Lab 1: MAP Image Restoration

Course Title: Image Processing II (Fall 2021)

Course Number: ECE 69400

Instructor: Prof. Charles A. Bouman

Author: **Zhankun Luo**

Question 3.2.3

2.1 MAP Estimation with Gaussian Priors

Question 2.1.1

Show that for the distribution of (2)

$$p(x) = \frac{1}{z(g,p,\sigma)} \exp\left\{ -\frac{1}{p\sigma^p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j} |x_i - x_j|^p \right\}$$

the normalizing constant has the form $z(g,p,\sigma) = z(g,p,1)\sigma^N$ where N is the number of lattice points (i.e. pixels).

solution

Let $\sigma = 1$ in (2), then we get the probability distribution

$$p(x) \mid_{\sigma=1} = \frac{1}{z(g,p,1)} \exp\left\{ -\frac{1}{p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j} |x_i - x_j|^p \right\}$$

Then we must have

$$1 = \int p(x) \mid_{\sigma=1} \prod_{i=1}^{N} \mathrm{d}x_i \Rightarrow$$

$$z(g,p,1) = \int \exp\left\{ -\frac{1}{p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j} |x_i - x_j|^p \right\} \prod_{i=1}^{N} \mathrm{d}x_i$$

$$1 = \int p(x) \mathrm{d}x \Rightarrow$$

$$\begin{aligned} \frac{z(g,p,\sigma)}{\sigma^N} &= \int \exp\left\{ -\frac{1}{p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j} \left| \frac{x_i - x_j}{\sigma} \right|^p \right\} \prod_{i=1}^{N} \mathrm{d}\frac{x_i}{\sigma} \\ &= \int \exp\left\{ -\frac{1}{p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j} |x_i - x_j|^p \right\} \prod_{i=1}^{N} \mathrm{d}x_i \end{aligned}$$

Compare them, we have

$$\frac{z(g,p,\sigma)}{\sigma^N} = z(g,p,1)$$

Thus, we prove $z(g,p,\sigma) = z(g,p,1)\sigma^N$

Question 2.1.2

Use the result of equation (5) to derive the ML estimate of σ^p shown in (12).

$$p(x) = \frac{1}{z(g,p,1)\sigma^N} \exp\left\{ -\frac{1}{p\sigma^p} \sum_{\{i,j\}\in\mathcal{C}} g_{i,j}|x_i - x_j|^p \right\}$$

solution

Write down the log likelihood function

$$\log\left(p(x)\right) = -\frac{N}{p}\log\left(\sigma^p\right) - \frac{\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}|x_i - x_j|^p}{p}\frac{1}{\sigma^p} - \log\left(z(g,p,1)\right)$$

Then, Let the derivative to be zero

$$\frac{\partial \log\left(p(x)\right)}{\partial \sigma^p} \Big|_{\sigma^p=\hat{\sigma}^p} = -\frac{N}{p\hat{\sigma}^p}\left[1 - \frac{\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}|x_i - x_j|^p}{N}\frac{1}{\hat{\sigma}^p}\right] = 0$$

So, we prove that (12)

$$\hat{\sigma}^p = \frac{1}{N}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}|x_i - x_j|^p$$

Question 2.1.3

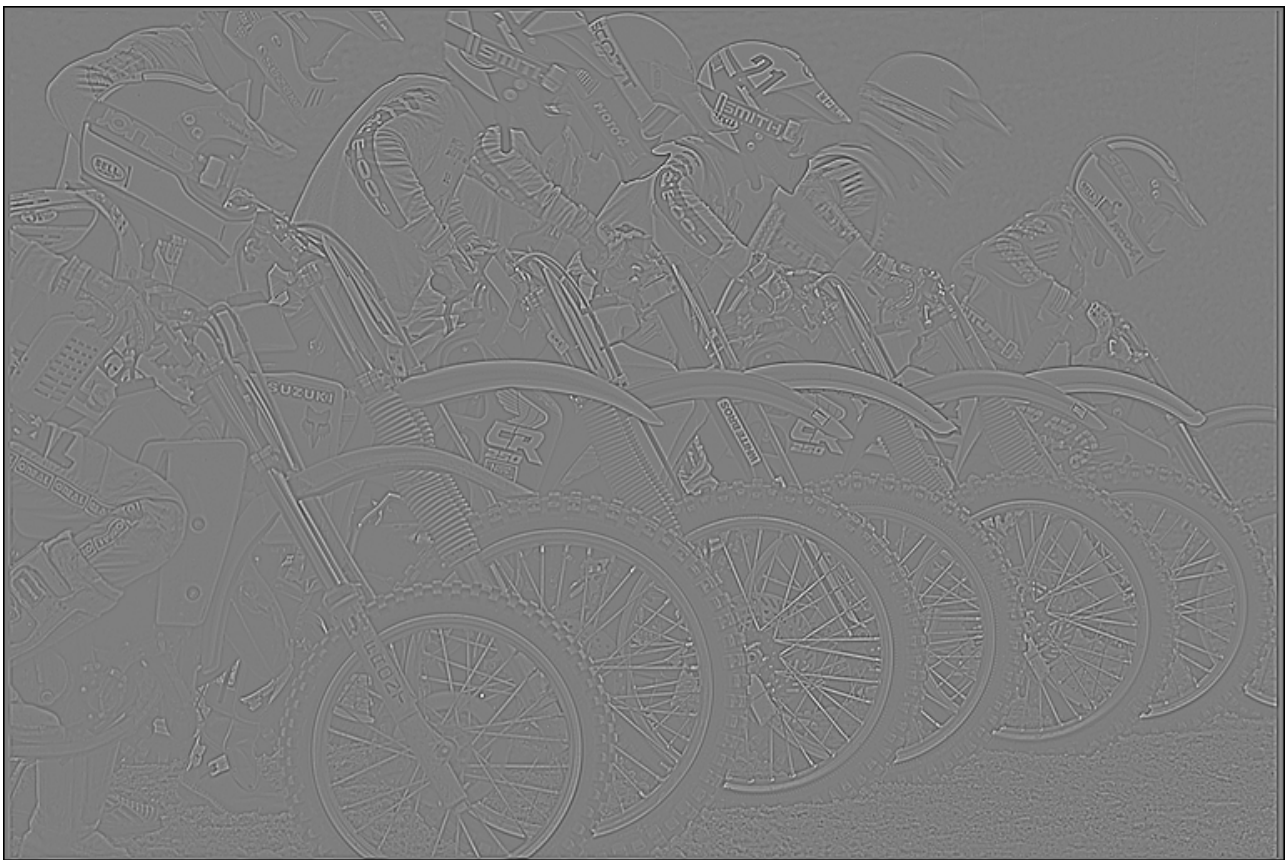
Compute the noncausal prediction error for the image img04.tif (should be **img04g.tif**)

$$e_i = x_i - \sum_{j \in \partial i} g_{i-j} x_j$$

and display it as an image by adding an offset of 127 to each pixel. Clip any value which is less than 0 or greater than 255 after adding the offset of 127.

solution

The causal prediction error + 127 is displayed below:



By running the following program **soln_2_1_3** with

```
./soln_2_1_3 img04g.tif
```

The corresponding function in **map.c** is

```
void compute_predict_error( struct TIFF_img *img, \
                            struct TIFF_img *img_error ) {
    int32_t **img_in,**img_out;
    int16_t W, H;
    /* allocate memory */
    W = img->width; H = img->height;
    get_TIFF( img_error, H, W, 'g' );
    img_in  = (int32_t **)get_img(W, H, sizeof(int32_t));
    img_out = (int32_t **)get_img(W, H, sizeof(int32_t));
    /* copy to array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_in[i][j] = img->mono[i][j];
        }
    }
    /* compute the error for each pixel of input image */
    for (int16_t j = 0; j < W; j++) { img_out[0][j]    =0; }
    for (int16_t j = 0; j < W; j++) { img_out[H-1][j]  =0; }
    for (int16_t i = 0; i < H; i++) { img_out[i][0]    =0; }
    for (int16_t i = 0; i < H; i++) { img_out[i][W-1]  =0; }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_out[i][j] = 12*img_in[i][j] \
                - 2*img_in[i][j-1] - 2*img_in[i][j+1];
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_out[i][j] -= 2*img_in[i-1][j] \
                + img_in[i-1][j-1] + img_in[i-1][j+1];
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_out[i][j] -= 2*img_in[i+1][j] \
                + img_in[i+1][j-1] + img_in[i+1][j+1];
            img_out[i][j] /= 12;
            img_out[i][j] += 127;
            img_out[i][j] = (img_out[i][j] < 0)? \
                0: ((img_out[i][j] > 255)? 255: img_out[i][j]);
```

```
        }
    }
    /* assign the error + 127 to the output image */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_error->mono[i][j] = img_out[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_in );
    free_img( (void**)img_out );
}
```

Question 2.1.4

Compute $\hat{\sigma}_{ML}$ the ML estimate of the scale parameter σ for values of p in the range $0.1 \leq p \leq 2$. Do not include cliques that fall across boundaries of the image.

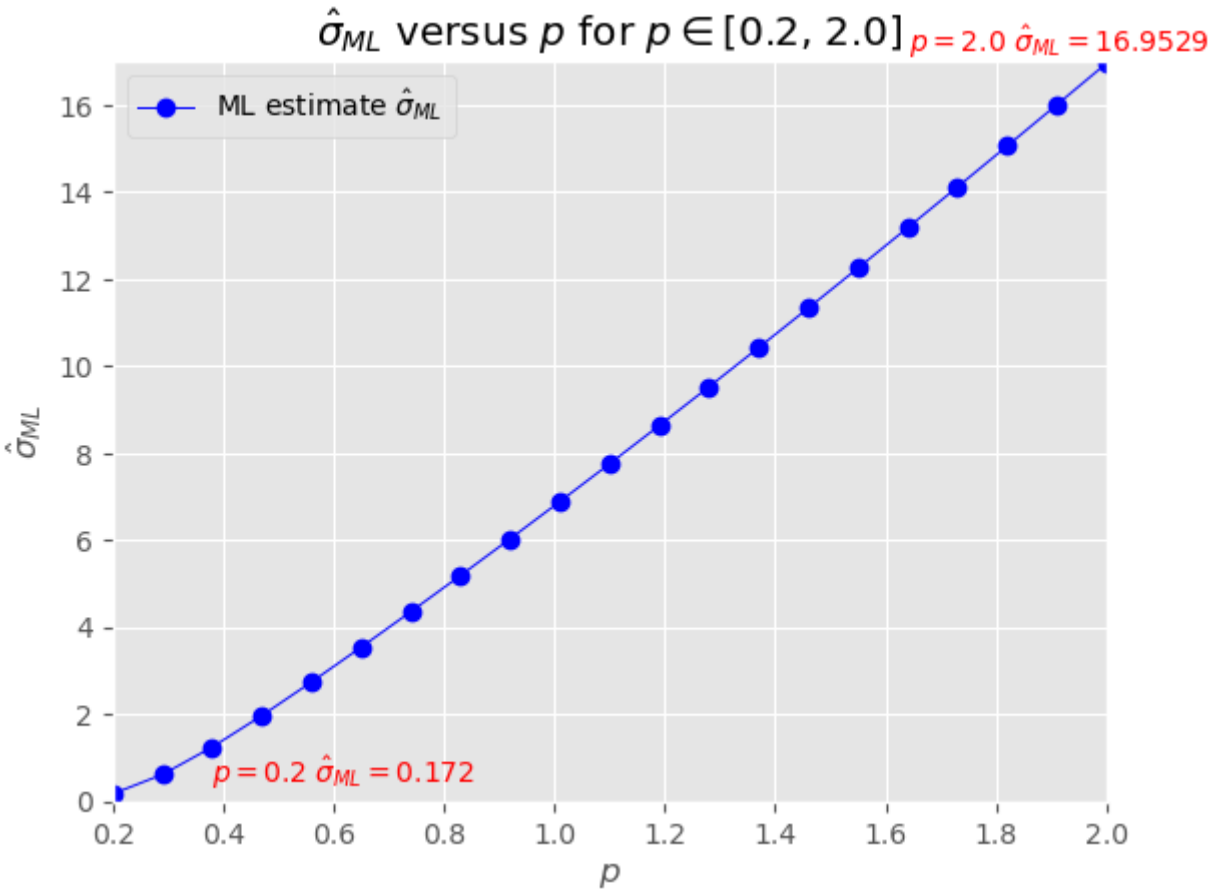
Plot $\hat{\sigma}_{ML}$ (not $\hat{\sigma}^{p}_{ML}$) versus p for p ranging from 0.1 to 2.

solution

By running the following program **soln_2_1_4** with, to export the computed prediction errors to **soln_2_1_4.csv**

```
./soln_2_1_4 img04g.tif
```

Then visualize the results of prediction errors with python script **vis_2_1_4.py**



The corresponding function in **map.c** is

```
double estimate_sigma_x ( struct TIFF_img *img, double p ) {
    int32_t **img1,**img2, N;
    int16_t W, H;
    double sigma_x = 0;
    /* allocate memory */
    W = img->width; H = img->height; N = W * H;
    img1 = (int32_t **)get_img(W, H, sizeof(int32_t));
    img2 = (int32_t **)get_img(W, H, sizeof(int32_t));
    /* copy to double array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img1[i][j] = img->mono[i][j];
            img2[i][j] = img->mono[i][j];
        }
    }
    /* computer the ML estimate for sigma_x */
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W; j++ ) {
            sigma_x+= pow(fabs(img1[i][j]-img1[i][j-1]), p)/(6*N);
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            sigma_x+= (2*pow(fabs(img1[i][j]-img2[i-1][j]), p)\
                + pow(fabs(img1[i][j]-img2[i-1][j-1]), p) \
                + pow(fabs(img1[i][j]-img2[i-1][j+1]), p))/(12*N);
        }
    }
    for (int16_t j = 1; j < W-1; j++ ) {
        sigma_x+= (2*pow(fabs(img1[H-2][j]-img2[H-1][j]), p) \
            + pow(fabs(img1[H-2][j]-img2[H-1][j-1]), p) \
            + pow(fabs(img1[H-2][j]-img2[H-1][j+1]), p))/(12*N);
    }
    /* free up the allocateed memory */
    free_img( (void**)img1 );
    free_img( (void**)img2 );
    return pow(sigma_x, 1/p);
}
```

