姓名：桑燊

学号：2014212128

班级：2014211602

图像处理大作业

1. 实验目的

1. 将前面所有的图像处理实验进行功能整合
2. 在图像处理时，将所有的图像、滤波器、旋转矩阵都转换为 Matrix 再处理， 并在 Matrix 类中增加相应的函数，以方便处理
3. 完成频域下的低通和高通滤波功能
4. 完成图片保存功能
5. 界面美化和软件的易用性优化
6. 上交工程和实验报告，使用统一的 markdown 模板

2. 实验过程

下面将从**基础功能、图像空间变换、图像空间滤波、空间滤波、几何变换**几个方面介绍，并附带说明一些附带功能如保存图片等。

2.1 基础功能

基础功能分为**打开、重置、上一张、下一张、保存、鼠标操作**等，初始界面如下所示：

打开图片

```
/**
 * button open
 */
void MainWindow::on_btn_open_clicked(){

    if(fileDialog->exec()){
        files = fileDialog->selectedFiles();

        QDir dir = fileDialog->directory();
        // string filter
        QStringList filters;
        filters << "*.jpg" << "*.bmp" << "*.png";
        images = dir.entryInfoList(filters);

        for(int i = 0; i < images.length(); i++){
            if(files[0] == images[i].absoluteFilePath()){
                curIndex = i;
            }
        }

        showImage(QImage(files[0]));
    }
}
```

首先需要一个`QFileDialog`对象，打开文件选择窗口，然后用一个`QStringList`将当前路径下图片文件的绝对路径保存下来，然后显示当前选择的图片，`showImage()`函数如下所示：

```
/**
 * show the image
 */
void MainWindow::showImage(QImage img){

    if(gpi){
        delete gpi;
        gpi = new MyMouseEvent();
    }

    gpi->setPixmap(QPixmap::fromImage(img));

    gs->addItem(gpi);

    ui->graphicsView->setScene(gs);

    reSet();
}
```

`reSet()`函数用来重置当前图片的大小和位置，使其适配于当前窗口，定义如下：

```
/**
 * reset the size and position of cur images to adapt to the window
 */
void MainWindow::reSet()
{
    // no image
    if(NULL == gpi){
        return;
    }

    gpi->reSet();

    // QGraphics adapt
    gs->setSceneRect(gpi->boundingRect());
    // graphicsView adapt
    ui->graphicsView->fitInView(gpi->boundingRect(), Qt::KeepAspectRatio);
}
```

重置图片

```
/**
 * button reset
 */
void MainWindow::on_btn_reset_clicked()
{
    reSet();
}
```

`reSet()`函数见上面定义。

上一张&下一张

```
/**
 * button last
 */
void MainWindow::on_btn_last_clicked()
{
    if(images.length() == 0){
        return;
    }

    showImage(QImage(images[(images.length() + --curIndex) %
images.length()].absoluteFilePath()));
}

/**
 * button next
 */
void MainWindow::on_btn_next_clicked()
{

    if(images.length() == 0){
        return;
    }

    showImage(QImage(images[(images.length() + ++curIndex) %
images.length()].absoluteFilePath()));
}
```

通过全局变量`curIndex`用来记录当前显示图片在`QStringList`中所处的序号，然后调取上一张和下一张图片。

保存图片

```
/**
 * save image
 */
void MainWindow::saveImage()
{
    QImage img;

    if(getDisplayImage(img)){

        QString filename = QFileDialog::getSaveFileName(this, tr("Save
Image"), "", tr("*.png;; *.jpg;;"));
        if(filename.isEmpty()){
            return;
        }else{
            if(!(img.save(filename))){
                return;
            }
        }
    }
}
```

