Computational Imaging

Objective: implement and evaluate continuous (variational) and discrete (MRF) formulations for image denoising

Students: Alex SZAPIRO & Gonzalo BECKER

Part I: Heat equation

Derive and implement the diffusion equation associated with the minimization of the regularisation energy $\int_\Omega \|\nabla u(p)\|^2 dp$.

Firstly, the discretized version of the variational energy is given by:

$$J(u) = \sum_{i,j} \|\nabla u_{i,j}\|^2$$

We then search for

$$u_{i,j} = \arg\min_{u_{i,j}} J(u)$$

which can be calculed be derivating as follows:

$$\frac{dJ(u)}{du_{i,j}} = \left(\frac{\partial}{\partial u_{i,j}} \left(\left(\frac{\partial u_{i,j}}{\partial x} \right)^2 + \left(\frac{\partial u_{i,j}}{\partial y} \right)^2 \right) \right)$$

$$\frac{dJ(u)}{du_{i,j}} = \left(\frac{\partial u_{i,j}}{\partial x} \frac{\partial}{\partial u_{i,j}} \left(\frac{\partial u_{i,j}}{\partial x} \right) + \frac{\partial u_{i,j}}{\partial y} \frac{\partial}{\partial u_{i,j}} \left(\frac{\partial u_{i,j}}{\partial y} \right) \right)$$

By using the fact that $\partial(\partial y/\partial x)/\partial y = y''(x)/y'(x)$, we finally obtain

$$\frac{dJ(u)}{du_{i,j}} = \left(\frac{\partial^2 u_{i,j}}{\partial x^2} + \frac{\partial^2 u_{i,j}}{\partial y^2} \right) = \nabla^2 u_{i,j}$$

The laplacian can the be replaced by a discretized version by applying a convolution by a 3x3 laplacian operator.

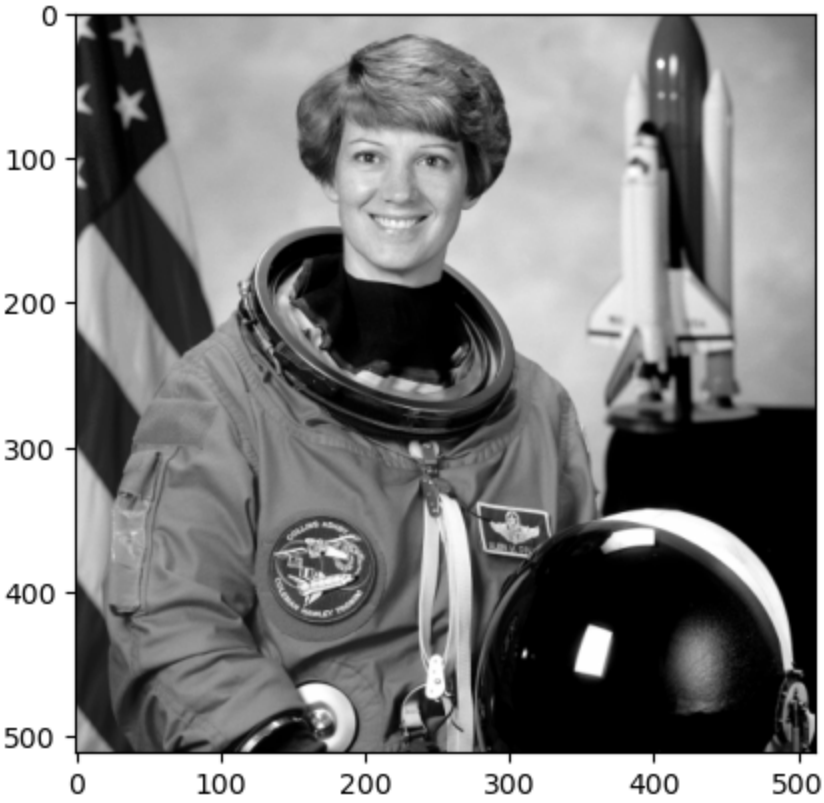
We then easily obtain the heat equation well-known relationship:

$$\frac{\partial u}{\partial t} = \lambda \nabla^2 u_{i,j}$$

Load an image using skimage module. Test images are available at: https://scikit-image.org/docs/dev/auto_examples/data/plot_general.html#sphx-glr-auto-examples-

data-plot-general-py. You may also select your own custom images.

In [28]:
```
import skimage
import matplotlib.pyplot as plt
import numpy as np

original_image = skimage.data.astronaut()
grayscale_image = skimage.color.rgb2gray(original_image)
plt.imshow(grayscale_image, cmap='gray')
plt.show()
```

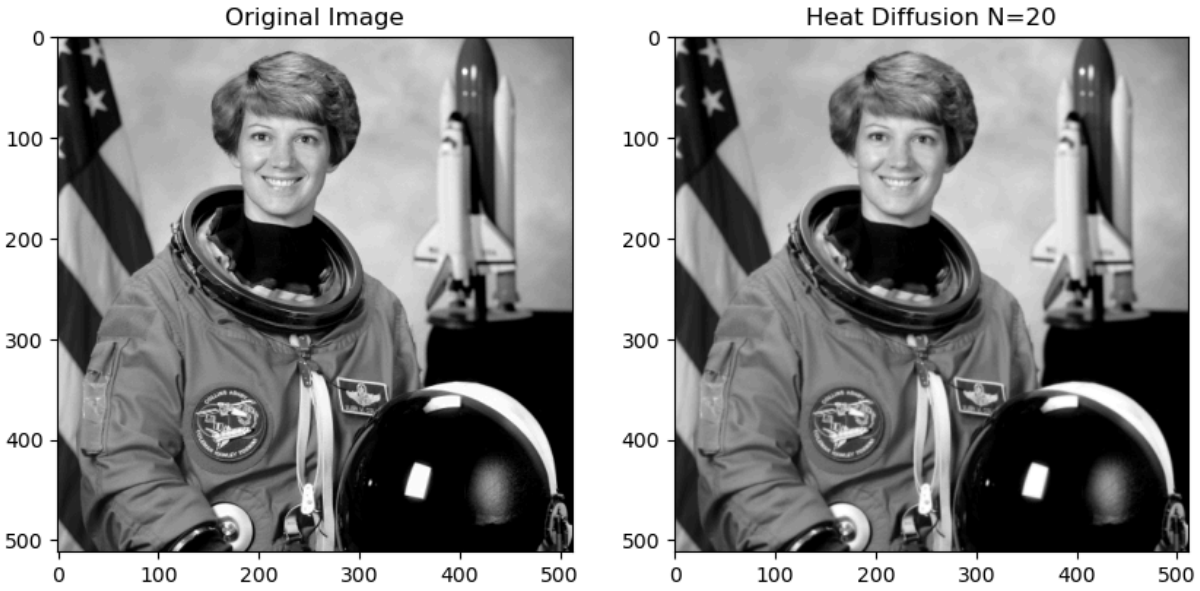


Apply a N-step diffusion process to the loaded image based on the heat equation Code this N-step diffusion process as a callable function using as input parameters, the image u, the number of iterations N and a scalar parameter λ

In [29]:
```
def computeHeatDiffusion(u,N,lambd):

    uf = np.copy(u)
    for n in range(N):
        lap_u = skimage.filters.laplace(u)
        uf = u - lambd * lap_u
        u = uf

    return uf
```

Visualize the result of the diffusion on an image and highlight through an approriate visualization and/or evaluation metrics the quality of the output image.

