In [1]:

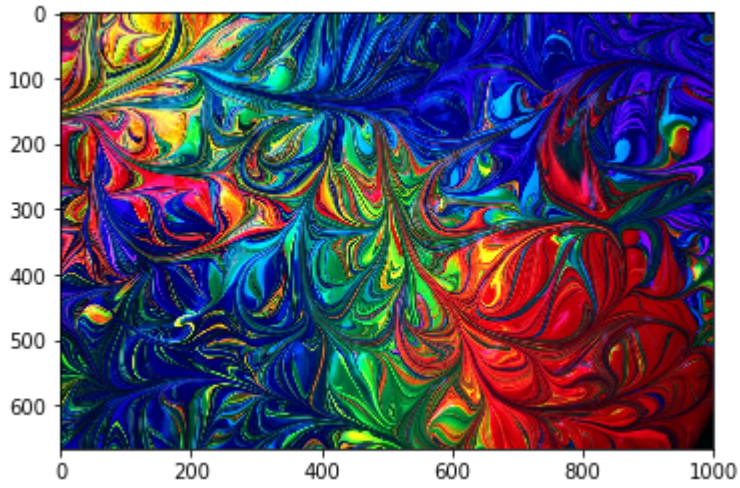
```
#Importações gerais
#import cv2
from matplotlib import pyplot as plt
import numpy as np
from PIL import Image
from statistics import mean, median, mode, stdev
```

1. Desenvolva um sistema para abrir, exibir, manipular e salvar imagens RGB com 24 bits/pixel (8 bits/componente/pixel). Não use bibliotecas ou funções especiais de processamento de imagens. Para os itens 1.3 a 1.8, duas formas de aplicação devem ser testadas: em RGB (banda a banda) e na banda Y, com posterior conversão para RGB.

In [2]:

```
#Carregando a imagem na memória, exemplo:
def carregaRGB(img_file):
    imgPIL = Image.open(img_file)
    pic = np.asarray(imgPIL)
    return pic

img = carregaRGB("rainbow.png")
print("Bits para cada unsigned int de RGB:", img.dtype)
plt.imshow(img)
plt.show()
```

```
Bits para cada unsigned int de RGB: uint8
```

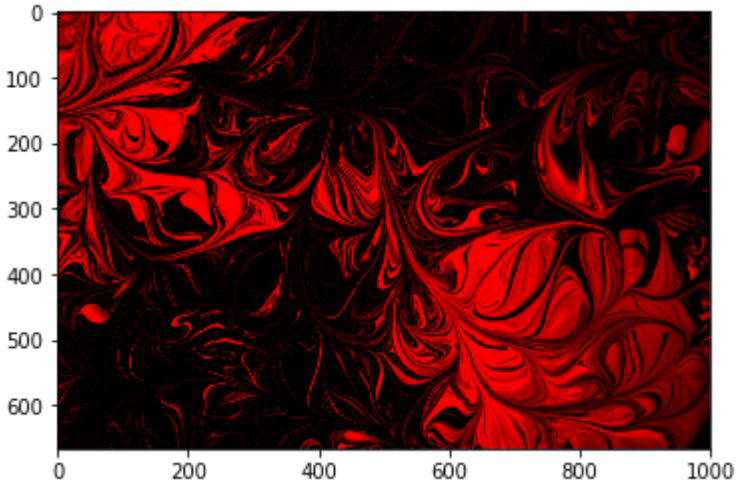


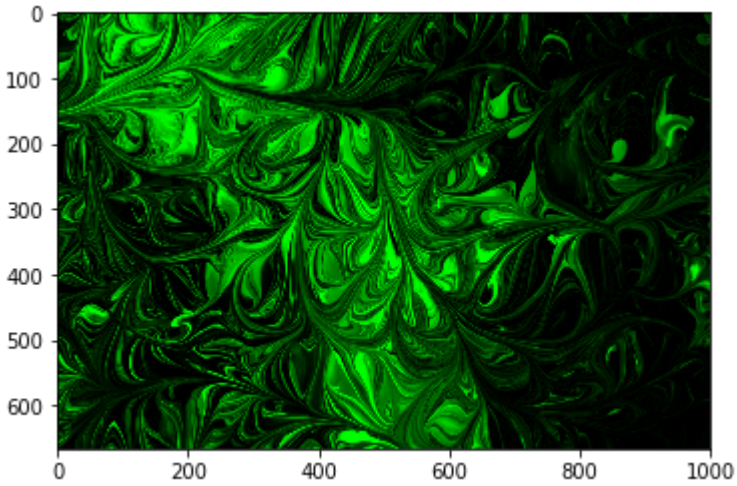
1.2. Exibição de bandas individuais (R, G e B) como imagens monocromáticas ou coloridas (em tons de R, G ou B, respectivamente)(Xerxes)

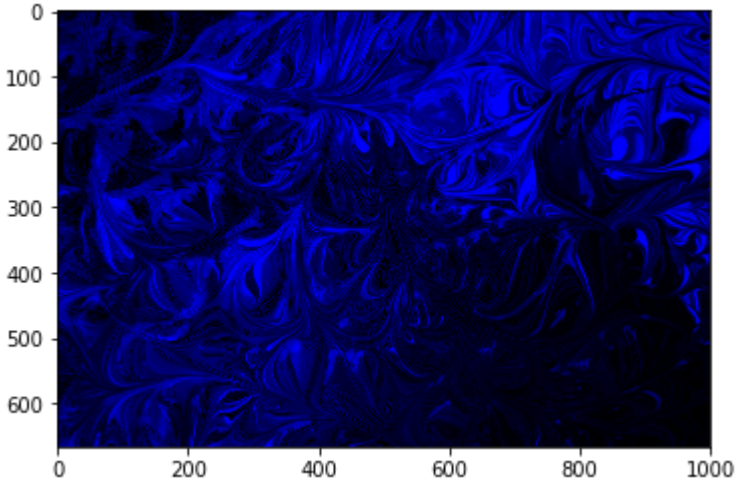
In [3]:

```
def separateRGB(img):
    #print('Iniciando separação de imagem...')
    img = Image.fromarray(np.uint8(img))
    #Pegamos cada 'Valor' em pixels
    data = img.getdata()
    #print('Data:', data)

    #Vamos então suprimir cada uma das bandas((255, 120, 65) -> (0, 120, 0) para G)
    r = [(d[0], 0, 0) for d in data]#Mantemos o valor de Vermelho e zeramos os valores
 de Verde e Azul

    g = [(0, d[1], 0) for d in data]#Mantemos o valor de Verde e zeramos os valores de
 Vermelho e Azul

    b = [(0, 0, d[2]) for d in data]#Mantemos o valor de Azul e zeramos os valores de V
ermelho e Verde

    #Salva imagens em R, G e B
    img.putdata(r)#Mudamos apenas os valores para R
    imgRed = img.copy()

    img.putdata(g)#Mudamos apenas os valores para G
    imgGreen = img.copy()

    img.putdata(b)#Mudamos apenas os valores para B
    imgBlue = img.copy()

    #print('Conversão Completa!')

    return r, g, b, imgRed, imgGreen, imgBlue

def separateYIQ(img):
    Ychannel = img[:,:,0]
    Ichannel = img[:,:,1]
    Qchannel = img[:,:,2]
    #print('Iniciando separação de imagem...')
    img = Image.fromarray(np.uint8(img))
    #Pegamos cada 'Valor' em pixels
    data = img.getdata()
    #print('Data:', data)

    #Vamos então suprimir cada uma das bandas((255, 120, 65) -> (0, 120, 0) para G)
    y = [(d[0], 0, 0) for d in data]#Mantemos o valor de Vermelho e zeramos os valores
 de Verde e Azul

    i = [(0, d[1], 0) for d in data]#Mantemos o valor de Verde e zeramos os valores de
 Vermelho e Azul

    q = [(0, 0, d[2]) for d in data]#Mantemos o valor de Azul e zeramos os valores de V
ermelho e Verde

    #print('Conversão Completa!')

    return y, i, q, Ychannel, Ichannel, Qchannel

img = carregaRGB("rainbow.png")

r, g, b, imgRed, imgGreen, imgBlue = separateRGB(img)
```

```
plt.imshow(imgRed)
plt.show()

plt.imshow(imgGreen)
plt.show()

plt.imshow(imgBlue)
plt.show()
```







