In [18]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import scipy
```

1. Two Dimensional Signal Processing

In [19]:
```
# Helper function for plotting a 2D array
def plot_2d(x):
  plt.imshow(x)
  plt.show()
```

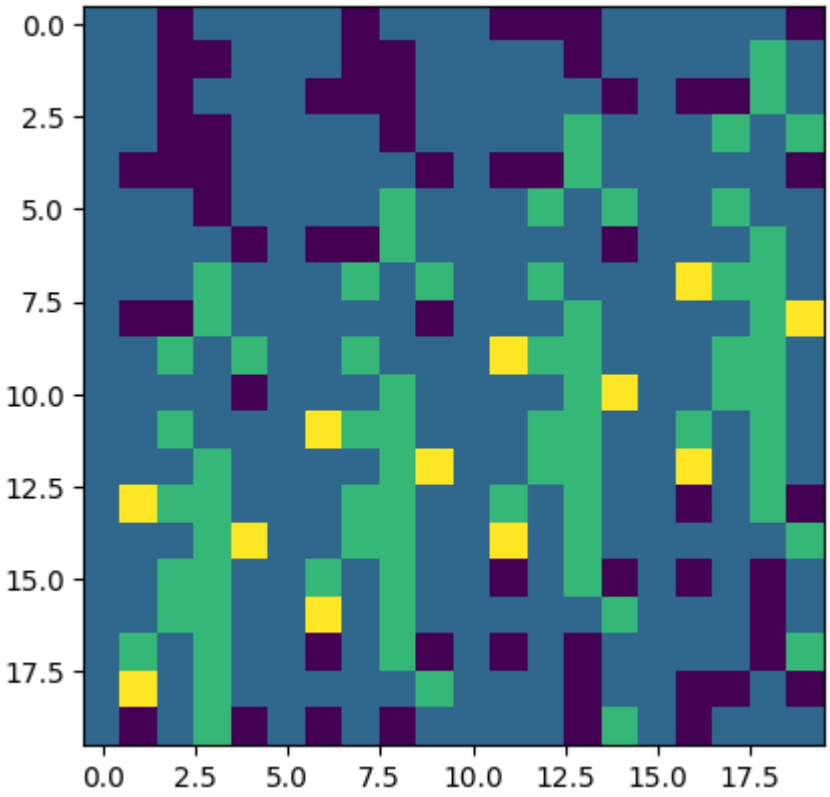
1.1 Inner products and orthogonality

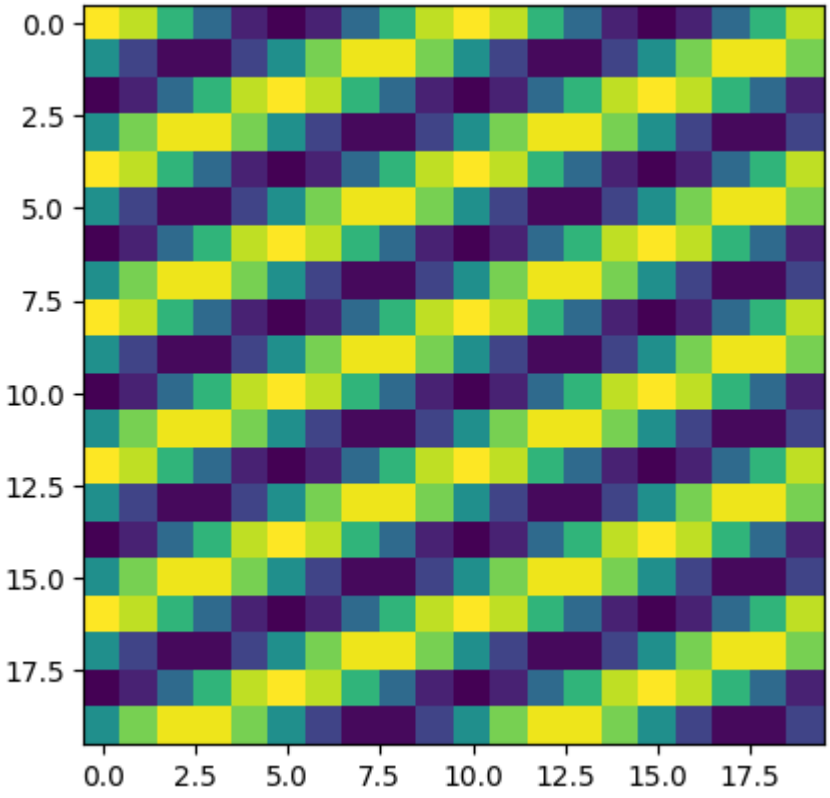
In [20]:
```
def inner_prod_2D(x, y):
  N = np.shape(x)[0]
  inner_prod = 0
  for i in range(N):
      for j in range(N):
          inner_prod += x[i,j] * np.conj(y[i,j])
  return inner_prod
```

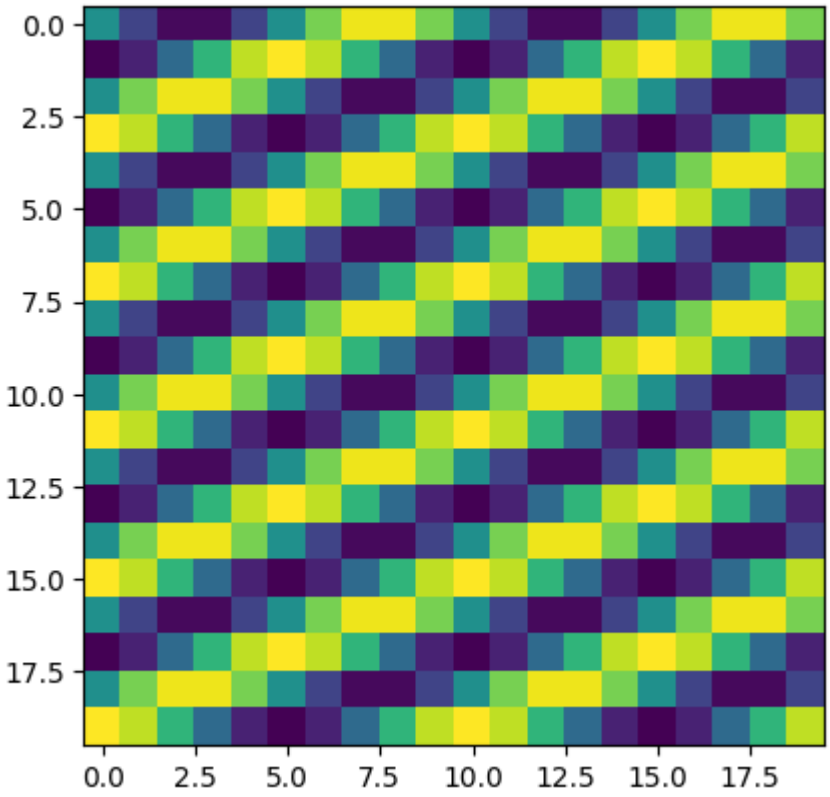
1.2 Discrete Complex Exponentials

In [21]:
```
def complex_exp_2D(N, k, l):
  """
  k, l are frequencies. k is frequency in x direction, l is y.
  N is the duration of the signal
  returns three matrices of size NxN, the first containing the complex values of
  e_kl,NN. the second is the real part; the third is the imaginary part.
  """
  x_tiles = np.tile(np.arange(N), (N, 1))
  y_tiles = np.tile(np.arange(N), (N, 1)).T
  partial_1 = k * x_tiles / N
  partial_2 = l * y_tiles / N
  cexp = 1/N * np.exp(-2j * np.pi * (partial_1 + partial_2))