In [30]:
```
N = 20
diffused_image = computeHeatDiffusion(grayscale_image,N,0.01)

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

plt.subplot(1,2,1)
plt.imshow(grayscale_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1,2,2)
plt.imshow(diffused_image, cmap='gray')
plt.title(f'Heat Diffusion'+f' N={N}')

plt.show()
```



In [31]:
```
from skimage.metrics import mean_squared_error

print("MSE:", mean_squared_error(grayscale_image, diffused_image))
```

```
MSE: 0.0002944475376190239
```

Part II: Heat equation for image denoising

Derive and implement the diffusion equation associated with image denoising according to the following cost function
$\min_u \int_\Omega \|u(p) - v(p)\|^2 dp + \alpha \int_\Omega \|\nabla u(p)\|^2 dp$, where v is a noisy image.

Firstly, the discretized version of the variational energy is given by:

$$J(u) = \frac{1}{2}\sum_{i,j}(u_{i,j} - v_{i,j})^2 + \frac{\alpha}{2}\sum_{i,j}\|\nabla u_{i,j}\|^2$$

We then search for

$$u_{i,j} = \arg\min_{u_{i,j}} J(u)$$

which can be calculed be derivating as follows:

$$\frac{dJ(u)}{du_{i,j}} = u_{i,j} - v_{i,j} + \frac{\alpha}{2}\left(\frac{\partial}{\partial u_{i,j}}\left(\left(\frac{\partial u_{i,j}}{\partial x}\right)^2 + \left(\frac{\partial u_{i,j}}{\partial y}\right)^2\right)\right)$$

$$\frac{dJ(u)}{du_{i,j}} = u_{i,j} - v_{i,j} + \alpha\left(\frac{\partial u_{i,j}}{\partial x}\frac{\partial}{\partial u_{i,j}}\left(\frac{\partial u_{i,j}}{\partial x}\right) + \frac{\partial u_{i,j}}{\partial y}\frac{\partial}{\partial u_{i,j}}\left(\frac{\partial u_{i,j}}{\partial y}\right)\right)$$

By using the fact that $\partial(\partial y/\partial x)/\partial y = y''(x)/y'(x)$, we finally obtain

$$\frac{dJ(u)}{du_{i,j}} = u_{i,j} - v_{i,j} + \alpha\left(\frac{\partial^2 u_{i,j}}{\partial x^2} + \frac{\partial^2 u_{i,j}}{\partial y^2}\right) = u_{i,j} - v_{i,j} + \alpha\left(\nabla^2 u_{i,j}\right)$$

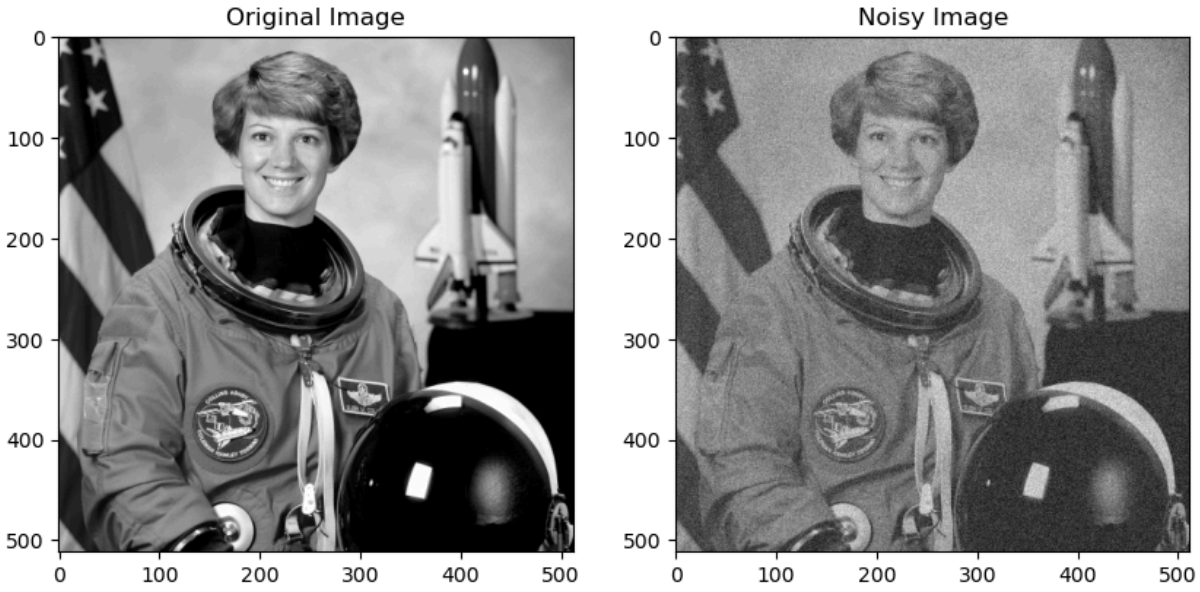
The laplacian can the be replaced by a discretized version by applying a convolution by a 3x3 laplacian operator.

By applying a simple gradient descent we obtain the diffusion equation for this case:

$$\frac{\partial u}{\partial t} = \lambda\left(u_{i,j} - v_{i,j} + \alpha\left(\nabla^2 u_{i,j}\right)\right)$$

Load an image and generate a noisy image using an additive Gaussian noise. You can use function np.random.normal() to generate the noise.

In [32]:
```
import numpy as np

sigma = 0.1

noise = np.random.normal(0, sigma, grayscale_image.shape)
noisy_image = grayscale_image + noise

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

plt.subplot(1,2,1)
plt.imshow(grayscale_image, cmap='gray')
plt.title('Original Image')

plt.subplot(1,2,2)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

plt.show()
```