O sistema deve ter as seguintes funcionalidade:

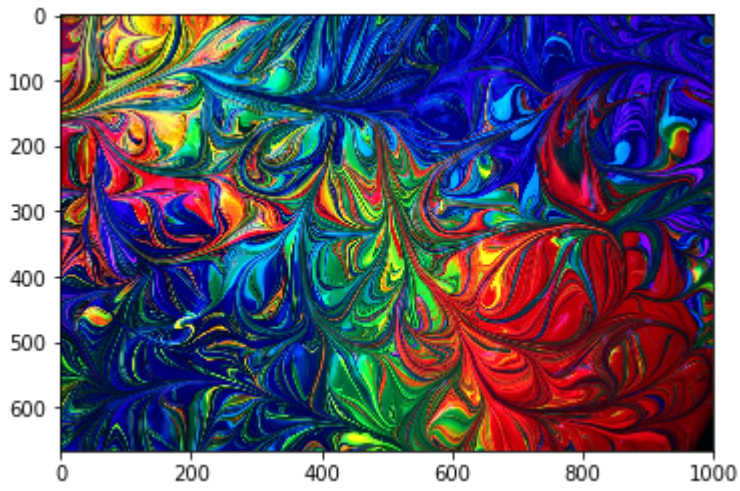
1.1. Conversão RGB-YIQ-RGB (cuidado com os limites de R, G e B na volta!)(Douglas)

In [4]:

```
#Converte uma imagem
'''
Y = 0.299R + 0.587G + 0.114B
I = 0.596R – 0.274G – 0.322B
Q = 0.211R – 0.523G + 0.312B
'''
#Função que realiza o truncamento de pixels para caso e vai arredondar para o teto se x
> 0.5 ou piso se x < 0.5
def trunca(number):
    if (number > 255):
        return 255

    elif(number < 0):
        return 0
    else:
        return int(round(number))

trunca_vetorizado = np.vectorize(trunca)


def converteRGB_YIQ(imgRGB):
    print("Conversão para RGB->YIQ")
    #Primeiro nos pegamos as imagens separadas na função 1.2 e transformamos elas em ar
rays
    r, g, b, imgRed, imgGreen, imgBlue = separateRGB(imgRGB)
    imgRed = np.asarray(imgRed)
    imgGreen = np.asarray(imgGreen)
    imgBlue = np.asarray(imgBlue)
    #Depois pegamos isolamos o canal que queremos usar para os calculos
    RedChannel = imgRed[:,:,0]
    GreenChannel = imgGreen[:,:,1]
    BlueChannel = imgBlue[:,:,2]

    #E aqui aplicamos as formulas em cada canal
    Ychannel = (0.299 * RedChannel) + (0.587 * GreenChannel) + (0.114 * BlueChannel)
    Ichannel = (0.596 * RedChannel) - (0.274 * GreenChannel) - (0.322 * BlueChannel)
    Qchannel = (0.211 * RedChannel) - (0.523 * GreenChannel) + (0.312 * BlueChannel)

    #Por ultimo nos empilhamos novamente os três canais para formar a imagem em YIQ
    imgYIQ = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)
    return imgYIQ

'''
R = 1.000 Y + 0.956 I + 0.621 Q
G = 1.000 Y – 0.272 I – 0.647 Q
B = 1.000 Y – 1.106 I + 1.703 Q
'''
def converteYIQ_RGB(imgYIQ):
    print("Conversão para YIQ->RGB")
    #Como a função que transforma em YIQ já os deixa em array, apenas precisamos isolar
os canais
    Ychannel = imgYIQ[:,:,0]
    Ichannel = imgYIQ[:,:,1]
    Qchannel = imgYIQ[:,:,2]

    #E após isso aplicar as formulas de conversão
    RedChannel = Ychannel + (0.956 * Ichannel) + (0.621 * Qchannel)
    GreenChannel = Ychannel - (0.274 * Ichannel) - (0.647 * Qchannel)
    BlueChannel = Ychannel - (1.106 * Ichannel) + (1.703 * Qchannel)
```

```
    imgRGB = np.stack((RedChannel, GreenChannel, BlueChannel), axis = 2)
    #Função vetorizada para maximizar desempenho aplicando em paralelo em cada pixel
    trunca_vetorizado = np.vectorize(trunca)
    #Fazemos então a normalização da imagem
    imgRGB = trunca_vetorizado(imgRGB)
    return imgRGB



#Carregamos a imagem e fazemos as conversões
img = carregaRGB("rainbow.png")
img = converteRGB_YIQ(img)


img = converteYIQ_RGB(img)
#Ela está pronta novamente para ser plottada
plt.imshow(img)
plt.show()
```

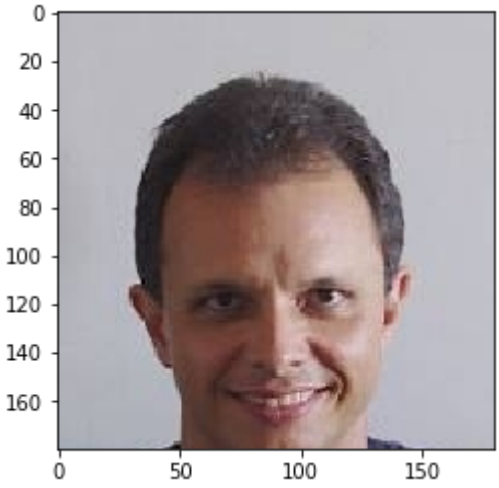
```
Conversão para RGB->YIQ
Conversão para YIQ->RGB
```



1.3. Negativo(Rebeca)

In [5]:

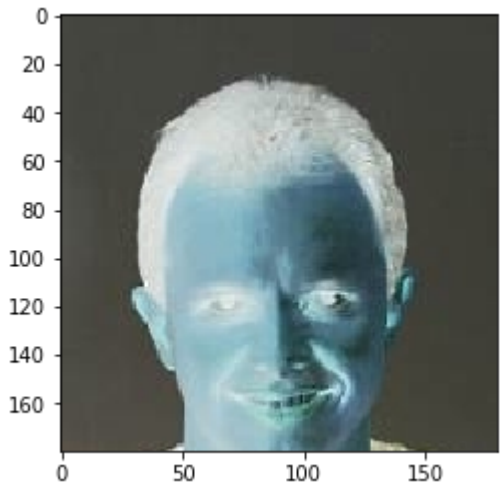
```
#Original
img = carregaRGB("leonardo.png")
plt.imshow(img)
plt.show()
```



