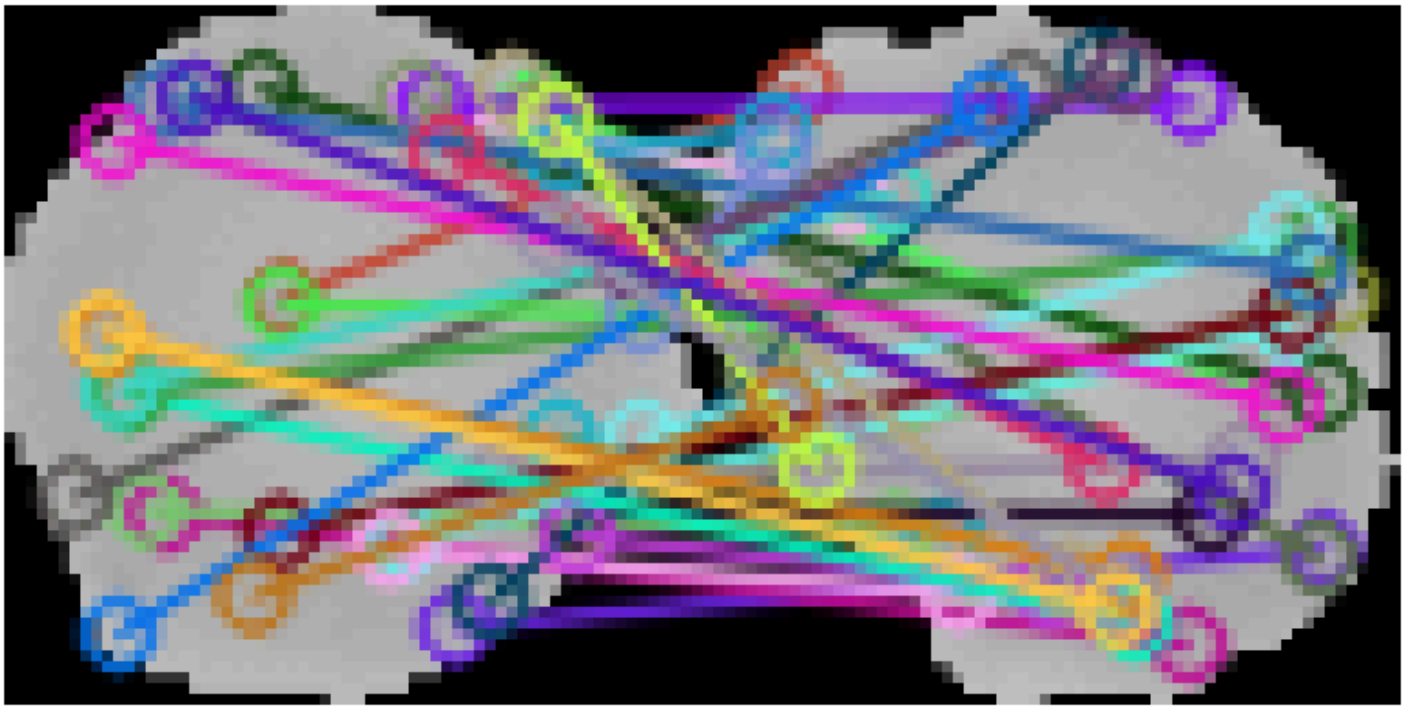
```
Extracting DAISY features for 153 keypoints.
Skipping keypoint at (5, 34) - out of bounds for patch extraction.
Skipping keypoint at (6, 43) - out of bounds for patch extraction.
Skipping keypoint at (7, 46) - out of bounds for patch extraction.
Skipping keypoint at (23, 57) - out of bounds for patch extraction.
Skipping keypoint at (28, 58) - out of bounds for patch extraction.
Skipping keypoint at (39, 58) - out of bounds for patch extraction.
Skipping keypoint at (45, 6) - out of bounds for patch extraction.
Skipping keypoint at (57, 10) - out of bounds for patch extraction.
Skipping keypoint at (57, 41) - out of bounds for patch extraction.
Skipping keypoint at (6, 14) - out of bounds for patch extraction.
Skipping keypoint at (7, 24) - out of bounds for patch extraction.
Skipping keypoint at (7, 45) - out of bounds for patch extraction.
Skipping keypoint at (18, 6) - out of bounds for patch extraction.
Skipping keypoint at (22, 56) - out of bounds for patch extraction.
Skipping keypoint at (44, 6) - out of bounds for patch extraction.
Skipping keypoint at (56, 10) - out of bounds for patch extraction.
Skipping keypoint at (56, 40) - out of bounds for patch extraction.
Skipping keypoint at (7, 23) - out of bounds for patch extraction.
Skipping keypoint at (7, 34) - out of bounds for patch extraction.
Skipping keypoint at (7, 41) - out of bounds for patch extraction.
Skipping keypoint at (7, 46) - out of bounds for patch extraction.
Skipping keypoint at (23, 7) - out of bounds for patch extraction.
Number of valid patches extracted: 131
Extracting DAISY features for 190 keypoints.
Skipping keypoint at (5, 18) - out of bounds for patch extraction.
Skipping keypoint at (7, 16) - out of bounds for patch extraction.
Skipping keypoint at (7, 45) - out of bounds for patch extraction.
Skipping keypoint at (57, 24) - out of bounds for patch extraction.
Skipping keypoint at (57, 30) - out of bounds for patch extraction.
Skipping keypoint at (6, 27) - out of bounds for patch extraction.
Skipping keypoint at (7, 45) - out of bounds for patch extraction.
Skipping keypoint at (18, 6) - out of bounds for patch extraction.