The visualization code **vis_2_1_4.py** is

```
import matplotlib.pyplot as plt
import csv
from os.path import join, abspath, dirname
from math import floor, ceil
if __name__ == "__main__":
    list_pow, list_sigma = [], []
    path_csv = join(dirname(dirname(abspath(__file__))),
"bin/soln_2_1_4.csv")
    with open(path_csv, newline='') as file_csv:
        reader = csv.DictReader(file_csv, delimiter=',')
        for row in reader:
            list_pow.append(float(row["pow"]))
            list_sigma.append(float(row["sigma"]))
    plt.plot(list_pow, list_sigma, 'bo-',\
        label=r'ML estimate $\hat{\sigma}_{ML}$', lw=0.7)
    plt.xlabel(r'$p$')
    plt.ylabel(r'${\hat{\sigma}}_{ML}$', multialignment='center')
    plt.title(r"$\hat{\sigma}_{ML}$ versus $p$ for $p\in$"\
        + f"[{min(list_pow)}, {max(list_pow)}]")
    plt.margins(0, 0)
    plt.xlim([min(list_pow), max(list_pow)])
    ylim_min,ylim_max=floor(min(list_sigma)),ceil(max(list_sigma))
    plt.ylim([ylim_min, ylim_max])
    plt.text(0.2*min(list_pow)+0.8*max(list_pow), max(list_sigma),\
            r'$p=$'+f'{max(list_pow)}'+r' $\hat{\sigma}_{ML}=$'\
            + str(round(max(list_sigma), 4)),\
            ha='left', va='bottom', color='r')
    plt.text(0.9*min(list_pow)+0.1*max(list_pow), min(list_sigma),\
            r'$p=$'+f'{min(list_pow)}'+r' $\hat{\sigma}_{ML}=$'\
            + str(round(min(list_sigma), 4)),\
            ha='left', va='bottom', color='r')
    plt.grid(True)
    plt.legend()
    path_save
=join(dirname(dirname(abspath(__file__))),"bin/soln_2_1_4.png")
    plt.savefig(path_save,
                bbox_inches='tight',
                pad_inches=0)
    plt.show()
```

2.2. MAP Estimation with Gaussian Priors

Question 2.2.1

Show that the cost function of (15) is strictly convex

solution

Find the Hessian derivative of $c(x)$ with respect to x_i

$$\frac{\partial c}{\partial x_i \partial x_j} = \frac{1}{\hat{\sigma}_W^2} I + \frac{1}{\sigma_x^2} B$$

Since both I, B are positive definite, so the Hessian of $c(x)$ is also positive definite

So, the (15) is strictly convex

Question 2.2.2

Derive equations (17) and (18) for the joint MAP estimation of x and σ_W^2

$$c(x) = \frac{1}{2\sigma_W^2}\sum_{i\in S}(y_i - x_i)^2 + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}(x_i - x_j)^2$$

solution

write down the negative likelihood function $-\log\left(p(x \mid y)\right) = c(x) - \text{const}$ with respect to x

let the derivative with respect to $\sigma_W^2, \hat{x}$ to be 0,

$$-\frac{\partial \log\left(p(x \mid y)\right)}{\partial \sigma_W^2}\ |_{\sigma_W^2=\hat{\sigma}_W^2, x=\hat{x}} = -\frac{1}{2(\hat{\sigma}_W^2)^2}\sum_{i\in S}\left(y_i - \hat{x}_i\right)^2 + \frac{N}{2\hat{\sigma}_W^2} = 0$$

$$\begin{aligned}
&-\frac{\partial \log\left(p(x \mid y)\right)}{\partial x_i}\ |_{\sigma_W^2=\hat{\sigma}_W^2, x=\hat{x}} = \frac{\partial c(x)}{\partial x_i}\ |_{\sigma_W^2=\hat{\sigma}_W^2, x=\hat{x}} \\
&= -\frac{1}{\hat{\sigma}_W^2}\left(y_i - \hat{x}_i\right) + \frac{1}{\sigma_x^2}\sum_{j\in\partial i} g_{i,j}\hat{x}_j \\
&= -\frac{N\left(y_i - \hat{x}_i\right)}{\sum_{i\in S}\left(y_i - \hat{x}_i\right)^2} + \frac{1}{\sigma_x^2}\sum_{j\in\partial i} g_{i,j}\hat{x}_j
\end{aligned}$$

The second equation is too complex for $\hat{x}_i, i = 1, \cdots, N$, to give the close form solution

From the first equation, we solve the $\hat{\sigma}_W^2$ as follows (17), and derive the equation (18)

$$\hat{\sigma}_W^2 = \frac{1}{N}\sum_{i\in S}\left(y_i - \hat{x}_i\right)^2$$

We assume the second equation the derivative of new $c(x)$ with respect to x_i at $x_i = \hat{x}_i, i = 1, \cdots, N$ (note $c(x)$ only depends on x, NOT depends on σ_W^2)

$$\frac{\partial c}{\partial x_i} = -\frac{N\left(y_i - x_i\right)}{\sum_{i\in S}\left(y_i - x_i\right)^2} + \frac{1}{\sigma_x^2}\sum_{j\in\partial i} g_{i,j}x_j$$

We can easily verify that such $c(x)$ of (19) satisfy the equations for derivatives above

$$c(x) = \frac{N}{2}\log\left(\sum_{i\in S}(y_i - x_i)^2\right) + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}(x_i - x_j)^2$$

and when $x = \hat{x}$, $\frac{\partial c}{\partial x_i}|_{x=\hat{x}} = 0$ and the new $c(x)$ reaches the minimal $c(x)|_{x=\hat{x}}$, then we prove (18)

$$\hat{x} = \arg\min_{x} \left\{ \frac{N}{2} \log \left(\sum_{i \in S} (y_i - x_i)^2 \right) + \frac{1}{2\sigma_x^2} \sum_{\{i,j\} \in \mathcal{C}} g_{i,j} (x_i - x_j)^2 \right\}$$

So, we prove (17), (18) above

Question 2.2.3

Does the application of update (20) followed by (21) reduce the cost function of (19)? Why?

$$c(x) = \frac{N}{2}\log\left(\sum_{i\in S}(y_i - x_i)^2\right) + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}(x_i - x_j)^2$$

$$\hat{\sigma}_W^2 \leftarrow \frac{1}{N}\sum_{i\in S}\left(y_i - \hat{x}_i\right)^2$$

$$\hat{x} \leftarrow \arg\min_x \left\{ \frac{1}{2\hat{\sigma}_W^2}\sum_{i\in S}\left(y_i - \hat{x}_i\right)^2 + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}\left(\hat{x}_i - \hat{x}_j\right)^2 \right\}$$

solution

Let's consider the $c(x)$ values for k-th iteration and $(k+1)$-th iteration

Denote the value of $\hat{x}, \hat{\sigma}_W^2$ at $k, (k+1)$-th iterations by $\hat{x}^{(k)}, \hat{x}^{(k+1)}$ and $\hat{\sigma}_W^2{}^{(k)}, \hat{\sigma}_W^2{}^{(k+1)}$

We introduce the Auxiliary function $\Delta(x, \sigma_W^2), f(x, \sigma_W^2)$as follows:

$$\Delta(x, \sigma_W^2) \equiv \frac{N}{2}\left[\frac{\frac{1}{N}\sum_{i\in S}(y_i - x_i)^2}{\sigma_W^2} - 1 - \log\left(\frac{\frac{1}{N}\sum_{i\in S}(y_i - x_i)^2}{\sigma_W^2}\right)\right]$$

$$\begin{aligned}
&f(x, \sigma_W^2) \equiv c(x) + \Delta(x, \sigma_W^2) \\
=&\frac{N}{2}\log\left(\sum_{i\in S}(y_i - x_i)^2\right) + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}(x_i - x_j)^2 \\
+&\frac{1}{2\sigma_W^2}\sum_{i\in S}(y_i - x_i)^2 - \frac{N}{2}\log\left(\sum_{i\in S}(y_i - x_i)^2\right) + \frac{N}{2}\log\left(\sigma_W^2\right) \\
-&\frac{N}{2} + \frac{N}{2}\log(N) \\
=&\left\{\frac{1}{2\sigma_W^2}\sum_{i\in S}(y_i - x_i)^2 + \frac{1}{2\sigma_x^2}\sum_{\{i,j\}\in\mathcal{C}} g_{i,j}(x_i - x_j)^2\right\} \\
+&\left[\frac{N}{2}\log\left(\sigma_W^2\right) - \frac{N}{2} + \frac{N}{2}\log(N)\right]
\end{aligned}$$

Notice that $t - 1 - \log(t) \geq 0, \forall t > 0$, and only when $t = 1, t - 1 - \log(t) = 0$

So, $\Delta(x, \sigma_W^2) \geq 0$, only when $\sigma_W^2 = \frac{1}{N}\sum_{i\in S}(y_i - x_i)^2$, we have $\Delta(x, \sigma_W^2)$=0

step 1: fix $\sigma_W^2 = \hat{\sigma}_W^{2\,(k+1)}$, find x for the minimal value of $f(x, \sigma_W^2)$

$$\begin{aligned}\hat{x}^{(k+1)} &\equiv \arg\min_x f(x, \hat{\sigma}_W^{2\,(k+1)}) \\ &= \arg\min_x \left\{ \frac{1}{2\hat{\sigma}_W^{2\,(k+1)}} \sum_{i \in S} (y_i - x_i)^2 + \frac{1}{2\sigma_x^2} \sum_{\{i,j\} \in \mathcal{C}} g_{i,j}(x_i - x_j)^2 \right\}\end{aligned}$$

Notice that $\hat{\sigma}_W^{2\,(k+1)} = \frac{1}{N} \sum_{i \in S} \left(y_i - \hat{x}_i^{(k)} \right)^2$, so we have $\Delta(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)})$=0

$$f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) = c(\hat{x}^{(k)}) + \Delta(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) = c(\hat{x}^{(k)})$$

Thus

$$f(\hat{x}^{(k+1)}, \hat{\sigma}_W^{2\,(k+1)}) \leq f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) = c(\hat{x}^{(k)})$$

step 2: fix $x = \hat{x}^{(k)}$, find σ_W^2 for the minimal value of $f(x, \sigma_W^2)$

$$\begin{aligned}\hat{\sigma}_W^{2\,(k+1)} &\equiv \arg\min_x f(x, \hat{\sigma}_W^{2\,(k)}) \equiv \arg\min_x \Delta(x, \hat{\sigma}_W^{2\,(k)}) \\ &= \frac{1}{N} \sum_{i \in S} \left(y_i - \hat{x}_i^{(k)} \right)^2\end{aligned}$$

And notice that $\Delta(\hat{x}^{(k)}, \sigma_W^2) \geq 0$, only when $\sigma_W^2 = \hat{\sigma}_W^{2\,(k+1)} = \frac{1}{N} \sum_{i \in S} \left(y_i - \hat{x}_i^{(k)} \right)^2$, we have $\Delta(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)})$=0

$$\begin{aligned}c(\hat{x}^{(k)}) &\equiv f(\hat{x}^{(k)}, \sigma_W^2) - \Delta(\hat{x}^{(k)}, \sigma_W^2) \\ &= f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k)}) - \Delta(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k)}) \\ &= f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) - \Delta(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) \\ &= f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)})\end{aligned}$$

Thus, we have

$$f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) = c(\hat{x}^{(k)}) \leq f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k)})$$

In the end, combine **step (1), (2)**, we proved that

$$f(\hat{x}^{(k+1)}, \hat{\sigma}_W^{2\,(k+1)}) \leq f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k+1)}) = c(\hat{x}^{(k)}) \leq f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k)})$$

So, we have

$$c(\hat{x}^{(k+1)}) \leq f(\hat{x}^{(k+1)}, \hat{\sigma}_W^{2\,(k+1)}) \leq c(\hat{x}^{(k)}) \leq f(\hat{x}^{(k)}, \hat{\sigma}_W^{2\,(k)}) \leq c(\hat{x}^{(k-1)})$$

We prove that the application of update (20), (21) reduce the cost function $c(x)$ of (19)

Question 2.2.4

Let $\left(\hat{x}, \hat{\sigma}_W^2\right)$ denote the exact solution to the optimization of equation (16). Is $\hat{\sigma}_W^2$ the ML estimate of σ_W^2 ? Is it the MAP estimate? Justify your answer

$$\begin{aligned}\left(\hat{x}, \hat{\sigma}_W^2\right) &= \arg\max_{x, \sigma_W^2} \left\{p_{X|Y}\left(x \mid y, \sigma_W^2\right)\right\} \\ &= \arg\max_{x, \sigma_W^2} \left\{\log p\left(y \mid x, \sigma_W^2\right) + \log p(x)\right\}\end{aligned}$$

solution

Let's consider ML estimate

the MAP estimate $\breve{\sigma}_W^2$ should be, since x and σ_W^2 are independent

$$\breve{\sigma}_W^2 = \arg\max_{\sigma_W^2} \left\{p_{\sigma_W^2}\left(\sigma_W^2\right)\right\}$$

So, we conclude: $\hat{\sigma}_W^2$ is **NOT** ML estimate

$$\hat{\sigma}_W^2 \neq \breve{\sigma}_W^2$$

Let's consider MAP estimate

the MAP estimate $\tilde{\sigma}_W^2$ should be, since x and σ_W^2 are independent

$$\begin{aligned}\tilde{\sigma}_W^2 &= \arg\max_{\sigma_W^2} \left\{p_{\sigma_W^2|X,Y}\left(\sigma_W^2 \mid x, y\right)\right\} \\ &= \arg\max_{\sigma_W^2} \left\{\log p\left(y \mid x, \sigma_W^2\right) + \log p(x) + \log p(\sigma_W^2) - \log\left(p(x, y)\right)\right\} \\ &= \arg\max_{\sigma_W^2} \left\{\log p\left(y \mid x, \sigma_W^2\right) + \log p(x) + \log p(\sigma_W^2)\right\}\end{aligned}$$

We have to add one more one more term $\log\left(p(\sigma_W^2)\right)$ during the MAP estimation, compared with (16)

So, we conclude: $\hat{\sigma}_W^2$ is **NOT** MAP estimate

$$\hat{\sigma}_W^2 \neq \tilde{\sigma}_W^2$$

2.3. MAP Estimation with Gaussian Priors

Question 2.3.1

Show that the costs resulting from ICD updates forms a monotone decreasing sequence that is bounded below.

$$c(x) = \frac{1}{2\sigma_W^2} \sum_{i \in S} (y_i - x_i)^2 + \frac{1}{2\sigma_x^2} \sum_{\{i,j\} \in \mathcal{C}} g_{i,j} (x_i - x_j)^2$$

Treat σ_W^2 as a constant in (15) cost function $c(x)$, then update $x_i \leftarrow x_i^{(k)}$ as follows, where the updated value x_i in the k-th iteration is denoted by $x_i^{(k)}$

$$x_i^{(k)} = \max \left\{ 0, \arg\min_{x_i} c\left(\{x_j^{(k)} \mid j < i\}, x_i, \{x_j^{(k-1)} \mid i < j\} \right) \right\}$$

$$= \max \left\{ 0, \arg\min_{x_i} \left\{ \frac{(y_i - x_i)^2}{2\sigma_W^2} + \frac{\sum_{\substack{j \in \partial i \\ j<i}} g_{i,j} \left(x_i - x_j^{(k)} \right)^2 + \sum_{\substack{j \in \partial i \\ i<j}} g_{i,j} \left(x_i - x_j^{(k-1)} \right)^2}{2\sigma_x^2} \right\} \right\}$$

$$= \max \left\{ 0, \frac{y_i + \frac{\sigma_W^2}{\sigma_x^2} \left[\sum_{\substack{j \in \partial i \\ j<i}} g_{i,j} x_j^{(k)} + \sum_{\substack{j \in \partial i \\ i<j}} g_{i,j} x_j^{(k-1)} \right]}{1 + \frac{\sigma_W^2}{\sigma_x^2}} \right\}$$

solution

Let's denote the cost function with the latest $x_i^{(k)}$ values in the k-th iteration by $c_i^{(k)}$

$$c_i^{(k)} \equiv c\left(\{x_j^{(k)} \mid j < i\}, x_i^{(k)}, \{x_j^{(k-1)} \mid i < j\} \right) = c\left(\{x_j^{(k)} \mid j \leq i\}, \{x_j^{(k-1)} \mid i < j\} \right)$$

Our goal is to prove following statements:

(i) $c_i^{(k)}$ is a monotone decreasing sequence for any fixed k-th iteration

$$c_N^{(k)} \leq c_{N-1}^{(k)} \leq \cdots \leq c_i^{(k)} \leq c_{i-1}^{(k)} \leq \cdots \leq c_1^{(k)}$$

(ii) $\{c_i^{(k)}\}_{i=1}^N$ is smaller than $\{c_i^{(k-1)}\}_{i=1}^N$

$$c_1^{(k)} \leq c_N^{(k-1)}$$

(iii) For any k, i , the value of $c_i^{(k)}$ is bounded below, with low bound L

$$c_i^{(k)} \geq L$$

Let's start to prove (i), (ii), (iii)

(i)(ii) consider $c_{i-1}^{(k)}$and $c_i^{(k)}$

case 1: $$\frac{y_i + \frac{\sigma_W^2}{\sigma_x^2}\left[\sum_{\substack{j \in \partial i \\ j<i}} g_{i,j} x_j^{(k)} + \sum_{\substack{j \in \partial i \\ i<j}} g_{i,j} x_j^{(k-1)}\right]}{1 + \frac{\sigma_W^2}{\sigma_x^2}} \geq 0$$

$$x_i^{(k)} = \arg\min_{x_i} c\left(\{x_j^{(k)} \mid j < i\}, x_i, \{x_j^{(k-1)} \mid i < j\}\right)$$

So, we have

$$c_i^{(k)} = c\left(\{x_j^{(k)} \mid j < i\}, x_i^{(k)}, \{x_j^{(k-1)} \mid i < j\}\right) \leq c\left(\{x_j^{(k)} \mid j < i\}, x_i, \{x_j^{(k-1)} \mid i < j\}\right)$$

Let the right hand side $x_i = x_i^{(k-1)}$

$$c_i^{(k)} \leq c\left(\{x_j^{(k)} \mid j < i\}, x_i^{(k-1)}, \{x_j^{(k-1)} \mid i < j\}\right) = c_{i-1}^{(k)}$$

Similarly, if $i = 1$ for $c_i^{(k)}$, we proved that under cases 1

$$c_1^{(k)} \leq c_N^{(k-1)}$$

case 2: $$\frac{y_i + \frac{\sigma_W^2}{\sigma_x^2}\left[\sum_{\substack{j \in \partial i \\ j<i}} g_{i,j} x_j^{(k)} + \sum_{\substack{j \in \partial i \\ i<j}} g_{i,j} x_j^{(k-1)}\right]}{1 + \frac{\sigma_W^2}{\sigma_x^2}} < 0$$

$$x_i^{(k)} = 0$$

Then we verify that $c\left(\{x_j^{(k)} \mid j < i\}, x_i, \{x_j^{(k-1)} \mid i < j\}\right)$ is strictly increasing for $x_i \geq 0$

$$\frac{\mathrm{d}c\left(\{x_j^{(k)} \mid j<i\}, x_i, \{x_j^{(k-1)} \mid i<j\}\right)}{\mathrm{d}x_i}$$

$$= \frac{\sigma_x^2+\sigma_W^2}{\sigma_x^2\sigma_W^2}\left\{ x_i - \frac{y_i + \frac{\sigma_W^2}{\sigma_x^2}\left[\sum_{\substack{j\in\partial i\\ j<i}} g_{i,j}x_j^{(k)} + \sum_{\substack{j\in\partial i\\ i<j}} g_{i,j}x_j^{(k-1)}\right]}{1+\frac{\sigma_W^2}{\sigma_x^2}} \right\} > 0$$

So, we have

$$x_i^{(k)} = 0 \le x_i^{(k-1)}$$

$$c_i^{(k)} = c\left(\{x_j^{(k)} \mid j \le i\}, \{x_j^{(k-1)} \mid i<j\}\right) \le c\left(\{x_j^{(k)} \mid j<i\}, \{x_j^{(k-1)} \mid i \le j\}\right) = c_{i-1}^{(k)}$$

Similarly, if $i=1$ for $c_i^{(k)}$, we proved that under cases 2

$$c_1^{(k)} \le c_N^{(k-1)}$$

Combine case (1)(2), we prove (i)(ii), thus we prove that the costs resulting from ICD updates forms a monotone decreasing sequence

$$c_N^{(k)} \le c_{N-1}^{(k)} \le \cdots \le c_i^{(k)} \le c_{i-1}^{(k)} \le \cdots \le c_1^{(k)}$$

$$c_1^{(k)} \le c_N^{(k-1)}$$

$\{c_i^{(k)}\}_{i=1}^N$ is smaller than $\{c_i^{(k-1)}\}_{i=1}^N$

And notice that $\sigma_x^2, \sigma_w^2 > 0, g_{i,j} \ge 0$

$$\begin{aligned} c_i^{(k)} &\equiv c\left(\{x_j^{(k)} \mid j \le i\}, \{x_j^{(k-1)} \mid i<j\}\right) \\ &= \frac{\left(y_i - x_i^{(k)}\right)^2}{2\sigma_W^2} + \frac{\sum\limits_{\substack{j\in\partial i\\ j<i}} g_{i,j}\left(x_i^{(k)} - x_j^{(k)}\right)^2 + \sum\limits_{\substack{j\in\partial i\\ i<j}} g_{i,j}\left(x_i^{(k)} - x_j^{(k-1)}\right)^2}{2\sigma_x^2} \\ &\ge 0 \end{aligned}$$

So, $c_i^{(k)}$ is bounded below, with low bound $L=0$, we prove (iii)

Question 2.3.2

Show that any local minimum of the cost function of (15) is also a global minimum.

$$c(x) = \frac{1}{2\sigma_W^2} \sum_{i \in S} (y_i - x_i)^2 + \frac{1}{2\sigma_x^2} \sum_{\{i,j\} \in \mathcal{C}} g_{i,j}(x_i - x_j)^2$$

solution

Treat σ_W^2 as a constant in (15) cost function $c(x)$

Find the Hessian matrix of $c(x)$ respect to x

$$\frac{\partial^2 c}{\partial x_i \partial x_j} = \frac{1}{\sigma_W^2} I + \frac{1}{\sigma_x^2} B$$

Where I and B are both positive definite, so the Hessian must be also positive definite

So, $c(x)$ is convex with respect to x for any $\forall x$

Thus, we prove that any local minimum of the cost function of (15) is also a global minimum for x

Question 2.3.3

use the monochrome image img04.tif as x and produce a noisy image y by adding i.i.d. Gaussian noise with mean zero and $\sigma_W^2 = 16^2$. Approximate y by truncating the pixels to the range $[0, \cdots 255]$. Print out the image Y.

solution

By running the following program **soln_2_3_3** with, to export the noisy image to **soln_2_3_3.tif**

```
./soln_2_3_3 img04g.tif
```

The corresponding function in **map.c** is

```
void add_noise( struct TIFF_img *img, struct TIFF_img *img_noisy,
double sigma_w ) {
    int32_t **img_out;
    int16_t W, H;
    /* allocate memory */
    W = img->width; H = img->height;
    get_TIFF( img_noisy, H, W, 'g' );
    img_out = (int32_t **)get_img(W, H, sizeof(int32_t));
    /* copy to array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_out[i][j] = img->mono[i][j];
        }
    }
    /* Set seed for random noise generator */
    srandom2(1);
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_out[i][j] += round(sigma_w * normal());
            img_out[i][j] = (img_out[i][j] < 0)? \
                        0: ((img_out[i][j] > 255)? \
                        255: img_out[i][j]);
        }
    }
    /* assign the noisy array to the output image */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_noisy->mono[i][j] = img_out[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_out );
}
```

Question 2.3.4

Compute the MAP estimate of X using 20 iterations of ICD optimization. Use $\sigma_x^2 = \hat{\sigma}_x^2$ the ML estimate of the scale parameter computed for $p = 2$, and $\sigma_W^2 = 16^2$. Print out the resulting MAP estimate.

solution

By running the following program **soln_2_3_4** with, to export the resulting MAP estimate image to **soln_2_3_4.tif** and the cost function to **soln_2_3_4.csv**

```
./soln_2_3_4 soln_2_3_3.tif
```

The corresponding function in **map.c** is

```
void iter_ICD_standard_quick(double **img_y, double **img_x, \
                             int16_t W, int16_t H,\
                             double sigma_w, double sigma_x) {
    double **img_t;
    double ratio = (sigma_x / sigma_w)*(sigma_x / sigma_w);
    /* allocate memory */
    img_t = (double **)get_img(W, H, sizeof(double));
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_t[i][j] = 2*img_x[i][j-1] \
                + 2*img_x[i][j+1];
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_t[i][j] += 2*img_x[i-1][j] \
                + img_x[i-1][j-1] + img_x[i-1][j+1];
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_t[i][j] += 2*img_x[i+1][j] \
                + img_x[i+1][j-1] + img_x[i+1][j+1];
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_t[i][j] = (img_t[i][j] \
                          + 12*ratio*img_y[i][j]) / (12*(1+ratio));
            img_t[i][j] = (img_t[i][j] < 0)? \
                        0: img_t[i][j];
        }
    }
    /* assign update the image_x */
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            img_x[i][j] = img_t[i][j];
        }
    }
    /* de-allocate memory */
```

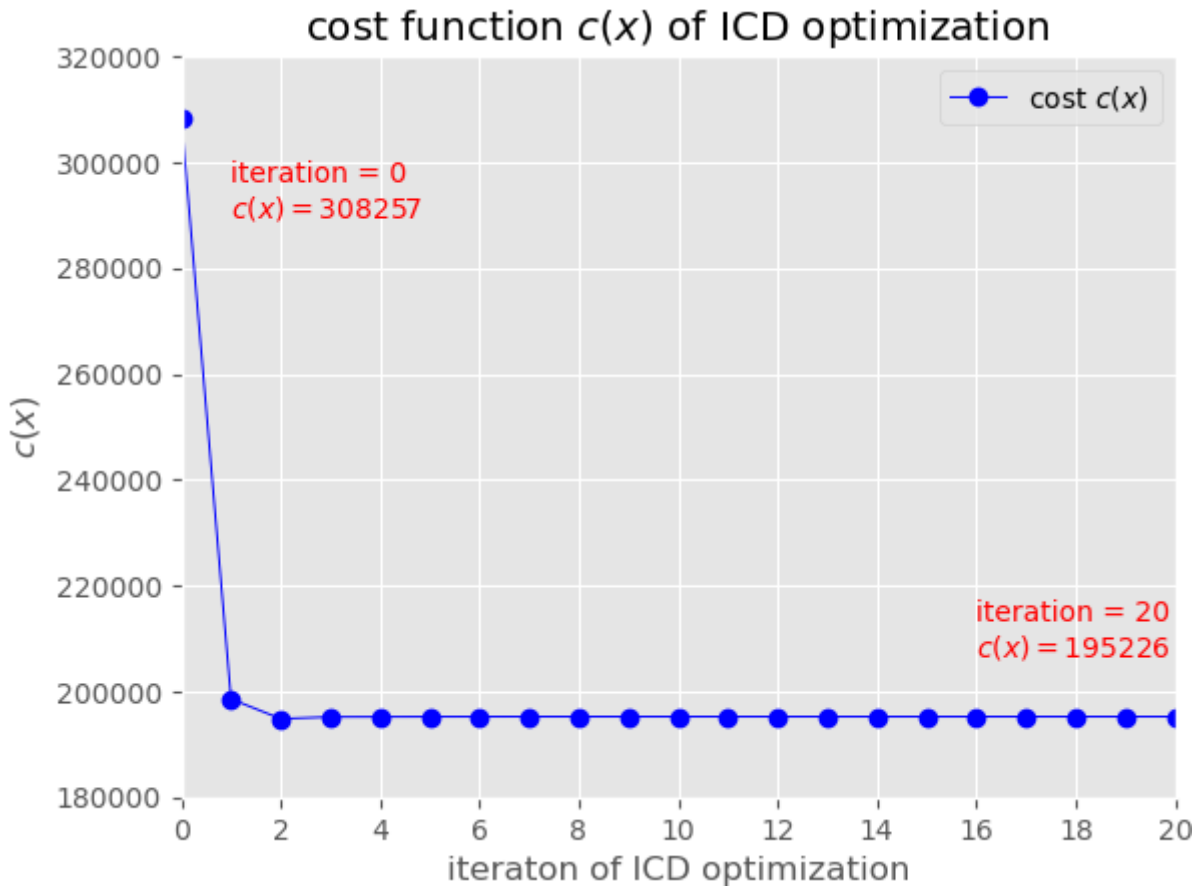
```
    free_img( (void**)img_t );
}
void clip_arr2img(double **array, struct TIFF_img *img) {
    int16_t W, H;
    W = img->width; H = img->height;
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img->mono[i][j] = round(array[i][j]);
            img->mono[i][j] = (img->mono[i][j] < 0)? \
                        0: ((img->mono[i][j] > 255)? \
                        255: img->mono[i][j]);
        }
    }
}
```

Question 2.3.5

Plot the cost function of (14) as a function of the iteration number for the experiment of step 4.

solution

Then visualize the results of cost functions in **soln_2_3_4.csv** with python script **vis_2_3_4.py**



The corresponding function in **map.c** is

```
double compute_cost(double **img_y, double **img_x, \
                    int16_t W, int16_t H,\
                    double sigma_w, double sigma_x) {
    double **img_t;
    double cost = 0.0;
    /* allocate memory */
    img_t = (double **)get_img(W, H, sizeof(double));
    /* copy to double array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
```

```
            img_t[i][j] = img_x[i][j];
        }
    }
    /* computer the cost function */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            double t = img_y[i][j]-img_x[i][j];
            cost +=  t*t;
        }
    }
    cost *= 12*(sigma_x / sigma_w)*(sigma_x / sigma_w);
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W; j++ ) {
            double t = img_x[i][j]-img_x[i][j-1];
            cost +=  t*t;
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            double r, s, t;
            r = img_x[i][j]-img_t[i-1][j];
            s = img_x[i][j]-img_t[i-1][j-1];
            t = img_x[i][j]-img_t[i-1][j+1];
            cost += (2*r*r + s*s + t*t);
        }
    }
    for (int16_t j = 1; j < W-1; j++ ) {
        double r, s, t;
        r = img_x[H-2][j]-img_t[H-1][j];
        s = img_x[H-2][j]-img_t[H-1][j-1];
        t = img_x[H-2][j]-img_t[H-1][j+1];
        cost += (2*r*r + s*s + t*t);
    }
    cost /= 2*12*sigma_x*sigma_x;
    /* free up the allocateed memory */
    free_img( (void**)img_t );
    return cost;
}
```

The visualization code **vis_2_3_4.py** is

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import csv
from os.path import join, abspath, dirname
from math import floor, ceil, log10
def plot_cost(path_csv, path_img):
    list_iter, list_cost = [], []
    path_csv_full = join(dirname(dirname(abspath(__file__))),
path_csv)
    with open(path_csv_full, newline='') as file_csv:
        reader = csv.DictReader(file_csv, delimiter=',')
        for row in reader:
            list_iter.append(int(row["iteration"]))
            list_cost.append(float(row["cost"]))
    plt.style.use('ggplot')
    ax = plt.figure().gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    plt.plot(list_iter, list_cost, 'bo-', \
        label=r'cost $c(x)$', lw=0.7)
    plt.xlabel('iteraton of ICD optimization')
    plt.ylabel(r'$c(x)$', multialignment='center')
    plt.title(r"cost function $c(x)$ of ICD optimization")
    plt.margins(0, 0)
    plt.xlim([min(list_iter), max(list_iter)])
    delta = 10 ** (floor(log10(max(list_cost)))-1)
    ylim_min, ylim_max = delta*(floor(min(list_cost)/delta)-1), \
                         delta*(ceil(max(list_cost)/delta)+1)
    plt.ylim([ylim_min, ylim_max])
    plt.text(0.2*list_iter[0]+0.8*list_iter[-1], \
             list_cost[-1]+delta,\
            'iteration = ' + f'{max(list_iter)}\n' + r'$c(x)=$' \
            + str(round(list_cost[-1])), \
            ha='left', va='bottom', color='r')
    plt.text(0.95*list_iter[0]+0.05*list_iter[-1], \
             list_cost[0]-2*delta,\
            'iteration = ' + f'{list_iter[0]}\n' + r'$c(x)=$' \
            + str(round(list_cost[0])), \
            ha='left', va='bottom', color='r')
    plt.grid(True)
    plt.legend()
```

```
    path_save = join(dirname(dirname(abspath(__file__))), path_img)
    plt.savefig(path_save,
                bbox_inches='tight',
                pad_inches=0)
    plt.show()

if __name__ == "__main__":
    path_csv1 = "bin/soln_2_3_4.csv"
    path_img1 = "bin/soln_2_3_4.png"
    plot_cost(path_csv1, path_img1)
```

Question 2.3.6

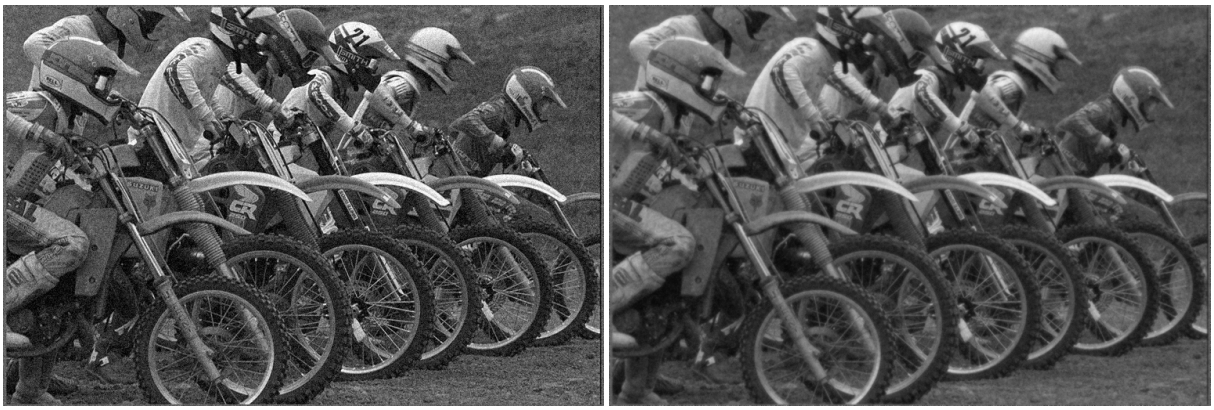
Repeat step 4 for $\sigma_x^2 = 5 \cdot \hat{\sigma}_x^2$, and $\sigma_x^2 = (1/5) \cdot \hat{\sigma}_x^2$

solution

By running the following program **soln_2_3_6** with, to export the resulting MAP estimate image to **soln_2_3_6a.tif, soln_2_3_6b.tif,** and the cost function to **soln_2_3_6a.csv, soln_2_3_6b.csv** for for $\sigma_x^2 = 5 \cdot \hat{\sigma}_x^2$, and $\sigma_x^2 = (1/5) \cdot \hat{\sigma}_x^2$ respectively

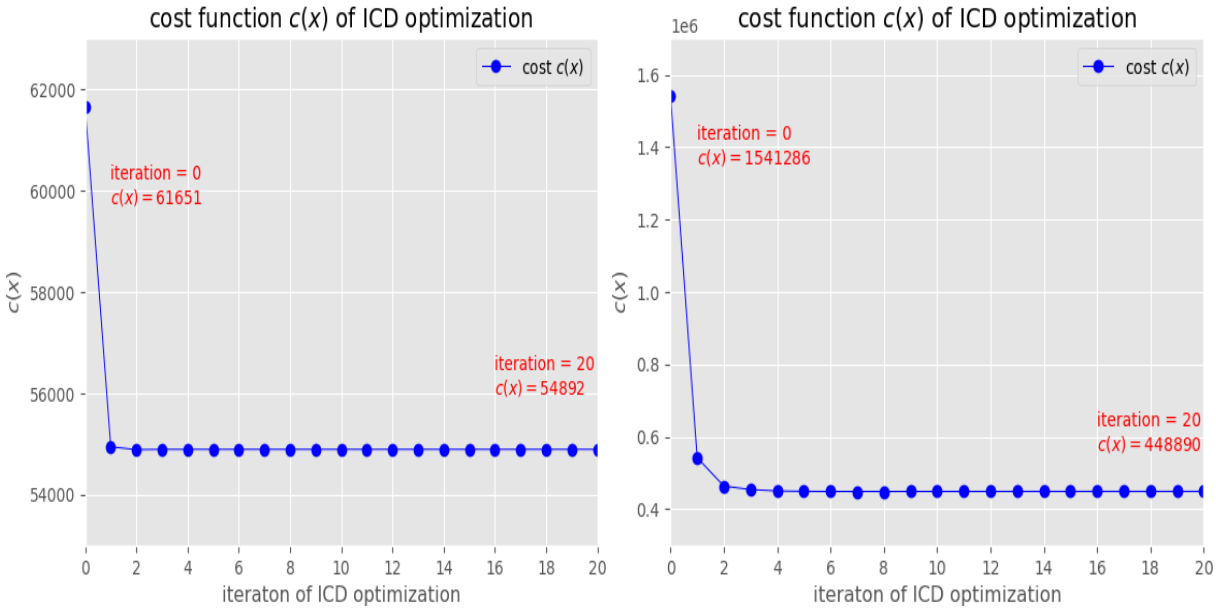
```
./soln_2_3_6 soln_2_3_3.tif
```

The left image is the resulting MAP estimate image for $\sigma_x^2 = 5 \cdot \hat{\sigma}_x^2$, and the right image is the one for $\sigma_x^2 = (1/5) \cdot \hat{\sigma}_x^2$



Then visualize the results of cost functions in **soln_2_3_6a.csv, soln_2_3_6b.csv** with python script **vis_2_3_6.py**

The left image is the cost functions for $\sigma_x^2 = 5 \cdot \hat{\sigma}_x^2$, and the right image is the one for $\sigma_x^2 = (1/5) \cdot \hat{\sigma}_x^2$



2.4. MAP Restoration from Blurred/Noisy Image with Gaussian Prior

Question 2.4.1

Use the monochrome image **img04.tif** as X and produce a blurred and noisy image Y. To do this, first apply the blurring filter of (22) with circular boundary conditions; then add noise with a variance of $\sigma_W^2 = 4^2$. Approximate Y by truncating the pixels to the range $[0, \cdots, 255]$. Print out the image Y

solution

By running the following program **soln_2_4_1** with, to export the blurred and noisy image to **soln_2_4_1.tif**

```
./soln_2_4_1 img04g.tif
```

The corresponding function in **map.c** is

```
void add_blur_noise( struct TIFF_img *img, \
                     struct TIFF_img *img_noisy, double sigma_w ) {
    int32_t **img_out, **img_in, **img_t;
    int16_t W, H;
    /* allocate memory */
    W = img->width; H = img->height;
    get_TIFF( img_noisy, H, W, 'g' );
    img_out = (int32_t **)get_img(W, H, sizeof(int32_t));
    img_in  = (int32_t **)get_img(W, H, sizeof(int32_t));
    img_t   = (int32_t **)get_img(W, H, sizeof(int32_t));
    /* copy to array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_out[i][j] = img->mono[i][j];
            img_in[i][j]  = img->mono[i][j];
            img_t[i][j]   = img->mono[i][j];
        }
    }
    /* blur the image */
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_t[i][j] *= 3;
            img_t[i][j] += 2*(img_in[i][j-1] + img_in[i][j+1])\
                        + img_in[i][j-2] + img_in[i][j+2];
        }
    }
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_out[i][j] *= 3;
            img_out[i][j] += 2*img_t[i-1][j];
        }
    }
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_out[i][j] += 2*img_t[i+1][j];
        }
    }
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_out[i][j] += img_t[i-2][j];
```

```
        }
    }
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_out[i][j] += img_t[i+2][j];
        }
    }
    /* Set seed for random noise generator, add noise to image */
    srandom2(1);
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_out[i][j] = round((img_out[i][j] \
                                + 81*sigma_w*normal())/81);
            img_out[i][j] = (img_out[i][j] < 0)? \
                        0: ((img_out[i][j] > 255)? \
                        255: img_out[i][j]);
        }
    }
    /* assign the noisy array to the output image */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_noisy->mono[i][j] = img_out[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_out );
    free_img( (void**)img_in  );
    free_img( (void**)img_t   );
}
```

Question 2.4.2

Compute the MAP estimate of X using 20 iterations for coordinate decent optimization with $\sigma_x^2 = \hat{\sigma}_x^2$ the ML estimate of σ_x^2 for $p = 2$, and $\sigma_W^2 = 4^2$. Print out the resulting MAP estimate.

solution

By running the following program to export the resulting MAP estimate image to **soln_2_4_2.tif** and the cost function to **soln_2_4_2.csv**

```
./soln_2_4_2 soln_2_4_1.tif
```

We have 2 versions of ICD with blurred image, one is **easy** must take **longer** time complete ICD iterations; the other **complex** version take **shorter** time. The second row in the table indicates Time cost for 20 iterations of ICD Algorithm with Blurring Filter, and cost function computation for each iteration.

	EASY VERSION	**COMPLEX VERSION**
function name in **map.c**	`iter_ICD_blur`	`iter_ICD_blur_quick`
Time cost (`second`)	4.4384	0.7905

The corresponding function for ICD with Blurring filter in **map.c** is (**easy version**, takes **longer** time)

```
void iter_ICD_blur(double **img_y, double **img_x, \
                   int16_t W, int16_t H,\
                   double sigma_w, double sigma_x) {
    double **img_t, **img_hy, **img_h2x_old, **img_gx_old;
    double **img_h2x, **img_gx;
    double ratio = (sigma_x / sigma_w)*(sigma_x / sigma_w);
    /* ||H||^2 the sum of all elements in blurring filter H */
    double H2 = (1*1 + 2*2 + 3*3 + 2*2 + 1*1) / 81.0;
    H2 *= H2;
    /* allocate memory */
    img_t       = (double **)get_img(W, H, sizeof(double));
    img_hy      = (double **)get_img(W, H, sizeof(double));
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* assign H * y to img_hy */
    conv_array_H(img_y, img_hy, W, H);
    /* ICD with blurred image */
    for (int16_t i=2; i < H-2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
            - point_conv_H2(img_t,i, j, W, H) + H2*img_t[i][j]) \
            + point_conv_G(img_t, i, j, W, H)) / (ratio*H2+1);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    /* assign update the img_x */
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_x[i][j] = img_t[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_t );
    free_img( (void**)img_hy );
}
```

```
double point_conv_G(double **img, int16_t i, int16_t j, \
                    int16_t W, int16_t H) {
    //assert((j-1 >= 0)&&(j+1 <= W-1)&&(i-1 >= 0)&&(i+1 <= H-1));
    double conv = 0.0;
    conv  = 2*(img[i][j-1] + img[i][j+1]);
    conv += 2*img[i-1][j] + (img[i-1][j-1] + img[i-1][j+1]);
    conv += 2*img[i+1][j] + (img[i+1][j-1] + img[i+1][j+1]);
    return conv/12;
}

double point_conv_H(double **img, int16_t i, int16_t j, \
                    int16_t W, int16_t H) {
    if ((j-2 < 0)||( j+2 > W-1 )||( i-2 < 0 )||( i+2 > H-1)){
        return img[i][j];
    }
    double a[5] = {0};
    double *p = a;
    for (int16_t row=i-2; row <= i+2; row++) {
        *p = 3*img[row][j] + 2*(img[row][j-1] \
           + img[row][j+1]) + img[row][j-2] + img[row][j+2];
        p++;
    }
    return ( 3*a[2] + 2*(a[1]+a[3]) + a[0] + a[4] ) / 81;
}

double point_conv_H2(double **img, int16_t i, int16_t j, \
                     int16_t W, int16_t H) {
    //assert((j-2 >= 0)&&(j+2 <= W-1)&&(i-2 >= 0)&&(i+2 <= H-1));
    double** img_t = (double **)get_img(5, 5, sizeof(double));
    for (int16_t dy = -2; dy <= 2; dy++) {
        for (int16_t dx = -2; dx <= 2; dx++) {
            img_t[2+dy][2+dx] = point_conv_H(img, i+dy, j+dx, W, H);
        }
    }
    double conv = point_conv_H(img_t, 2, 2, 5, 5);
    free_img( (void**)img_t );
    return conv;
}
```

The corresponding function for ICD with Blurring filter in **map.c** is (**complex version**, takes **shorter** time)

```
void iter_ICD_blur_quick(double **img_y, double **img_x, \
                    int16_t W, int16_t H,\
                    double sigma_w, double sigma_x) {
    double **img_t, **img_hy, **img_h2x_old, **img_gx_old;
    double **img_h2x, **img_gx;
    double **reg_up  = (double **)get_img(W, 4, sizeof(double));
    double **reg_low = (double **)get_img(W, 4, sizeof(double));
    double ratio = (sigma_x / sigma_w)*(sigma_x / sigma_w);
    /* ||H||^2 the sum of all elements in blurring filter H */
    double H2 = (1*1 + 2*2 + 3*3 + 2*2 + 1*1) / 81.0;
    H2 *= H2;
    /* allocate memory */
    img_t       = (double **)get_img(W, H, sizeof(double));
    img_hy      = (double **)get_img(W, H, sizeof(double));
    img_h2x_old = (double **)get_img(W, H, sizeof(double));
    img_gx_old  = (double **)get_img(W, H, sizeof(double));
    img_h2x     = (double **)get_img(W, H, sizeof(double));
    img_gx      = (double **)get_img(W, H, sizeof(double));
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* assign H * y to img_hy */
    conv_array_H(img_y, img_hy, W, H);
    /* assign the x^{old} part of G * x to img_gx_old */
    /* x = [x^{new}, x^{old}], where x^{old} for last iteration */
    /* and x^{new} for current iteration */
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_gx_old[i][j] = 2*img_x[i][j+1];
        }
    }
    for (int16_t i=H/2-3; i >= 4; i--) {
        for (int16_t j=4; j < W-4; j++) {
            img_gx_old[i][j] += 2*img_x[i-1][j] \
                + img_x[i-1][j-1] + img_x[i-1][j+1];
            img_gx_old[i][j] /= 12;
        }
```

```
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_gx_old[i][j] += 2*img_x[i+1][j] \
                + img_x[i+1][j-1] + img_x[i+1][j+1];
            img_gx_old[i][j] /= 12;
        }
    }
    /* assign the x^{old} part of H*H*x to img_h2x_old */
    /* x = [x^{new}, x^{old}], where x^{old} for last iteration */
    /* and x^{new} for current iteration */
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 19*img_x[i][j] \
                + 16*(img_x[i][j+1]+img_x[i][j-1])\
                + 10*(img_x[i][j+2]+img_x[i][j-2]) \
                + 4*(img_x[i][j+3]+img_x[i][j-3]) \
                + img_x[i][j+4]+img_x[i][j-4];
        }
    }
    for (int16_t i=H/2-3; i >= 4; i--) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 16*img_h2x_old[i-1][j]\
                + 10*img_h2x_old[i-2][j] \
                + 4*img_h2x_old[i-3][j] + img_h2x_old[i-4][j];
        }
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 16*img_h2x_old[i+1][j]\
                + 10*img_h2x_old[i+2][j] \
                + 4*img_h2x_old[i+3][j] + img_h2x_old[i+4][j];
        }
    }
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            double t = 19*img_x[i][j] + 16*img_x[i][j+1]\
                + 10*img_x[i][j+2] \
                + 4*img_x[i][j+3] + img_x[i][j+4];
            img_h2x_old[i][j] += 19*t;
            img_h2x_old[i][j] /= 81*81;
        }
    }
```

```
    /* ICD process */
    /* step 1: compute x^{new} in the mid 4 rows */
    for (int16_t i=H/2-2; i < H/2+2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) + H2*img_t[i][j]) \
                + point_conv_G(img_t, i, j, W, H)) / (ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=H/2-2; i < H/2+2; i++) {
        for (int16_t j=4; j < W-4; j++) {
            int16_t r_up = i-(H/2-2);
            int16_t r_low= (H/2)+1-i;
            reg_up[r_up][j]     = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])\
                +10*(img_t[i][j-2]+img_t[i][j+2])\
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
            reg_low[r_low][j]   = reg_up[r_up][j];
        }
    }
    /* step 2: process upper chunk + lower chunk of img */
    for (int16_t i=H/2-3; i >= 4; i--) {
        int16_t r0 = (i-((H/2-3)%4))%4, r1, r2, r3;
        r1=(r0+1)%4; r2=(r0+2)%4; r3 = (r0+3)%4;
        for (int16_t j=4; j < W-4; j++) {
            img_h2x[i][j] = 16*reg_up[r0][j]\
                    +10*reg_up[r1][j] + 4*reg_up[r2][j]\
                    +reg_up[r3][j];
            img_h2x[i][j] /= 81*81;
            img_h2x[i][j] += img_h2x_old[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            img_gx[i][j] = 2*img_t[i+1][j] \
                + img_t[i+1][j-1] + img_t[i+1][j+1];
            img_gx[i][j] /= 12;
            img_gx[i][j] += img_gx_old[i][j];
        }
        /* update the estimate img_t */
        for (int16_t j=4; j < W-4; j++) {
```

```
            double h2x, gx;
            h2x = 16*img_t[i][j-1] + 10*img_t[i][j-2] \
                + 4*img_t[i][j-3]\
                +img_t[i][j-4];
            h2x *= 19; h2x /= 81*81;
            h2x += img_h2x[i][j];
            gx  = img_t[i][j-1] / 6;
            gx  += img_gx[i][j];
            img_t[i][j] = (ratio*(img_hy[i][j] \
                - h2x + H2*img_t[i][j]) \
                + gx) / (ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            reg_up[r3][j] = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])\
                +10*(img_t[i][j-2]+img_t[i][j+2])\
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
        }
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        int16_t j=2;
        int16_t r0 = (((H/2+2)%4)+4-(i%4))%4, r1, r2, r3;
        r1=(r0+1)%4; r2=(r0+2)%4; r3 = (r0+3)%4;
        for (int16_t j=4; j < W-4; j++) {
            img_h2x[i][j] = 16*reg_low[r0][j]\
                    +10*reg_low[r1][j] + 4*reg_low[r2][j]\
                    +reg_low[r3][j];
            img_h2x[i][j] /= 81*81;
            img_h2x[i][j] += img_h2x_old[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            img_gx[i][j] = 2*img_t[i-1][j] \
                + img_t[i-1][j-1] + img_t[i-1][j+1];
            img_gx[i][j] /= 12;
            img_gx[i][j] += img_gx_old[i][j];
        }
        /* update the estimate img_t */
        for (j=4; j < W-4; j++) {
            double h2x, gx;
```

```
            h2x = 16*img_t[i][j-1] + 10*img_t[i][j-2] \
                + 4*img_t[i][j-3]\
                +img_t[i][j-4];
            h2x *= 19; h2x /= 81*81;
            h2x += img_h2x[i][j];
            gx  = img_t[i][j-1] / 6;
            gx  += img_gx[i][j];
            img_t[i][j] = (ratio*(img_hy[i][j] \
                 - h2x + H2*img_t[i][j]) \
                + gx) / (ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            reg_low[r3][j] = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])\
                +10*(img_t[i][j-2]+img_t[i][j+2])\
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
        }
    }
    /* step 3: post-process for head rows + tail rows/cols */
    /* rows */
    for (int16_t i=2; i < 4; i++) {
        for (int16_t j=2; j < W-2; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) + H2*img_t[i][j]) \
                + point_conv_G(img_t, i, j, W, H)) / (ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=H-4; i < H-2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) + H2*img_t[i][j]) \
                + point_conv_G(img_t, i, j, W, H)) / (ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
```

```
    }
    /* cols */
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=2; j < 4; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) + H2*img_t[i][j])\
                + point_conv_G(img_t, i, j, W, H)) /(ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=W-4; j < W-2; j++) {
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + point_conv_G(img_t, i, j, W, H)) /(ratio*H2+1);
            img_t[i][j] = (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    /* assign update the img_x */
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_x[i][j] = img_t[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_t );
    free_img( (void**)img_hy );
    free_img( (void**)img_h2x_old );
    free_img( (void**)img_gx_old );
    free_img( (void**)img_h2x );
    free_img( (void**)img_gx );
    free_img( (void**)reg_up );
    free_img( (void**)reg_low );
}
```

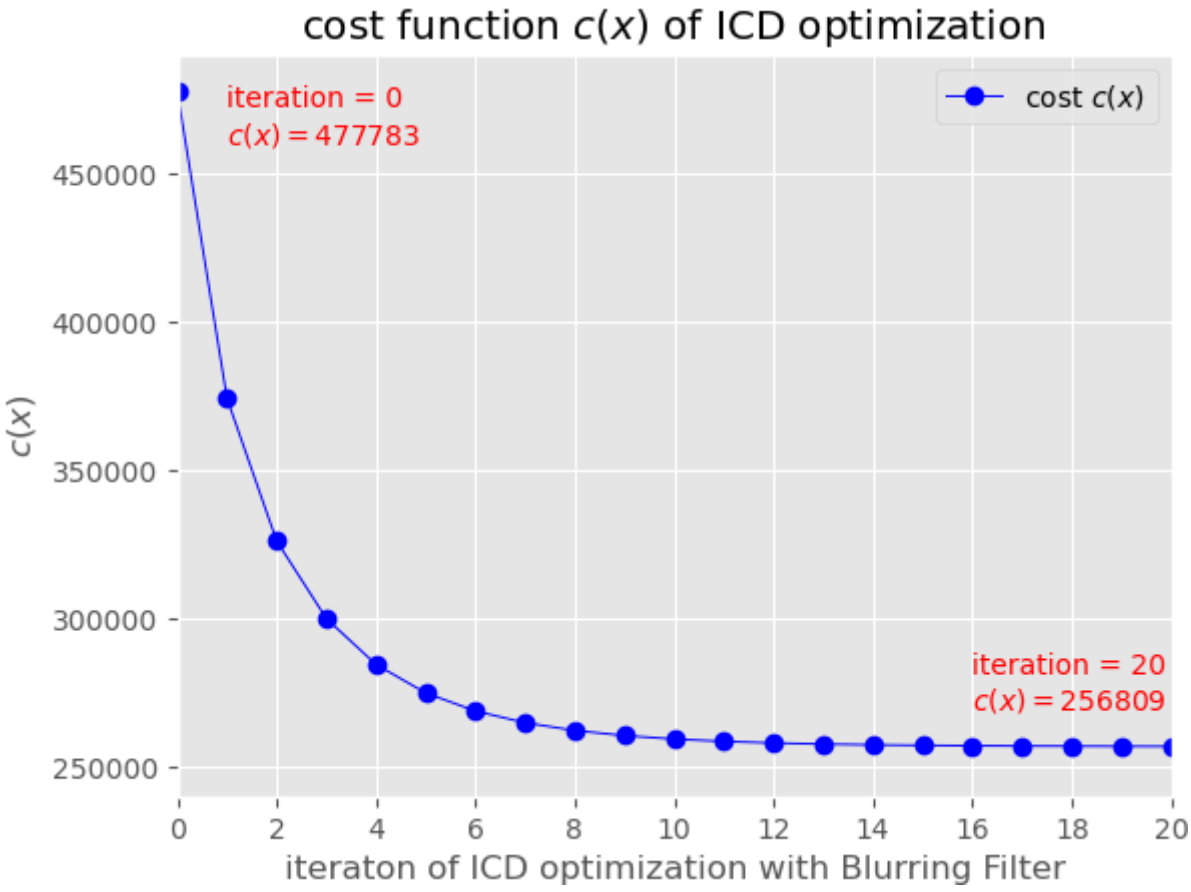
Question 2.4.3

Plot the cost function of equation 23 as a function of iteration number

solution

Then visualize the results of cost functions in **soln_2_4_2.csv** with python script **vis_2_4_2.py**

The below image is the cost functions for $\sigma_x^2 = \hat{\sigma}_x^2, \sigma_w^2 = 4^2$



The corresponding function for **cost function computation** in **map.c** is

