ECE 285 – Project B Total Variation

In [1]:
```
%load_ext autoreload
%autoreload 2
import numpy as np
import numpy.fft as npf
import matplotlib
import matplotlib.pyplot as plt
import time
import imagetools.projectB as im
starfish = plt.imread('assets/starfish.png')
flowers = plt.imread('assets/flowers.png')
ball = plt.imread('assets/ball.png')
%matplotlib notebook
```

1 Operators

1

```
class Identity(LinearOperator):
    def __init__(self, ishape, oshape=None):
        LinearOperator.__init__(self, ishape, oshape=None)
        self._ishape = ishape
        self.nu = kernel("box",tau=0)
        self.mu = np.flip(self.nu,1)[::-1]
        self.H = kernel2fft(self.nu, ishape[0], ishape[1])
        self.H_star = kernel2fft(self.mu, ishape[0], ishape[1])
    def __call__(self, x):
        return convolvefft(x, self.H)
    def adjoint(self, x):
        return convolvefft(x, self.H_star)
    def gram(self, x):
        return convolvefft(x, self.H*self.H_star)
    def gram_resolvent(self, x, tau):
        res_lbd = 1 / (1 + tau * self.H*self.H_star)
        return convolvefft(x, res_lbd)
```

2

```
class Convolution(LinearOperator):
    def __init__(self, ishape, nu, separable=None, oshape=None):
        LinearOperator.__init__(self, ishape, oshape=None)
        self._ishape = ishape
        self.nu = nu
        self._separable = separable
        self.mu = np.flip(self.nu,1)[::-1]
        self.H = kernel2fft(self.nu, ishape[0], ishape[1],
self._separable)
```

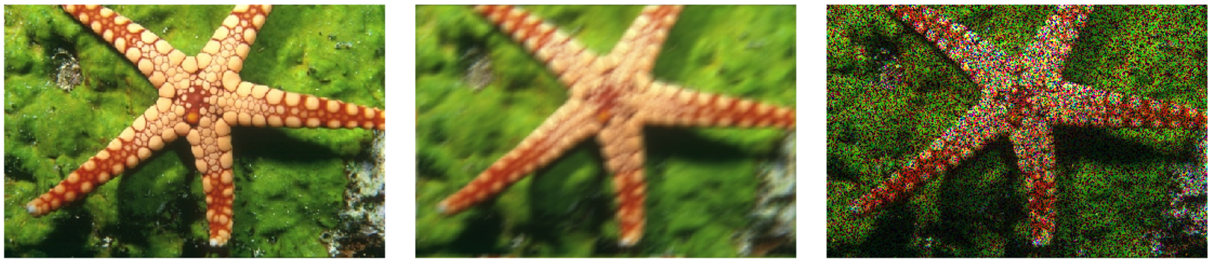
```
        self.H_star = kernel2fft(self.mu, ishape[0], ishape[1],
self._separable)
    def __call__(self, x):
        return convolvefft(x, self.H)
    def adjoint(self, x):
        return convolvefft(x, self.H_star)
    def gram(self, x):
        return convolvefft(x, self.H*self.H_star)
    def gram_resolvent(self, x, tau):
        res_lbd = 1 / (1 + tau * self.H*self.H_star)
        return convolvefft(x, res_lbd)
```

3

```
class RandomMasking(LinearOperator):
    def __init__(self, ishape, p):
        LinearOperator.__init__(self, ishape, oshape=None)
        self._p = p
        self._mask = np.random.choice([0,1], size=ishape, p=[self._p, 1-
self._p])
    def __call__(self, x):
        return np.multiply(self._mask, x)
    def adjoint(self, x):
        return np.multiply(self._mask, x)  # since conjugate of random
mask is the same mask
    def gram(self, x):
        return np.multiply(self._mask, x)
    def gram_resolvent(self,x,tau):
        return x + (tau * self(x))
```

4

In [3]:
```
x0 = starfish
I = im.Identity(x0.shape)
nu = im.kernel('motion')
C = im.Convolution(x0.shape, nu)
R = im.RandomMasking(x0.shape, 0.4)
x1 = I(x0)
x2 = C(x0)
x3 = R(x0)
fig, axes = plt.subplots(ncols=3, figsize=(16,4))
im.show(x1, ax=axes[0])
im.show(x2, ax=axes[1])
im.show(x3, ax=axes[2])
fig.show()
```