Skipping keypoint at (25, 7) - out of bounds for patch extraction.
Skipping keypoint at (36, 6) - out of bounds for patch extraction.
Skipping keypoint at (7, 44) - out of bounds for patch extraction.
Number of valid patches extracted: 179
Extracting DAISY features for 150 keypoints.
Skipping keypoint at (5, 36) - out of bounds for patch extraction.
Skipping keypoint at (6, 20) - out of bounds for patch extraction.
Skipping keypoint at (6, 26) - out of bounds for patch extraction.
Skipping keypoint at (6, 39) - out of bounds for patch extraction.
Skipping keypoint at (7, 42) - out of bounds for patch extraction.
Skipping keypoint at (11, 6) - out of bounds for patch extraction.
Skipping keypoint at (13, 6) - out of bounds for patch extraction.
Skipping keypoint at (21, 56) - out of bounds for patch extraction.
Skipping keypoint at (25, 7) - out of bounds for patch extraction.
Skipping keypoint at (26, 58) - out of bounds for patch extraction.
Skipping keypoint at (28, 7) - out of bounds for patch extraction.
Skipping keypoint at (33, 5) - out of bounds for patch extraction.
Skipping keypoint at (36, 7) - out of bounds for patch extraction.
Skipping keypoint at (39, 56) - out of bounds for patch extraction.
Skipping keypoint at (42, 57) - out of bounds for patch extraction.
Skipping keypoint at (50, 56) - out of bounds for patch extraction.
Skipping keypoint at (56, 35) - out of bounds for patch extraction.
Skipping keypoint at (56, 39) - out of bounds for patch extraction.
Skipping keypoint at (56, 51) - out of bounds for patch extraction.
Skipping keypoint at (58, 23) - out of bounds for patch extraction.
Skipping keypoint at (58, 44) - out of bounds for patch extraction.
Skipping keypoint at (6, 38) - out of bounds for patch extraction.
Skipping keypoint at (7, 8) - out of bounds for patch extraction.
Skipping keypoint at (7, 28) - out of bounds for patch extraction.
Skipping keypoint at (10, 6) - out of bounds for patch extraction.
Skipping keypoint at (13, 6) - out of bounds for patch extraction.
Skipping keypoint at (34, 56) - out of bounds for patch extraction.
Skipping keypoint at (49, 56) - out of bounds for patch extraction.
Number of valid patches extracted: 122
```