In [6]:

```
#Nativo em RGB
#Transforma uma imagem "normal" em uma negativa
def negativeFilter(img_file):

    print('Aplicando filtro negativo...')

    #Transformamos as imagens isoladas R,G,B resultantes do separateRGB para arrays
    r, g, b, imgRed, imgGreen, imgBlue = separateRGB(img_file)
    imgRed = np.asarray(imgRed)
    imgGreen = np.asarray(imgGreen)
    imgBlue = np.asarray(imgBlue)

    #Isolamos os canais de cada cor e calculamos o negativo de cada um
    negativeRed = 255 - imgRed[:,:,0]
    negativeGreen = 255 - imgGreen[:,:,1]
    negativeBlue = 255 - imgBlue[:,:,2]

    #Empilhamos os resultantes dos calculos do negativo
    negativeImg = np.stack((negativeRed, negativeGreen, negativeBlue), axis = 2)
    print("Filtro negativo aplicado com sucesso!")
    return negativeImg

#Carregamos a imagem
img = carregaRGB("leonardo.png")
negative = negativeFilter(img)

#Plottamos e salvamos imagem
plt.imshow(negative)
plt.show()
negative = Image.fromarray(np.uint8(negative), "RGB")
negative.save("leonardoNegative.png")
```

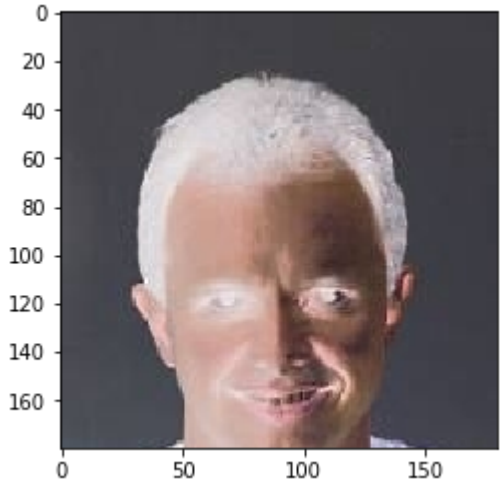
```
Aplicando filtro negativo...
Filtro negativo aplicado com sucesso!
```



In [7]:

```
#Negativo em YIQ
#Transforma uma imagem "normal" em negativa apenas na banda Y
def negativeFilterY(img_file):
    #Converte a imagem RGB em imagem YIQ
    img_file = converteRGB_YIQ(img_file)

    #A imagem já vem em array, só isolamos os canais para fazer o que desejamos
    #Inverte apenas a banda Y
    print("Aplicando negativo na banda Y...")
    negativeYchannel = 255.0 - img_file[:,:,0]
    Ichannel = img_file[:,:,1]
    Qchannel = img_file[:,:,2]

    #Empilhamos as bandas, dessa vez com Ynegativo
    negativeY = np.stack((negativeYchannel, Ichannel, Qchannel), axis = 2)
    print("Filtro negativo em Y aplicado com sucesso...")
    return negativeY

#Carregamos a imagem
img = carregaRGB("leonardo.png")

#Aplicamos o filtro
negativeY = negativeFilterY(img)

#Reconvertemos para RGB e truncamos a imagem
img = converteYIQ_RGB(negativeY)

#Plotamos e salvamos

plt.imshow(img)
plt.show()
negativeY = Image.fromarray(np.uint8(img), "RGB")
negativeY.save("leonardoNegativeY.png")
```

```
Conversão para RGB->YIQ
Aplicando negativo na banda Y...
Filtro negativo em Y aplicado com sucesso...
Conversão para YIQ->RGB
```

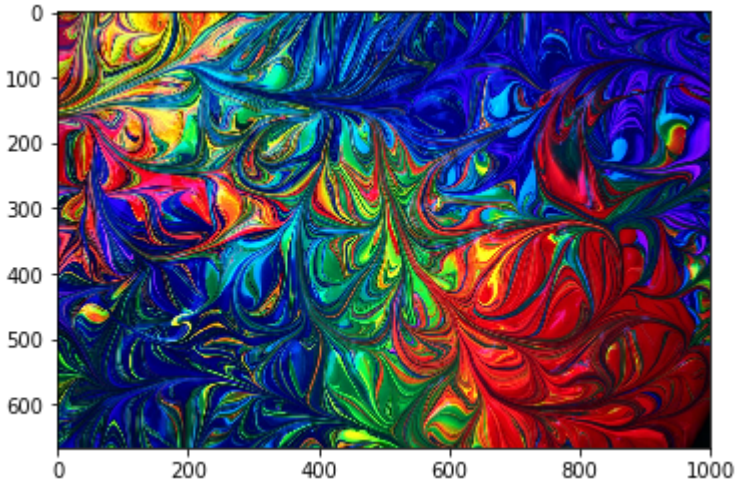


1.4. Controle de brilho aditivo (valor do pixel resultante = valor do pixel original + c, c inteiro) (cuidado com os limites de R, G e B!)(Pedro)

In [9]:

```
#Brilho adivitivo em RGB
#Adiciona um brilho aditivo a partir de uma constante c
def brilhoAditivo(img, c):
    data = img.getdata()

    brilho = [(limite(int(d[0] + c)), limite(int(d[1]) + c), limite(int(d[2]))) for d i
n data]

    return brilho

def limite(n):
    if n > 255:
        n = 255
    elif n < 0:
        n = 0

    return n

img = Image.open("rainbow.png")
print('Antes:')
plt.imshow(img)
plt.show()

c = 50
brilho = np.asarray(brilhoAditivo(img))
print('Depois:')
plt.imshow(brilho.reshape(667, 1000, 3))
plt.show()
```

Antes:

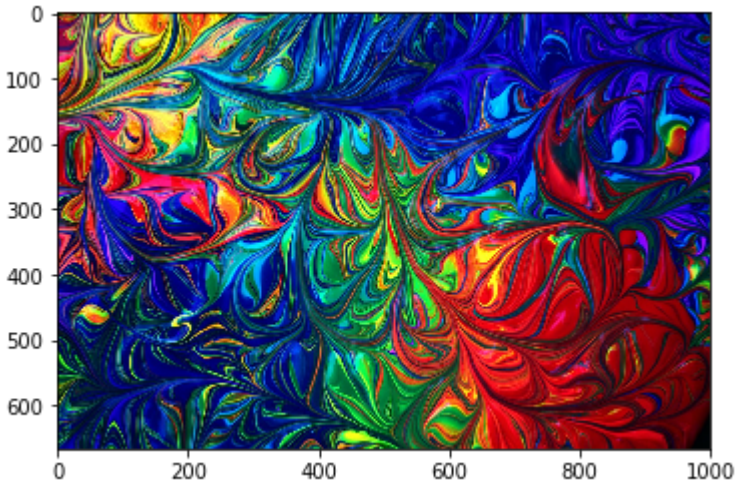



```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-9-553001c85cbc> in <module>
     22
     23 c = 50
---> 24 brilho = np.asarray(brilhoAditivo(img))
     25 print('Depois:')
     26 plt.imshow(brilho.reshape(667, 1000, 3))

TypeError: brilhoAditivo() missing 1 required positional argument: 'c'
```