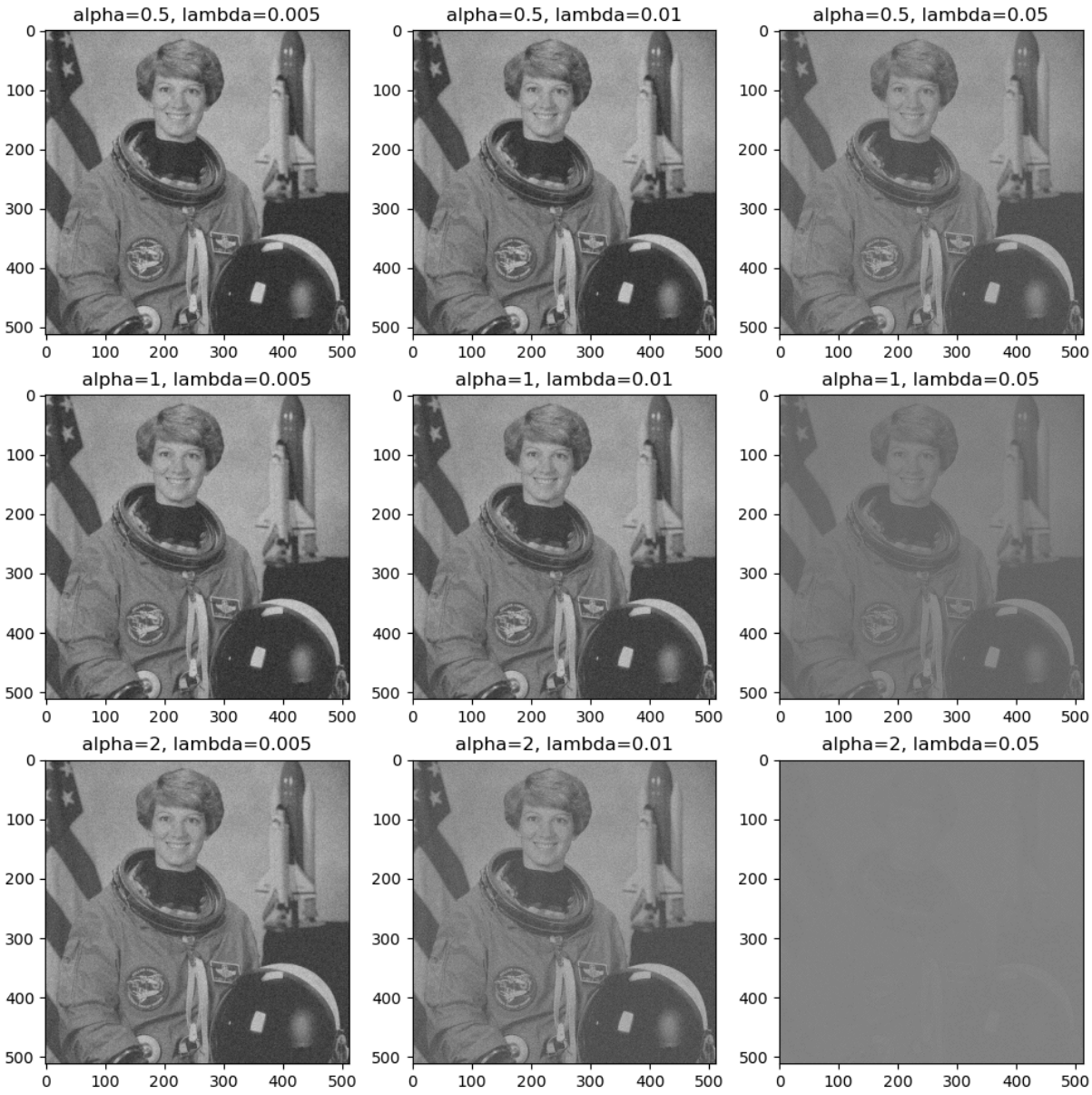


Implement a diffusion equation to solve the above minimization through a function denoisingHeatDiffusion(v,N,alpha) with v the noisy image, N the number of iterations of the diffusion, α the weighing factor of the regularization term and λ the gradient step

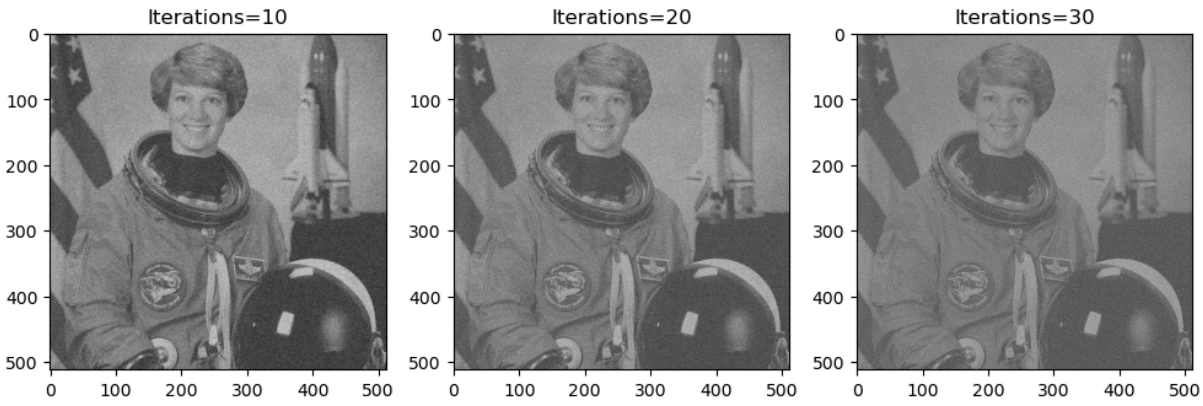
In [33]:
```
def denoisingHeatDiffusion(v,N,alpha,lambd):
    u = v
    for i in range(N):
        u = u - lambd*(2*(u-v) - alpha*skimage.filters.laplace(u))

    return u
```

Run denoising experiments and evaluate the quality of the denoising process w.r.t. the main parameters of the denoising scheme

In [34]:
```
# Effect of alpha and lambda variation for a fixed number of iterations
n_iterations = 10

alpha_list = [0.5, 1, 2]
lambda_list = [0.005, 0.01, 0.05]

fig, axes = plt.subplots(3, 3, figsize=(12, 12))
ax = axes.ravel()

for i, alpha in enumerate(alpha_list):
    for j, lambda_ in enumerate(lambda_list):
        x_denoised = denoisingHeatDiffusion(noisy_image, n_iterations, alpha
        ax[i*3+j].imshow(x_denoised, cmap=plt.cm.gray)
        ax[i*3+j].set_title("alpha={}, lambda={}".format(alpha, lambda_))
```



In [35]:
```
# Vary number of iterations

n_iterations_list = [10, 20, 30]
lambda_ = 0.01
alpha = 1

fig, axes = plt.subplots(1, 3, figsize=(12, 4))
ax = axes.ravel()

for i, n_iterations in enumerate(n_iterations_list):
    x_denoised = denoisingHeatDiffusion(noisy_image, n_iterations, alpha, la
    ax[i].imshow(x_denoised, cmap=plt.cm.gray)
    ax[i].set_title("Iterations={}".format(n_iterations))
```



We can clearly see in both plots that the last row is more diffused than the others. This is due to the fact that a higher number of iterations or a higher λ increases the diffusion process.

In [36]:
```
def psnr(u, v):
    return skimage.metrics.peak_signal_noise_ratio(u, v, data_range=u.max()
```

Part III: TV regularization

Derive and implement the diffusion equation associated with the minimization of the regularisation energy $\int_\Omega \sqrt{\epsilon + \|\nabla u(p)\|^2} dp$. We recall that for an energy $\int_\Omega \rho(\|\nabla u(p)\|^2) dp$ the associated diffusion equation is given by:

$$\frac{\partial u}{\partial t} = div\left(\rho'(\|\nabla u(p)\|)\frac{\nabla u(p)}{\|\nabla u(p)\|}\right)$$

Compare the numerical schemes derived from (i) the discretized version of the analytical expression of the diffusion operator, and (ii) the discretized version of the divergence operator applied to $\rho'(\|\nabla u(p)\|)\frac{\nabla u(p)}{\|\nabla u(p)\|}$.