Keypoint Matches

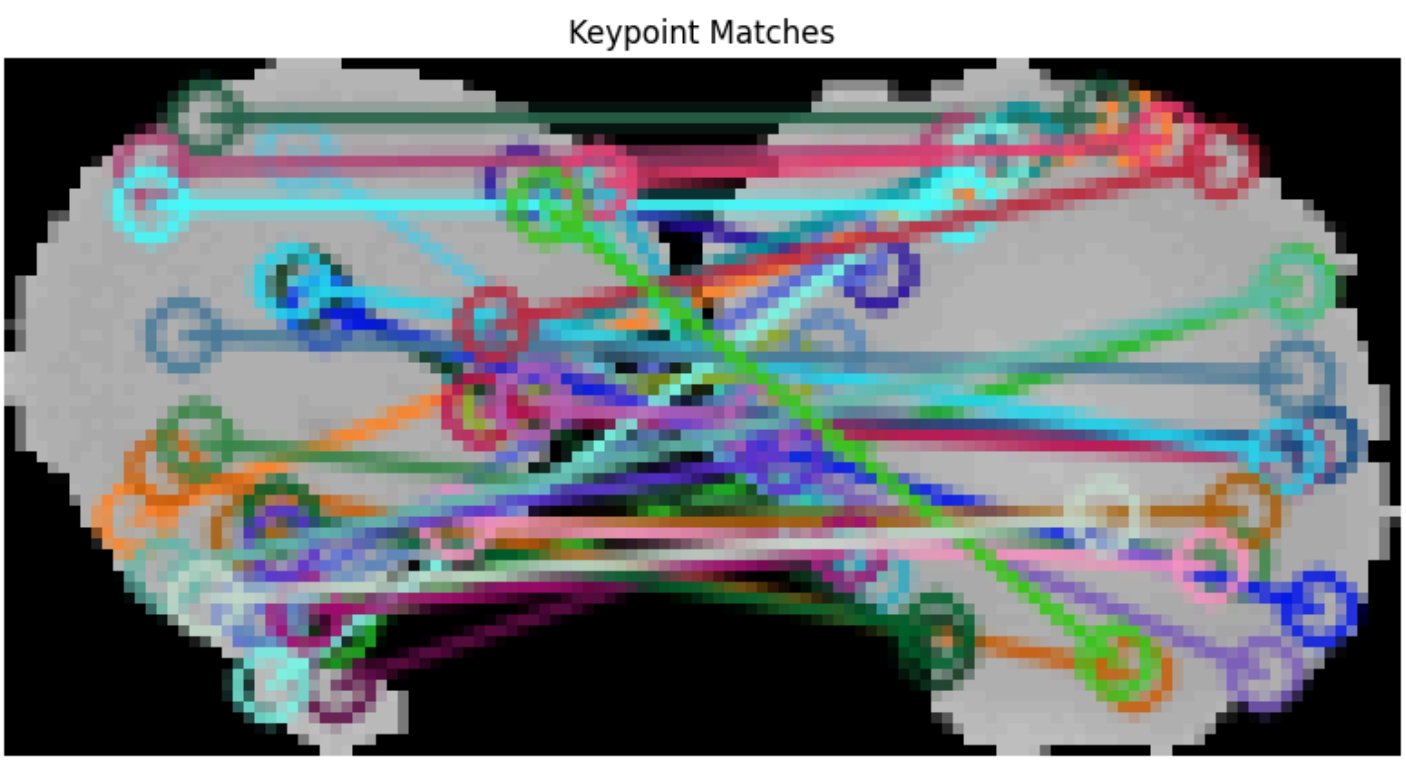


```
Image 0 and Image 1:
Number of keypoints in image 0: 153
Number of keypoints in image 1: 190
Number of matches: 37
```

Keypoint Matches



```
Image 0 and Image 2:
Number of keypoints in image 0: 153
Number of keypoints in image 2: 150
Number of matches: 39
```




```
Image 1 and Image 2:
Number of keypoints in image 1: 190
Number of keypoints in image 2: 150
Number of matches: 35
```

Analysis for Keypoint Matching between uninfected Images

Keypoint Matching Between Uninfected Images:

1. **Image 0 and Image 1**:
   - Keypoints in Image 0: 153
   - Keypoints in Image 1: 190
   - Matches: 37
2. **Image 0 and Image 2**:
   - Keypoints in Image 0: 153
   - Keypoints in Image 2: 150
   - Matches: 39
3. **Image 1 and Image 2**:
   - Keypoints in Image 1: 190
   - Keypoints in Image 2: 150
   - Matches: 35

Keypoint Matching Between Parasitized and Uninfected Images:

- **Parasitized Image vs. Uninfected Image**:
  - Keypoints in Parasitized Image: 255
  - Keypoints in Uninfected Image: 144
  - Matches: 43

Comparison:

1. **Keypoint Detection**:

- **Uninfected Images**: The number of keypoints detected in the uninfected images ranged from **150 to 190**, indicating less complex texture and feature variation compared to the parasitized images (which had keypoint counts of 231-256).
- **Parasitized vs. Uninfected**: Parasitized images have significantly more keypoints (up to 255), highlighting their greater feature richness compared to the uninfected images.

2. **Keypoint Matching**:

- **Uninfected Images**: The number of matches between uninfected images ranged from **35 to 39**, which is slightly lower than the matches found between parasitized images (52-61). This suggests that uninfected images, while somewhat similar to each other, have fewer shared distinctive features than parasitized images.
- **Parasitized vs. Uninfected**: The match count between parasitized and uninfected images (43 matches) is comparable to the matches found between uninfected images, indicating some shared characteristics between these image categories. However, the parasitized images still present more distinctive and feature-rich characteristics in comparison.

3. **Key Observations**:

- **Fewer Keypoints and Matches in Uninfected Images**: The lower keypoint counts and match numbers in uninfected images reinforce that these images have fewer distinct features compared to parasitized images. This is consistent across both within-category and between-category comparisons.
- **Feature Similarity in Uninfected Images**: The moderate number of matches between uninfected images (35-39) suggests some level of similarity, but less than that seen between parasitized images.

Conclusion:

- **Uninfected Images** exhibit fewer detectable keypoints and lower match counts, which may be attributed to their simpler structure and fewer distinct textures.
- **Parasitized vs. Uninfected**: The number of matches between parasitized and uninfected images is close to that between uninfected images, but the parasitized images consistently show more keypoints and higher match counts when compared within their category, reflecting their more complex and distinct features.

Keypoint Matching vs Pairwise Distances

Computational Cost:

- **Pairwise Distance**: This method is simpler and generally faster because it computes distances based on the entire DAISY descriptor extracted for the image. However, this approach can overlook important local variations since it treats the image as a whole.
- **Keypoint Matching**: Keypoint-based matching is computationally more expensive as it extracts DAISY features at multiple localized keypoints and compares them individually. Despite the higher computational cost, this method captures more detailed information, which is critical for our task of distinguishing between parasitized and uninfected cells.

Feature Specificity:

- **Pairwise Distance**: Since it compares overall image descriptors, this method might not be sensitive to localized features or small regions of interest, potentially leading to misclassifications. For instance, in malaria detection, the entire image is condensed into a single feature vector, which might obscure critical differences between infected and uninfected cells. We noted many false negatives and positives with this approach, resulting in an accuracy far below our expectations for the algorithm we aim to achieve.

- **Keypoint Matching**: By focusing on extracting and matching features at specific localized keypoints, keypoint matching is more robust in identifying small but crucial features. In the context of classifying malaria-infected cells, the infection may manifest in distinct regions, such as altered structures or shapes within the cells. Keypoint

matching can effectively capture these subtle changes, making it more adept at distinguishing between parasitized and uninfected cells, even when such differences are not apparent in the overall image. As shown in the graphics, keypoint matching successfully identified these subtle structures and shapes.

Accuracy:

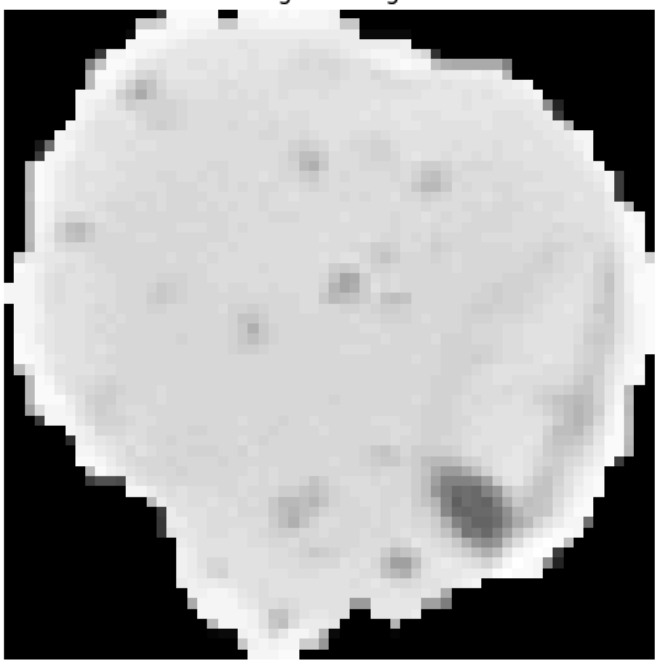
- **Pairwise Distance**: While this method may perform well with globally similar images, it struggles to correctly classify images where small, localized differences are essential. Subtle alterations caused by infection may not be reflected in the overall descriptor, leading to misclassifications.

- **Keypoint Matching**: Keypoint matching outperforms pairwise distance for this specific task because it emphasizes fine, localized similarities and differences. In malaria-infected cells, these critical differences—such as alterations in cell structure or texture—are better captured by focusing on keypoints rather than the entire image. Thus, keypoint matching is more robust and accurate for distinguishing between parasitized and uninfected cells, which is crucial for reliable classification in this context.

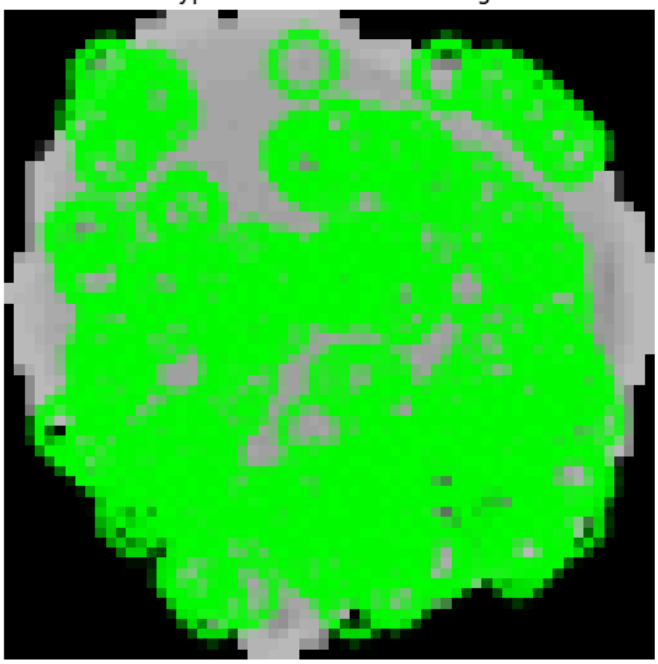
Overall:

- Keypoint matching clearly performed better than pairwise distance. It effectively extracts and matches features at specific localized keypoints, ensuring robust identification of smaller features critical for classification. Malaria cells are often characterized by minute details, so accurate identification is essential. Although keypoint matching is computationally more expensive, the trade-off is justified by the improved accuracy and robustness in feature identification, making it a superior choice for this classification task.

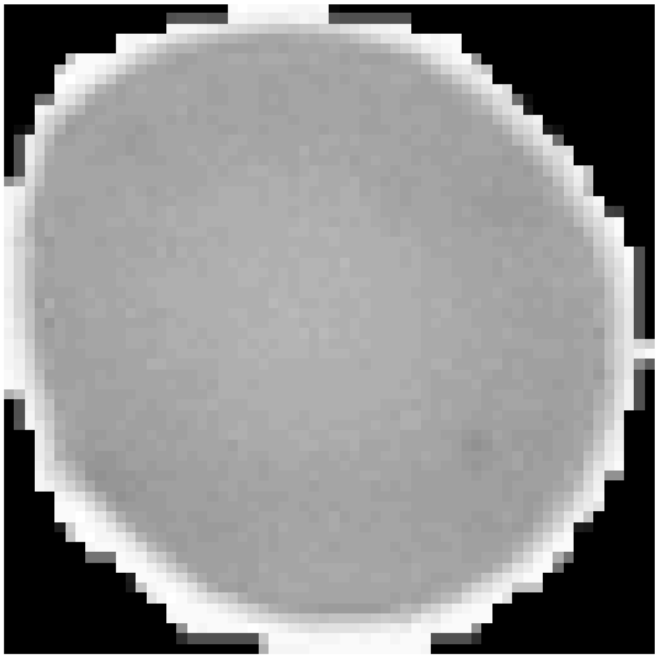
Original Image



Keypoints in Parasitized Image



Original Image



Keypoints in Uninfected Image