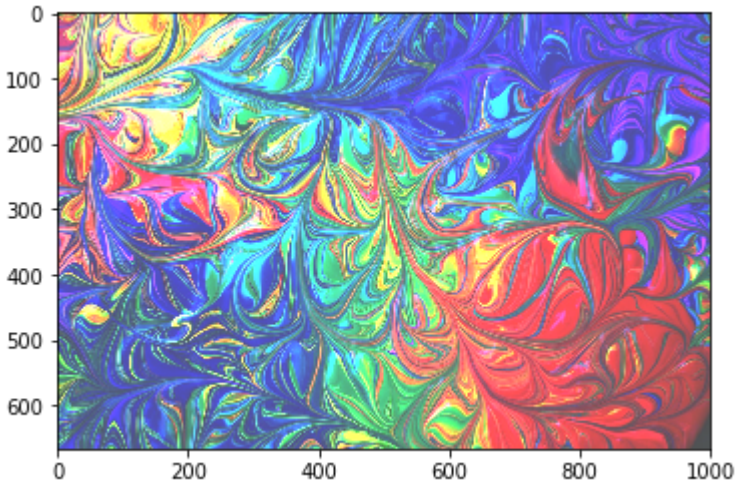
In [9]:

```
#Brilho aditivivo em YIQ
#Adiciona um brilho aditivo a partir de uma constante c
def brilhoAditivoYIQ(img_YIQ, c):
    #Funções auxiliar de soma
    def soma(x, c):
        return x + c
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img_YIQ)

    sum_vectorized = np.vectorize(soma)

    Ychannel = sum_vectorized(Ychannel, c)

    img = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)

    return img

img = Image.open("rainbow.png")
print('Antes:')
plt.imshow(img)
plt.show()


print('Depois:')
img = converteRGB_YIQ(img)

print("Brilho aditivo apenas na banda Y..")
img = brilhoAditivoYIQ(img, 80)

img = converteYIQ_RGB(img)

plt.imshow(img)
plt.show()
```

```
Antes:
```



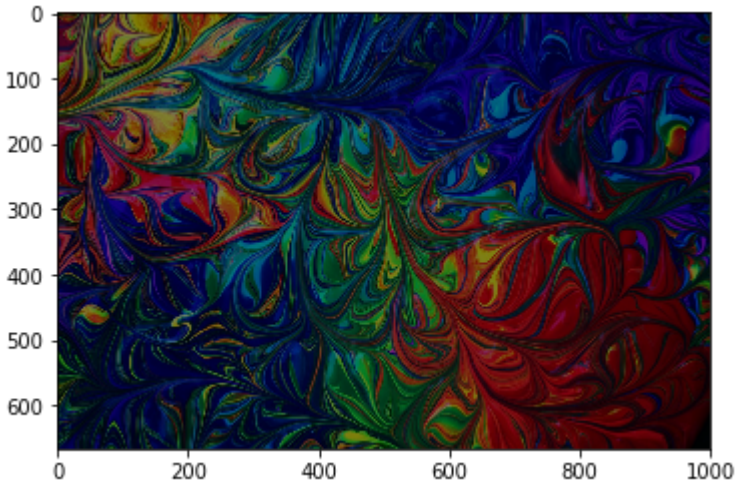

```
Depois:
Conversão para RGB->YIQ
Brilho aditivo apenas na banda Y..
Conversão para YIQ->RGB
```



1.5. Controle de brilho multiplicativo (valor do pixel resultante = valor do pixel original x c, c real não negativo) (cuidado com os limites de R, G e B!)(Douglas)

In [10]:

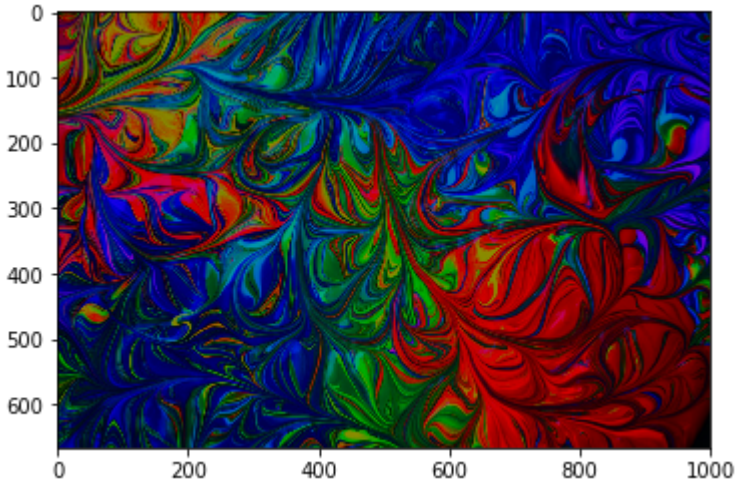
```
#BrilhoMult em RGB
def brilhoMult(img, c):
    #Funções auxiliar de multiplicação
    def Multiply(x, c):
        return x * c
    #Função auxiliar de truncamento e arredondamento
    def trunca(number):
        if (number > 255):
            return 255
        elif(number < 0):
            return 0
        else:
            return int(round(number))
    #Primeiro a gente cria o modelo de vetorização para as funções auxiliares
    multiply_vectorized = np.vectorize(Multiply)
    trunca_vectorized = np.vectorize(trunca)
    #Aplicamos
    img = trunca_vectorized(multiply_vectorized(img, c))
    return img

img = carregaRGB("rainbow.png")
img = brilhoMult(img, 0.5)

plt.imshow(img)
plt.show()
```



In [11]:

```
#BrilhoMult em YIQ

def brilhoMultYIQ(img_YIQ, c):
    #Funções auxiliar de multiplicação
    def Multiply(x, c):
        return x * c
    #Função auxiliar de truncamento e arredondamento
    def trunca(number):
        if (number > 255):
            return 255
        elif(number < 0):
            return 0
        else:
            return int(round(number))
    #Separação dos canais YIQ
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img_YIQ)

    #Primeiro a gente cria o modelo de vetorização para as funções auxiliares

    multiply_vectorized = np.vectorize(Multiply)
    #Aplicamos a multiplicação apenas na banda Y
    print("Aplicando multiplicação apenas na banda Y...")
    #print(Ychannel[:5,:5])
    Ychannel = multiply_vectorized(Ychannel, c)
    print("Recriação da imagem YIQ, com valores de Y alterados..")
    #print(Ychannel[:5,:5])
    img = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)

    return img


img = carregaRGB("rainbow.png")

img = brilhoMultYIQ(converteRGB_YIQ(img), 0.5)

print("Conversão YIQ->RGB")
img = converteYIQ_RGB(img)

plt.imshow(img)
plt.show()
```

```
Conversão para RGB->YIQ
Aplicando multiplicação apenas na banda Y...
Recriação da imagem YIQ, com valores de Y alterados..
Conversão YIQ->RGB
Conversão para YIQ->RGB
```



1.6. Convolução m x n com bias (viés, offset). Testar com filtros Média e Sobel. (Xerxes)

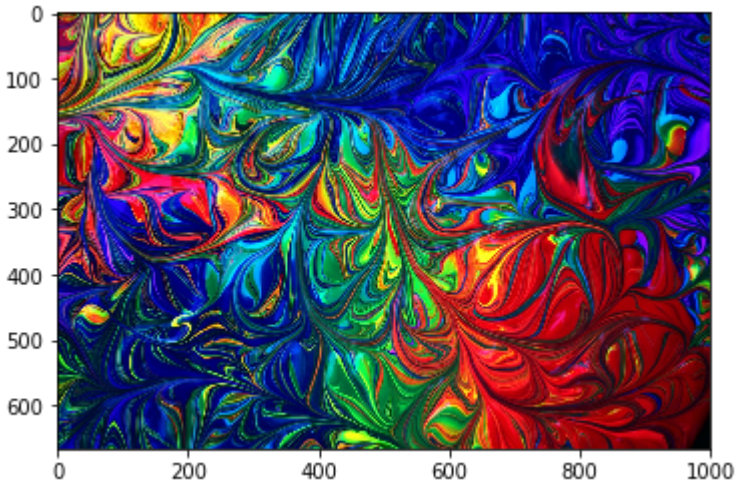
In [11]:

```
#Convolução em RGB
def convolution(mascara):
    mascara.reverse()
    return mascara

def trunca(number):
    if (number > 255):
        return 255
    elif(number < 0):
        return 0
    else:
        return int(round(number))

def filtroMedia(mascara):
    r_total = 0
    g_total = 0
    b_total = 0

    for pixel in mascara:
        #print('Pixel:', pixel)

        r = pixel[0]
        g = pixel[1]
        b = pixel[2]

        r_total += r
        g_total += g
        b_total += b

    r_total /= 9
    g_total /= 9
    b_total /= 9
    pixel = (trunca(r_total), trunca(g_total), trunca(b_total))
    return pixel

def filtroSobel(mascara, tipo_sobel):
    #print('Mascara:', mascara)

    #tipo_sobel:
    #Vertical
    #Horizontal

    if (tipo_sobel == 'Vertical'):
        sobel = [-1, 0, 1, -2, 0, 2, -1, 0, 1]#sobel Vertical
    elif (tipo_sobel == 'Horizontal'):
        sobel = [-1, -2, -1, 0, 0, 0, 1, 2, 1]#sobel Horizontal

    r_total = 0
    g_total = 0
    b_total = 0

    counter = 0
    for pixel in mascara:
        #print('Pixel:', pixel)
        r = pixel[0] * sobel[counter]
        g = pixel[1] * sobel[counter]
        b = pixel[2] * sobel[counter]

        #pixel_total += pixel
```

