This is exactly the same as the normal equation we have for projections.

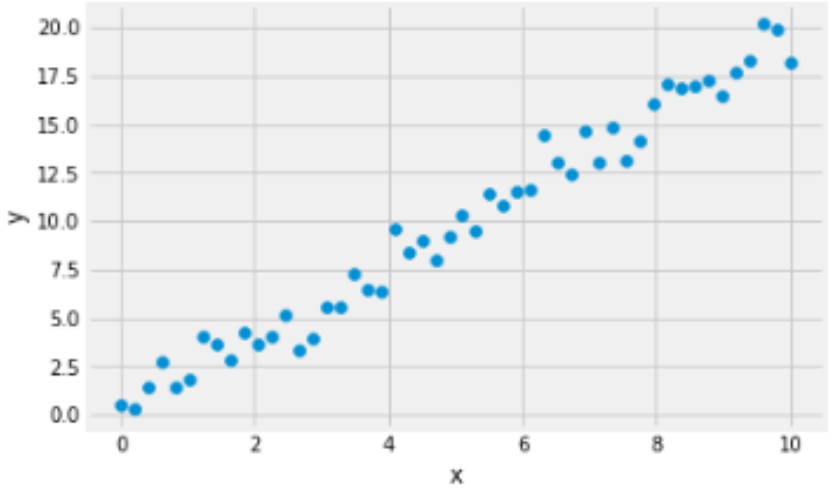
This means that if we solve for $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$, we would find the best $\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$, i.e. the $\boldsymbol{\theta}$ which minimizes our objective.

Let's put things into perspective. Consider that we want to predict the true coefficient $\boldsymbol{\theta}$ of the line $\boldsymbol{y} = \boldsymbol{\theta}^T\boldsymbol{x}$ given only $\boldsymbol{X}$ and $\boldsymbol{y}$. We do not know the true value of $\boldsymbol{\theta}$.

Below, in a two dimensional plane, we shall generate 50 points on a line passing through the origin and with $\boldsymbol{\theta}$ (which is slope in this case) = 2. Then, we shall add some noise to all the points so that all the points do not end up being on the same line (if all the points are on the same line, it would make finding $\boldsymbol{\theta}$ extremely easy).

Note: In this particular example, $\boldsymbol{\theta}$ is a scalar. Still, we can represent it as an $\mathbb{R}^1$ vector.

In [27]:
```
x = np.linspace(0, 10, num=50)
theta = 2
def f(x):
    random = np.random.RandomState(42) # we use the same random seed so we get deterministic output
    return theta * x + random.normal(scale=1.0, size=len(x)) # our observations are corrupted by some noise, so that we do not g

y = f(x)
plt.scatter(x, y);
plt.xlabel('x');
plt.ylabel('y');
```



Now, we shall calculate $\hat{\theta}$ using the formula which we derived above.

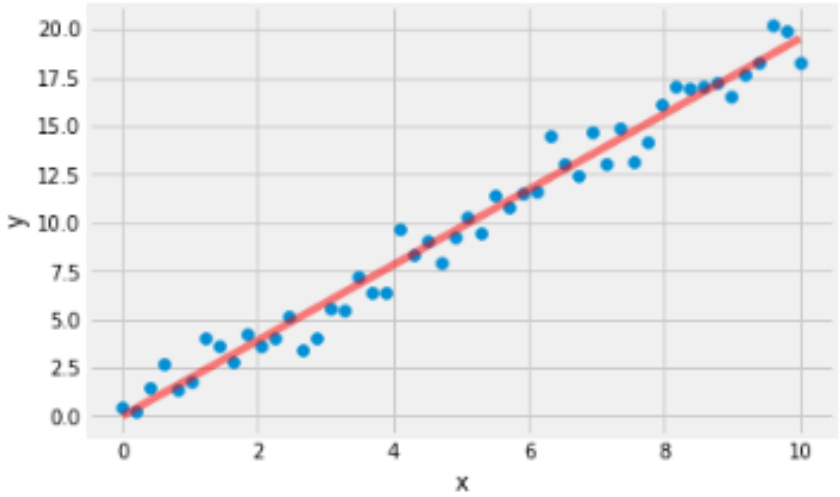
In [28]:
```
X = x.reshape(-1,1) # size N x 1
Y = y.reshape(-1,1) # size N x 1

# maximum likelihood estimator
theta_hat = np.linalg.inv(X.T @ X) @ X.T @ Y
print('Inferred slope =', theta_hat[0, 0])
```

```
Inferred slope = 1.951585423289831
```

We can show how our $\hat{\theta}$ fits the line.

In [29]:
```
fig, ax = plt.subplots()
ax.scatter(x, y);
xx = [0, 10]
yy = [0, 10 * theta_hat[0,0]]
ax.plot(xx, yy, 'red', alpha=.5);
ax.set(xlabel='x', ylabel='y');
print("theta = %f" % theta)
print("theta_hat = %f" % theta_hat)
```

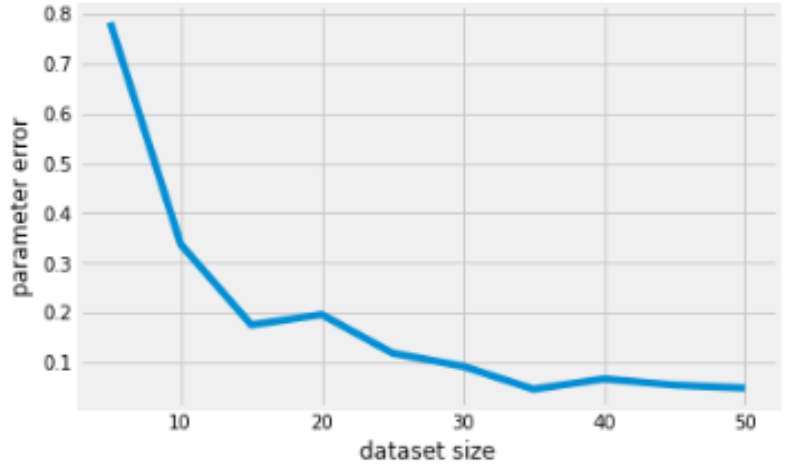
```
theta = 2.000000
theta_hat = 1.951585
```



Suppose that we calculate $\hat{\theta}$ multiple times, each time taking increasing number of datapoints into consideration. How would you expect $\|\hat{\theta} - \theta\|$ to vary as the number of datapoints increases?

Make your hypothesis, and complete the code below to confirm it!

In [30]:
```
theta_error = []
size = []

for i in range(5, 51, 5):
    # Take the first i points from X and Y
    X_i = X[:i]
    Y_i = Y[:i]

    # Calculate theta_hat for X_i and Y_i
    # Feel free to look at how we did it above in case you are stuck
    theta_hat = np.linalg.inv(X_i.T@ X_i) @ X_i.T @ Y_i

    # Removing any excess dimensions from theta_hat
    theta_hat = np.squeeze(theta_hat)

    # Append the error to the end of theta_error
    # We have already done this for you
    theta_error.append(abs(theta - theta_hat))
    size.append(i)

plt.plot(size, theta_error)
plt.xlabel("dataset size")
plt.ylabel("parameter error");
```



As you can see, $\|\hat{\theta} - \theta\|$ generally decreases with an incrase in the dataset size.

In [15]:
```
# for numerical reasons which you shall see in week 4, we normalize the dataset
mean = faces.mean(axis=0)
std = faces.std(axis=0)
faces_normalized = (faces - mean) / std
```

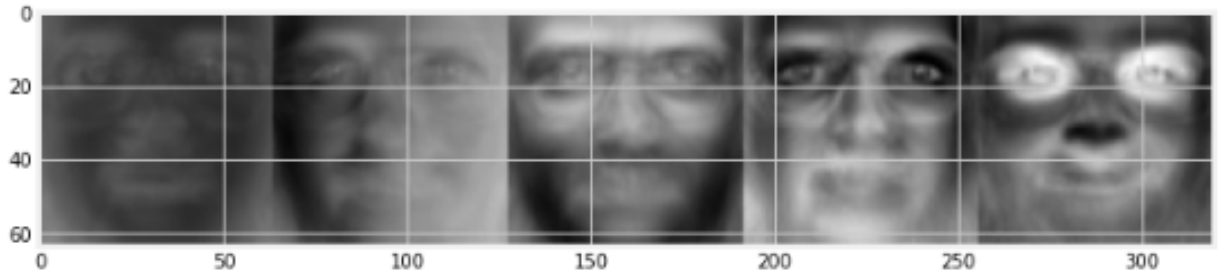
The data for the basis has been saved in a file named `eigenfaces.npy`, first we load it into the variable B.

In [16]:
```
B = np.load('eigenfaces.npy')[:50] # we use the first 50 basis vectors --- you should play around with this.
print("the eigenfaces have shape {}".format(B.shape))
```

```
the eigenfaces have shape (50, 64, 64)
```

Each instance in B is a `64x64' image, an "eigenface", which we determined using an algorithm called Principal Component Analysis. Let's visualize a few of those "eigenfaces".

In [17]:
```
plt.figure(figsize=(10,10))
plt.imshow(np.hstack(B[:5].reshape(-1, 64, 64)), cmap='gray');
```



Take a look at what happens if we project our faces onto the basis B spanned by these 50 "eigenfaces". In order to do this, we need to reshape B from above, which is of size (50, 64, 64), into the same shape as the matrix representing the basis as we have done earlier, which is of size (4096, 50). Here 4096 is the dimensionality of the data and 50 is the number of data points.

Then we can reuse the functions we implemented earlier to compute the projection matrix and the projection. Complete the code below to visualize the reconstructed faces that lie on the subspace spanned by the "eigenfaces".

In [18]:
```
@interact(i=(0, 10))
def show_face_face_reconstruction(i):
    original_face = faces_normalized[i].reshape(64, 64)
    # reshape the data we loaded in variable `B`
    B_basis = B.reshape(B.shape[0], -1).T
    face_reconstruction = project_general(faces_normalized[i], B_basis).reshape(64, 64)
    plt.figure()
    plt.imshow(np.hstack([original_face, face_reconstruction]), cmap='gray')
    plt.show()
```

```
interactive(children=(IntSlider(value=5, description='i', max=10), Output()), _dom_classes=('widget-interact',…
```

What would happen to the reconstruction as we increase the dimension of our basis?

Modify the code above to visualize it.

In the above you used a specially selected basis for your projections. What happens if you simply use a random basis instead?

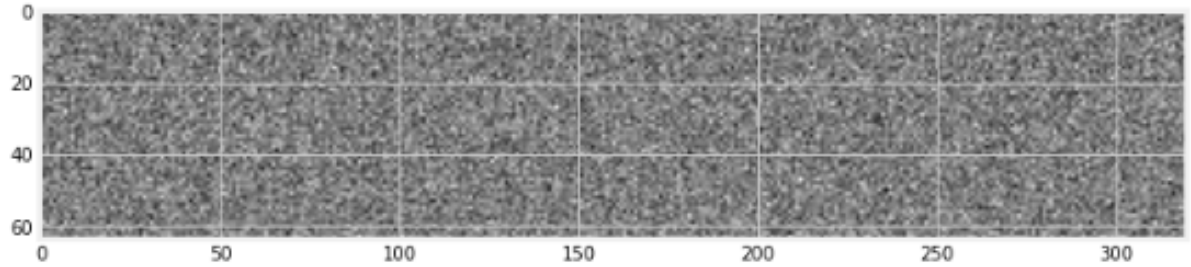
Below, you will project the data onto 50 randomly generated basis vectors. Before you run the code cells below, can you predict the results?

In [19]:
```
B_random = np.random.randn(*B.shape)
print("the random basis has shape {}".format(B_random.shape))
```

```
the random basis has shape (50, 64, 64)
```

As before, we shall visualize the "faces" represented by the basis vectors.

In [20]:
```
plt.figure(figsize=(10,10))
plt.imshow(np.hstack(B_random[:5].reshape(-1, 64, 64)), cmap='gray');
```



As you can see, the basis vectors do not store faces but only store random noise.

Let us now try to project the faces onto this basis.

In [21]:
```
@interact(i=(0, 10))
def show_face_face_reconstruction(i):
    original_face = faces_normalized[i].reshape(64, 64)
    # reshape the data we loaded in variable `B`
    B_basis = B_random.reshape(B_random.shape[0], -1).T
    face_reconstruction = project_general(faces_normalized[i], B_basis).reshape(64, 64)
    plt.figure()
    plt.imshow(np.hstack([original_face, face_reconstruction]), cmap='gray')
    plt.show()
```

```
interactive(children=(IntSlider(value=5, description='i', max=10), Output()), _dom_classes=('widget-interact',…
```

Were these the results you expected? You can see how important it is to use a technique like PCA when selecting your basis for projection!

3. Least squares regression (optional)

In this section, we shall apply the concept of projection to finding the optimal parameters of a least squares regression model.

Consider the case where we have a linear model for predicting housing prices. We are predicting the housing prices based on features in the housing dataset. If we denote the features as $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$ and collect them into a vector $\boldsymbol{x}$, and the price of the houses as y. Assume that we have a prediction model in the way such that $\hat{y}_i = f(\boldsymbol{x}_i) = \boldsymbol{\theta}^T \boldsymbol{x}_i$.

If we collect the dataset into a (N, D) data matrix $\boldsymbol{X}$(where N is the number of houses and D is the number of features for each house), we can write down our model like this:

$$\begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix} \boldsymbol{\theta} = \begin{bmatrix} y_1 \\ \vdots \\ y_2 \end{bmatrix},$$

i.e.,

$$\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{y}.$$

Note that the data points are the *rows* of the data matrix, i.e., every column is a dimension of the data.

Our goal is to find the best $\boldsymbol{\theta}$ such that we minimize the following objective (least square).

$$\begin{aligned} &\sum_{i=1}^{n} \|\bar{y_i} - y_i\|^2 \\ &= \sum_{i=1}^{n} \|\boldsymbol{\theta}^T \boldsymbol{x}_i - y_i\|^2 \\ &= (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}). \end{aligned}$$

If we set the gradient of the above objective to $\mathbf{0}$, we have

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}) &= \mathbf{0} \\ \nabla_{\boldsymbol{\theta}} (\boldsymbol{\theta}^T \boldsymbol{X}^T - \boldsymbol{y}^T)(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}) &= \mathbf{0} \\ \nabla_{\boldsymbol{\theta}} (\boldsymbol{\theta}^T \boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}^T \boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{\theta}^T \boldsymbol{X}^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y}) &= \mathbf{0} \\ 2\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{X}^T \boldsymbol{y} &= \mathbf{0} \\ \boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\theta} &= \boldsymbol{X}^T \boldsymbol{y}. \end{aligned}$$

The solution that gives zero gradient solves (which we call the maximum likelihood estimator) the following equation:

$$\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T \boldsymbol{y}.$$