5

In [4]:
```
x = starfish
y = flowers

x_ic = im.Identity(x.shape)(x)
y_ia = im.Identity(y.shape).adjoint(y)
print("The statement for Identity operator:",np.isclose(np.sum(x_ic*y),np.s

nu = im.kernel('motion')
x_cc = im.Convolution(x.shape, nu)(x)
y_ca = im.Convolution(y.shape, nu).adjoint(y)
print("The statement for Convolution operator:",np.isclose(np.sum(x_cc*y),n

x_rc = im.RandomMasking(x.shape, 1)(x)
y_ra = im.RandomMasking(y.shape, 1)(y)
print("The statement for RandomMasking operator:",np.isclose(np.sum(x_rc*y)
```

```
The statement for Identity operator: True
The statement for Convolution operator: True
The statement for RandomMasking operator: True
```

6

In [5]:
```
t = 0.1
xi = im.Identity(x.shape)
x_il = xi.gram_resolvent(x+t*xi.gram(x), t)
print("The statement for Identity operator:",np.allclose(x_il, x))

nu = im.kernel('motion')
xc = im.Convolution(x.shape, nu)
x_cl = xc.gram_resolvent(x+t*xc.gram(x), t)
print("The statement for Convolution operator:",np.allclose(x_cl, x))

xr = im.Identity(x.shape, 0.4)
x_rl = xr.gram_resolvent(x+t*xr.gram(x), t)
print("The statement for RandomMasking operator:",np.allclose(x_rl, x))
```

```
The statement for Identity operator: True
The statement for Convolution operator: True
The statement for RandomMasking operator: True
```

2 Smoothed Total-Variation

7

```
Total-Variation promotes piece-wise constant solutions due to the nature
that TV smooths out large ‘flat’ regions in the image by only storing the
gradient of the image to do so. This is since the loss function, which is
the gradient of x, is simply comparing the difference between a pixel and
its neighbors, thus only detecting edges in the image.  Since there are
relatively few edges in relation to image size, this leads TV to produce
piece-wise constant solutions.
```

8

1. Given:

$$E(x) = \frac{1}{2}||y - Hx||_2^2 + \tau||\nabla x||_1$$

2. Approximately,

$$|\nabla x| = \sqrt{|\nabla x|^2 + \varepsilon}$$

3. For the first part,

$$\begin{aligned}\nabla\frac{1}{2}||y - Hx||_2^2 &= \nabla\frac{1}{2}(||Hx||^2 + ||y||^2 - 2 < Hx, y >) \\ &= \nabla\frac{1}{2}(< x, H^*Hx > +||y||^2 - 2 < x, H^*y >) \\ &= \frac{1}{2}((H^*H + H^*H)x - 2H^*y) \\ &= H^*(Hx - y)\end{aligned}$$

4. For the second part,

$$\nabla\sqrt{|\nabla x|^2+\varepsilon} = -\nabla\cdot\left(\frac{\nabla x}{\sqrt{|\nabla x|^2+\varepsilon}}\right)$$
$$= -div\left(\frac{\nabla x}{\sqrt{|\nabla x|^2+\varepsilon}}\right)$$

5. Therefore,

$$\nabla E(x) = H^*(Hx-y) - div\left(\frac{\nabla x}{\sqrt{|\nabla x|^2+\varepsilon}}\right)$$

9

```
def total_variation(y, sig, H = None, m=400, scheme = 'gd', rho=1,
return_energy = False):
    tau = rho * sig
    epsilon = sig**2 * 0.001
    if H is None:
        H = Identity(y.shape)

    laplacian = kernel('laplacian2')
    L = H.norm2()**2 + (tau/np.sqrt(epsilon)) * np.sqrt(np.sum(laplacian
** 2))
    gamma = 1 / L

    def energy(H, x, y, tau):
        op1 = 0.5 * np.sum((y-H(x))** 2)
        op2 = tau * np.sum(np.gradient(x))
        return op1 + op2

    def loss(H, x, y, tau, epsilon):
        op1 = H.gram(x) - H.adjoint(y)
        gradX = grad(x)
        op2 = tau * np.sum((gradX / np.sqrt(np.abs(gradX)**2 + epsilon)),
axis=2)
        return op1 - op2

    x = H.adjoint(y)

    if scheme is 'gd':
        if return_energy:
            e = []
            for i in range(m):
                e.append(energy(H,x,y,tau))
                x = x - gamma * loss(H,x,y,tau,epsilon)
            return x, e

        else:
            for i in range(m):
                print("iteration %d"% i)
                x = x - gamma * loss(H,x,y,tau,epsilon)
            return x
```

10

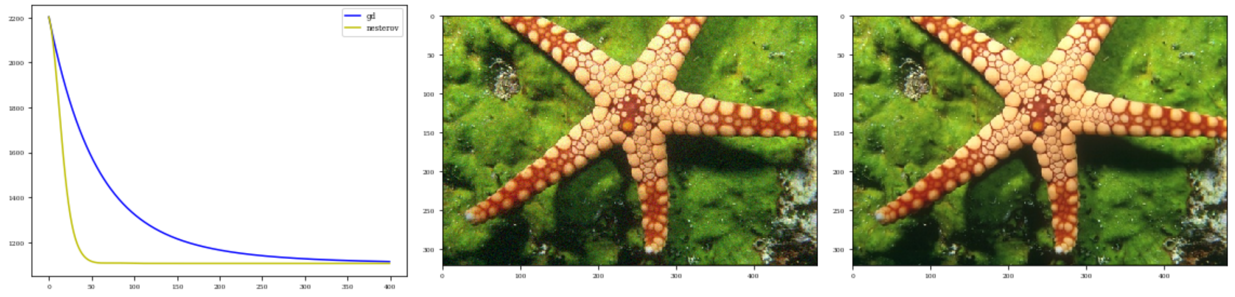
```
def total_variation(y, sig, H = None, m = 400, scheme = 'gd', rho=1,
return_energy = False):
    ...
    elif scheme is 'nesterov':
        xBar = H.adjoint(y)
        prevX = x

        def calcT1(t):
            return (1 + np.sqrt(1 + 4*t**2)) / 2

        def calcU(t, t1):
            return (t-1)/t1

        t = 1
        t1 = calcT1(t)

        if return_energy:
            e = []
            for i in range(m):
                e.append(energy(H,x,y,tau))
                prevX = x
                x = xBar - gamma * loss(H,xBar,y,tau,epsilon)
                xBar = x + calcU(t,t1)*(x-prevX)
                t = t1
                t1 = calcT1(t)
            return x, e

        else:
            for i in range(m):
                print("iteration %d"% i)
                prevX = x
                x = xBar - gamma * loss(H,xBar,y,tau,epsilon)
                xBar = x + calcU(t,t1)*(x-prevX)
                t = t1
                t1 = calcT1(t)
            return x
```

11

In [9]:
```
x0 = starfish
sig = 10/255
y = x0 + sig * np.random.randn(*x.shape)
x1, e1 = im.total_variation(y, sig, H = None, m = 400, scheme = 'gd', rho=1
x2, e2 = im.total_variation(y, sig, H = None, m = 400, scheme = 'nesterov',
```

In [10]:
```
fig, axes = plt.subplots(ncols=3, figsize=(16,4))
plt.subplot(1,3,1)
plt.plot(e1,'b-',e2,'y-')
plt.legend(labels = ['gd', 'nesterov'], loc = 'upper right')
plt.subplot(1,3,2)
plt.imshow(y)
plt.subplot(1,3,3)
plt.imshow(x0)
fig.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
```



12

In [11]:
```
print("For gradient descent, the energy after 50 iterations compared to the
print("For Nesterov acceleration, the energy after 50 iterations compared t
```

```
For gradient descent, the energy after 50 iterations compared to the orig
inal energy 0.4276907676344612
For Nesterov acceleration, the energy after 50 iterations compared to the
original energy 0.008277000210652558
```

m = 50 iterations is not enough for gradient descent, but is enough for Nesterov acceleration.

13

In [12]:
```
x0 = starfish
p = 0.4
sig = 2/255

R = im.RandomMasking(x0.shape, p)
y2 = R(x0)

x21, e21 = im.total_variation(y2, sig, H = None, m = 400, scheme = 'gd', rh
x22, e22 = im.total_variation(y2, sig, H = None, m = 400, scheme = 'nesterc
print("For the random masking and gradient descent, the energy after 50 ite
print("For the random masking and Nesterov acceleration, the energy after 5

nu = im.kernel('motion')
y3 = im.convolve(x0, nu, 'periodical')
x31, e31 = im.total_variation(y3, sig, H = None, m = 400, scheme = 'gd', rh
x32, e32 = im.total_variation(y3, sig, H = None, m = 400, scheme = 'nesterc
print("For the motion blur and gradient descent, the energy after 50 iterat
print("For the motion blur and Nesterov acceleration, the energy after 50 i
```

```
For the random masking and gradient descent, the energy after 50 iteratio
ns compared to the original energy 0.6368973894459802
For the random masking and Nesterov acceleration, the energy after 50 ite
rations compared to the original energy 0.017836780698799917
For the motion blur and gradient descent, the energy after 50 iterations
compared to the original energy 0.43749955197842727
For the motion blur and Nesterov acceleration, the energy after 50 iterat
ions compared to the original energy 0.009392941860200807
```

The statement still holds: m = 50 iterations is not enough for gradient descent, but is enough for Nesterov acceleration.

3 Advanced solvers for Total-Variation

14

```
def softthresh(z, t):
    z[np.abs(z) <= t] = 0
    return z - np.sign(z)*t
```

15

```
def total_variation(y, sig, H = None, m = 400, scheme = 'gd', rho = 1,
return_energy = False):
    ...
    elif scheme is 'admm':
        #initialize iteratable variables
        gamma = 1
        xBar = H.adjoint(y)
        zBar = Grad(y.shape)(xBar)
        dX = np.zeros(y.shape)
        dZ = np.zeros(zBar.shape)

        e = []
        for i in range(m):
            x = H.gram_resolvent(xBar + dX + gamma*H.adjoint(y), gamma)
            e.append(energy(H,x,y,tau))
            z = softthresh(zBar + dZ, gamma*tau)
            xBar = Grad(y.shape).gram_resolvent(x-
dX+Grad(zBar.shape).adjoint(z-dZ), 1)
            zBar = Grad(y.shape)(xBar)
            dX = dX - x + xBar
            dZ = dZ - z + zBar

        if return_energy:
            return x, e
        else:
            return x
```

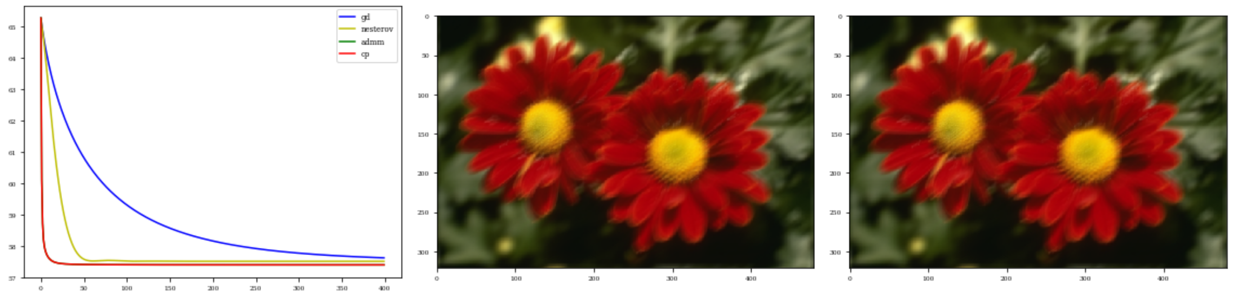
16

```
def total_variation(y, sig, H = None, m = 400, scheme = 'gd', rho = 1,
return_energy = False):
    ...
    elif scheme is 'cp':
        #initialize iteratable variables
        gamma = 1
        theta = 1
        k = 1/(Grad(y.shape).norm2()**2)
        x = H.adjoint(y)
        z = np.zeros(Grad(y.shape)(y).shape)
        v = np.zeros(y.shape)

        e = []
        for i in range(m):
            e.append(energy(H,x,y,tau))
            prevX = np.copy(x)
            zBar = z + k * Grad(y.shape)(v)
            z = zBar - softthresh(zBar, tau)
            xBar = x - gamma * Grad(z.shape).adjoint(z)
            x = H.gram_resolvent(xBar + gamma*H.adjoint(y), gamma)
            v = x + theta*(x-prevX)

        if return_energy:
            return x, e
        else:
            return x
```

17

In [40]:
```
x0 = flowers
nu = im.kernel('motion')
y = im.convolve(x0, nu, 'periodical')
sig = 2/255
x1, e1 = im.total_variation(y, sig, H = None, m = 400, scheme = 'gd', rho=1
x2, e2 = im.total_variation(y, sig, H = None, m = 400, scheme = 'nesterov',
x3, e3 = im.total_variation(y, sig, H = None, m = 400, scheme = 'admm', rhc
x4, e4 = im.total_variation(y, sig, H = None, m = 400, scheme = 'cp', rho=1
```

In [44]:
```
fig, axes = plt.subplots(ncols=3, figsize=(16,4))
plt.subplot(1,3,1)
plt.plot(e1,'b-',e2,'y-', e3, 'g-', e4, 'r-')
plt.legend(labels = ['gd', 'nesterov', 'admm', 'cp'], loc = 'upper right')
plt.subplot(1,3,2)
plt.imshow(y)
plt.subplot(1,3,3)
plt.imshow(x4)
fig.show()
```



18

Chambolle-Pock method is equivalent to ADMM if H = I. But in the general case, Chambolle-Pock is usually faster, since solving the subproblems of ADMM is harder.

4 Total Generalized Variation

19

When $\zeta = 0$,

$$E(x) = \frac{1}{2}||y - Hx||_2^2 + \tau \min_z(||\nabla x||_1 + ||div z||_1)$$

However, since the entire expression is inside a minimum with respect to z, the minimum z will be found that sets the $||div z||_1$ to be 0, which is just the 0 vector for z. Therefore, TGV is equivalent to TV when $\zeta = 0$.

20

To understand why TGV promotes piece-wise affine solutions, it is helpful to understand why TV does not promote affine solutions. TV does not promote affine solutions since it is assuming that the gradient of x can only be taken in two directions (right and down, due to the definition of the grad kernel), this produces square artifacts in the TV reconstruction since by nature of the algorithm it is assuming that images consist of similar colorred square patches. However, in TGV we discard this assumption with the addition of the z vector. The z vector now allows us to take the gradient in any direction, and, since it is minimized in the TGV expression, it chooses the direction z that minimizes the loss. By doing this, we would assume that the resulting images will no longer have square artifacts in the result since the directions of TGV are not limited to the x,y axes.

21

This one is pretty simple, just take the given formula, substitute the matrices, and perform matrix multiplication to get the original TV formula. There is one step in the second term of the sum where we make use that $||\cdot||_1$ is a linear operator (since it is a sum) so we can separate out the $divz$. Otherwise, the $\min_z$ from the original expression is lost, but that is acceptable since we are using a given z in the form of X = (x,z).