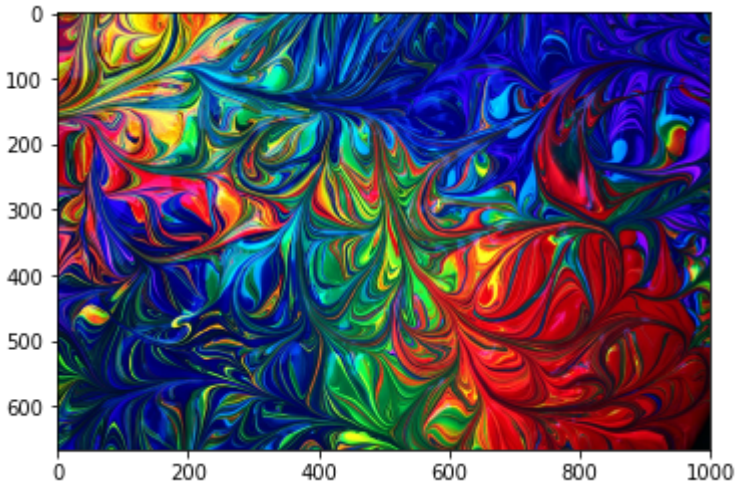
```
            r_total += r
            g_total += g
            b_total += b

            counter += 1

        pixel = (trunca(r_total), trunca(g_total), trunca(b_total))
        return pixel

    def mxn(img, m, n, tipo_filtro):

        #tipo_filtro:
        #1 = Media
        #2 = Sobel Vertical
        #3 = Sobel Horizontal

        img = Image.fromarray(np.uint8(img), "RGB")
        mascara = [(0)] * m * n

        #print(mascara, type(mascara))
        width, height = img.size
        tamh = (m*n / m)
        tamv = (m*n / n)
        v = int(tamv - (tamv/2))
        h = int(tamh - (tamh/2))

        filteredImage = Image.new("RGB",(width,height),"white") #Para trocar o tipo de imag
    em (RGB, Grayscale), só mudar o "RGB"

        counter = 0

        for i in range(h, width - h):
            for j in range(v, height - v):
                for l in range(m):
                    for c in range(n):
                        mascara[counter] = img.getpixel((i - (h - c), j - (v - l)))
                        counter += 1


                counter = 0
                #Colocar o que vc quer fazer com a mascara aqui

                #Aplicação da Convolução
                mascara = convolution(mascara)

                #Aplicação dos Filtros
                if (tipo_filtro == 1):
                    pixel = filtroMedia(mascara)
                elif (tipo_filtro == 2):
                    pixel = filtroSobel(mascara, 'Vertical')
                elif (tipo_filtro == 3):
                    pixel = filtroSobel(mascara, 'Horizontal')

                filteredImage.putpixel((i,j),(pixel)) #Preenche cada pixel com o valor da m
    ediana na nova imagem

        #Colocar retorno do metodo
        return filteredImage

    def main():
        #img = Image.open("leonardo.png")
```

```
    img = carregaRGB("rainbow.png")

    print('Imagem Original:')
    plt.imshow(img)
    plt.show()

    #tipo_filtro:
    #1 = Media
    #2 = Sobel Vertical
    #3 = Sobel Horizontal

    m = 3
    n = 3

    for count in range(1, 4):
        tipo_filtro = count

        imagem_filtrada = mxn(img, m, n, tipo_filtro)

        if (tipo_filtro == 1):
            nome_imagem = 'Filtro Média'

        elif (tipo_filtro == 2):
            nome_imagem = 'Filtro Sobel Vertical'

        elif (tipo_filtro == 3):
            nome_imagem = 'Filtro Sobel Horizontal'

        print(nome_imagem + ':')
        plt.imshow(imagem_filtrada)
        plt.show()

        imagem_filtrada.save(nome_imagem + '.png')

if __name__ == "__main__":
    main()
```

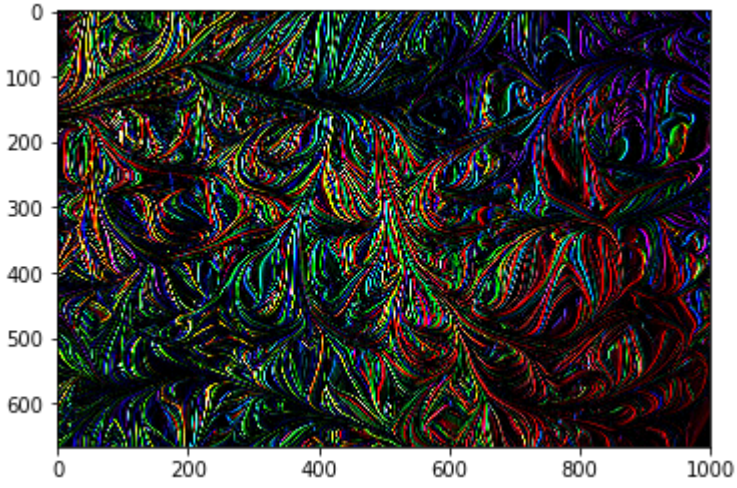
Imagem Original:



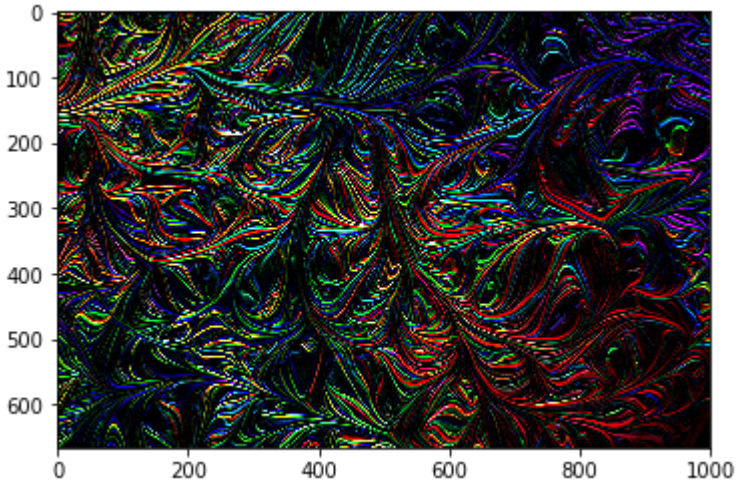
Filtro Média:



Filtro Sobel Vertical:



Filtro Sobel Horizontal:


0
100
200
300
400
500
600
0
200
400
600
800
1000

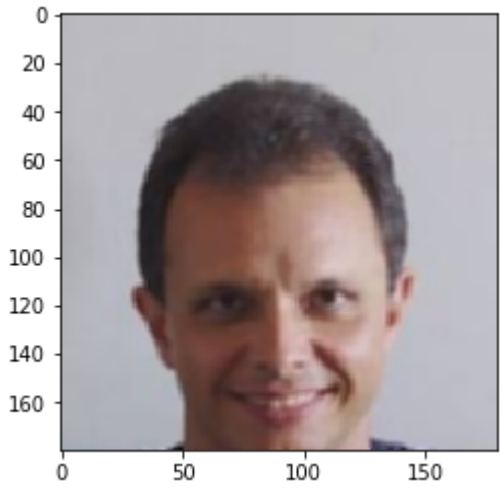
In [13]:

```
#Convolução em YIQ

def convolution(mascara):
    mascara.reverse()
    return mascara


def filtroMediaY(mascara):
    y_total = 0

    for pixel in mascara:
        #print('Pixel:', pixel)

        y_total += pixel

    y_total /= 9

    return y_total

def filtroSobelY(mascara, tipo_sobel):
    #print('Mascara:', mascara)

    #tipo_sobel:
    #Vertical
    #Horizontal

    if (tipo_sobel == 'Vertical'):
        sobel = [-1, 0, 1, -2, 0, 2, -1, 0, 1]#sobel Vertical
    elif (tipo_sobel == 'Horizontal'):
        sobel = [-1, -2, -1, 0, 0, 0, 1, 2, 1]#sobel Horizontal

    y_total = 0

    counter = 0
    for pixel in mascara:
        #print('Pixel:', pixel)
        y_total += pixel * sobel[counter]

        counter += 1

    return y_total

def mxnY(img, m, n, tipo_filtro):
    #img = converteRGB_YIQ(carregaRGB("rainbow.png"))
    #m = 3
    #n = 3
    #tipo_filtro = 1
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img)

    #tipo_filtro:
    #1 = Media
    #2 = Sobel Vertical
    #3 = Sobel Horizontal

    #Ychannel = Image.fromarray(Ychannel)
    mascara = [(0)] * m * n

    #print(mascara, type(mascara))
    width, height = img.shape[0], img.shape[1]
```

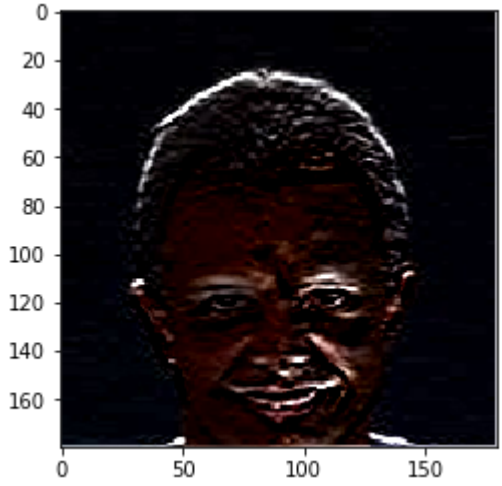
```
    tamh = (m*n / m)
    tamv = (m*n / n)
    v = int(tamv - (tamv/2))
    h = int(tamh - (tamh/2))
    #print(h)
    #print(v)

    #filteredImage = Image.new("RGB",(width,height),"white") #Para trocar o tipo de ima
gem (RGB, Grayscale), só mudar o "RGB"
    #filteredImage = np.zeros((Ychannel.shape[0], Ychannel.shape[1]), dtype = np.float6
4)
    filteredImage = Ychannel.copy()
    counter = 0
    for i in range(h, width - h):
        for j in range(v, height - v):
            for l in range(m):
                for c in range(n):
                    mascara[counter] = Ychannel[i - (h - c), j - (v - l)]
                    counter += 1


            counter = 0
            #Colocar o que vc quer fazer com a mascara aqui

            #Aplicação da Convolução
            mascara = convolution(mascara)
            #Aplicação dos Filtros
            if (tipo_filtro == 1):
                pixel = filtroMediaY(mascara)
            elif (tipo_filtro == 2):
                pixel = filtroSobelY(mascara, 'Vertical')
            elif (tipo_filtro == 3):
                pixel = filtroSobelY(mascara, 'Horizontal')
            mascara = [(0)] * m * n
            filteredImage[i][j] = pixel #Preenche cada pixel com o valor da mediana na
 nova imagem

    #Colocar retorno do metodo
    return filteredImage


img = converteRGB_YIQ(carregaRGB("leonardo.png"))

#y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img)

#tipo_filtro:
#1 = Media
#2 = Sobel Vertical - Só fuciona com mxn = 3x3
#3 = Sobel Horizontal - Só fuciona com mxn = 3x3
for i in range(1, 4):
    tipo_filtro = i
    m = 3
    n = 3

    #imagem_filtrada = mxn(img, m, n, tipo_filtro)
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img)

    Ychannel = mxnY(img, m, n, tipo_filtro)

    imagem_filtrada = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)
```

```
    imagem_filtrada = converteYIQ_RGB(imagem_filtrada)


    if (tipo_filtro == 1):
        nome_imagem = 'Filtro Média'
        print(nome_imagem)

    elif (tipo_filtro == 2):
        nome_imagem = 'Filtro Sobel Vertical'
        print(nome_imagem)

    elif (tipo_filtro == 3):
        nome_imagem = 'Filtro Sobel Horizontal'
        print(nome_imagem)

    plt.imshow(imagem_filtrada)
    plt.show()

    imagem_filtrada = Image.fromarray(np.uint8(imagem_filtrada), "RGB")
    imagem_filtrada.save(nome_imagem + '.png')
```

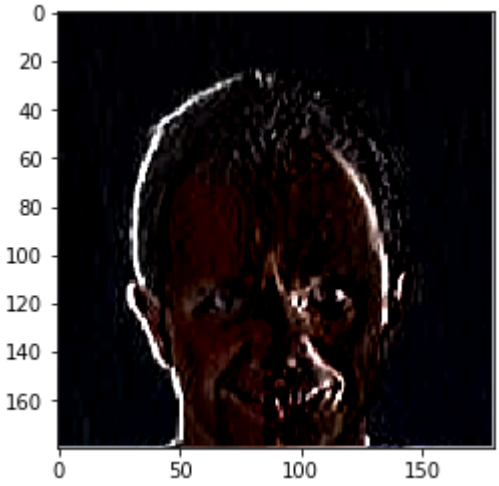
```
Conversão para RGB->YIQ
Conversão para YIQ->RGB
Filtro Média
```




```
Conversão para YIQ->RGB
Filtro Sobel Vertical
```




```
Conversão para YIQ->RGB
Filtro Sobel Horizontal
```