```
double compute_cost_blur(double **img_y, double **img_x, \
                         int16_t W, int16_t H,\
                         double sigma_w, double sigma_x) {
    double **img_t;
    double cost = 0.0;
    /* allocate memory */
    img_t = (double **)get_img(W, H, sizeof(double));
    /* assign H * x to img_t */
    conv_array_H(img_x, img_t, W, H);
```

```
    /* compute the cost function: |y-H*x|^2 part */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            double t = img_y[i][j]-img_t[i][j];
            cost +=  t*t;
        }
    }
    cost *= 12*(sigma_x / sigma_w)*(sigma_x / sigma_w);
    /* copy to double array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* compute the cost function: x neighbor part */
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W; j++ ) {
            double t = img_x[i][j]-img_x[i][j-1];
            cost +=  t*t;
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            double r, s, t;
            r = img_x[i][j]-img_t[i-1][j];
            s = img_x[i][j]-img_t[i-1][j-1];
            t = img_x[i][j]-img_t[i-1][j+1];
            cost += (2*r*r + s*s + t*t);
        }
    }
    for (int16_t j = 1; j < W-1; j++ ) {
        double r, s, t;
        r = img_x[H-2][j]-img_t[H-1][j];
        s = img_x[H-2][j]-img_t[H-1][j-1];
        t = img_x[H-2][j]-img_t[H-1][j+1];
        cost += (2*r*r + s*s + t*t);
    }
    cost /= 2*12*sigma_x*sigma_x;
    /* free up the allocateed memory */
    free_img( (void**)img_t );
    return cost;
}
```

3.1 Basic Techniques for MAP Restoration with non-Gaussian Prior

Question 3.1.1

Use the noisy and blurred image from Section 2.4 as the image y, and compute the MAP estimate of X using 20 iterations of coordinate decent optimization with $p = 1.20, \sigma_x^{1.2} = \hat{\sigma}_x^{1.2}$ (i.e. the ML estimate of σ for $p = 1.2$), $\sigma_W^2 = 4^2$, and err = 1e-7. Print out the resulting MAP estimate

solution

The times cost of ICD and cost function computation for 20 iterations is around **60** seconds for one run.

By running the following program **soln_3_1_1** with command line

```
./soln_3_1_1 soln_2_4_1.tif
```

to export the resulting MAP estimate image to **soln_3_1_1.tif** and the cost function to **soln_3_1_1.csv**

The corresponding function of ICD algorithm with Blurring filter and GGMRF prior in **map.c** is

```
void iter_ICD_blur_GGMRF(double **img_y, double **img_x, \
                    int16_t W, int16_t H,\
                    double sigma_w, double sigma_x, double p) {
    double **img_t, **img_hy, **img_h2x_old, **img_gx_old;
    double **img_h2x, **img_gx;
    double ratio = pow(sigma_x, p) / (sigma_w*sigma_w);
    /* ||H||^2 sum of all elements in blurring filter H */
    double H2 = (1*1 + 2*2 + 3*3 + 2*2 + 1*1) / 81.0;
    H2 *= H2;
    /* allocate memory */
    img_t       = (double **)get_img(W, H, sizeof(double));
    img_hy      = (double **)get_img(W, H, sizeof(double));
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* assign H * y to img_hy */
    conv_array_H(img_y, img_hy, W, H);
    /* ICD with blurred image and  GGMRF prior */
    for (int16_t i=2; i < H-2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            /* compute convolution product H * e of H and e=y-H*x */
            /* H * e = H*y - H*H* x */
            double *a_t = (double *)calloc(25, sizeof(double));
            for (int16_t dy = -2; dy <= 2; dy++) {
                for (int16_t dx = -2; dx <= 2; dx++) {
                    if ((j+dx-2 < 0) || ( j+dx+2 > W-1 ) \
                        || ( i+dy-2 < 0 ) || ( i+dy+2 > H-1)) {
                        *(a_t+5*(dy+2)+dx+2) = img_t[i+dy][j+dx];
                    } else {
                        double a[5] = {0};
                        double *pa = a;
                        for (int16_t row=i+dy-2; row<=i+dy+2; row++){
                            *pa = 3*img_t[row][j+dx] \
                                + 2*(img_t[row][j+dx-1] \
                                + img_t[row][j+dx+1]) \
                                + img_t[row][j+dx-2] \
                                + img_t[row][j+dx+2];
```

```
                    pa++;
                }
                *(a_t+5*(dy+2)+dx+2) = \
                    (3*a[2]+2*(a[1]+a[3])+a[0]+a[4])/81;
            }
        }
    }
    double a2[5] = {0};
    double *pa = a2;
    for (int16_t row=0; row <= 4; row++) {
        *pa = 3*(*(a_t+5*row+2)) + 2*((*(a_t+5*row+1)) \
            + (*(a_t+5*row+3))) + (*(a_t+5*row)) + \
            (*(a_t+5*row+4));
        pa++;
    }
    free( a_t );
    double conv = \
        (3*a2[2]+2*(a2[1]+a2[3])+a2[0]+a2[4])/81;/*H*H*x*/
    double he = img_hy[i][j];
    he-=conv; /* H * e = H*y - H*H* x */
    /* find max/min elements in neighbor of img_t[i][j] */
    double high = (img_t[i][j-1] > img_t[i][j+1])? \
        img_t[i][j-1]: img_t[i][j+1];
    for (int16_t dx = -1; dx <= 1; dx++) {
        if (img_t[i-1][j+dx] > high) {high=img_t[i-1][j+dx];}
        if (img_t[i+1][j+dx] > high) {high=img_t[i+1][j+dx];}
    }
    double low = (img_t[i][j-1] < img_t[i][j+1])? \
        img_t[i][j-1]: img_t[i][j+1];
    for (int16_t dx = -1; dx <= 1; dx++) {
        if (img_t[i-1][j+dx] < low) {low=img_t[i-1][j+dx];}
        if (img_t[i+1][j+dx] < low) {low=img_t[i+1][j+dx];}
    }
    /* compare and update low/high bound with threshold */
    double thr = img_t[i][j]; thr += he/H2;
    if (low  > thr) { low  = thr; }
    if (high < thr) { high = thr; }
    /* pass parameters of function */
    Parameters param = {.ratio=ratio, .H2=H2, .he=he, .p=p};
    for (int16_t dy=-1; dy <= 1; dy++) {
        for (int16_t dx=-1; dx <= 1; dx++) {
            param.x_old[dy+1][dx+1] = img_t[i+dy][j+dx];
        }
```

```
            }
            /* obtain the optimal solution */
            int code;
            img_t[i][j] = solve(f_GGMRF,&param,low,high,1e-7,&code);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    /* assign update the img_x */
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_x[i][j] = img_t[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_t );
    free_img( (void**)img_hy );
}


double sign(double x) {
    if (x > 0.0) return 1.0;
    if (x < 0.0) return -1.0;
    return x;
}


static double f_GGMRF(double x, void * pblock) {
    Parameters * ptr = (Parameters *) pblock;
    double value = -(ptr->he) + (ptr->H2)*(x-(ptr->x_old[1][1]));
    value *= (ptr->ratio);
    double** prod = (double **)get_img(3, 3, sizeof(double));
    for (int16_t i=0; i < 3; i++) {
        for (int16_t j=0; j < 3; j++) {
            double d = x-(ptr->x_old[i][j]);
            prod[i][j] = pow(fabs(d), (ptr->p)-1) * sign(d);
        }
    }
    double conv = 0.0;
    conv  = 2*(prod[1][0] + prod[1][2]);
    conv += 2*prod[0][1] + (prod[0][0] + prod[0][2]);
    conv += 2*prod[2][1] + (prod[2][0] + prod[2][2]);
```

```
    conv /= 12;
    value += conv;
    free_img( (void**)prod );
    return value;
}
```

Question 3.1.2

Produce two restorations using 20 ICD iterations with the parameters of problem 1 above, but with $\sigma_x = 5 \cdot \hat{\sigma}_x$ and $\sigma_x = (1/5) \cdot \hat{\sigma}_x$ Print out the resulting MAP estimates.

solution

By running the following program **soln_3_1_2** with command line

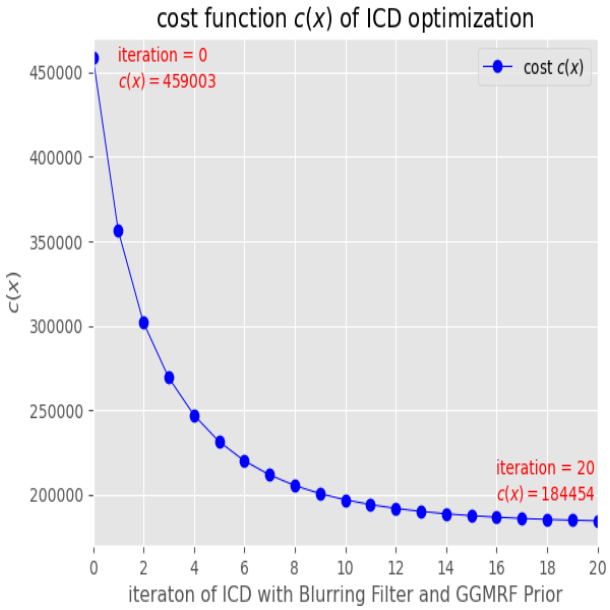
```
./soln_3_1_2 soln_2_4_1.tif
```

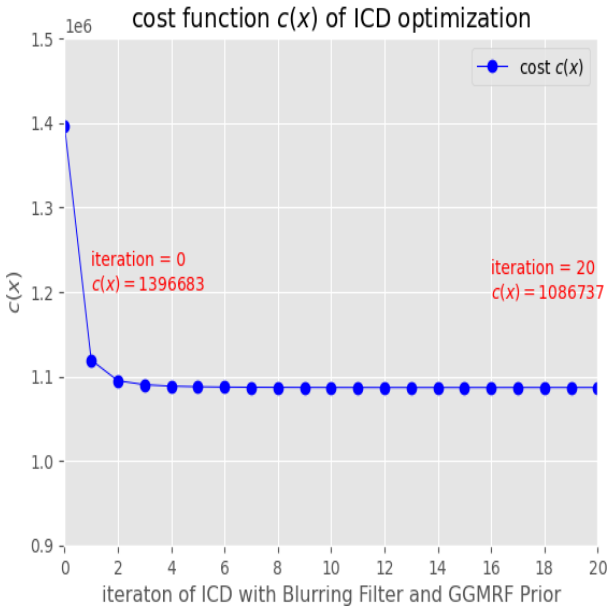
to export the resulting MAP estimate image to **soln_3_1_2a.tif, soln_3_1_2b.tif,** and the cost function to **soln_3_1_2a.csv, soln_3_1_2b.csv** for $\sigma_x = 5 \cdot \hat{\sigma}_x$ and $\sigma_x = (1/5) \cdot \hat{\sigma}_x$ respectively

The left image is the resulting MAP estimate image for $\sigma_x = 5 \cdot \hat{\sigma}_x$, and the right image is the one for $\sigma_x = (1/5) \cdot \hat{\sigma}_x$


Then visualize the results of cost functions in **soln_3_1_2a.csv, soln_3_1_2b.csv** with python script **vis_3_1_2.py**

The left image is the cost functions for $\sigma_x = 5 \cdot \hat{\sigma}_x$, and the right image is the one for $\sigma_x = (1/5) \cdot \hat{\sigma}_x$



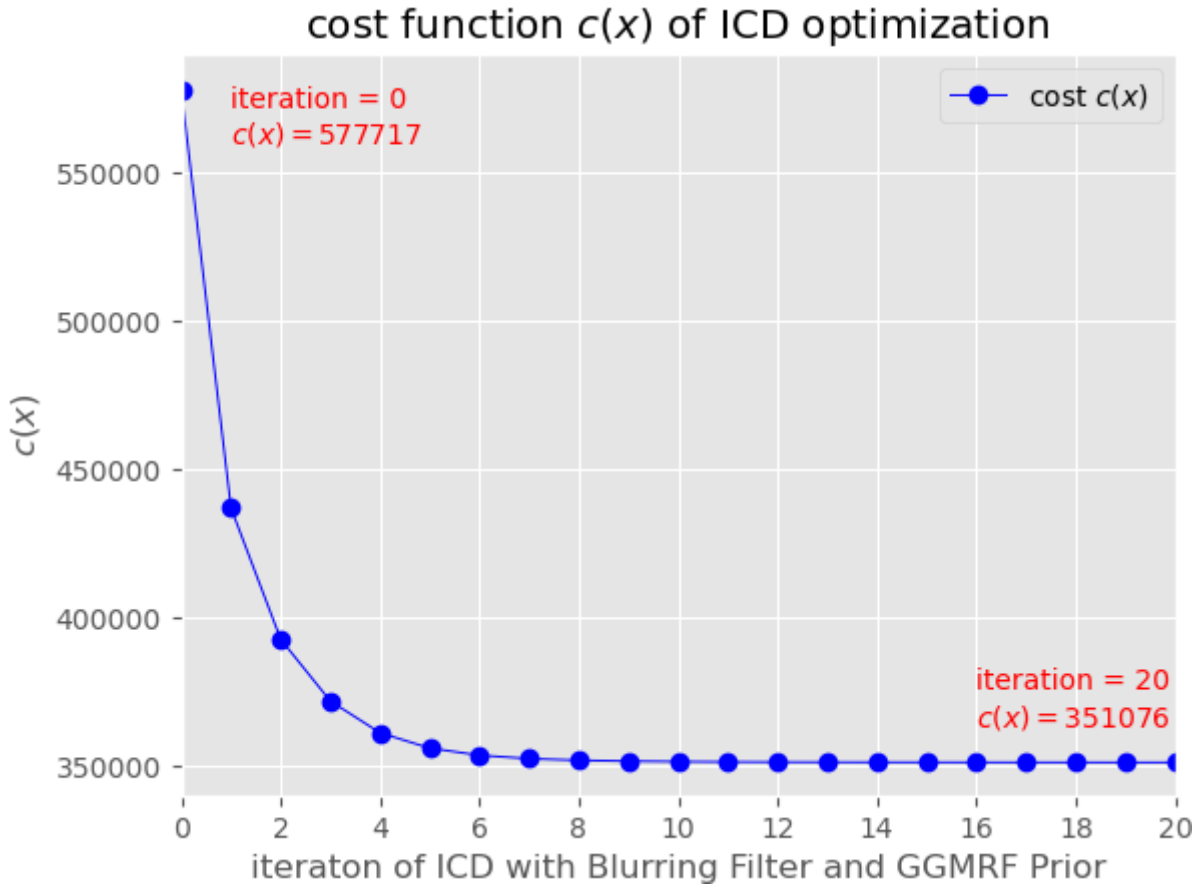


Question 3.1.3

Plot the cost function of equation 31 as a function of iteration number.

solution

Then visualize the results of cost functions in **soln_3_1_1.csv** with python script **vis_3_1_1.py**. The below image is the **cost function** for $\sigma_x^{1.2} = \hat{\sigma}_x^{1.2} = 8.732012^{1.2}, \sigma_w^2 = 4^2$ for 20 iterations



The corresponding function for **cost function computation** in **map.c** is

