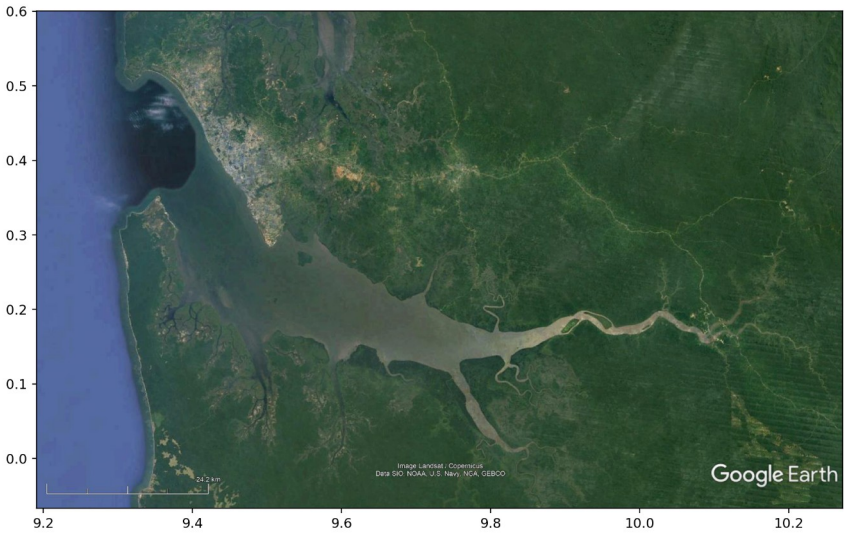
Analisando dados com Python

Usando Imagens: georeferenciamento

Parte 1

V1. 2021.12.22



Carlos A.F. Schettini

Laboratório de Oceanografia Costeira e Estuarina

LOCostE / FURG

O objetivo deste tutorial é demonstrar como georeferenciar uma imagem.

Georeferenciamento de uma imagem é uma transformação espacial entre diferentes sistemas de coordenadas. ‘Geo’, no caso, é de algum sistema arbitrário (coordenadas dos pixels da imagem) para coordenadas geográficas (latitude & longitude). O assunto é extremamente complexo, pois tem as projeções, distorções de imagens e outras coisas… (Dunning-Kruger! Leia mais a respeito!). Para fins geográficos/cartográficos, o caminho é usar um sistema geográfico de informações (SIG), tal como o Qgis. Se precisar fazer algo com maior precisão, existem bibliotecas para fazer isso no Python (Cartopy, Rasterio) e acesso à ferramentas externas (GDAL). Para processos repetitivos, é melhor fazer no Python do que no Qgis. Por exemplo, extrair séries temporais a partir de imagens de satélite.

Porém, para começar, vamos fazer algo muito básico para entender o princípio da coisa! Usando o Google Earth, escolha algum canto do mundo, e dê um zoom para que tenha uma escala grande. (Isso confunde um pouco: 1:1.000.000 é uma escala pequena comparada com 1:1.000, que é uma escala grande! O ‘milhão’ e o ‘mil’ são denominadores!) Isto porque para uma escala grande (e.g, 1:20.000), podemos aproximar a superfície da Terra como um plano. Quando menor for a escala, maior vai ser a distorção. Latitude e longitude são coordenadas esféricas, mas, em escalas grandes podem ter aproximações planas com erros aceitáveis!

No Google Earth, tenha certeza que está com uma visão ortogonal da superfície. Não pode estar oblíqua. E, também, que está orientada para o norte! Uma vez escolhido o local, usando 'Edit', 'Copy Image' ou Ctrl+Alt+C, copie a imagem e cole em um programa de edição de imagens. Eu uso o Paint.net (gratuito!). Salve a imagem. Eu salvo no formato 'png'. Mas poderia ser tiff, jpg ou bmp… Mas, ATENÇÃO! Neste exercício precisamos de duas imagens exatamente do mesmo local (superfície da Terra!), uma com as linhas das coordenadas e outra, absolutamente na mesma posição, sem as linhas. Então, tem repetir o procedimento de copiar e colar duas vezes, tendo o cuidado de não mover a imagem no Google Earth. Para adicionar ou remover as linhas no Google Eearth dá para ser com o Ctrl + L.