  cexp_real = np.real(cexp)
  cexp_imag = np.imag(cexp)
  return cexp, cexp_real, cexp_imag
```

In [22]:
```
# test
signal = complex_exp_2D(20, 2, 5)
plot_2d(abs(signal[0]))
plot_2d(signal[1])
plot_2d(signal[2])
```


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

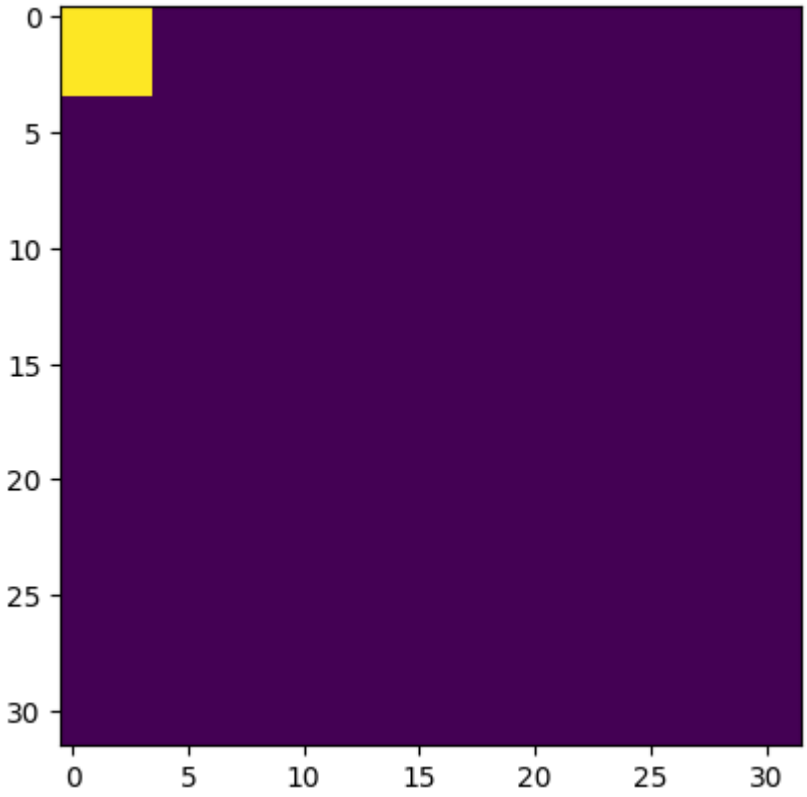

0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



1.3 Unit Energy 2-D Square Pulse

In [23]:
```
def sq_pulse_2D(N, L):
  pulse = np.zeros([N, N])
  for i in range(L):
    for j in range(L):
      pulse[i, j] = 1/(L*L)
  n_samples = N*N
  ####YOUR CODE HERE#####

  return pulse, n_samples
```

In [24]:
```
# TODO:  Plot the two-dimensional square pulse for N = 32 and L = 4
pulse, n_samples = sq_pulse_2D(32, 4)
plot_2d(pulse)
```

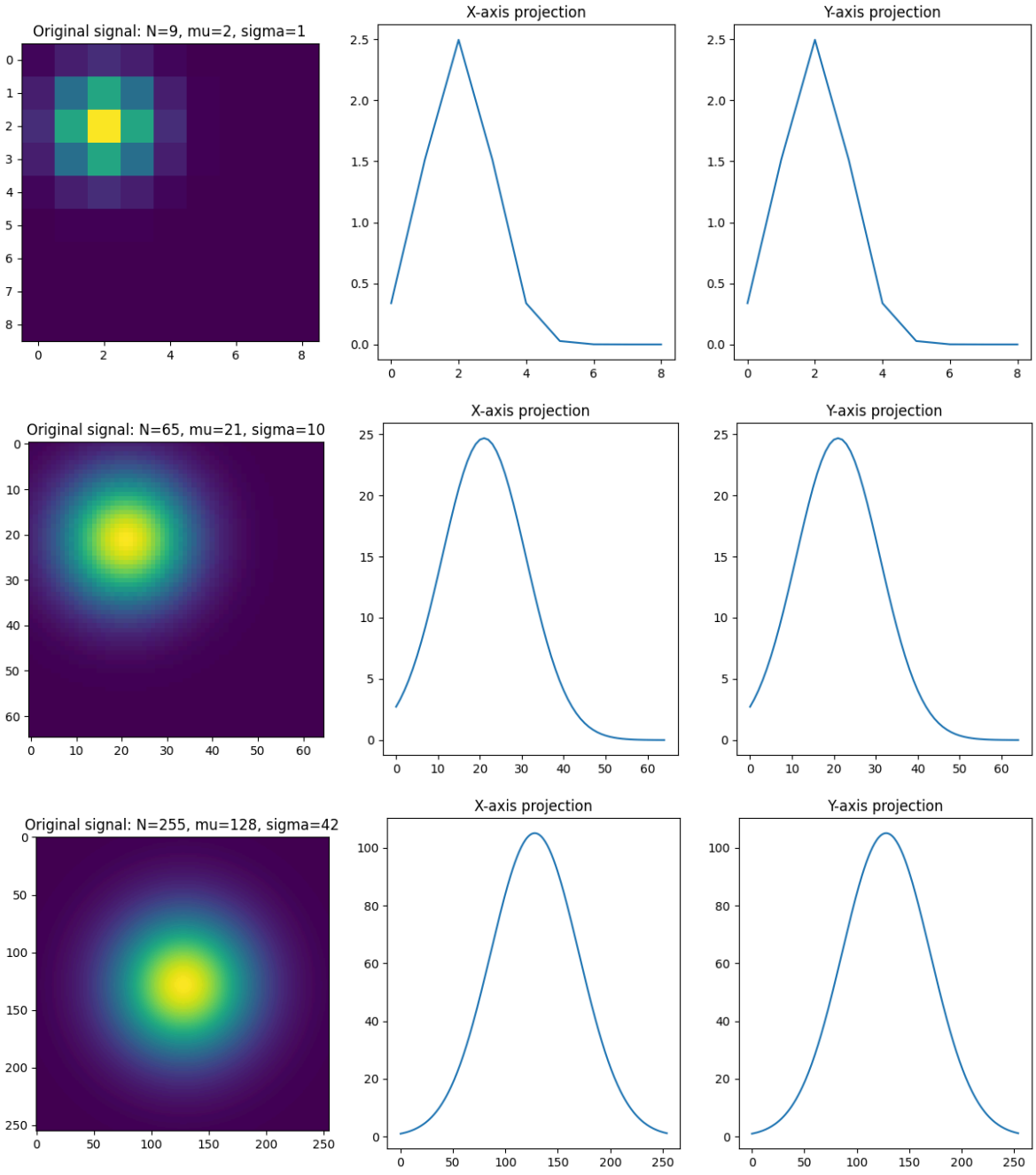


1.4 Two-Dimensional Gaussian Signals

In [25]:
```
def Gaussian_2D(N, mu, sigma):
  """N: size
  mu: mean
  sigma: stdev