```
double compute_cost_blur_GGMRF(double **img_y, double **img_x, \
                        int16_t W, int16_t H,\
                        double sigma_w, double sigma_x, double p) {
    double **img_t;
    double cost = 0.0;
    /* allocate memory */
    img_t = (double **)get_img(W, H, sizeof(double));
    /* assign H * x to img_t */
    conv_array_H(img_x, img_t, W, H);
    /* compute the cost function: |y-H*x|^2 part */
```

```
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            double t = img_y[i][j]-img_t[i][j];
            cost +=  t*t;
        }
    }
    cost *= 12 * p*pow(sigma_x, p) / (2*sigma_w*sigma_w);
    /* copy to double array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* compute the cost function: x neighbor part */
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W; j++ ) {
            double t = fabs(img_x[i][j]-img_x[i][j-1]);
            cost +=  pow(t, p);
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            double r, s, t;
            r = fabs(img_x[i][j]-img_t[i-1][j]);
            s = fabs(img_x[i][j]-img_t[i-1][j-1]);
            t = fabs(img_x[i][j]-img_t[i-1][j+1]);
            cost += (2*pow(r, p) + pow(s, p) + pow(t, p));
        }
    }
    for (int16_t j = 1; j < W-1; j++ ) {
        double r, s, t;
        r = fabs(img_x[H-2][j]-img_t[H-1][j]);
        s = fabs(img_x[H-2][j]-img_t[H-1][j-1]);
        t = fabs(img_x[H-2][j]-img_t[H-1][j+1]);
        cost += (2*pow(r, p) + pow(s, p) + pow(t, p));
    }
    cost /= 12 * p*pow(sigma_x, p);
    /* free up the allocateed memory */
    free_img( (void**)img_t );
    return cost;
}
```

3.2 MAP Restoration using Majorization to Optimize non-Gaussian Cost Function

Question 3.2.1

Use the same noisy and blurred image from Sections 2.4 and 3.1 as the image y, and compute the MAP estimate of X using 20 iterations of coordinate decent optimization with symmetric bound majorization as specified above. Use the parameters $p = 1.2, q = 2, \sigma_x^{1.2} = \hat{\sigma}_x^{1.2}$ (i.e. the ML estimate of σ for $p = 1.2$), and $\sigma_W^2 = 4^2$. Print out the resulting MAP estimate

solution

We set $T = 1$ as the default value in out program **soln_3_2_1.c**

The times cost of ICD and cost function computation for 20 iterations is around **6** seconds for one run, much faster than the **60** seconds time cost of the method in previous section.

By running the following program **soln_3_2_1** with command line

```
./soln_3_2_1 soln_2_4_1.tif
```

to export the resulting MAP estimate image to **soln_3_2_1.tif** and the cost function to **soln_3_2_1.csv**

The corresponding function of ICD algorithm with with Majorization of GGMRF Prior in **map.c** is

```
void iter_ICD_blur_qGGMRF_quick(double **img_y, double **img_x, \
                    int16_t W, int16_t H,\
                    double sigma_w, double sigma_x, \
                    double p, double q, double T) {
    double **img_t, **img_hy, **img_h2x_old, \
        **img_gx_old, **img_g_old;
    double **img_h2x, **img_gx, **img_g;
    double **reg_up  = (double **)get_img(W, 4, sizeof(double));
    double **reg_low = (double **)get_img(W, 4, sizeof(double));
    double ratio = 1.0 / (2*sigma_w*sigma_w);
    /* ||H||^2 the sum of all elements in blurring filter H */
    double H2 = (1*1 + 2*2 + 3*3 + 2*2 + 1*1) / 81.0;
    H2 *= H2;
    /* allocate memory */
    img_t       = (double **)get_img(W, H, sizeof(double));
    img_hy      = (double **)get_img(W, H, sizeof(double));
    img_h2x_old = (double **)get_img(W, H, sizeof(double));
    img_gx_old  = (double **)get_img(W, H, sizeof(double));
    img_g_old   = (double **)get_img(W, H, sizeof(double));
    img_h2x     = (double **)get_img(W, H, sizeof(double));
    img_gx      = (double **)get_img(W, H, sizeof(double));
```

```
    img_g       = (double **)get_img(W, H, sizeof(double));
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* assign H * y to img_hy */
    conv_array_H(img_y, img_hy, W, H);
    /* assign the x^{old} part of G * x, G
    (b_{s, r} * x_{r} , b_{s, r}) to img_gx_old */
    /* x = [x^{new}, x^{old}], where x^{old} for last iteration */
    /* and x^{new} for current iteration */
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            double xs = img_x[i][j];
            double r = get_btilde(img_x[i][j+1]-xs, \
                     1.0/6, sigma_x, p, q, T);
            img_gx_old[i][j] = r*img_x[i][j+1];
            img_g_old[i][j]  = r;
        }
    }
    for (int16_t i=H/2-3; i >= 4; i--) {
        for (int16_t j=4; j < W-4; j++) {
            double xs = img_x[i][j];
            double r = get_btilde(img_x[i-1][j]-xs, \
                     1.0/6, sigma_x, p, q, T), \
                   s = get_btilde(img_x[i-1][j-1]-xs, \
                     1.0/12, sigma_x, p, q, T), \
                   t = get_btilde(img_x[i-1][j+1]-xs, \
                     1.0/12, sigma_x, p, q, T);
            img_gx_old[i][j] += r*img_x[i-1][j] \
                + s*img_x[i-1][j-1] + t*img_x[i-1][j+1];
            img_g_old[i][j]  += r + s + t;
        }
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            double xs = img_x[i][j];
            double r = get_btilde(img_x[i+1][j]-xs, \
                     1.0/6, sigma_x, p, q, T), \
                   s = get_btilde(img_x[i+1][j-1]-xs, \
                     1.0/12, sigma_x, p, q, T), \
                   t = get_btilde(img_x[i+1][j+1]-xs, \
```

```
                    1.0/12, sigma_x, p, q, T);
            img_gx_old[i][j] += r*img_x[i+1][j] \
                + s*img_x[i+1][j-1] + t*img_x[i+1][j+1];
            img_g_old[i][j]  += r + s + t;
        }
    }
    /* assign the x^{old} part of H*H*x to img_h2x_old */
    /* x = [x^{new}, x^{old}], where x^{old} for the last iteration
*/
    /* and x^{new} for current iteration */
    for (int16_t i=0; i < H; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 19*img_x[i][j] \
                + 16*(img_x[i][j+1]+img_x[i][j-1])\
                + 10*(img_x[i][j+2]+img_x[i][j-2]) \
                + 4*(img_x[i][j+3]+img_x[i][j-3]) \
                + img_x[i][j+4]+img_x[i][j-4];
        }
    }
    for (int16_t i=H/2-3; i >= 4; i--) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 16*img_h2x_old[i-1][j]\
                + 10*img_h2x_old[i-2][j] \
                + 4*img_h2x_old[i-3][j] + img_h2x_old[i-4][j];
        }
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            img_h2x_old[i][j] = 16*img_h2x_old[i+1][j]\
                + 10*img_h2x_old[i+2][j] \
                + 4*img_h2x_old[i+3][j] + img_h2x_old[i+4][j];
        }
    }
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=4; j < W-4; j++) {
            double t = 19*img_x[i][j] + 16*img_x[i][j+1]\
                + 10*img_x[i][j+2] \
                + 4*img_x[i][j+3] + img_x[i][j+4];
            img_h2x_old[i][j] += 19*t;
            img_h2x_old[i][j] /= 81*81;
        }
    }
    /* ICD process */
```

```
    /* step 1: compute x^{new} in the mid 4 rows */
    for (int16_t i=H/2-2; i < H/2+2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            double gx, g;
            point_btilde_G(img_t, i, j, W, H,\
                    sigma_x, p, q, T, \
                    &gx, &g);
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=H/2-2; i < H/2+2; i++) {
        for (int16_t j=4; j < W-4; j++) {
            int16_t r_up = i-(H/2-2);
            int16_t r_low= (H/2)+1-i;
            reg_up[r_up][j]     = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])\
                +10*(img_t[i][j-2]+img_t[i][j+2])\
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
            reg_low[r_low][j]   = reg_up[r_up][j];
        }
    }
    /* step 2: process upper chunk + lower chunk of img */
    for (int16_t i=H/2-3; i >= 4; i--) {
        int16_t r0 = (i-((H/2-3)%4))%4, r1, r2, r3;
        r1=(r0+1)%4; r2=(r0+2)%4; r3 = (r0+3)%4;
        for (int16_t j=4; j < W-4; j++) {
            img_h2x[i][j] = 16*reg_up[r0][j]\
                    +10*reg_up[r1][j] + 4*reg_up[r2][j]\
                    +reg_up[r3][j];
            img_h2x[i][j] /= 81*81;
            img_h2x[i][j] += img_h2x_old[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            double xs = img_t[i][j];
            double r = get_btilde(img_t[i+1][j]-xs, \
                     1.0/6, sigma_x, p, q, T), \
```

```
                s = get_btilde(img_t[i+1][j-1]-xs, \
                  1.0/12, sigma_x, p, q, T), \
                t = get_btilde(img_t[i+1][j+1]-xs, \
                  1.0/12, sigma_x, p, q, T);
            img_gx[i][j]  = r*img_t[i+1][j] \
                + s*img_t[i+1][j-1] + t*img_t[i+1][j+1];
            img_g[i][j]   = r + s + t;
            img_gx[i][j] += img_gx_old[i][j];
            img_g[i][j]  += img_g_old[i][j];
        }
        /* update the estimate img_t */
        for (int16_t j=4; j < W-4; j++) {
            double h2x, gx, g;
            h2x = 16*img_t[i][j-1] + 10*img_t[i][j-2] \
                + 4*img_t[i][j-3]\
                +img_t[i][j-4];
            h2x *= 19; h2x /= 81*81;
            h2x += img_h2x[i][j];
            double t = \
                get_btilde(img_t[i][j-1]-img_t[i][j],\
                           1.0/6, sigma_x, p, q, T);
            gx = t*img_t[i][j-1];
            g  = t;
            gx  += img_gx[i][j];
            g   += img_g[i][j];
            img_t[i][j] = (ratio*(img_hy[i][j] \
                - h2x + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            reg_up[r3][j] = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])
                +10*(img_t[i][j-2]+img_t[i][j+2]) \
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
        }
    }
    for (int16_t i=H/2+2; i < H-4; i++) {
        int16_t j=2;
        int16_t r0 = (((H/2+2)%4)+4-(i%4))%4, r1, r2, r3;
```

```
        r1=(r0+1)%4; r2=(r0+2)%4; r3 = (r0+3)%4;
        for (int16_t j=4; j < W-4; j++) {
            img_h2x[i][j] = 16*reg_low[r0][j]\
                    +10*reg_low[r1][j] + 4*reg_low[r2][j]\
                    +reg_low[r3][j];
            img_h2x[i][j] /= 81*81;
            img_h2x[i][j] += img_h2x_old[i][j];
        }
        for (int16_t j=4; j < W-4; j++) {
            double xs = img_t[i][j];
            double r = get_btilde(img_t[i-1][j]-xs, \
                     1.0/6, sigma_x, p, q, T), \
                   s = get_btilde(img_t[i-1][j-1]-xs, \
                     1.0/12, sigma_x, p, q, T), \
                   t = get_btilde(img_t[i-1][j+1]-xs, \
                     1.0/12, sigma_x, p, q, T);
            img_gx[i][j]  = r*img_t[i-1][j] \
                + s*img_t[i-1][j-1] + t*img_t[i-1][j+1];
            img_g[i][j]   = r + s + t;
            img_gx[i][j] += img_gx_old[i][j];
            img_g[i][j]  += img_g_old[i][j];
        }
        /* update the estimate img_t */
        for (j=4; j < W-4; j++) {
            double h2x, gx, g;
            h2x = 16*img_t[i][j-1] \
                + 10*img_t[i][j-2] + 4*img_t[i][j-3]\
                    +img_t[i][j-4];
            h2x *= 19; h2x /= 81*81;
            h2x += img_h2x[i][j];
            double t = get_btilde(img_t[i][j-1]-img_t[i][j], \
                     1.0/6, sigma_x, p, q, T);
            gx = t*img_t[i][j-1];
            g  = t;
            gx  += img_gx[i][j];
            g   += img_g[i][j];
            img_t[i][j] = (ratio*(img_hy[i][j] \
                - h2x + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
```