By taking the previous equation and doing some algebra, we obtain the following result:

$$\frac{\partial \mu}{\partial t} = \operatorname{div}\left(\rho'(\|\nabla\mu(p)\|)\frac{\nabla\mu(p)}{\|\nabla\mu(p)\|}\right)$$

$$= \operatorname{div}\left(\frac{\|\nabla\mu(p)\|}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\nabla\mu(\rho)}{\|\nabla\mu(p)\|}\right)$$

$$= \operatorname{div}\left(\frac{\nabla\mu(p)}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\right)$$

$$= \frac{\partial}{\partial x}\left(\frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\partial\mu}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\partial\mu}{\partial y}\right)$$

$$\Rightarrow \frac{\partial}{\partial x}\left(\frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\partial\mu}{\partial x}\right) = \frac{\partial}{\partial x}\left(\frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\right)\cdot\frac{\partial\mu}{\partial x} + \left(\frac{1}{\sqrt{\varepsilon+\|\nabla\mu(}\right.$$

$$\left.\frac{\partial}{\partial x}\left(\frac{1}{\sqrt{\varepsilon+\left(\frac{\partial\mu}{\partial x}\right)^2+\left(\frac{\partial\mu}{\partial y}\right)^2}}\right)\frac{\partial\mu}{\partial x} + \frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\partial^2\mu}{\partial x^2}\right)$$

$$= \frac{\partial\mu}{\partial x}\cdot\frac{1}{(\varepsilon+\|\nabla\mu(p)\|^2)^{3/2}}\cdot\left(\frac{\partial^2\mu}{\partial x^2}+\frac{\partial^2\mu}{\partial x\partial y}\right) - \frac{1}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}}\frac{\partial^2\mu}{\partial x^2}$$

$$\Rightarrow \frac{\partial\mu}{\partial t} = \frac{\Delta\mu}{\sqrt{\varepsilon+\|\nabla\mu(p)\|^2}} - \frac{1}{(\varepsilon+\|\nabla\mu(p)\|^2)^{3/2}}\left(\frac{\partial\mu}{\partial x}\cdot\left(\frac{\partial^2\mu}{\partial x^2}+\frac{\partial^2\mu}{\partial x\partial y}\right)+\frac{\partial\mu}{\partial y}\cdot\right.$$

In [65]:
```
def TVregularization(u,N,lambd,epsilon):

    uu = u.copy()
    for i in range(N):
        gy, gx = np.gradient(uu)
        gxy, gxx = np.gradient(gx)
        gyy, _ = np.gradient(gy)
        gu_abs2 = gx**2 + gy**2

        F = (gxx**2+gyy**2)/np.sqrt(epsilon+gu_abs2) -1/(epsilon+gu_abs2)**(

        uu = uu + lambd*F

    uu = (uu - np.min(uu))/(np.max(uu) - np.min(uu))

    return uu
```

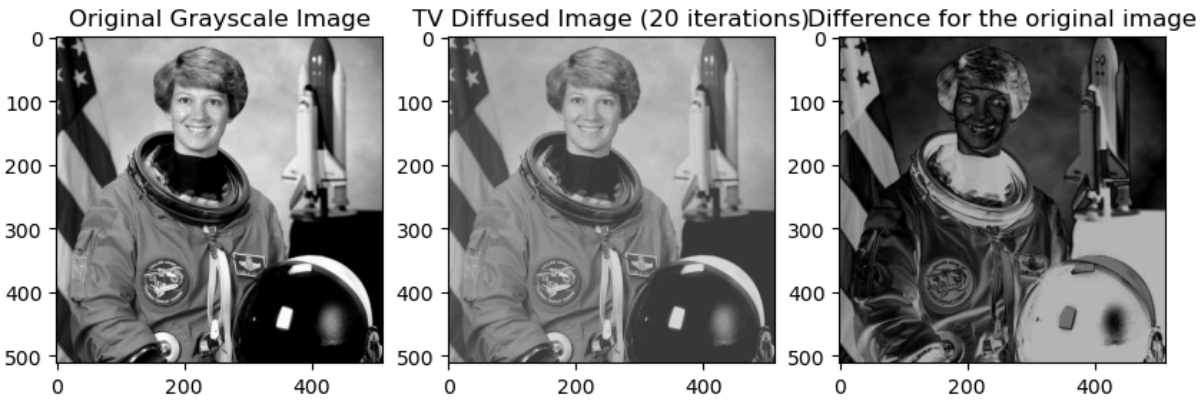
Run regularization experiments and evaluate the quality of the denoising process w.r.t. the main parameters of the regularization scheme

In [66]:
```
N = 20
l = 0.01
```

```
epsilon = 0.2
diffused_tv_original = TVregularization(grayscale_image, N, l, epsilon)
```

In [67]:
```
import matplotlib.pyplot as plt

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

# Original grayscale image
plt.subplot(2, 3, 1)
plt.imshow(grayscale_image, cmap='gray')
plt.title('Original Grayscale Image')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 2)
plt.imshow(diffused_tv_original, cmap='gray')
plt.title('TV Diffused Image (20 iterations)')


diff_orig = np.abs(diffused_tv_original - grayscale_image)
plt.subplot(2, 3, 3)
plt.imshow(diff_orig, cmap='gray')
plt.title('Difference for the original image')
```

Out[67]:
```
Text(0.5, 1.0, 'Difference for the original image')
```



We can see from the difference of the images that edges of the image are preserved in contrast to the homogeneous regions. This is due to the anisotropy of the image. In addition, the small details of the image are preserved, such as the rocket in the background.

Part IV: TV denoising

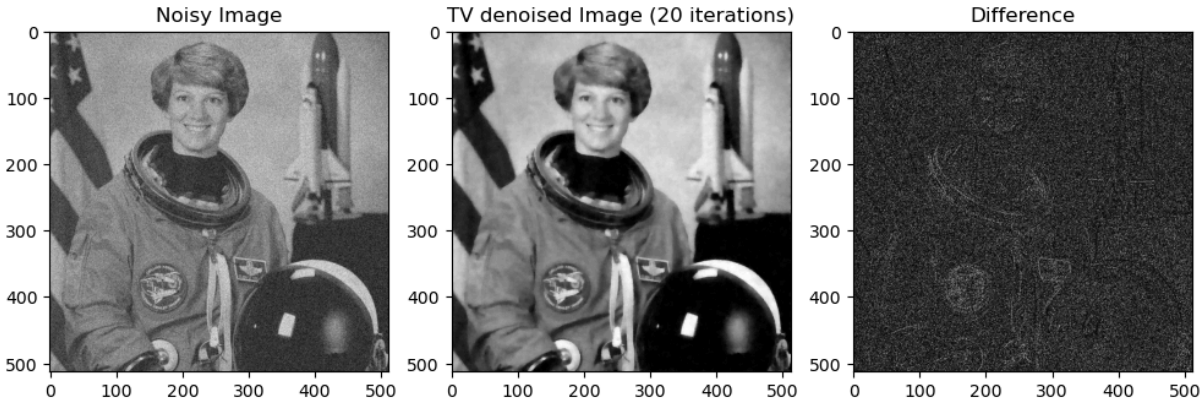
Implement and benchmark TV denoising schemes, including a comparison to the Laplacian diffusion process.

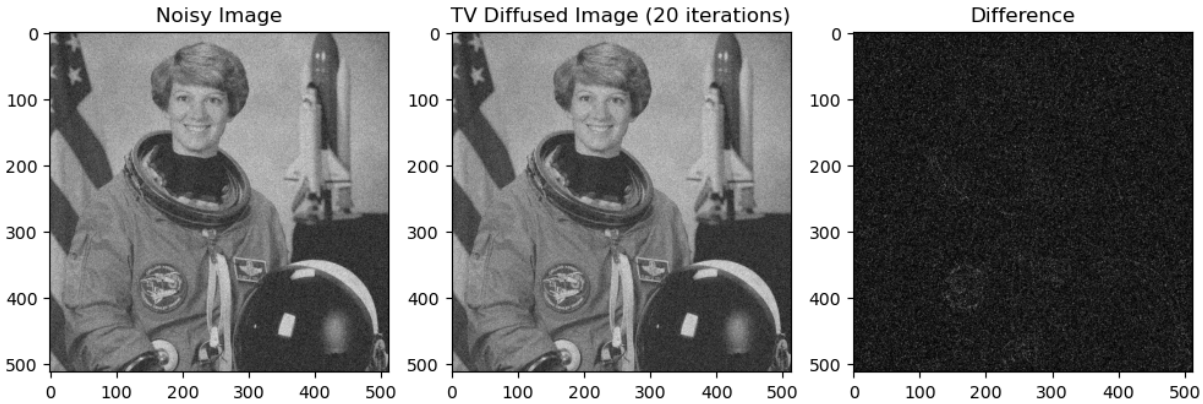
Illustrate through one or two examples the key properties of the diffusion associated with the TV denoising

In [72]:
```
def denoisingTV(v, N, lambda_, alpha, epsilon):

    g = v.copy()
    if N > 0:
        delta_t = 1 / N
    for n in range(N):
        gy, gx = np.gradient(g)
        grad_u_abs2 = np.square(gy) + np.square(gx)

        _, gxx = np.gradient( gx / np.sqrt( epsilon + grad_u_abs2))
        gyy, _ = np.gradient( gy / np.sqrt( epsilon + grad_u_abs2))

        g = g - delta_t * lambda_ * ( 2 * (g-v) - alpha * (gxx + gyy))

    g[g > 1.0] = 1.0
    g[g < 0] = 0
    return g
```

In [94]:
```
N = 20
l = 0.4
alpha = 0.6
epsilon = 0.001
denoised_TV_image = denoisingTV(noisy_image, N, l, alpha, epsilon)
N = 20
l = 0.005
epsilon = 0.2
alpha = 0.1
denoised_heat_eq = denoisingHeatDiffusion(noisy_image, N, alpha, l)

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

# Original grayscale image
plt.subplot(2, 3, 1)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 2)
plt.imshow(denoised_TV_image, cmap='gray')
plt.title('TV denoised Image ({} iterations)'.format(N))

# TV diffused image after 10 iterations
plt.subplot(2, 3, 3)
plt.imshow(np.abs(noisy_image - denoised_TV_image), cmap='gray')
plt.title('Difference')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 4)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 5)
plt.imshow(denoised_heat_eq, cmap='gray')
plt.title('TV Diffused Image ({} iterations)'.format(N))
```

```
# TV diffused image after 10 iterations
plt.subplot(2, 3, 6)
plt.imshow(np.abs(noisy_image - denoised_heat_eq), cmap='gray')
plt.title('Difference')

denoisingHeatDiffusion(noisy_image, n_iterations, alpha, lambda_)

plt.tight_layout()
plt.show()
```

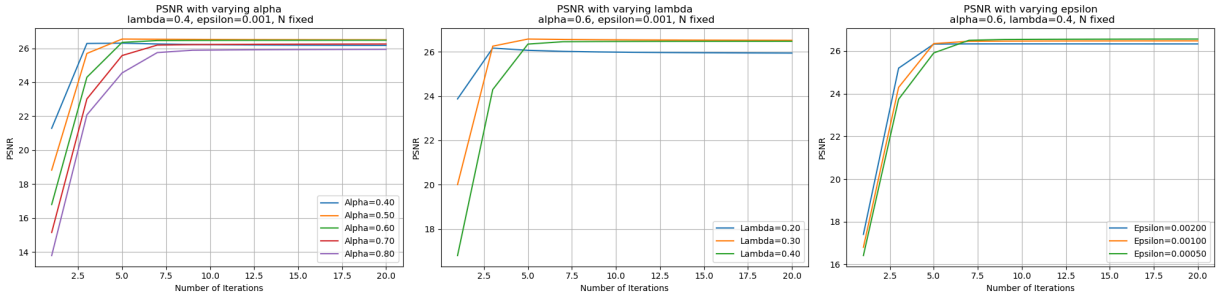




As can be observed in the images, the heat equation generates a diffusion in an isotropic way, meaning that the difference between the denoised and noisy images appears to be quite random. Meanwhile, the TV regularization shows a lower denoising effect near edges, which is clearly observed in the difference between both images.

In [96]:
```
N = 20
l = 0.4
alpha = 0.6
epsilon = 0.001

N_range = np.linspace(1, 20, num=10, dtype=int)
alphas = [0.4, 0.5, 0.6, 0.7, 0.8]  # Three values for alpha
lambdas = [0.2, 0.3, 0.4]  # Three values for lambda
epsilons = [0.002, 0.001, 0.0005]  # Three values for epsilon
```

```
plt.figure(figsize=(20, 5))

# Vary alpha
plt.subplot(1, 3, 1)
for a, alpha_i in enumerate(alphas):
    PSNR_alpha = []
    for N_iter in N_range:
        denoised_image = denoisingTV(noisy_image, N_iter, l, alpha_i, epsilo
        PSNR_alpha.append(psnr(denoised_image, grayscale_image))

    plt.plot(N_range, PSNR_alpha, label="Alpha={:.2f}".format(alpha_i))

plt.title("PSNR with varying alpha\n lambda={}, epsilon={}, N fixed".format(
plt.xlabel("Number of Iterations")
plt.ylabel("PSNR")
plt.legend()
plt.grid()

# Vary lambda
plt.subplot(1, 3, 2)
for i, lambd_i in enumerate(lambdas):
    PSNR_lambda = []
    for N_iter in N_range:
        denoised_image = denoisingTV(noisy_image, N_iter, lambd_i, alpha, ep
        PSNR_lambda.append(psnr(denoised_image, grayscale_image))

    plt.plot(N_range, PSNR_lambda, label="Lambda={:.2f}".format(lambd_i))

plt.title("PSNR with varying lambda\n alpha={}, epsilon={}, N fixed".format(
plt.xlabel("Number of Iterations")
plt.ylabel("PSNR")
plt.legend()
plt.grid()

# Vary epsilon
plt.subplot(1, 3, 3)
for e, eps_i in enumerate(epsilons):
    PSNR_epsilon = []
    for N_iter in N_range:
        denoised_image = denoisingTV(noisy_image, N_iter, l, alpha, eps_i)
        PSNR_epsilon.append(psnr(denoised_image, grayscale_image))

    plt.plot(N_range, PSNR_epsilon, label="Epsilon={:.5f}".format(eps_i))

plt.title("PSNR with varying epsilon\n alpha={}, lambda={}, N fixed".format(
plt.xlabel("Number of Iterations")
plt.ylabel("PSNR")
plt.legend()
plt.grid()


plt.tight_layout()
plt.show()
```