鼠标事件

基本功能还实现了鼠标的放大、缩小、移动等操作，新建一个类`MyMouseEvent`，关键代码如下所示：

```
void MyMouseEvent::mouseMoveEvent(QGraphicsSceneMouseEvent *event){

    setPos(pos() + mapToParent(event->pos()) - mapToParent(event-
>lastPos()));
}

void MyMouseEvent::wheelEvent(QGraphicsSceneWheelEvent *event){

    int delta = event->delta();
    double factor = scale();
    double rate = 0.1;

    if(delta > 0){
        factor *= (1 + rate);
    }else if(delta < 0){
        factor *= (1 - rate);
    }
    setTransformOriginPoint(boundingRect().width()/2,
boundingRect().height()/2);
    setScale(factor);
}

void MyMouseEvent::reSet()
{
    setPos(0, 0);
    setScale(1.0);
}
```

将`MainWindow`里面的`QGraphicsSceneItem`对象替换成`MyMouseEvent`即可，这样就可以通过鼠标操作图片Item了。

这样整个程序基本的功能就实现了。

2.2 图像空间变换

新建一个`ImageProcessing`类专门用作图像的处理，其中空间变换中主要包括**RGB转灰度、图像反转、LOG变换、直方图均衡（RGB、gray、HSI）**。

RGB转灰度图

```
/**
 * grb ==> gray
 */
QImage ImageProcessing::rbg2gray(const QImage & img)
{
//    QImage ret(img);
//    for(int i = 0; i< ret.width(); i++){
//        for(int j = 0; j < ret.height(); j++){
//            QRgb rgb = img.pixel(i, j);

//            int grayValue = qGray(rgb);
//            ret.setPixelColor(i, j, qRgb(grayValue, grayValue,
grayValue));
//        }
//    }

//    return ret;

    return Matrix<int>::toQImage(Matrix<int>::fromQImage(img, 'h'));
}
```

RGB转灰度只需要将每一个像素点的RGB转成灰度输出即可，当然上面代码用了图像和矩阵的互转，`toQImage`和`fromImage`方法将在后面提到。

图像反转

```
/**
 * reverse the color
 */
QImage ImageProcessing::pixelReverse(const QImage & img)
{
    QImage ret(img);
    for(int i = 0; i< ret.width(); i++){
        for(int j = 0; j < ret.height(); j++){
            QRgb rgb = img.pixel(i, j);

            ret.setPixelColor(i, j, qRgb(-qRed(rgb), -qGreen(rgb), -
qBlue(rgb)));
        }
    }
    return ret;
}
```

图像反转即将每一个像素点的值取负即可。

LOG变换

```
/**
 * log transform for a picture
 */
QImage ImageProcessing::logTransformation(const QImage & img ,int c = 1)
{
    QImage ret(img);

    for(int i = 0; i< ret.width(); i++){
        for(int j = 0; j < ret.height(); j++){

            QRgb rgb = img.pixel(i, j);

            int grayValue = c * log(qGray(rgb)/255.0 + 1) * 255;
            ret.setPixelColor(i, j, qRgb(grayValue, grayValue, grayValue));
        }
    }
    return ret;
}
```

LOG变换将每一个像素通过公式 $g = c \times log(g/255 + 1) \times 255$ 进行处理。

直方图均衡

直方图均衡氛围三种：灰度直方图均衡、RGB直方图均衡和HSI直方图均衡

灰度直方图均衡

```
/**
 * histequ for gray
 */
QImage ImageProcessing::histEquilibrium(const QImage & img){

    QImage ret(img);

    int width = img.width();
    int height = img.height();
    int N = width * height;

    // calculate gray-sum of every gray-level
    int hist[256];
    std::fill(hist, hist + 256, 0);

    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            hist[qGray(img.pixel(i, j))]++;
        }
    }

    int map[256];
    double sum = 0;
    for(int i = 0; i < 256; i++){
        sum += hist[i];
        map[i] = round(sum / N * 255);
    }

    // map
    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            int g = map[qGray(img.pixel(i, j))];
            ret.setPixel(i, j, qRgb(g, g, g));
        }
    }
    return ret;
}
```