Agora temos duas imagens idênticas, porém uma tem as linhas de coordenadas e a outra não. Isso é para que quando quisermos trabalhar com a imagem, podemos optar por ter ou não os grids. Geralmente não! Especialmente se quisermos plotar alguma coisa sobre ela, como vetores de correntes ou isolinhas.

Para carregar e ver as imagens no Jupyter, usamos o pacote Matplotlib.pyplot (Código 1), mas há outros para fazer isso (OpenCV, Pillow, scikit-image). Observe as imagens e tente (se esforce) 'ler' o conteúdo do problema que temos. Aqui entra a tal da abstração… as pessoas ficam entretidas com o conteúdo 'superficial' da imagens, ou seja, as cores e feições, e ficam por aí. Inicialmente é necessário entender que uma 'imagem' não é nada mais do que uma matriz de números. Nós já geramos imagens antes (tutorial CTD), e estas não são conceitualmente diferentes. O que muda é que estas foram geradas a partir de um sensor em um satélite, que, para todos os propósitos é equivalente a uma máquina fotográfica como tem no seu celular. O sensor

capta radiação refletida em um conjunto de sensores discretos (no sentido de oposto de contínuo!), cada qual fornecendo um pixel (==> **PI**cture x **EL**ement!). Isto tudo de maneira digital, ou seja, a quantidade de radiação captada é convertida em um número (que por sua vez pode ser convertidos em bits → bytes → computação → Python!

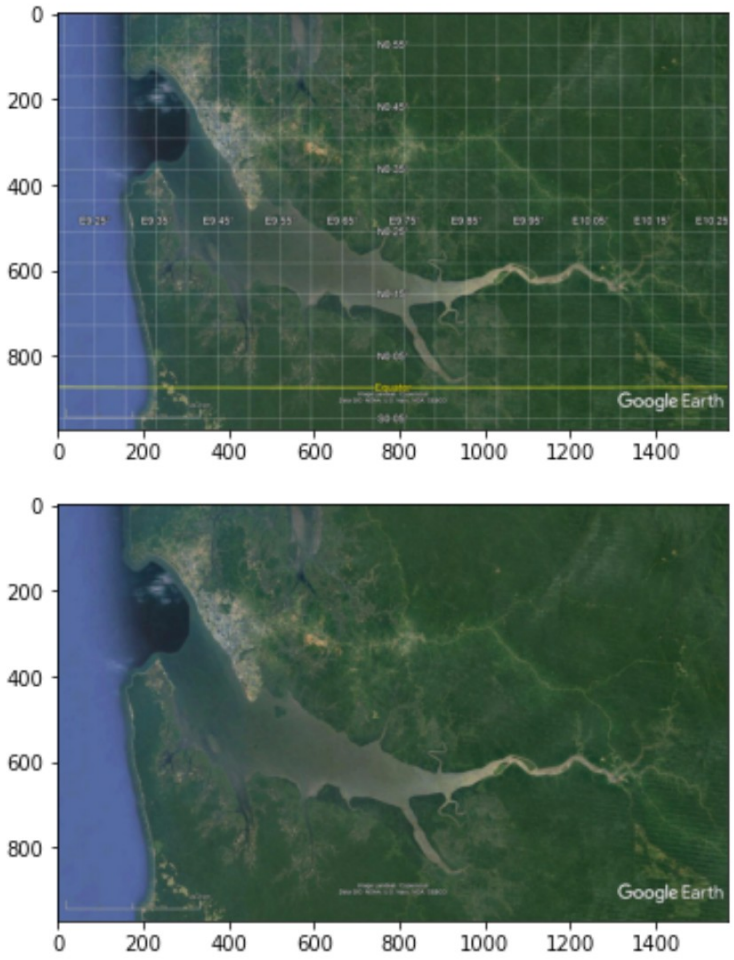
Sensores em satélites são bem mais sofisticados que os do seu celular, e podem captar comprimentos de ondas específicos. Mas, as imagens do Google Earth, independente do satélite que as gerou, e salvas no formato PNG ou qualquer que seja, são compostas por não uma mas três matrizes representando a quantidade de vermelho, verde e azul (RGB – red/green/blue), que quando sobrepostos fornecem qualquer outra cor do espectro do visível. Cada matriz tem um número de linhas e colunas, e cada elemento da matriz pode ter valores entre 0 e 255 (ou 2^8), ou valores derivados destes, dependendo do formato. Sendo matrizes, podemos carregar e processar tais quais, com procedimentos muito similares ao já utilizados, como o slicing (Código 2).

Então, retornando para a 'abstração', o que temos quando carregamos uma imagem colorida é uma matriz 3D, com linhas, colunas e camadas. No nosso caso, como a imagem foi capturada do Google Earth, perpendicular na vertical e orientada para o norte, podemos 'abstrair' que as coordenadas no eixo x são relacionadas a longitude, e no eixo y a latitude. Mas, os números são obviamente diferentes. Os números que vemos nos 'ticks' dos eixos representam os pixels, e não coordenadas. E, para quem não percebeu, a orientação do eixo y está invertida (cartesianamente falando!).

```python
import matplotlib.pyplot as plt
```

```python
img_cg = plt.imread('imagem_com_grid.png')
img_sg = plt.imread('imagem_sem_grid.png')
```

```python
plt.imshow(img_cg)
plt.show()
plt.imshow(img_sg)
plt.show()
```



Código 1: Carregando e visualizando imagens.

```
print(type(img_cg))
img_cg.shape
```

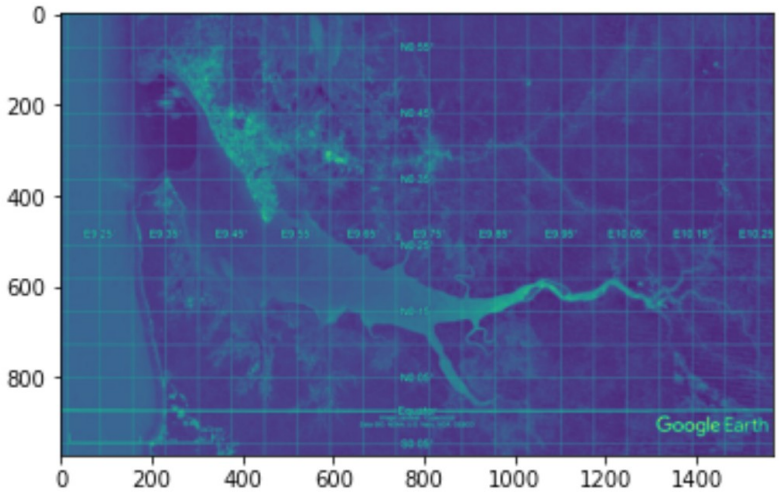
```
<class 'numpy.ndarray'>
(971, 1568, 3)
```

```
b = img_cg[:,:,0].squeeze()
print(b[:3,:3])
```

```
[[0.30588236 0.30588236 0.30588236]
 [0.3137255  0.3137255  0.3137255 ]
 [0.3019608  0.3019608  0.3019608 ]]
```

```
plt.imshow(b)
```

```
<matplotlib.image.AxesImage at 0x2a5d0cb0160>
```

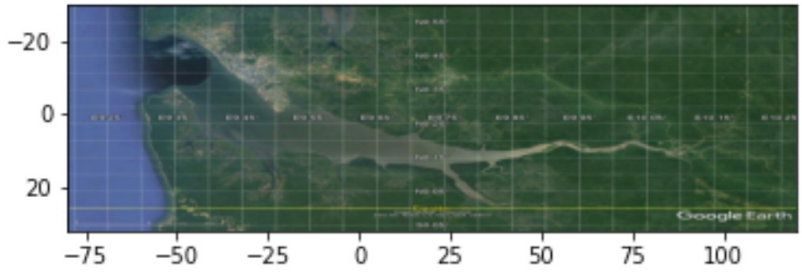


Código 2: Explorando uma imagem.

Então nosso problema é como converter estes números que representam a numeração de pixels da imagem em números que metafisicamente representam a posição deste bits/bytes na

superfície da Terra, em coordenadas geográficas. Talvez ajude a entender que eu posso informar ao Matplotlib absolutamente qualquer número real para os vértices da imagem, e não importa a orientação (Código 3). Mas, queremos fornecer os vértices que estejam atrelados ao sistema geográfico!

```
fig, ax = plt.subplots()
ax.imshow(img_cg, extent=[-80,120,32,-30])
plt.show()
```



Código 3: Mudando as coordenadas da imagem com 'extent'.

Esta atividade é uma das que eu uso em meus cursos de programação ou análise de dados, pois inicialmente os alunos acham que isto é algo extremamente complexo e é necessário conhecimento profundo de matemática avançada e programação. Mas não… é necessário inicialmente a ‘abstração’ para entender o problema sob a imagem colorida, e álgebra linear básica.

Considerando que a imagem é plana, podemos considerar que os valores de pixel e coordenadas são linearmente relacionados. Ou, um pode ser convertido no outro se soubermos a equação da reta que os relaciona. O que é necessário para obter uma equação da reta? Segundo

Euclides (~ 300 a.C.), bastam dois pontos. Então, para converter os valores de pixel do eixo x em longitude, eu preciso de dois pontos na imagem que eu conheça os valores de pixel e longitude (Figura 1). O mesmo se aplica para o eixo y. Ou seja, precisamos de duas equações de reta, uma para longitude e outra para a latitude.

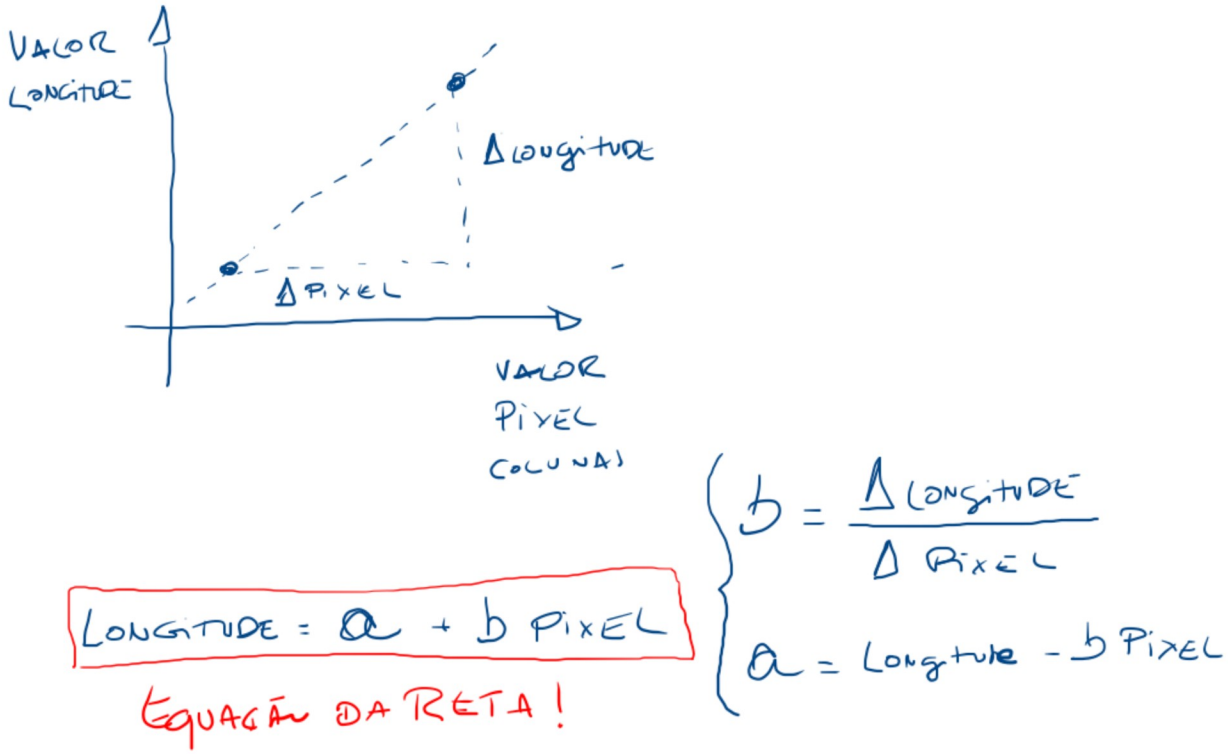


Figura 1: Equação da reta!

O que certamente confunde, possivelmente pela falta de abstração, é que temos duas dimensões que precisam ser tratadas separadamente. Uma vez entendido o problema central, agora vem outro problema: como obter os valores de pixel e coordenadas para construirmos as equações! Para isso pegamos uma imagem com grid. Em qualquer intersecção de linhas de longitude e latitude saberemos as coordenadas geográficas. Mas, para obtermos os valores

equivalentes em pixel, temos que capturar essa informação das imagens. Para isso temos a ferramenta do Matplotlit 'ginput'. Mas… não tão simples. Sem querer entrar em minúcias (não são, de fato, minúcias! O vale do desespero aqui é bem fundo!), o Jupyter é um software bastante complexo, ainda mais por ser multi-plataforma (Windows, Linux e OS). E, cada plataforma tem suas regras de como os softwares interagem com o hardware. O monitor é um hardware (!), e para usar o ginput precisamos usar um modo gráfico ativo offline, ou seja, externo à janela que contém o Jupyter. Em outras palavras, o gráfico tem que abrir uma janela separada, e estar no modo ativo, onde podemos dar zoom, mover, etc. Para fazer isso, a solução que eu encontrei no StackOverflow (https://stackoverflow.com/questions/41403406/matplotlib-use-of-ginput-on-jupyter-matplotlib-notebook) é importando o matplotlib (sem o 'pyplot'), e ativar o 'TkAgg',

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

. Mas que raios seria 'TkAgg'… Tk é de Tkinter, que é o 'Graphical User Interface' (GUI) padrão do Python. Então, aparentemente quando a gente está usando o Jupyter, o GUI não é o Tkinter… !?!? Eu não faço idéia de qual seja, mas isso é para dar a noção da profundidade do vale. Enfim, use isso, e como sugerido pela resposta no StackOverflow, quando quiser fazer um gráfico em uma janela ativa (externa), inclua no início da célula `%matplotlib qt` (qt?), e quando quiser que o gráfico fique 'embutido' no notebook, como usualmente usamos, inclua no início da célula `%matplotlib inline`

O que queremos fazer, resumindo, é abrir uma janela com a imagem com o grid e capturar dois pontos na diagonal, com o 'plt.ginput' para fornecer as variações de pixel na horizontal e vertical. Pontos estes localizados nas interseções das coordenadas geográficas que poderemos capturar visualmente. Executando o Código 4, obteremos a Figura 2.

```
%matplotlib qt
plt.imshow(img_cg)
pontos = plt.ginput(2, timeout=-1)
plt.show()
```

Código 4: Usando o 'ginput'.

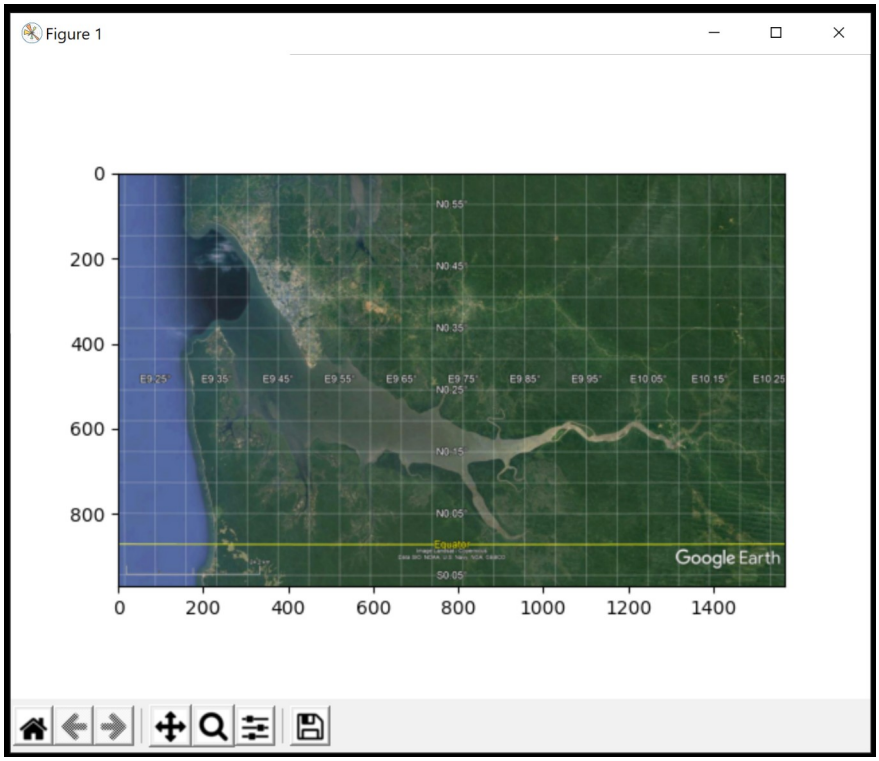


Figura 2: Janela aberta com o Código 4.

Como podem facilmente notar, a figura é acopanhada de uma barra de ferramentas, como zoom e salvar. Como nós chamamos o ‘ginput’, e especificamos 2, o código está aguardando dois clics do botão esquerdo do mouse sobre a janela aberta. Caso precise, use o zoom. Mire, e clique em dois pontos, um do lado inferior-esquerdo e outro do lado superior-direito. Depois do

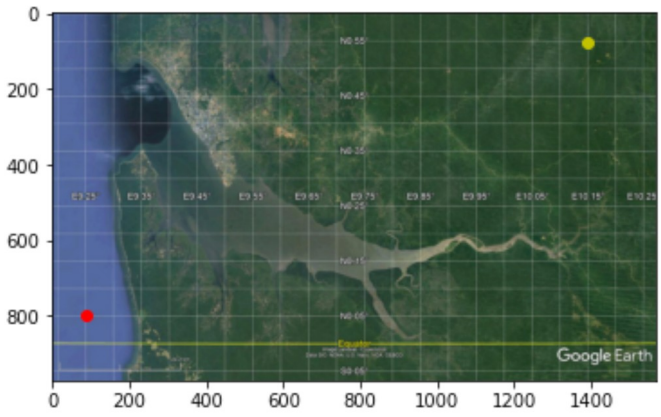
segundo clique, aparentemente não acontece nada, mas os pontos capturados já foram armazendos na variável ‘pontos’. Para ver se deu certo: Código 5.

```
print(pontos)
```

```
[(86.19660596026495, 799.6908112582782), (1390.375, 76.26365894039736)]
```

```
%matplotlib inline
fig, ax = plt.subplots()
ax.imshow(img_cg)
plt.plot(pontos[0][0], pontos[0][1], 'or')
plt.plot(pontos[1][0], pontos[1][1], 'oy')
```

```
[<matplotlib.lines.Line2D at 0x1df1e005ac0>]
```



Código 5: Conferindo os pontos capturados.

O ‘ginput’ retorna tuplas de coordenadas, por isso ficam entre ( ), diferente de listas, que ficam entre [ ]. Agora, fornecemos manualmente (olhando e digitando!) os valores das coordenadas geográficas destes dois pontos (Código 6). Não foi mencionado antes, mas as coordenadas geográficas devem estar em graus e décimos de graus!

```
# pontos coordenadas pixels a partir do ginput
pt1_p = pontos[0]
pt2_p = pontos[1]

# pontos coordenadas geográficas capturados manualmnete
pt1_g = [9.25, 0.05]
pt2_g = [10.15, 0.55]
```

Código 6: Pares de pontos em coordenadas pixel (_p) e geográficas (_g).

Agora é construir as equações das retas e usá-las para achar as coordenadas dos vértices, ou, quando o pixel é 1 ou o número total de linhas (latitude) e colunas (longitude) (Código 7)!

```
# equação da reta
# Y = a + b X
# b = delta Y / delta X
# a = Y - b X
```

```
dx_g = pt2_g[0] - pt1_g[0]
dx_p = pt2_p[0] - pt1_p[0]

b_x = dx_g / dx_p
a_x = pt1_g[0] - b_x * pt1_p[0]
```

```
dy_g = pt2_g[1] - pt1_g[1]
dy_p = pt2_p[1] - pt1_p[1]

b_y = dy_g / dy_p
a_y = pt1_g[1] - b_y * pt1_p[1]
```

```
linhas, colunas, _ = img_cg.shape

x_1 = a_x + 1*b_x
x_2 = a_x + colunas*b_x

y_1 = a_y + 1*b_y
y_2 = a_y + linhas*b_y
```

Código 7: Construindo e usando as equações de reta!

Se não houve erros (!!), o resultado será (Código 8, Figura 3).

```
%matplotlib qt
fig, ax = plt.subplots()
ax.imshow(img_cg, extent=[x_1, x_2, y_2, y_1])
plt.show()
```

Código 8: Visualizando a imagem georeferenciada.

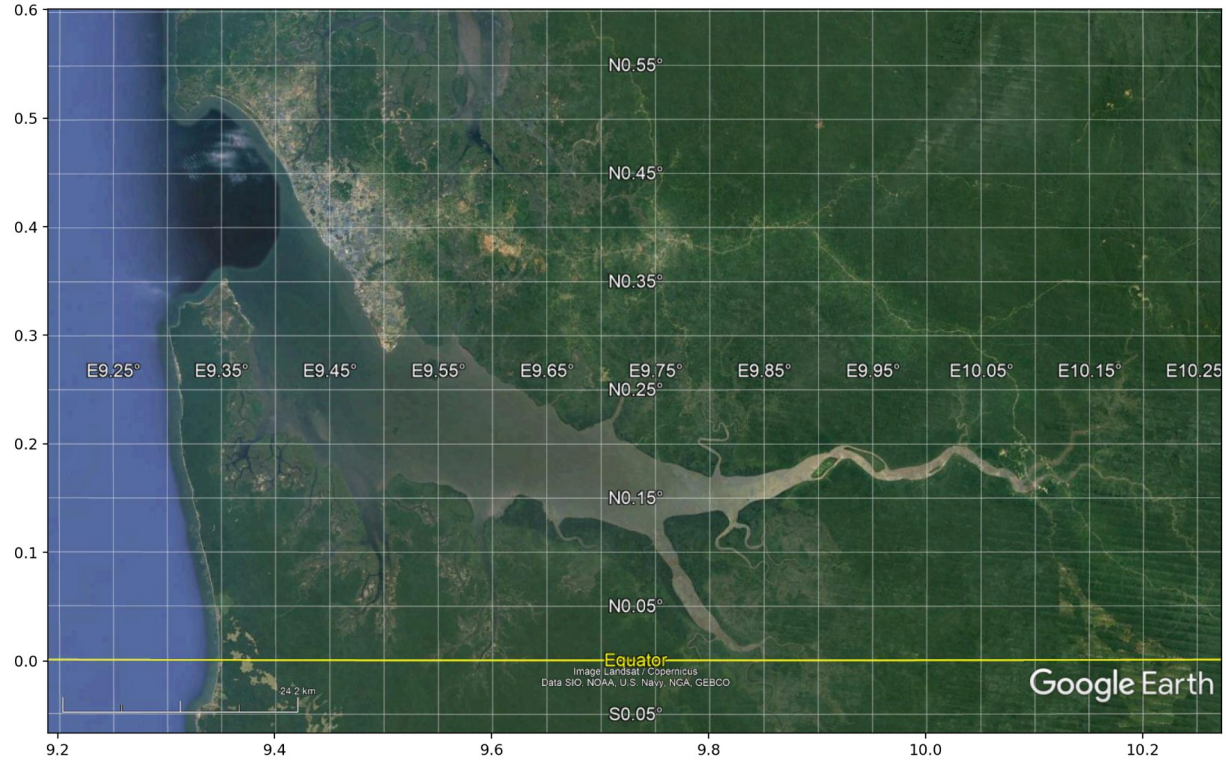


Figura 3: Imagem georeferenciada.

A qualidade do produto final depende muito da precisão da captura dos pontos da imagem. Mas podemos observar que o resultado está muito razoável. Para validar e avaliar o erro, podemos captura alguns pontos notáveis nesta imagem com o ‘ginput’ e comparar com a

localização fornecida pelo Google Earth. Se estiver satisfeito com o resultado, basta usar a imagem sem grid para o que desejar (Figura 4)!

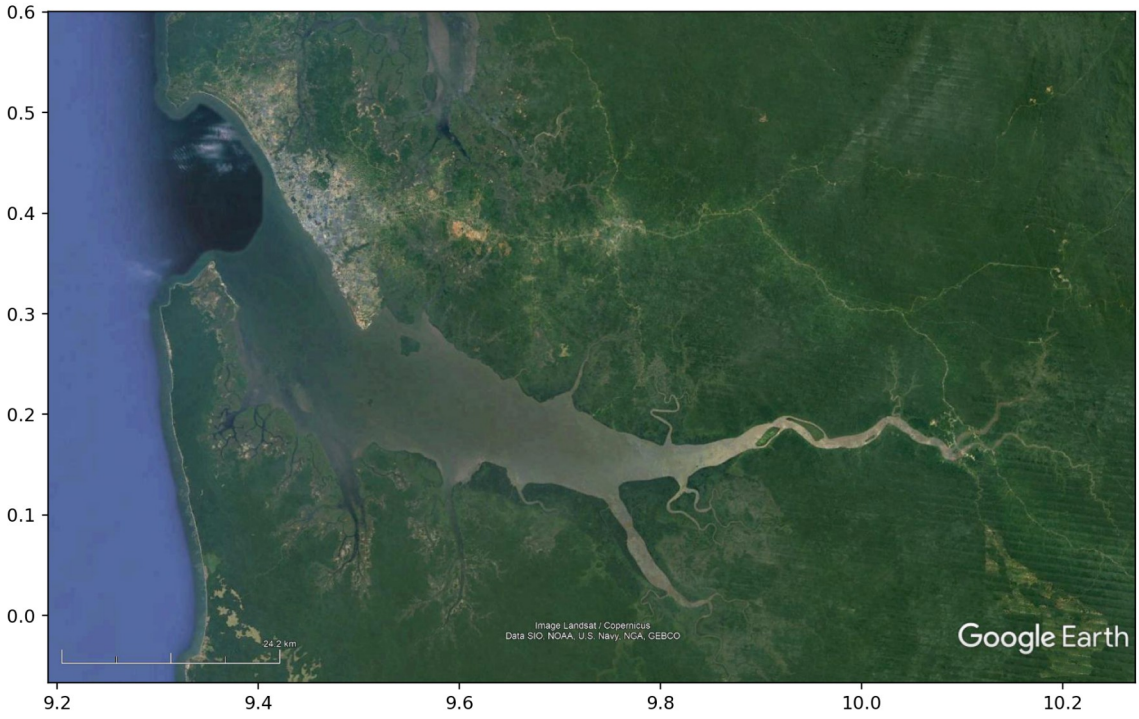


Figura 4: Imagem georefernciada sem grid.

Enfatizando que este procedimento não é recomendado para fins cartográficos. Mas é muito útil para apresentação de resultados sobre a imagem. Era deste jeito que eu fazia para, por exemplo, obter a área da superfície livre de estuários ou área de manguezais. Até que eu aprendi a usar o Qgis que certamente é uma ferramenta mais apropriada para este tipo de coisa! Também é importante dizer que em várias situações, em vez de fazer o georeferenciamento ‘braçal’ no Jupyter, a mesma coisa pode ser feita no Qgis. E, se é necessário usar a figura no Jupyter, dá para exportar a figura do Qgis no formato ‘geotif’, que nada mais é um ‘tif’ com metadata informando

os vértices da imagem, e mais importante, atrelado a uma projeção específica. Em coordenadas geográficas, usamos usualmente a WGS 84 / EPSG 4326.