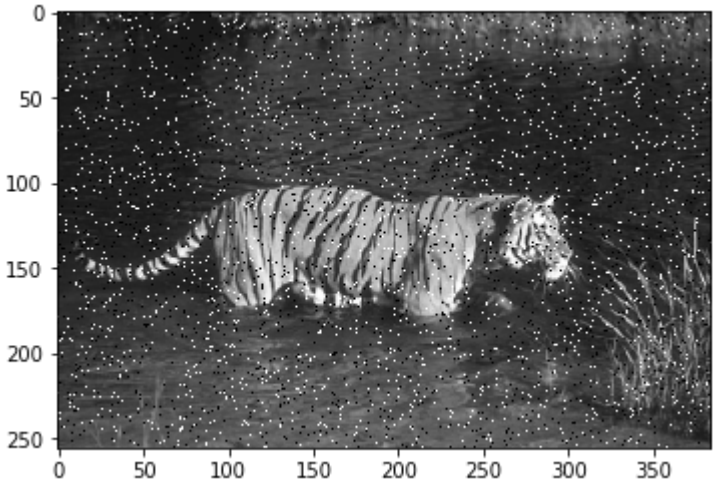


1.7. Filtro mediana m x n. (Rebeca)

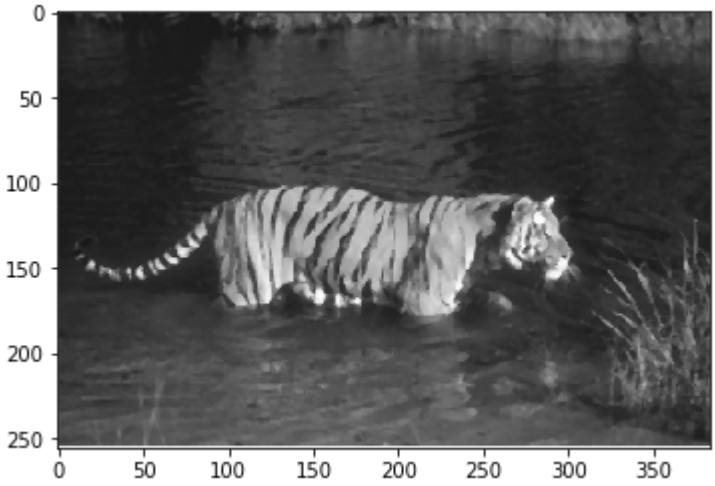
In [14]:

```
#Filtro de Mediana, recebe como par
def medianFilter(originalImg, mMask, nMask):

    img = Image.fromarray(np.uint8(originalImg), "RGB")
    #Cria a mascara
    m = mMask
    n = nMask
    mascara = [(0)] * m*n
    print("Mascara's size: ")
    print(len(mascara))
    #print(mascara, type(mascara))
    width, height = img.size
    #Determina a parte corrente da imagem a ser percorrida
    tamh = (m*n / m)
    tamv = (m*n / n)
    v = int(tamv - (tamv/2))
    h = int(tamh - (tamh/2))

    #Cria imagem nova em branco
    filteredImage = Image.new("RGB",(width,height),"white")

    counter = 0

    #Percorre a imagem original com a mascara
    for i in range(h, width - h):
        for j in range(v, height - v):
            for l in range(m):
                for c in range(n):
                    mascara[counter] = img.getpixel((i - (h - c), j - (v - l))) #percor
re na vertical
                    counter += 1 #passa pra próxima coluna na horizontal
            counter = 0
            #Chama a função que encontra a mediana de cada banda
            mR, mG, mB = mediana(mascara)
            filteredImage.putpixel((i,j),(mR, mG, mB)) #Preenche cada pixel com o valor
das medianas de cada banda

    print("Filtro da mediana aplicado com sucesso")
    return filteredImage

def mediana(mascara):
    rValue = []
    gValue = []
    bValue = []
    #Percorre os valores de cada banda que estão na mascara
    for valor in mascara:
        r = valor[0]
        g = valor[1]
        b = valor[2]

        #Salva os valores de R, G e B da máscara numa lista
        rValue.append(r)
        gValue.append(g)
        bValue.append(b)

    '''Encontra a mediana de cada valor,
    lembrando que a função median já trata
    o caso da lista ser par!'''
    rValue = int(median(rValue))
```

```
    gValue = int(median(gValue))
    bValue = int(median(bValue))

    return rValue, gValue, bValue

#Carregamos a imagem RGB
img = carregaRGB("ruido.png")
plt.imshow(img)
plt.show()

#Aplicamos o filtro
mascara = medianFilter(img, 3, 3)
plt.imshow(mascara)
plt.show()

#Imagem Salva
medianFilter = Image.fromarray(np.uint8(img), "RGB")
medianFilter.save("medianFilter.png")
```



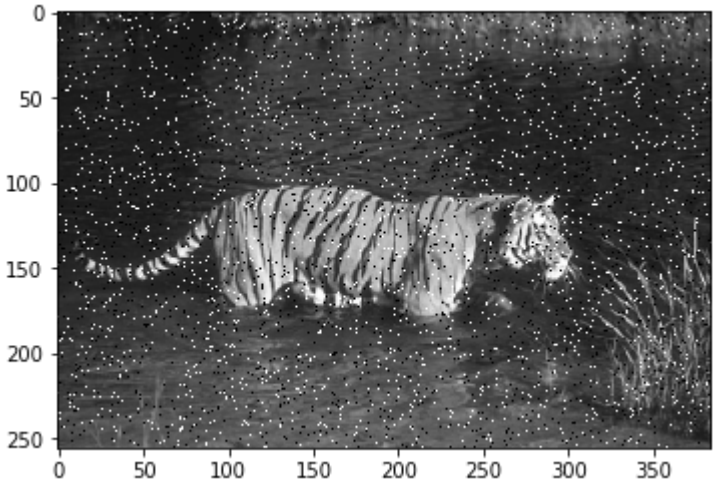

```
Mascara's size:
9
Filtro da mediana aplicado com sucesso
```



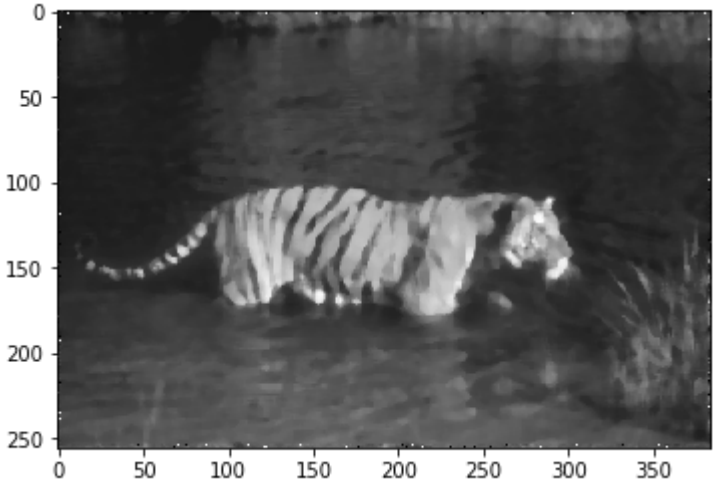
In [53]:

```
def medianFilterY(img, m, n):

    #Separamos as bandas para usar apenas a Y
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img)

    #Criamos a máscara
    mascara = [(0)] * m * n
    print("Mascara's size: ")
    print(len(mascara))

    #Determinamos o tamanho da imagem e o offset
    width, height = img.shape[0], img.shape[1]
    tamh = n
    tamv = m
    v = int(tamv - (tamv/2))
    h = int(tamh - (tamh/2))

    #Criamos a banda que será a filtrada
    filteredY = Ychannel.copy()

    counter = 0
    try:
        #Percorre a imagem original com a mascara
        for i in range(h, width - h):
            for j in range(v, height - v):
                for l in range(m):
                    for c in range(n):
                        mascara[counter] = Ychannel[i - (h - c), j - (v - l)] #percorre
na vertical
                        counter += 1 #passa para a próxima coluna na horizontal

                counter = 0

                #Aplicamos o filtro
                pixel = medianaY(mascara)
                filteredY[i][j] = pixel #Preenche cada pixel com o valor da mediana na
 nova imagem
    except Exception as error:
        print('Error:', error)
    #retornamos a banda filtrada
    print("Filtro da Mediana aplicado com sucesso - Banda y")
    return filteredY


def medianaY(mascara):
    yValue = []
    #Percorremos os valores de Y na mascara
    for valor in mascara:
        y = valor

        #Salvamos os valores de Y numa lista
        yValue.append(y)

    '''Encontra a mediana dos valores de Y,
    lembrando que a função median já trata
    o caso da lista ser par!'''
    yValue = median(yValue)

    return yValue
```

```
#Carrega a imagem
img = carregaRGB("ruido.png")
plt.imshow(img)
plt.show()

#Converte pra YIQ e separa os canais, só usaremos a banda Y
img = converteRGB_YIQ(img)
y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img)

#Aplicamos o filtro
Ychannel = medianFilterY(img, 5, 5)

#Únimos a banda filtrada às demais e convertemos pra RGB
filteredImage = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)
imagem_filtrada = converteYIQ_RGB(filteredImage)
plt.imshow(imagem_filtrada)
plt.show()

#ImagemSalva
medianFilterY = Image.fromarray(np.uint8(img), "RGB")
medianFilterY.save("medianFilterY.png")
```




```
Conversão para RGB->YIQ
Mascara's size:
25
Filtro da Mediana aplicado com sucesso - Banda y
Conversão para YIQ->RGB
```