首先统计各阶灰度的值，然后归一化，然后做映射即可。

彩色直方图均衡

```
/**
 * histequ for rgb
 */
QImage ImageProcessing::histEquilibriumForRgb(const QImage & img){

    QImage ret(img);

    int width = img.width();
    int height = img.height();
    int N = width * height;

    // count histgram
    int hist[3][256];
    std::fill(hist[0], hist[0] + 3 * 256, 0);

    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            hist[0][qRed(img.pixel(i, j))]++;
            hist[1][qGreen(img.pixel(i, j))]++;
            hist[2][qBlue(img.pixel(i, j))]++;
        }
    }

    // calculate
    int map[3][256];
    double sum[3] = {0};
    for(int i = 0; i < 256; i++){
        sum[0] += hist[0][i];
        sum[1] += hist[1][i];
        sum[2] += hist[2][i];

        map[0][i] = round(sum[0] / N * 255);
        map[1][i] = round(sum[1] / N * 255);
        map[2][i] = round(sum[2] / N * 255);
    }

    // map the pixel
    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            int r = map[0][qRed(img.pixel(i, j))];
            int g = map[1][qGreen(img.pixel(i, j))];
            int b = map[2][qBlue(img.pixel(i, j))];
            ret.setPixel(i, j, qRgb(r, g, b));
        }
    }
    return ret;
}
```

思路和灰度一样，只是将RGB三个通道分开处理即可。

HSI直方图均衡

```
/**
 * histequ in HSI
 */
QImage ImageProcessing::histEquilibriumByHSI(const QImage & img){

    QImage ret(img);

    int width = img.width();
    int height = img.height();
    int N = width * height;

    // count histgram
    int hist[256];
    std::fill(hist, hist + 256, 0);

    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){

            hist[(int)(ImageProcessing::Rgb2Hsi(img.pixel(i, j)).i *
255)]++;
        }
    }

    // calculate
    int map[256];
    double sum = 0;
    for(int i = 0; i < 256; i++){
        sum += hist[i];
        map[i] = round(sum / N * 255);
    }

    // map the pixel
    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){
            HSI temp = ImageProcessing::Rgb2Hsi(img.pixel(i,j));
            temp.i = map[(int)(ImageProcessing::Rgb2Hsi(img.pixel(i, j)).i *
255)] / 255.0;

            ret.setPixel(i, j, ImageProcessing::Hsi2Rgb(temp));
        }
    }
    return ret;
}
```