The PSNR increases as the iterations of the algorithm run, which is expected. Also, we note:

- At lambda = 0.3 the converged value of the PSNR is higher than at lambda equal to 0.2 and 0.4
- The epsilon value doesn't seem to affect significantly the converged value of the PSNR, but it does affect the convergence speed.
- At Alpha equal to 0.6 the converged PSNR seems to be maximal (up to a resolution of 0.1). Also, too low values of alpha can make the PSNR actually decrease, which of course is to be avoided.

Part V: Perona-Malik diffusion (Bonus)

Implement and benchmark diffusion schemes defined as:

$$\frac{\partial u}{\partial t} = div\left(c(\|\nabla u(p)\|)\nabla u(p)\right)$$

with different choices for function $c()$: $c(x) = 1 - exp(-x^2/K^2)$,
$c(x) = \left[1 + x^2/K^2\right]^{-1/2}$, $c(x) = \left[1 + x^2/K^2\right]^{-1}$.

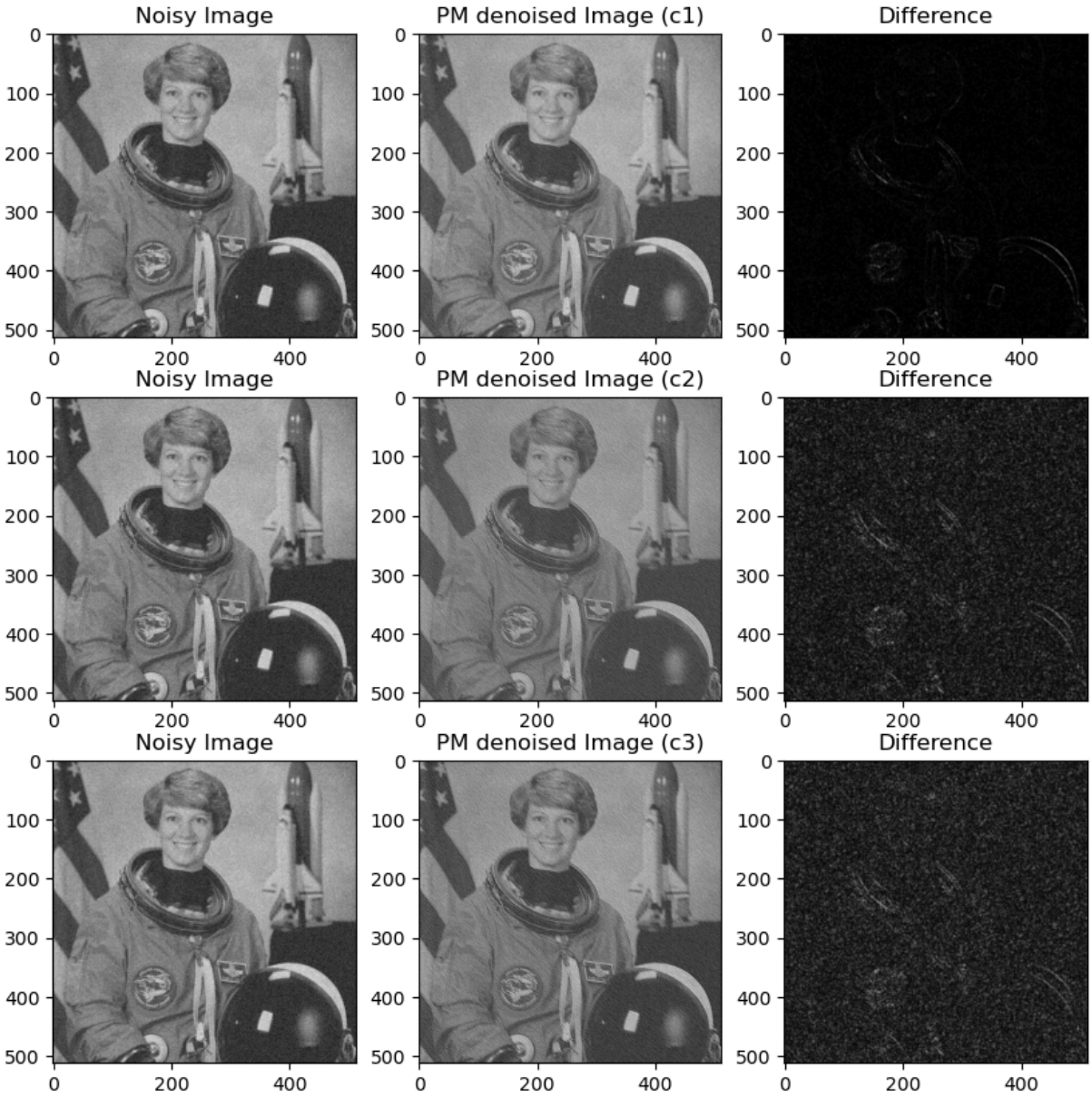
In [43]:
```
def c1(x, k=1):
    return 1 - np.exp(-x/(k*k))

def c2(x, k=1):
    return np.reciprocal(np.sqrt(1 + x / (k*k)))

def c3(x, k=1):
    return np.reciprocal(1 + x / (k*k))

def PeronaMalikDiffussion(u, N, lambda_, function_=(None,1)):

    for i in range(N):
        # Compute the gradient
        gy, gx = np.gradient(u)
        gu_abs = np.sqrt(gx**2 + gy**2)

        # Compute the diffusion function
        c = function_(gu_abs, k=1)
```

```
        # Update the image
        u = u + lambda_ * (np.gradient(c*gx)[0] + np.gradient(c*gy)[1])

    return u
```