```
        for (int16_t j=4; j < W-4; j++) {
            reg_low[r3][j] = 19*img_t[i][j] \
                +16*(img_t[i][j-1]+img_t[i][j+1])
                +10*(img_t[i][j-2]+img_t[i][j+2]) \
                + 4*(img_t[i][j-3]+img_t[i][j+3])\
                +img_t[i][j-4]+img_t[i][j+4];
        }
    }
    /* step 3: post-process for head rows + tail rows/cols */
    /* rows */
    for (int16_t i=2; i < 4; i++) {
        for (int16_t j=2; j < W-2; j++) {
            double gx, g;
            point_btilde_G(img_t, i, j, W, H,\
                    sigma_x, p, q, T, \
                    &gx, &g);
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=H-4; i < H-2; i++) {
        for (int16_t j=2; j < W-2; j++) {
            double gx, g;
            point_btilde_G(img_t, i, j, W, H,\
                    sigma_x, p, q, T, \
                    &gx, &g);
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    /* cols */
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=2; j < 4; j++) {
```

```
            double gx, g;
            point_btilde_G(img_t, i, j, W, H,\
                    sigma_x, p, q, T, \
                    &gx, &g);
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    for (int16_t i=4; i < H-4; i++) {
        for (int16_t j=W-4; j < W-2; j++) {
            double gx, g;
            point_btilde_G(img_t, i, j, W, H,\
                    sigma_x, p, q, T, \
                    &gx, &g);
            img_t[i][j] = (ratio*(img_hy[i][j]\
                -point_conv_H2(img_t,i, j, W, H) \
                + H2*img_t[i][j]) \
                + gx) / (ratio*H2+g);
            img_t[i][j] =  (img_t[i][j] < 0)?\
                        0: (img_t[i][j] > 255)?\
                        255:img_t[i][j];
        }
    }
    /* assign update the img_x */
    for (int16_t i = 2; i < H-2; i++ ) {
        for (int16_t j = 2; j < W-2; j++ ) {
            img_x[i][j] = img_t[i][j];
        }
    }
    /* de-allocate memory */
    free_img( (void**)img_t );
    free_img( (void**)img_hy );
    free_img( (void**)img_h2x_old );
    free_img( (void**)img_gx_old );
    free_img( (void**)img_g_old );
    free_img( (void**)img_h2x );
    free_img( (void**)img_gx );
    free_img( (void**)img_g );
```

```
    free_img( (void**)reg_up );
    free_img( (void**)reg_low );
}


void point_btilde_G(double **img, int16_t i, int16_t j, \
                    int16_t W, int16_t H, \
                    double sigma_x, double p, double q, double T, \
                    double* ptr_gx, double* ptr_g) {
    //assert((j-1 >= 0)&&(j+1 <= W-1)&&(i-1 >= 0)&&(i+1 <= H-1));
    double xs = img[i][j];
    double r = get_btilde(img[i+1][j]-xs, 1.0/6, sigma_x, p, q, T),\
           s = get_btilde(img[i+1][j-1]-xs, 1.0/12, sigma_x,p,q,T),\
           t = get_btilde(img[i+1][j+1]-xs, 1.0/12, sigma_x,p,q,T);
    *ptr_gx = r*img[i+1][j] + s*img[i+1][j-1] + t*img[i+1][j+1];
    *ptr_g  = r + s + t;
    r = get_btilde(img[i-1][j]-xs, 1.0/6, sigma_x, p, q, T);
    s = get_btilde(img[i-1][j-1]-xs, 1.0/12, sigma_x, p, q, T);
    t = get_btilde(img[i-1][j+1]-xs, 1.0/12, sigma_x, p, q, T);
    *ptr_gx += r*img[i-1][j] + s*img[i-1][j-1] + t*img[i-1][j+1];
    *ptr_g  += r + s + t;
    s = get_btilde(img[i][j-1]-xs, 1.0/6, sigma_x, p, q, T);
    t = get_btilde(img[i][j+1]-xs, 1.0/6, sigma_x, p, q, T);
    *ptr_gx += s*img[i][j-1] + t*img[i][j+1];
    *ptr_g  += s + t;
    return;
}
```

Question 3.2.2

Produce two restorations using 20 ICD iterations with the parameters of problem 1 above, but with $\sigma_x = 5 \cdot \hat{\sigma}_x$ and $\sigma_x = (1/5) \cdot \hat{\sigma}_x$ Print out the resulting MAP estimates.

solution

By running the following program **soln_3_2_2** with command line

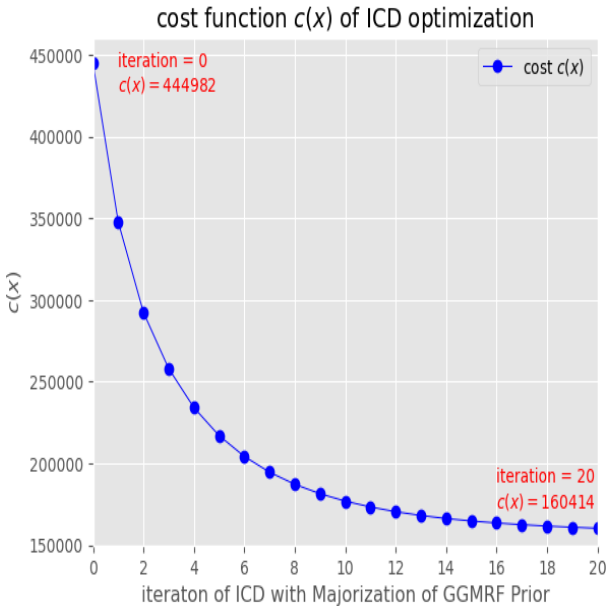
```
./soln_3_2_2 soln_2_4_1.tif
```

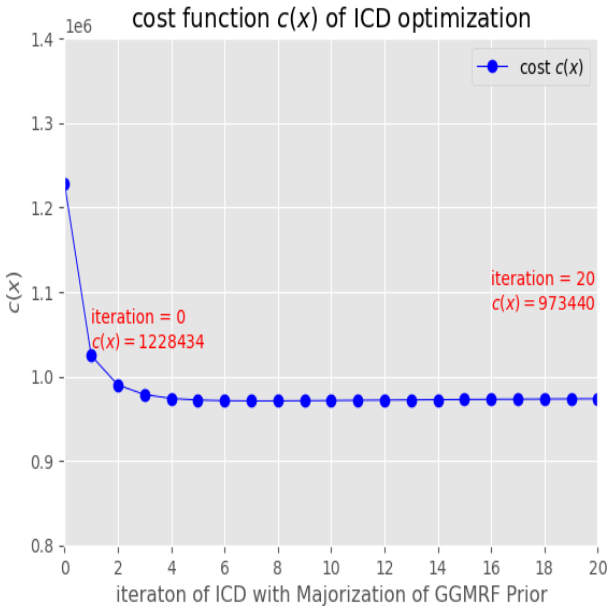
to export the resulting MAP estimate image to **soln_3_2_2a.tif, soln_3_2_2b.tif,** and the cost function to **soln_3_2_2a.csv, soln_3_2_2b.csv** for $\sigma_x = 5 \cdot \hat{\sigma}_x$ and $\sigma_x = (1/5) \cdot \hat{\sigma}_x$ respectively

The left image is the resulting MAP estimate image for $\sigma_x = 5 \cdot \hat{\sigma}_x$, and the right image is the one for $\sigma_x = (1/5) \cdot \hat{\sigma}_x$


Then visualize the results of cost functions in **soln_3_2_2a.csv, soln_3_2_2b.csv** with python script **vis_3_2_2.py**

The left image is the cost functions for $\sigma_x = 5 \cdot \hat{\sigma}_x$, and the right image is the one for $\sigma_x = (1/5) \cdot \hat{\sigma}_x$



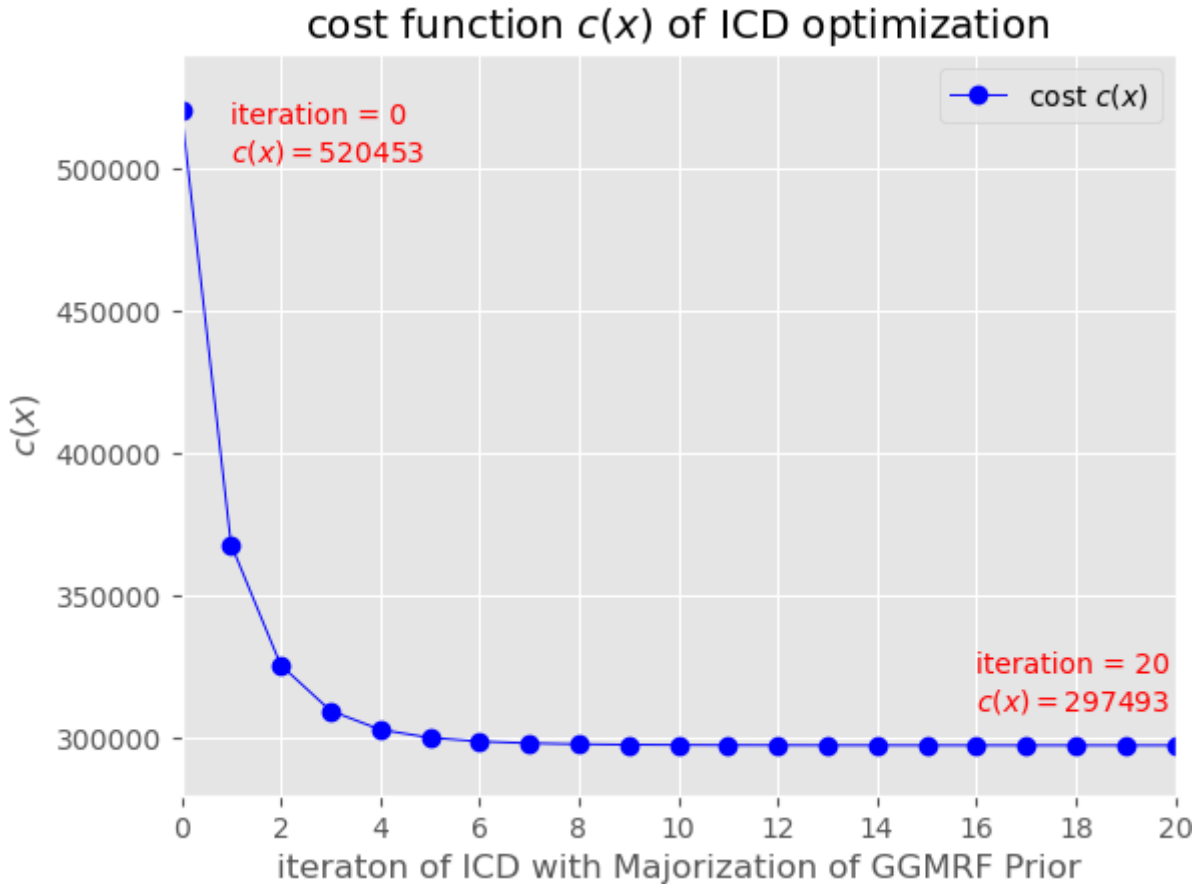


Question 3.2.3

Plot the cost function of equation 36 as a function of iteration number.

solution

Then visualize the results of cost functions in **soln_3_2_1.csv** with python script **vis_3_2_1.py**. The below image is the **cost function** for $\sigma_x^{1.2} = \hat{\sigma}_x^{1.2} = 8.732012^{1.2}, \sigma_w^2 = 4^2$ for 20 iterations



The corresponding function for **cost function computation** in **map.c** is

```
double compute_cost_blur_qGGMRF(double **img_y, double **img_x, \
                        int16_t W, int16_t H,\
                        double sigma_w, double sigma_x, \
                        double p, double q, double T) {
    double **img_t;
    double cost = 0.0;
    /* allocate memory */
    img_t = (double **)get_img(W, H, sizeof(double));
    /* assign H * x to img_t */
    conv_array_H(img_x, img_t, W, H);
```

```
    /* compute the cost function: |y-H*x|^2 part */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            double t = img_y[i][j]-img_t[i][j];
            cost +=  t*t;
        }
    }
    cost *= 12 / (2*sigma_w*sigma_w);
    /* copy to double array */
    for (int16_t i = 0; i < H; i++ ) {
        for (int16_t j = 0; j < W; j++ ) {
            img_t[i][j] = img_x[i][j];
        }
    }
    /* compute the cost function: x neighbor part */
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W; j++ ) {
            cost += get_rho(img_x[i][j]-img_x[i][j-1], \
                            1, sigma_x, p, q, T);
        }
    }
    for (int16_t i = 1; i < H-1; i++ ) {
        for (int16_t j = 1; j < W-1; j++ ) {
            double xs = img_x[i][j];
            double r = get_rho(img_t[i-1][j]-xs, \
                               1, sigma_x, p, q, T),\
                   s = get_rho(img_t[i-1][j-1]-xs, \
                               1, sigma_x, p, q, T),\
                   t = get_rho(img_t[i-1][j+1]-xs, \
                               1, sigma_x, p, q, T);
            cost += (2*r + s + t);
        }
    }
    for (int16_t j = 1; j < W-1; j++ ) {
        double xs = img_x[H-2][j];
        double r = get_rho(img_t[H-1][j]-xs, \
                           1, sigma_x, p, q, T), \
               s = get_rho(img_t[H-1][j-1]-xs, \
                           1, sigma_x, p, q, T),\
               t = get_rho(img_t[H-1][j+1]-xs, \
                           1, sigma_x, p, q, T);
        cost += (2*r + s + t);
    }
```

```
    cost /= 12;
    /* free up the allocateed memory */
    free_img( (void**)img_t );
    return cost;
}
```