思路和上面一样，只是需要将RGB图像转到HSI空间处理，然后在转回到RGB做映射，`Rgb2Hsi`和`Hsi2Rgb`方法如下：

```
/**
 * RGB ==> HSI
 */
HSI ImageProcessing::Rgb2Hsi(const QRgb rgb){

    HSI hsi;

    // nromalization
    double R = qRed(rgb) / 255.0;
    double G = qGreen(rgb) / 255.0;
    double B = qBlue(rgb) / 255.0;

    // get the range
    double min = std::min(std::min(R, G), B);
    double max = std::max(std::max(R, G), B);
    double deltaMax = max - min;

    double H;
    double S;
    double I = (max + min) / 2;

    if (deltaMax == 0 ){
        H = 0;
        S = 0;
    }else{
        H = 0;
        if (I < 0.5)
            S = deltaMax / (max + min);
        else
            S = deltaMax / (2 - max - min);

        double deltaR = (((max - R) / 6.0) + (deltaMax / 2.0)) / deltaMax;
        double deltaG = (((max - G) / 6.0) + (deltaMax / 2.0)) / deltaMax;
        double deltaB = (((max - B) / 6.0) + (deltaMax / 2.0)) / deltaMax;

        if (R == max){
            H = deltaB - deltaG;
        }else if (G == max){
            H = 1 / 3.0 + deltaR - deltaB;
        }else if (B == max){
            H = 2 / 3.0 + deltaG - deltaR;
        }

        if (H < 0)
            H += 1;
        if (H > 1)
            H -= 1;
    }
```

```
    hsi.h = H;
    hsi.s = S;
    hsi.i = I;

    return hsi;
}

/**
 * HSI ==> RGB
 */
QRgb ImageProcessing::Hsi2Rgb(const HSI hsi){

    double H = hsi.h;
    double S = hsi.s;
    double I = hsi.i;

    int R = 0;
    int G = 0;
    int B = 0;

    double v1 = 0;
    double v2 = 0;

    if (S == 0) {
        R = I * 255;
        G = I * 255;
        B = I * 255;
    }else {
        if (I < 0.5){
            v2 = I * (1 + S);
        }else{
            v2 = (I + S) - (S * I);
        }

        v1 = 2 * I - v2;

        R = 255 * ImageProcessing::Hue2Rgb(v1, v2, H + 1/3.0);
        G = 255 * ImageProcessing::Hue2Rgb(v1, v2, H);
        B = 255 * ImageProcessing::Hue2Rgb(v1, v2, H - 1/3.0);
    }
    return qRgb(R, G, B);
}

/**
 * Hue ==> Rgb
 */
double ImageProcessing::Hue2Rgb(double v1, double v2,double vH)
{
    if (vH < 0)
```

```
            vH += 1;
        if (vH > 1)
            vH -= 1;

        if ((6 * vH) < 1)
            return v1 + (v2 - v1) * 6 * vH;

        if ((2 * vH) < 1)
            return v2;

        if ((3 * vH) < 2)
            return v1 + (v2 - v1) * (2 / 3.0 - vH) * 6;

        return v1;
    }
```

在`Hsi2Rgb`中涉及到空间`Hue`到`RGB`的转换。

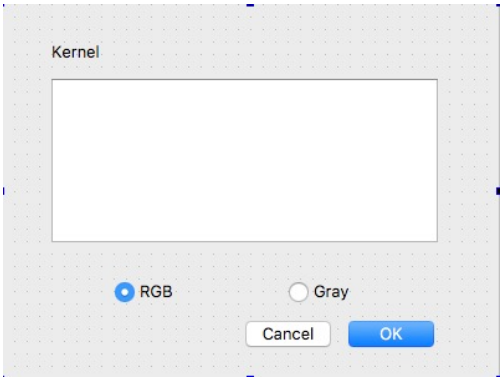
2.3 图像空间滤波

图像空间滤波分为**线性滤波、高斯滤波、双边滤波、中值滤波、RGB和灰度分别的膨胀和腐蚀**等操作。

线性滤波

线性滤波相对最为复杂，因为线性滤波输入为一个未知大小的滤波器，需要进行正则表达式匹配才能得出结果。

首先创建一个新的设计师界面类用来做线性滤波的弹窗，界面如下；



在输入框中输入滤波器的Kernel，通过`QRadioButton`选择滤波模式是RGB还是灰度。

界面类的关键代码如下：

```
void SpacialFilterCernelInput::on_buttonBox_accepted()
{
    QString str = ui->kernelEdit->toPlainText();

    if(str != ""){

        if(ui->rBtnRGB->isChecked()){
            patten = "RGB";
        }else if(ui->rBtnGray->isChecked()){
            patten = "Gray";
        }
        emit(confirmed(str, patten));
    }
}
```

这是OK按钮对应的槽函数，首先获取到Kernel对应的字符串以及当前模式，然后还通过`comfirmed`信号传递出去。

接下来，`MainWindow`中接收到这个信号并且做处理，连接信号-槽：

```
connect(sfDlg,SIGNAL(confirmed(QString,
QString)),this,SLOT(on_SpacialFilterCernelInput_confirmed(QString,
QString)));
```