In [97]:

```
N1 = 10
lambda_1 = 0.1
N2 = 10
lambda_2 = 0.1
N3 = 10
lambda_3 = 0.1
denoised_PM_image_c1 = PeronaMalikDiffussion(noisy_image, N1, lambda_1, c1)
denoised_PM_image_c2 = PeronaMalikDiffussion(noisy_image, N2, lambda_2, c2)
denoised_PM_image_c3 = PeronaMalikDiffussion(noisy_image, N3, lambda_3, c3)

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

# Original grayscale image
plt.subplot(3, 3, 1)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# PM diffused image after 20 iterations
plt.subplot(3, 3, 2)
plt.imshow(denoised_PM_image_c1, cmap='gray')
plt.title('PM denoised Image (c1)'.format(N1, lambda_1))

# Difference
plt.subplot(3, 3, 3)
plt.imshow(np.abs(noisy_image - denoised_PM_image_c1), cmap='gray')
plt.title('Difference')

# Original grayscale image
plt.subplot(3, 3, 4)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# PM diffused image after 20 iterations
plt.subplot(3, 3, 5)
plt.imshow(denoised_PM_image_c2, cmap='gray')
plt.title('PM denoised Image (c2)'.format(N2, lambda_2))

# Difference
plt.subplot(3, 3, 6)
plt.imshow(np.abs(noisy_image - denoised_PM_image_c2), cmap='gray')
plt.title('Difference')

# Original grayscale image
plt.subplot(3, 3, 7)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# PM diffused image after 20 iterations
plt.subplot(3, 3, 8)
plt.imshow(denoised_PM_image_c3, cmap='gray')
plt.title('PM denoised Image (c3)'.format(N3, lambda_3))
```

```
# Difference
plt.subplot(3, 3, 9)
plt.imshow(np.abs(noisy_image - denoised_PM_image_c3), cmap='gray')
plt.title('Difference')

fig.tight_layout()
plt.show()
```



Part VI: MRF-based image denoising

This section aims to implement and test Markov Random Field (MRF) algorithms for image denoising. Based on discretized version of variational energy:

$$\int_{\Omega} \|u(p) - v(p)\|^2 dp + \alpha \int_{\Omega} \|\nabla u(p)\|^2 dp$$

derive the observation lieklihood model $P(V|U)$ and the Markovian prior $P(U)$ of the associated MRF formulation. Derive an iterative denoising algorithm based on the maximization of the following posterior likelihood:

$$P(U_p|V, U_q,\ q \neq p)$$

Implement this algorithm and compare to the Laplacian diffusion process.

$$J(u) = \frac{1}{2}\sum_{i,j}(u_{i,j} - v_{i,j})^2 + \frac{\alpha}{2}\sum_{i,j}\|\nabla u_{i,j}\|^2$$

When assuming a gaussian distribution in a MRF, we have $V|U \sim \mathcal{N}(U, \sigma^2)$, because of the assumtion $V = U + N$ where $N \sim \mathcal{N}(0, \sigma^2)$. In this bayesian framework, we can also define, for a given β,

$$\mathcal{P}(U) = \frac{1}{Z_\beta}e^{-\alpha\ TV(U)} = \frac{1}{Z_\beta}\exp\left(-\alpha\sum_{i,j}|\nabla u_{ij}|^2\right)$$

$$\mathcal{L}(U) = \frac{1}{Z}\mathcal{P}(V|U) = \frac{1}{Z}\exp\left(-\frac{1}{2}\sum_{i,j}\left(\frac{(v_{ij} - u_{ij})^2}{\sigma^2}\right)\right)$$

For the algorithm, it would be possible to solve for

$$\hat{u_{i,j}} = \arg\max_{u \in N(u_{i,j})} \mathcal{P}(U|V)$$

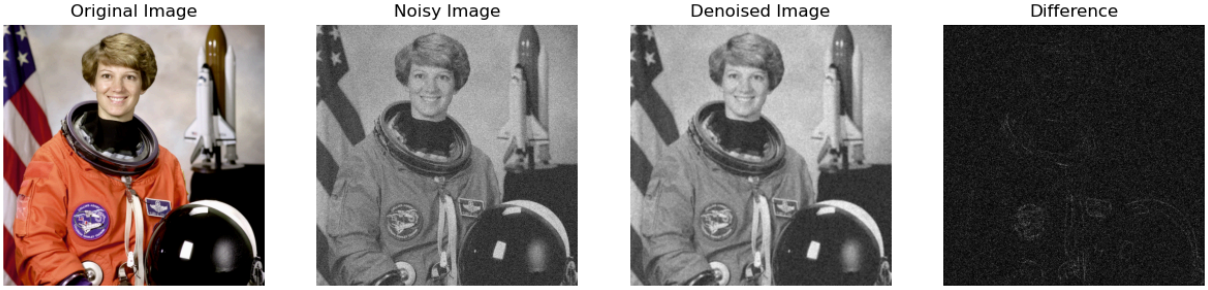
where

$$\mathcal{P}(U|V) \propto \mathcal{P}(V|U)\ \mathcal{P}(U)$$

However, instead of takin the maximum element in the neighborhood of $u_{i,j}$, we will choose in this case to weight all neighbors according to the posteriori distribution $\mathcal{P}(U|V)$. Then, each pixel value in the denoised image is updated using a weighted sum of the noisy pixel values within the local neighborhood. In a similar fashion to the ICM (Iterative Conditional Mode) algorithm, this algorithm will be deterministic.

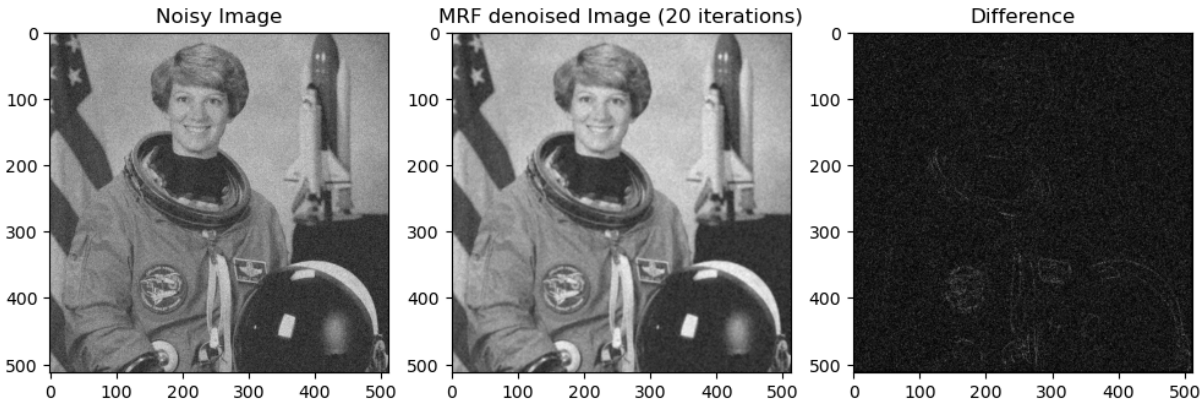
In [45]:
```python
def gaussian_likelihood(image, noisy_image, sigma):
    likelihood = np.exp(-0.5 * np.square(image - noisy_image) / sigma**2)
    return likelihood

def gaussian_prior(image, alpha):
    grad_x, grad_y = np.gradient(image)
    prior = np.exp(-0.5 * (np.square(grad_x) + np.square(grad_y)) * alpha)
    return prior

def denoise_MRF(noisy_image, alpha, sigma, iterations=100):
    denoised_image = np.copy(noisy_image)
    height, width = noisy_image.shape
```