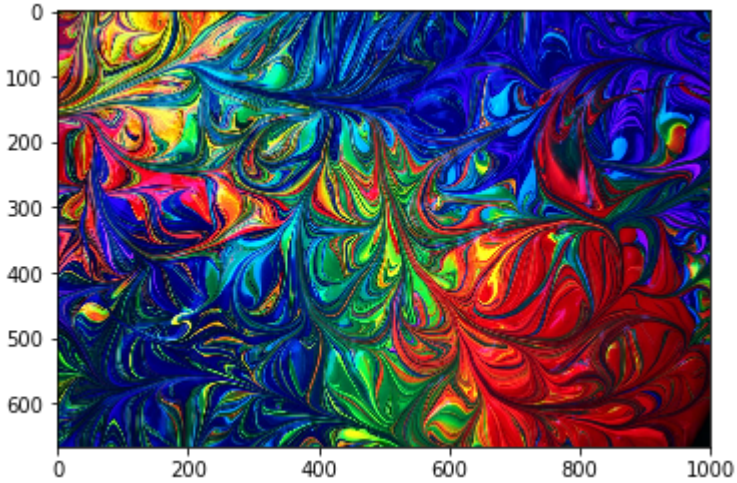


1.8. Limiarização com limiar m escolhido pelo usuário. (Pedro)

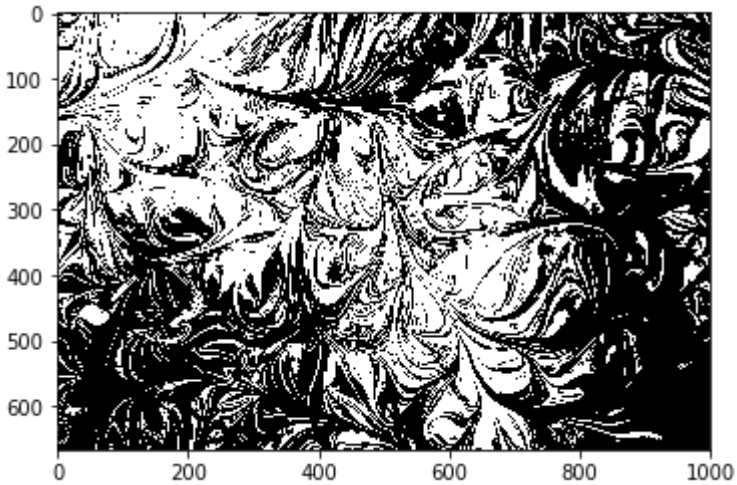
In [15]:

```
#Limiarização em RGB
m = int(input('Digite o limiar m:  '))

def limiarizacao(img, m):
    m = m
    data = img.getdata()

    l = [(limiar(int(d[0]), int(d[1]), int(d[2]), m)) for d in data]

    return l

def limiar(n1, n2, n3, m):
    soma = n1 + n2 + n3
    m = 3 * m
    if soma >= m:
        n1 = 255
        n2 = 255
        n3 = 255
    elif soma < m:
        n1 = 0
        n2 = 0
        n3 = 0

    return n1, n2, n3

img = Image.open("rainbow.png")
print('Antes:')
plt.imshow(img)
plt.show()

brilho = np.asarray(limiarizacao(img, 70))
print('Depois:')
plt.imshow(brilho.reshape(667, 1000, 3))
plt.show()
```

```
Digite o limiar m:      70
Antes:
```



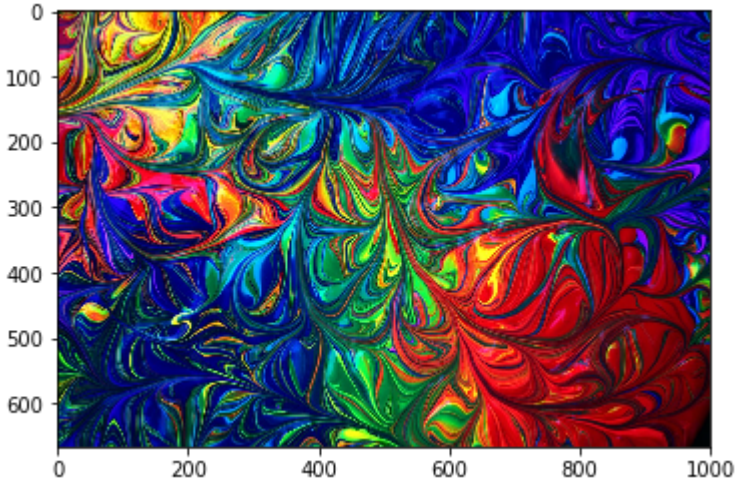

```
Depois:
```



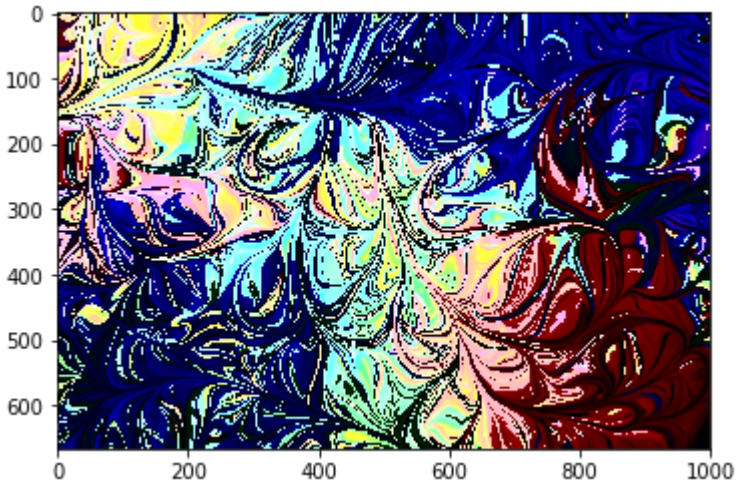
In [17]:

```
#Limiarização em YIQ
m = int(input('Digite o limiar m:  '))

def limiarizacaoY(img_YIQ, m):
    def limiarY(y, m):
        if y >= m:
            y = 255
        elif y < m:
            y = 0
        return y
    m = m
    y, i, q, Ychannel, Ichannel, Qchannel = separateYIQ(img_YIQ)

    limiar = np.vectorize(limiarY)

    Ychannel = limiar(Ychannel, m)

    img = np.stack((Ychannel, Ichannel, Qchannel), axis = 2)

    return img

img = Image.open("rainbow.png")
print('Antes:')
plt.imshow(img)
plt.show()

img = converteRGB_YIQ(img)
img = limiarizacaoY(img, m)

print('Depois:')

img = converteYIQ_RGB(img)

plt.imshow(img)
plt.show()
```

```
Digite o limiar m:      70
Antes:
```




```
Conversão para RGB->YIQ
Depois:
Conversão para YIQ->RGB
```



In [ ]: