فایل helper.py

همراه تمامی کدها، یک فایل helper.py قرار دارد که به منظور خوانایی کدهاست. تابع هایی که در فایل اصلی کدها هستند (Qi.py) در فایل helper پیاده سازی میشوند تا کد اصلی کوتاه تر و خواناتر باشد. در توضیحات سوال ها، هم کدهای اصلی هم تابع های پیاده سازی شده را تا جای ممکن توضیح میدهم.

کد اصلی در داخل یک for اجرا میشود که برای اجرا روی ۲ مثال پیاده سازی شده است.

عکس سورس، عکس تارگت و عکس ماسک را لود میکنیم.

مطابق آنچه که در کلاس تدریس شد و در اسلاید های درس آمده است، برای حل معادله پواسن و بدست آوردن تک تک پیکسل ها از این معادله،میتوان از Discrete Poisson Solver استفاده نمود:

Discrete Poisson Solver

$$\text{for all } p \in \Omega \quad |N_p| f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq}$$

Known:

$v_{pq} = gradient(p) - gradient(q)$

f^* = constrained border pixels

N_p = list of neighbors of pixel p

Unknown:

f = reconstructed pixels

Write equation as $Af = b$

- A is a large space matrix mostly of 0s and ± 1s
- b is a vector of the righthandside of the above equation

Matlab can solve (for small images) in a single command: $f = A\backslash b$

در این روش یک ماتریس خاص A وجود دارد که در ادامه توضیح میدهم که از کجا می آید و چطور آن را میسازم.

هدف حل معادله Af = b است. این معادله در واقع حل همان معادله پواسون را به ما می دهد.

هدف از متد Poisson Blending این است که گرادیان عکس تارگت در جایی که قرار است عکس سورس قرار بگیرد، همانند عکس سورس شود.

پس ما به چنین چیزی نیازمندیم:

$$\nabla f = \nabla g$$

ماتریس A ماتریسی است که برای ما همچون عملگر گرادیان عمل میکند. بدین منظور ضرب ماتریس A در عکس سورس را بدست آوردم که این موضوع را نشان دهم:

بنابراین در معادله $Af = b$ که b در واقع Ag است، هدف همان برابر بودن گرادیان هاست. یعنی جوری f را بدست آوریم که گرادیان دو طرف برابر شود.

برای ساختن ماتریس A از ویکی پدیا مربوط به Discrete Poisson Solver استفاده کردم:

https://en.wikipedia.org/wiki/Discrete_Poisson_equation

طبق این صفحه، ماتریس A چنین شکلی دارد:

$$A = \begin{bmatrix} D & -I & 0 & 0 & 0 & \dots & 0 \\ -I & D & -I & 0 & 0 & \dots & 0 \\ 0 & -I & D & -I & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -I & D & -I & 0 \\ 0 & \dots & \dots & 0 & -I & D & -I \\ 0 & \dots & \dots & \dots & 0 & -I & D \end{bmatrix},$$

I is the $m \times m$ identity matrix, and D, also $m \times m$, is given by:

$$D = \begin{bmatrix} 4 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 4 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 4 & -1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 4 & -1 & 0 \\ 0 & \dots & \dots & 0 & -1 & 4 & -1 \\ 0 & \dots & \dots & \dots & 0 & -1 & 4 \end{bmatrix},$$

که ماتریس A در واقع یک ماتریس mn در mn است. به عبارت دیگر هر سطر این ماتریس، شامل اطلاعات تمام عکس ماست. و براساس آن برای تک تک پیکسل های f یک جواب پیدا میکند.

برای ساختن ماتریس A ابتدا از numpy استفاده کردم:

```
# D = build_D_matrix(m)
# Eye_matrix = np.eye(m, m)
#
# A = np.zeros((m * n, m * n))
# # put D in A diagonal
# for i in range(n):
#     A[i:i + m, i:i + m] = D
#
# # put -I's in A
# for i in range(n - 1):
#     A[i * m:i * m + m, (i + 1) * m:(i + 1) * m + m] = -Eye_matrix
#     A[(i + 1) * m:(i + 1) * m + m, i * m:i * m + m] = -Eye_matrix
# this way doesn't work! memory error!
# after search at internet, I understand I must use scipy.sparse for building matrices that use compressing methods.
```

اتفاقی که افتاد این بود که برنامه بدون هیچ اروری متوقف میشد : /

بعد از بررسی cpu و مموریم، به این نتیجه رسیدم که مموری کم میاره.

بنابراین بعد از سرچ کردن در اینترنت، فهمیدم که باید از پکیج scipy و لایبرری sparse استفاده کنم.

که کد ساختن ماتریس A در نهایت بدین صورت شد:

```python
def build_A_matrix(n, m):
    D = build_D_matrix(m)
    Eye_matrix = np.eye(m, m)
    A = sparse.lil_matrix((m * n, m * n))
    # put D in A diagonal
    for i in range(n):
        A[i * m:i * m + m, i * m:i * m + m] = D
    # put -I's in A
    for i in range(n - 1):
        A[i * m:i * m + m, (i + 1) * m:(i + 1) * m + m] = -Eye_matrix
        A[(i + 1) * m:(i + 1) * m + m, i * m:i * m + m] = -Eye_matrix

    return A


def build_D_matrix(m):
    D = sparse.lil_matrix((m, m))
    for i in range(m):
        for j in range(m):
            if i == j:
                D[i, j] = 4
            elif i == j + 1 or i == j - 1:
                D[i, j] = -1
    return D
```

که باتوجه به ساختاری که در ویکی پدیا آمده بود پیاده سازی کردم.

بعد از ساختن ماتریس A، آن را در تصویر سورس ضرب میکنم (دلیل آن را بالاتر گفتم)

مسئله بعدی این است که ما میخواهیم در خارج از ماسک، خروجی دقیقا همان تصویر تارگت ما باشد. بدین منظور باید A رو طوری تغییر بدیم، که در خارج از ماسک، همانند ماتریس همانی عمل کند. دقت کنیم که همانطور که گفتم هر سطر ماتریس A شامل کل تصویر است(یک بردار mn تایی) بنابراین برای تغییر قسمت های خارج از ماسک بدین صورت عمل میکنیم:

```python
def change_A(A, mask):
    """
    Outside the mask region, we want get target pixels as result. So we change these corresponding parts of A to
    I (identical matrix). So in Af = b equation, we will get target pixels for pixels that are outside the mask region.
    For this subject, we must change 4 on diagonal to 1 and -1s to 0 for each row of A matrix.

    :param A: the A matrix
    :param mask: our mask
    :return: changed A
    """
    height, width = mask.shape[:2]

    for y in range(1, height - 1):
        for x in range(1, width - 1):
            if mask[y, x] == 0:
                k = y * width
                k += x
                A[k, k] = 1
                A[k, k + width] = 0
                A[k, k - width] = 0
                A[k, k + 1] = 0
                A[k, k - 1] = 0

    # Compress matrix A
    A = A.tocsc()
    return A
```

همینطور در هنگام محاسبه ماتریس A.dot(S) (محاسبه b در معادله Af=b) باید به این موضوع دقت شود که میخواهیم قسمت های خارج از ماسک دقیقا عکس تارگت ما باشد. بدین منظور در تابع solve_equation این کار را کردم:

```python
# outside the mask, we want b pixels is exactly like target image
b[mask_flat == 0] = target_flat[mask_flat == 0]
```

در نهایت باید برای هرکدام از چنل ها معادله Af=b را حل کنیم و ریزالت را خروجی دهیم.