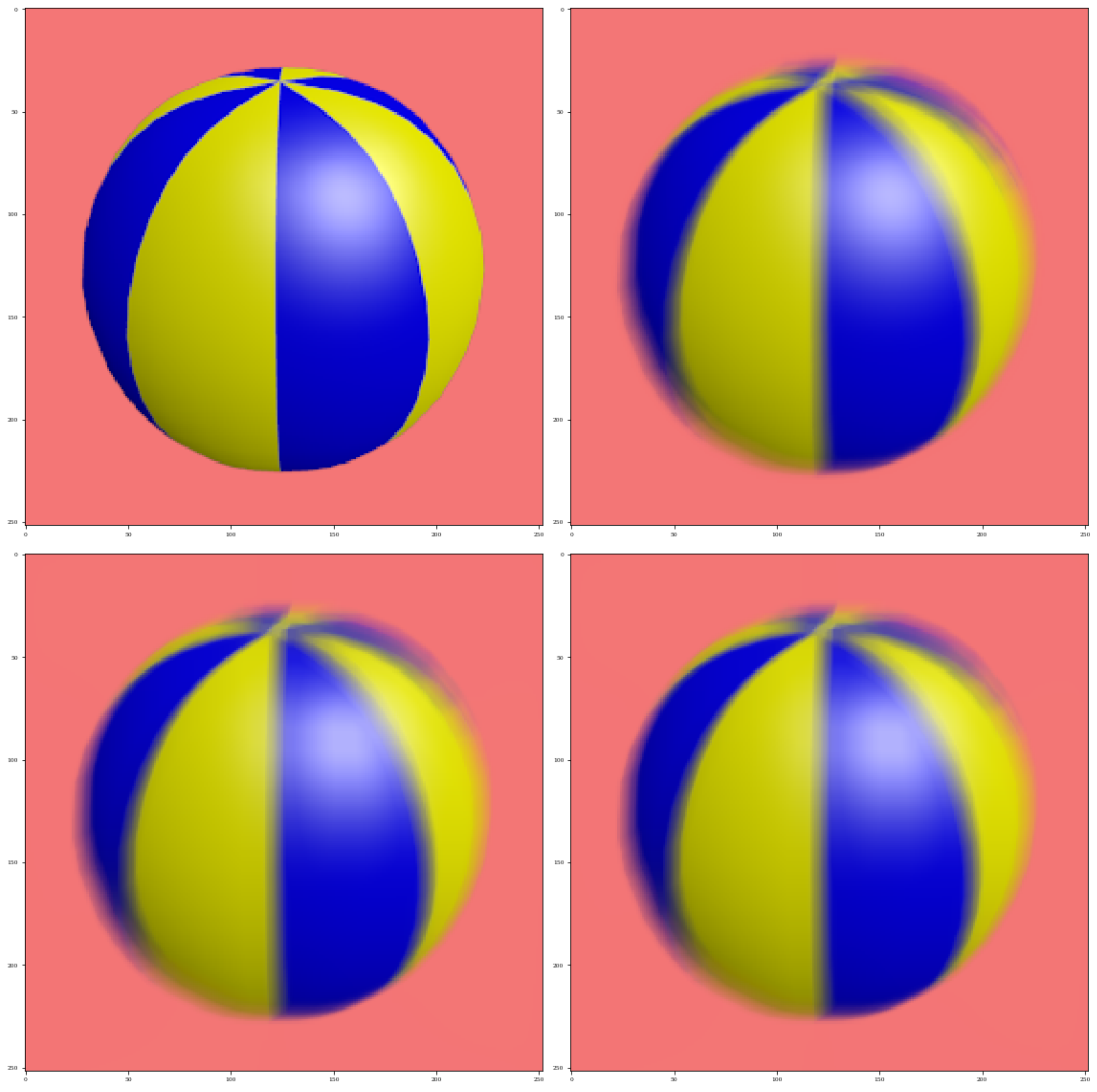
22

```
def tgv(y, sig, H=None, zeta=.1, rho=1, m=400, return_energy=False):
    def energyGen(H,x,y,Gamma,tau):
        op1 = 0.5 * np.sum(np.square(y-H(x[0])))
        t = Gamma(x)
        op2 = tau * (np.sum(np.abs(t[0])) + np.sum(np.abs(t[1])))
        return op1 + op2

    if H is None:
        H = Identity(y.shape)

    gamma = 1
    G = Gamma(y.shape, zeta)
    tau = rho * sig
    gr = Grad(y.shape)
    zeroGrad = np.zeros(gr(y).shape)
    zeroImg = np.zeros(y.shape)

    X = [None, None]
    Z = [None, None]
    ZBar = [None, None]
    XBar = [None, None]
    DX = [None, None]
    DZ = [None, None]

    XBar[0] = H.adjoint(y)
    XBar[1] = np.copy(zeroGrad)

    ZBar[0] = gr(XBar[0])
    ZBar[1] = np.copy(zeroImg)
    DX[0] = np.copy(zeroImg)
    DX[1] = np.copy(zeroGrad)
    DZ[0] = np.copy(zeroGrad)
    DZ[1] = np.copy(zeroImg)

    e = []
    for i in range(m):
        star = XBar[0] + DX[0] + gamma * H.adjoint(y)
        X[0] = star + gamma * H.gram(star)
        X[1] = XBar[1] + DX[1]

        Z[0] = softthresh(ZBar[0] + DZ[0], gamma*tau)
        Z[1] = softthresh(ZBar[1] + DZ[1], gamma*tau)

        star2 = [None, None]
        inner = [None, None]
        inner[0] = Z[0] - DZ[0]
        inner[1] = Z[1] - DZ[1]
        t = G.adjoint(inner)
        star2[0] = X[0] - DX[0] + t[0]
        star2[1] = X[1] - DX[1] + t[1]
        t2 = G.gram_resolvent(star2, 1)
        XBar[0] = t2[0]
        XBar[1] = t2[1]

        t = G(XBar)
        ZBar[0] = t[0]
        ZBar[1] = t[1]

        DX[0] = DX[0] - X[0] + XBar[0]
        DX[1] = DX[1] - X[1] + XBar[1]

        DZ[0] = DZ[0] - Z[0] + ZBar[0]
        DZ[1] = DZ[1] - Z[1] + ZBar[1]

        e.append(energyGen(H,X,y,G,tau))

    if return_energy:
        return X, e
    else:
        return X
```

23

In [51]:
```
x0 = ball
nu = im.kernel('motion')
y = im.convolve(x0, nu, 'periodical')
sig = 10/255
x1, e1 = im.total_variation(y, sig, H = None, m = 400, scheme = 'admm', rhc
x2, e2 = im.tgv(y, sig, H=None, zeta=.1, rho=1, m=400, return_energy = True
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize=(16,16))
plt.subplot(2,2,1)
plt.imshow(x0)
plt.subplot(2,2,2)
plt.imshow(y)
plt.subplot(2,2,3)
plt.imshow(x1)
plt.subplot(2,2,4)
plt.imshow(x2)
fig.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
```


0
50
100
150
200
250