In [44]:

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import svd
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from tqdm import tqdm
FIGSIZE = (12, 8)
```

Problem 1

In this problem we will look at image compression using SVD, following the lines of the well-known "Eigenfaces" experiment. The basic concept is to represent an image (in grayscale) of size $m \times n$ as an $m \times n$ real matrix M. SVD is then applied to this matrix to obtain U, S, and V such that $M = U S V^T$. Here U and V are the matrices whose columns are the left and right singular vectors respectively, and S is a diagonal $m \times n$ matrix consisting of the singular values of M. The number of non-zero singular values is the rank of M. By using just the largest k singular values (and corresponding left and right singular vectors), one obtains the best rank-k approximation to M.

The following code returns the dataset of 400 images.

In [39]:

```
data = datasets.fetch_olivetti_faces();
images = data.images
```

(a) Given an $m \times n$ image M and its rank-k approximation A, we can measure the reconstruction error using mean ℓ_1 error: $$ \text{error}_{\ell_1}(M, A) = \frac{1}{mn}\| M - A \|_1 = \frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}| M_{i,j} - A_{i,j} |. $$ For $k = 1, \dots, 30$, take the average rank-k reconstruction error over all images in the dataset, and plot a curve of average reconstruction error as a function of k.

In [40]:

```
# Returns the best rank-k approximation to M
def svd_reconstruct(M, k):
    # TODO: Complete this!
    # Advice: pass in full_matrices=False to svd to avoid dimensionality issues
    u, s, v = svd(M, full_matrices=False)
    M_rec = u[:, :k] @ np.diag(s[:k]) @ v[:k, :]
    return M_rec

def reconstruction_error(M, M_rec):
    return np.mean(np.abs(M - M_rec))
```

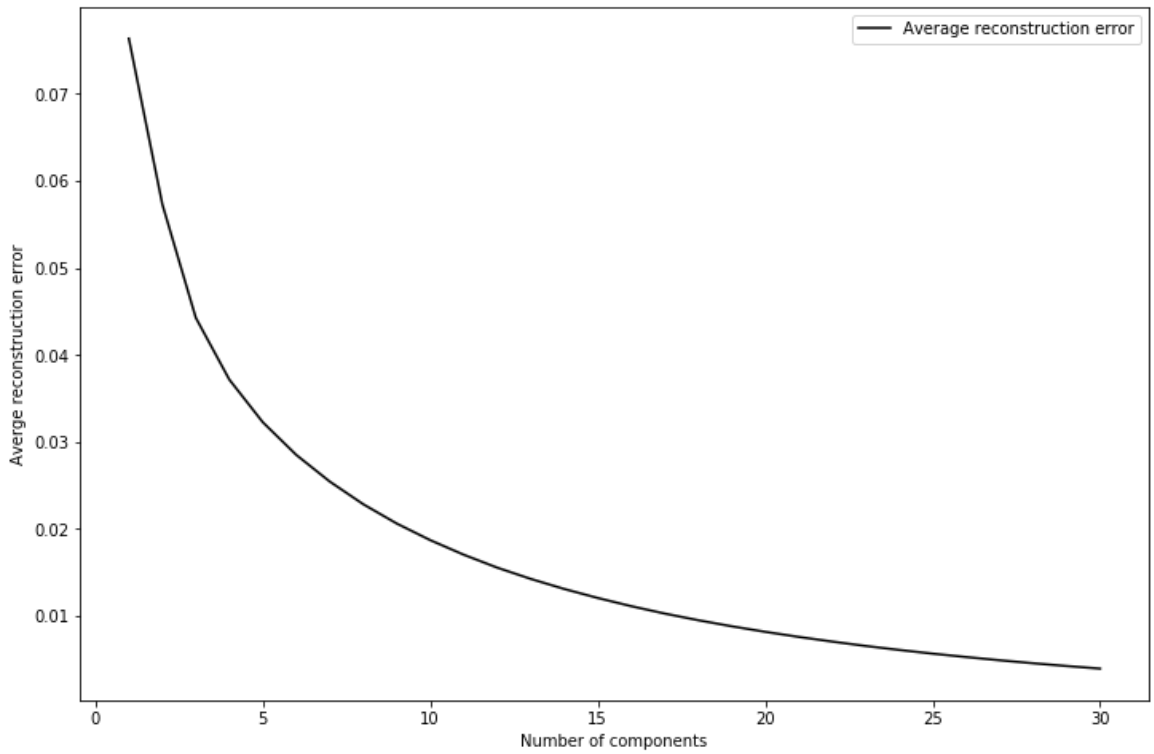
In [46]:

```
ks = list(range(1, 31))
errors = []
for k in tqdm(ks):
    error_k = 0
    for img in images:
        M_rec = svd_reconstruct(img, k)
        error_k += reconstruction_error(img, M_rec)
    errors.append(error_k / len(images))
```

```
100%|██████████| 30/30 [00:09<00:00,  3.32it/s]
```

In [47]:

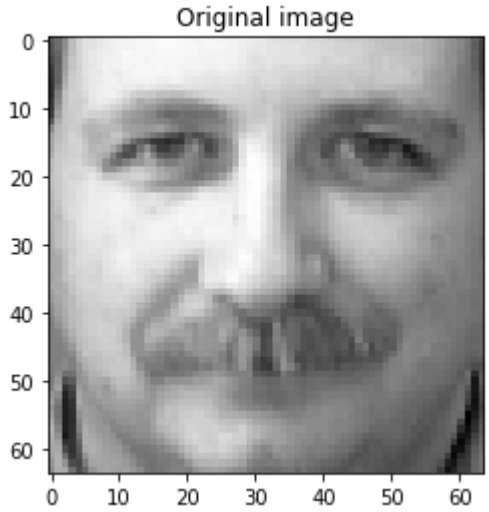
```
plt.figure(figsize=FIGSIZE)
plt.plot(ks, errors, c='k', label="Average reconstruction error")
plt.xlabel("Number of components")
plt.ylabel("Averge reconstruction error")
plt.legend()
plt.show();
```

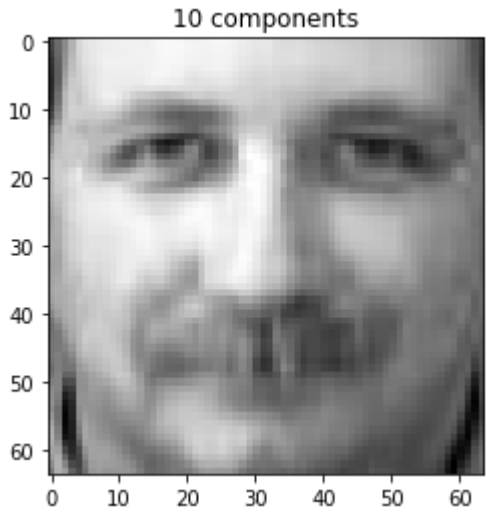


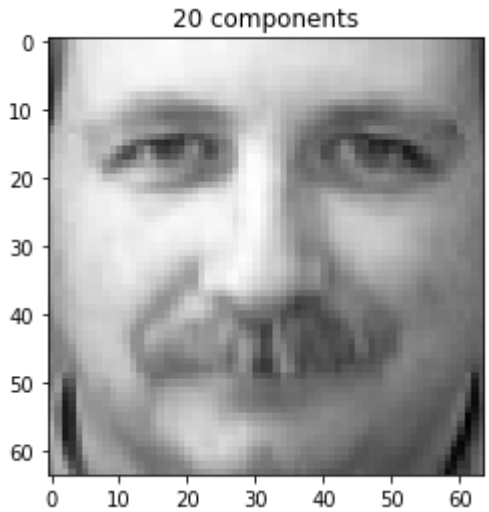
(b) Pick any image in the dataset, and display the following side-by-side as images: the original, and the best rank-k approximations for $k = 10, 20, 30, 40$. You will find the `imshow` method in matplotlib useful for this; pass in `cmap='gray'` to render in grayscale. Feel free to play around further.

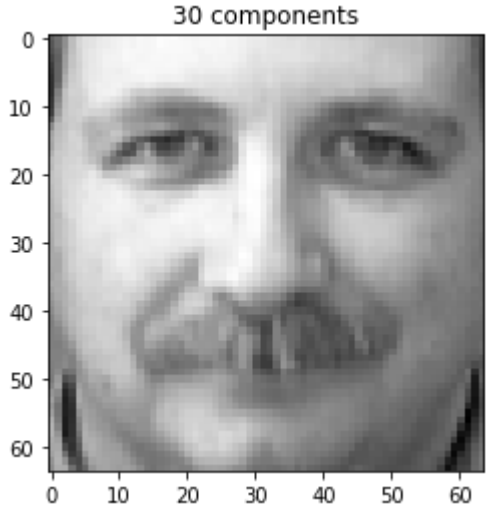
In [49]:

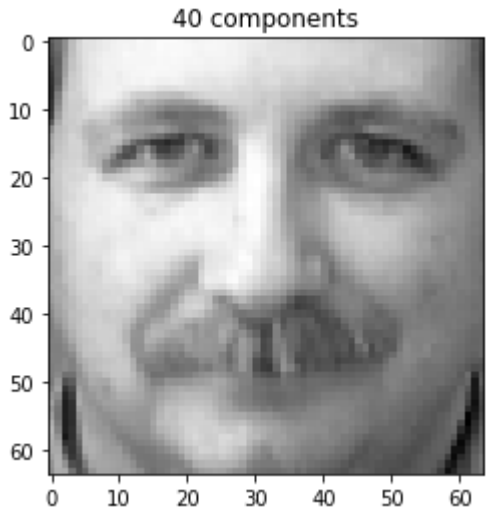
```
image = images[np.random.randint(len(images))]
ks = [10, 20, 30, 40]
images_reconstructed = [svd_reconstruct(image, k) for k in ks]

plt.figure()
plt.title("Original image")
plt.imshow(image, cmap='gray')
for i, img in enumerate(images_reconstructed):
    plt.figure()
    plt.title(f"{ks[i]} components")
    plt.imshow(img, cmap='gray')
```


Original image
0
10
20
30
40
50
60


10 components
0
10
20
30
40
50
60


20 components
0
10
20
30
40
50
60





Problem 2

In this problem we visualize the Wisconsin breast cancer dataset in two dimensions using PCA. First, rescale the data so that every feature has mean 0 and standard deviation 1 across the various points in the dataset. You may find `sklearn.preprocessing.StandardScaler` useful for this. Next, compute the top two principal components of the dataset using PCA, and for every data point, compute its coordinates (i.e. projections) along these two principal components. You should do this in two ways:

1. By using SVD directly. Do not use any PCA built-ins.
2. By using `sklearn.decomposition.PCA`.

The two approaches should give exactly the same result, and this also acts as a check. (But note that the signs of the singular vectors may be flipped in the two approaches since singular vectors are only determined uniquely up to sign. If this happens, flip signs to make everything identical again.)

Your final goal is to make a scatterplot of the dataset in 2 dimensions, where the x-axis is the first principal component and the y-axis is the second. Color the points by their diagnosis (malignant or benign). Do this for both approaches. Your plots should be identical. Does the data look roughly separable already in 2 dimensions?

In [51]:

```
cancer = datasets.load_breast_cancer()
scaler = StandardScaler()
cancer.data = scaler.fit_transform(cancer.data)
```

In [52]:

```
n_components = 2
u, s, v = svd(cancer.data, full_matrices=False)
cancer_reconstructed_svd = []
for el in cancer.data:
    x = -np.sum(el * v[0])
    y = -np.sum(el * v[1])
    cancer_reconstructed_svd.append([x, y])
cancer_reconstructed_svd = np.array(cancer_reconstructed_svd)
```

In [53]:

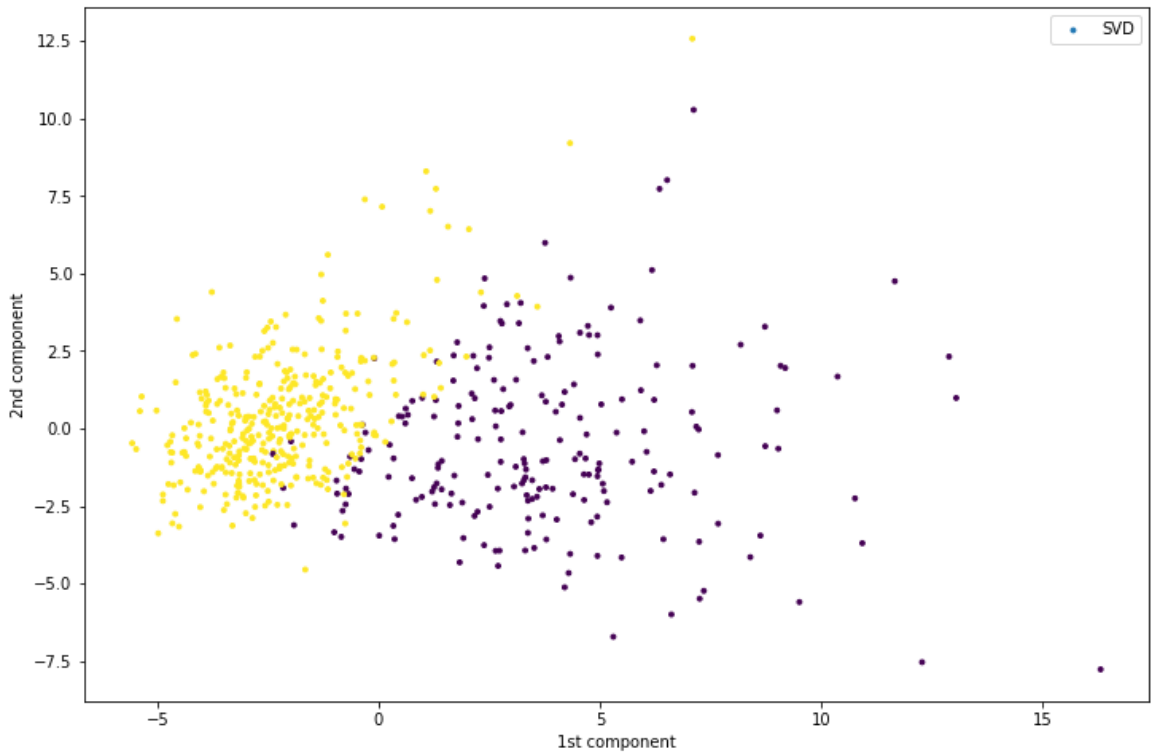
```
pca = PCA(n_components=n_components)
cancer_reconstructed_pca = pca.fit_transform(cancer.data)
```

In [54]:

```
# sanity check
assert np.allclose(cancer_reconstructed_pca, cancer_reconstructed_svd)
```

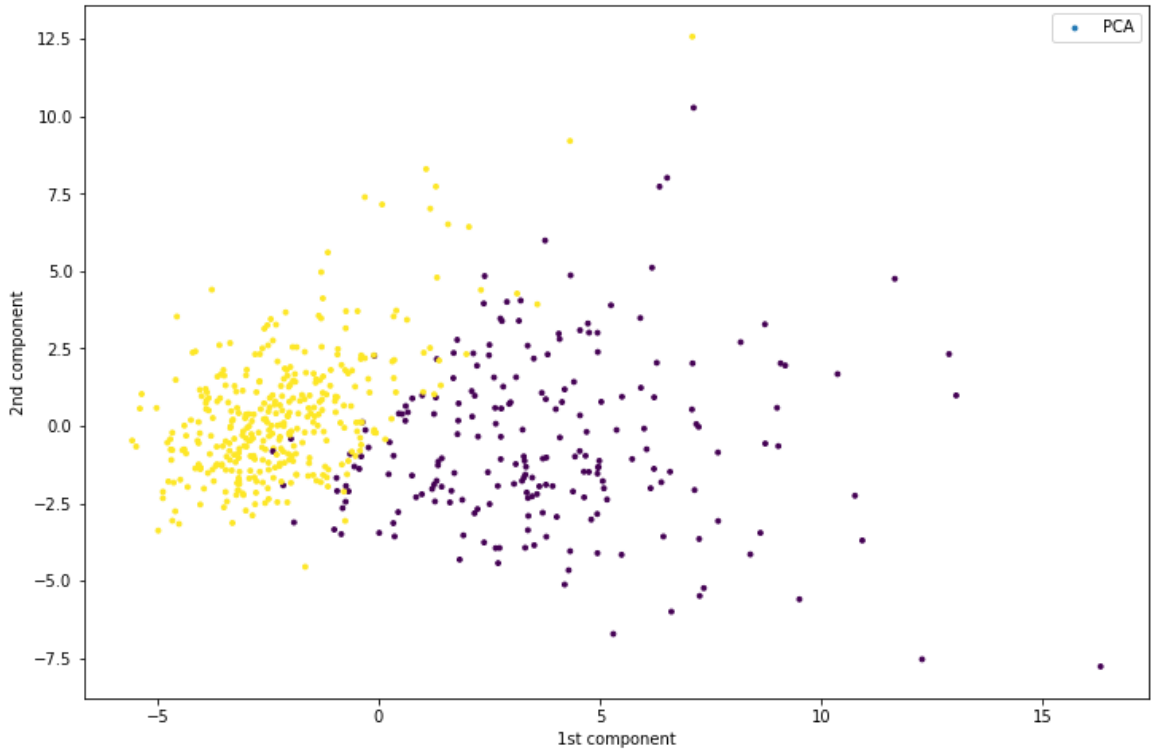
In [55]:

```
plt.figure(figsize=FIGSIZE)
plt.scatter(cancer_reconstructed_svd[:, 0], cancer_reconstructed_svd[:, 1],
            c=cancer.target, s=8, label="SVD")
plt.legend()
plt.xlabel("1st component")
plt.ylabel("2nd component")
plt.show()
```



In [56]:

```
plt.figure(figsize=FIGSIZE)
plt.scatter(cancer_reconstructed_pca[:, 0], cancer_reconstructed_pca[:, 1],
            c=cancer.target, s=8, label="PCA")
plt.legend()
plt.xlabel("1st component")
plt.ylabel("2nd component")
plt.show()
```



A: Yes, data looks quite separable even using two dimensions. Although some misclassifications still possible.