  outputs a 2D array representing a Gaussian pulse of size NxN
  """
  x_tiles = np.tile(np.arange(N), (N, 1))
  y_tiles = np.tile(np.arange(N), (N, 1)).T

  pulse = np.exp(-((x_tiles - mu) ** 2 + (y_tiles - mu) ** 2)/ (2 * sigma**2))
  n_samples = N ** 2

  return pulse, n_samples
```

In [26]:
```
# Plot the two dimensional Gaussian for the requested parameters and project it

def plot_signal_and_projections(x, title):
  """x is a 2D array"""
  # make three horizontal figures
  fig, axs = plt.subplots(1, 3, figsize=(15, 5))
  axs[0].imshow(x)
  axs[0].set_title("Original signal: " + title)
  axs[1].plot(np.sum(x, axis=0))
  axs[1].set_title("X-axis projection")
  axs[2].plot(np.sum(x, axis=1))
  axs[2].set_title("Y-axis projection")


plot_signal_and_projections(Gaussian_2D(9, 2, 1)[0], "N=9, mu=2, sigma=1")
```

```
plot_signal_and_projections(Gaussian_2D(65, 21, 10)[0], "N=65, mu=21, sigma=10")
plot_signal_and_projections(Gaussian_2D(255, 128, 42)[0], "N=255, mu=128, sigma=
```

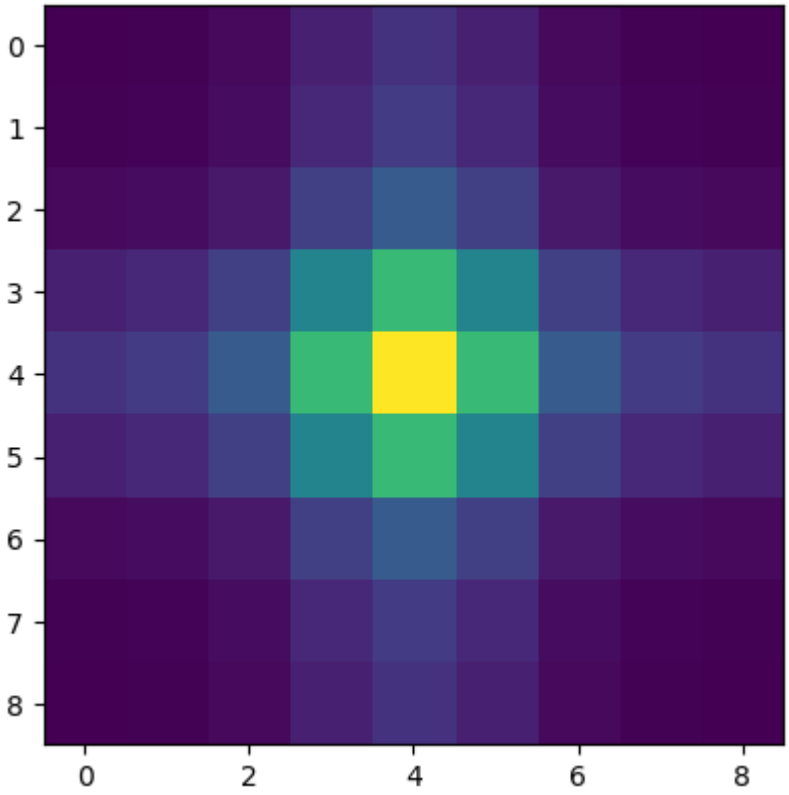


Comment: When a two-dimensional Gaussian pulse is projected onto one dimension, it resembles the general shape of a one dimensional gaussian pulse.

1.5 DFT in two dimensions

In [27]:
```
def compute_DFT_2D(x):
  # X is a 2D array containing the coefficients of the 2D DFT of the NxN dimensi
  N = np.shape(x)[0]
  X = np.zeros([N, N], dtype=np.complex128)
  for k in range(N):
    for l in range(N):
      cexp, _, _ = complex_exp_2D(N, k, l)
      X[k, l] = inner_prod_2D(x, cexp)
  ####YOUR CODE HERE#####
  return X
```

In [28]:
```
# Plot your results on the two-dimensional plane
x = Gaussian_2D(9, 0, 2)
X = compute_DFT_2D(x[0])
# FFTshift rearranges it so that the 0 frequency is at the center instead of the
# of the array, making the visualization more intuitive.
# it changes the coord system from "0-to-N-1" to "-N/2 to N/2-1"
plot_2d(abs(np.fft.fftshift(X)))
```

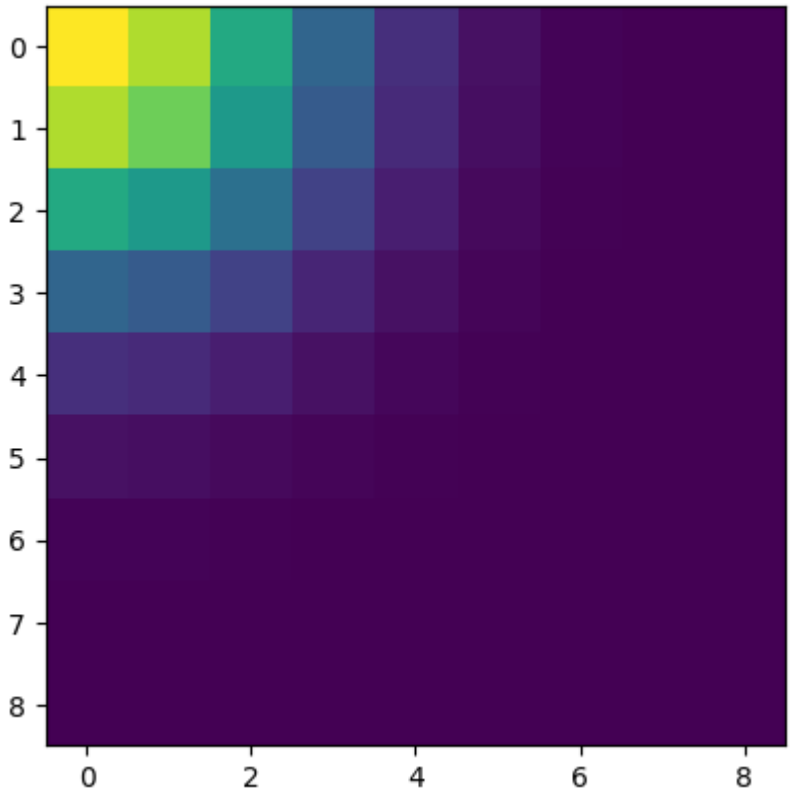


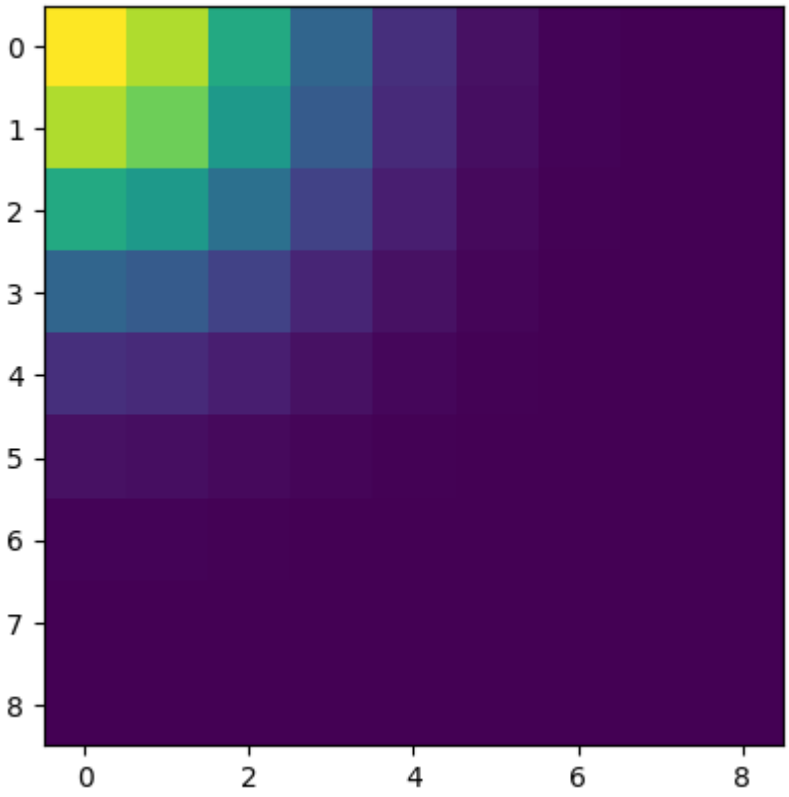
1.6 iDFT in Two Dimensions

In [29]:
```
def compute_iDFT_2D(X):

  # x is an NxN dimensional signal whose DFT is given by X
  # X has dimensions of... NxN
  # it goes from 0 to N-1 along each dimension.
  N = np.shape(X)[0]
  x = np.zeros((N, N), dtype=np.complex128)

  for m in range(N):
      for n in range(N):
          sum_val = 0
```

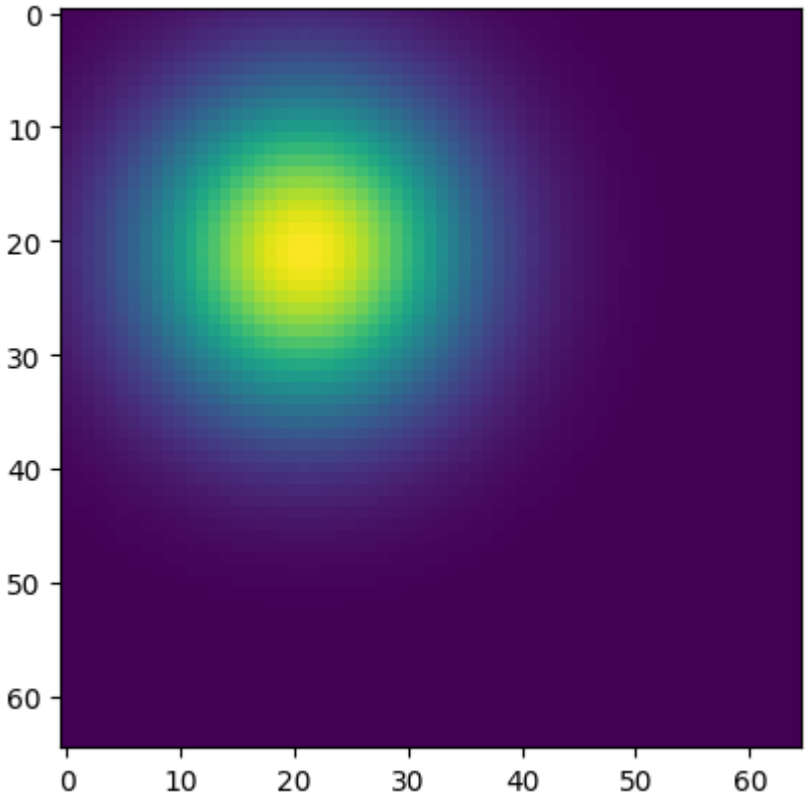
```
        # Process the first half of frequencies
        for k in range(N//2 + 1):  # 0 to N/2
            for l in range(N):
                exponent = 2j * np.pi * (m * k / N + n * l / N)
                current_term = X[k, l] * np.exp(exponent)

                # Add contribution from the conjugate pair
                # Skip for k=0 (which are their own conjugates)
                if k > 0:
                    conj_k = (N - k) % N
                    conj_l = (N - l) % N
                    conj_exponent = 2j * np.pi * (m * conj_k / N + n * conj_l
                    current_term += np.conj(X[k, l]) * np.exp(conj_exponent)

                sum_val += current_term

        # convert coordinate system to match original
        x[N-m if m != 0 else m, N-n if n != 0 else n] = (sum_val / N)

    return x
```

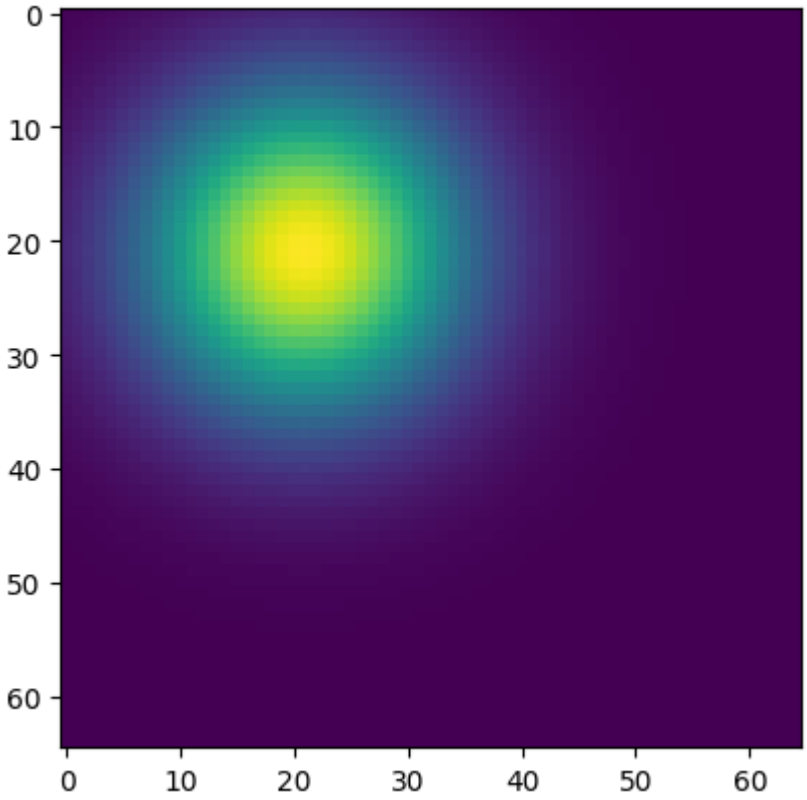
In [30]:
```
x_tilde = compute_iDFT_2D(X)
plot_2d(abs(x_tilde))
plot_2d(x[0])
```





In [31]:
```
x2 = Gaussian_2D(65, 21, 10)
X2 = compute_DFT_2D(x2[0])
x_tilde2 = compute_iDFT_2D(X2)
plot_2d(abs(x_tilde2))
plot_2d(x2[0])
```





Comment: In both cases the original and IDFT output are identical, as expected, suggesting that both the DFT and IDFT implementations are correct.

In [32]:
```
print(np.sum(abs(x_tilde - x[0])))
print(np.sum(abs(x_tilde2 - x2[0])))
```

```
3.95952178477438e-14
2.129100885209922e-11
```

2. Image Filtering and de-noising

2.1 Spatial de-noising

First, read and Display the images linked on the lab page using the function 'plt.imread'.

In [33]:
```
# Display orginal images
imgA = plt.imread("imgA.png")
imgB = plt.imread("imgB.png")
plt.imshow(imgA)
plt.axis('off')
plt.show()
plt.imshow(imgB)
plt.axis('off')
plt.show()
```

Then implement Gaussian filtering by convolving the image with a gaussian pulse. You can import and use the function 'scipy.signal.convolve2d'.

In [34]:
```
def gaussian_filter(x, N, mu, sigma):
  ####YOUR CODE HERE#####
  pulse, _ = Gaussian_2D(N, mu, sigma)
  filtered_signal = scipy.signal.convolve2d(x, pulse, mode='same')
  return filtered_signal
```

In [38]:
```
Ns = [7,13,25]
for N in Ns:
  mu = (N-1)/2
```

```
  sigma = (N-1)/6
  imgA_filtered = gaussian_filter(imgA, N, mu, sigma)
  imgB_filtered = gaussian_filter(imgB, N, mu, sigma)
  print("N = ", N)
  plt.imshow(imgA_filtered)
  plt.axis('off')
  plt.show()
  plt.imshow(imgB_filtered)
  plt.axis('off')
  plt.show()
```

```
N =  7
```

```
N =  13
```

N = 25

Comment: the denoising is dependent on sigma. I have experminted and increased sigma and decreased it. As we increase sigma the image loses details and harshly get denoised and vice versa. Conclusion, denoising is dependent on sigma.