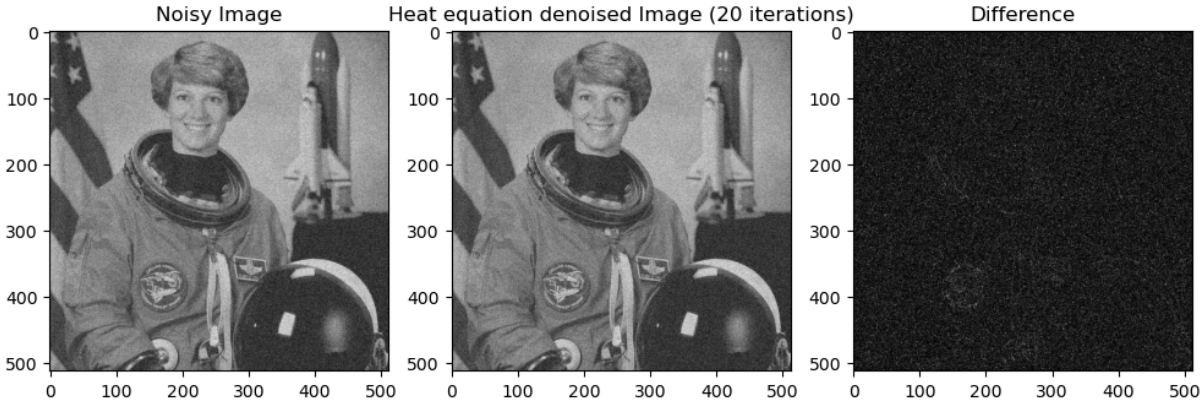
```
    for _ in range(iterations):
        for j in range(1, height - 1):
            for i in range(1, width - 1):

                likelihood = gaussian_likelihood(denoised_image[j, i], noisy
                prior = gaussian_prior(denoised_image[j-1:j+2, i-1:i+2], alp
                posterior = likelihood * prior
                posterior /= np.sum(posterior)

                # Update pixel value
                denoised_image[j, i] = np.sum(posterior * noisy_image[j-1:j+

    return denoised_image

# Denoise using MRF algorithm
alpha = 0.1
sigma = 0.1 # Using the same sigma as the one used for the noise generation
N = 3
x_denoised = denoise_MRF(noisy_image, alpha, sigma, N)

# Plot results
plt.figure(figsize=(15, 5))
plt.subplot(1, 4, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')
plt.axis('off')

plt.subplot(1, 4, 3)
plt.imshow(x_denoised, cmap='gray')
plt.title('Denoised Image')
plt.axis('off')

plt.subplot(1, 4, 4)
plt.imshow(np.abs(x_denoised - noisy_image), cmap='gray')
plt.title('Difference')
plt.axis('off')

plt.show()
```



In [50]:

```
N = 20
l = 0.005
epsilon = 0.2
alpha = 0.1
denoised_heat_eq = denoisingHeatDiffusion(noisy_image, N, alpha, l)

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

# Original grayscale image
plt.subplot(2, 3, 1)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 2)
plt.imshow(x_denoised, cmap='gray')
plt.title('MRF denoised Image ({} iterations)'.format(N))

# TV diffused image after 10 iterations
plt.subplot(2, 3, 3)
plt.imshow(np.abs(noisy_image - x_denoised), cmap='gray')
plt.title('Difference')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 4)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Image')

# TV diffused image after 10 iterations
plt.subplot(2, 3, 5)
plt.imshow(denoised_heat_eq, cmap='gray')
plt.title('Heat equation denoised Image ({} iterations)'.format(N))

# TV diffused image after 10 iterations
plt.subplot(2, 3, 6)
plt.imshow(np.abs(noisy_image - denoised_heat_eq), cmap='gray')
plt.title('Difference')

denoisingHeatDiffusion(noisy_image, n_iterations, alpha, lambda_)

plt.tight_layout()
plt.show()
```





As can be observed, the MRF is also efficient at denoising while preserving the main features of the image. In this case, the α parameter defines the smoothness of the denoised image, higher values being more smooth.

Bonus: Using a m-estimator instead of the quadratic prior (cf. https://hal.inria.fr/inria-00350297/document) and an reweigthed iterative least-square algorithm, implement a generalization of the above MRF formulation which accounts for a more appropriate prior than a Gaussian prior.

```
`# This is formatted as code`
```

Part VII: Variational minimization using autograd tools in pytorch

Link to autograd tutorial:
https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

In [ ]:
```
def TVregularization(u,N,lambd,epsilon):
```

```
    uu = u.copy()
    for i in range(N):
        gy, gx = np.gradient(uu)
        gxy, gxx = np.gradient(gx)
        gyy, _ = np.gradient(gy)
        gu_abs2 = gx**2 + gy**2

        F = (gxx**2+gyy**2)/np.sqrt(epsilon+gu_abs2) -1/(epsilon+gu_abs2)**(

        uu = uu + lambd*F

    uu = (uu - np.min(uu))/(np.max(uu) - np.min(uu))

    return uu
```

In [ ]:
```
import torch

def variational_cost(u_tensor):

    sobel_x = torch.Tensor(np.array([1,0,-1,2,0,-2,1,0,-1]).reshape(3,3))
    sobel_y = torch.Tensor(np.array([1,2,1,0,0,0,-1,-2,-1]).reshape(3,3))

    sobel_x = sobel_x.view(1,1,3,3)
    sobel_y = sobel_y.view(1,1,3,3)

    gx  = torch.nn.functional.conv2d(u_tensor, sobel_x.type(torch.DoubleTens
    gy  = torch.nn.functional.conv2d(u_tensor, sobel_y.type(torch.DoubleTens

    grad     = torch.pow(gx,2) + torch.pow(gy,2)
    var_cost = grad.sum()

    return var_cost
```

In [ ]:
```
## implement a gradient descent algorithm from the automatic differentiation
def gradient_descent(u,v, N, alpha, lambd):

    uf = v.copy()
    delta = 1/N
    loss = []

    # Perform the gradient descent optimization
    for i in range(N):
        uf = torch.from_numpy(uf)
        uf = uf.view(1, 1, v.shape[0], v.shape[1]).requires_grad_(True)

        cost = variational_cost(uf)
        cost.backward()
        grad = uf.grad
        grad = grad.view(v.shape[0],v.shape[1])

        loss.append(cost.item())

        uf = uf.view(v.shape[0], v.shape[1]).detach().numpy()
        grad = grad.detach().numpy()
        uf = uf - delta * lambd * (2 * (uf-v) + alpha * grad)
```

```
    return uf, loss
```

In [ ]:
```
## Check the difference between the gradient computed using autograd vs. the

n_iter = 10
alpha = 0.1
lambda_ = 0.05

denoised_analytic = denoisingHeatDiffusion(noisy_image,n_iter,alpha,lambda_)
denoised_autograd,loss = gradient_descent(grayscale_image,noisy_image,n_iter

denoised_autograd = np.clip(denoised_autograd, 0,1)

print("MSE between the analytical calculation and autograd: ", mean_squared_

fig, axs = plt.subplots(1,2, figsize=(14,8))
axs[0].imshow(denoised_analytic, cmap='gray')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[0].set_title("Denoising computed normally")
axs[1].imshow(denoised_autograd, cmap='gray')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[1].set_title("Denoising computed with Autograd")
plt.show()
```

```
MSE between the analytical calculation and autograd:  0.0028577186649803997
```