槽函数如下：

```
/**
 * get parameters from sfDLg and process image
 */
void MainWindow::on_SpacialFilterCernelInput_confirmed(QString str, QString
patten){

    int colNum = 0;

    // get the data from str
    std::vector<std::vector<double>> filterData = getFilterData(str,
colNum);

    // vector2matrix
    Matrix<double> filter(filterData.size(), colNum, 0);
    for(int i = 0; i < filterData.size(); i++){
        for(int j = 0; j < colNum; j++){
            filter(i, j) = filterData[i][j];
        }
    }
    ImageProcessing::filterNormalization(filter);
//    test for vector2matrix
//    qDebug() << "QVector:    " << filterData;
//    std::cout << "Matirx:" << std::endl << filter << std::endl;

    QImage img;
    if(getDisplayImage(img)){

        img = ImageProcessing::linearSpacialFilter(img, filter, patten);
        showImage(img);
    }
}
```

该函数首先将获取信号传递过来的参数，然后将字符串通过`getFilterData()`函数处理成一个二维向量，再转为`Matrix`对象，进行滤波器的归一化后，进行图像的处理。`getFilterData()`如下：

```
std::vector<std::vector<double> > MainWindow::getFilterData(QString & str,
int & colNum){

    std::vector<std::vector<double>> vec;

    // split into lines
    QStringList qsl = str.replace("\n",
"").trimmed().split(QRegExp("\\s*;"));

    // first line
    std::vector<double> vecLine = getFilterDataOfEveryLine(qsl[0], colNum);
    vec.push_back(vecLine);

    // from second line
    for(int i = 1; i < qsl.length() && qsl[i] != ""; i++){

        int curColNum = 0;
        std::vector<double> vecLine = getFilterDataOfEveryLine(qsl[i],
curColNum);
        if(curColNum == colNum){
            vec.push_back(vecLine);
        }
    }
    return vec;
}

std::vector<double> MainWindow::getFilterDataOfEveryLine(QString &str, int &
colNum){

    std::vector<double> vec;

    // split into single item
    QStringList qsl =
str.replace("\n","").trimmed().split(QRegExp("\\s*,"));

    // number of current line
    colNum = qsl.length();
    for(int i = 0; i < qsl.length() && qsl[i] != ""; i++){

        double d = (double)qsl.at(i).toDouble();

        vec.push_back(d);
    }
    return vec;
}
```

先将字符串通过一些符号切割成行的形式，这样就能对每一行进行单独处理，每一行中再将其分割成单独的字符并且转为`double`即可。

获取到矩阵以后，就可以进行空间滤波了，在`ImageProcessing`类中进行滤波，添加静态函数`linearSpacialFilter()`如下：

```
/**
 * linear spatial fiter
 */
QImage ImageProcessing::linearSpacialFilter(const QImage & img, const
Matrix<double> filter, QString patten){

    QImage ret(img);

    int nRow = filter.getNRow();
    int nCol = filter.getNCol();
    if(nRow * nCol % 2 == 0){
        return ret;
    }

    // padding with zero
    QImage paddedImage = ImageProcessing::zeroPadding(ret, nCol, nRow);

    return ImageProcessing::filterImage(paddedImage, filter, patten);
}
```

首先需要根据滤波器的大小对图片进行边缘扩充，有三种方法：补零、重复、镜像。

分别定义如下：

补零

```
/**
 * zero padding
 */
QImage ImageProcessing::zeroPadding(const QImage & img, const int nCol,
const int nRow){

    QImage ret = QImage(img.width() + nCol - 1, img.height() + nRow - 1,
img.format());

    for(int i = 0; i < ret.width(); i++){
        for(int j = 0; j < ret.height(); j++){
            if(i >= nCol/2 && i < ret.width() - nCol/2 && j >= nRow/2 && j <
ret.height() - nRow/2){
                ret.setPixel(i, j, img.pixel(i - nCol/2, j - nRow/2));
            }else{
                ret.setPixel(i, j, qRgb(0, 0, 0));
            }
        }
    }

    return ret;
}
```

重复

```
/**
 * repear padding
 */
QImage ImageProcessing::repeatPadding(const QImage & img, const int nCol,
const int nRow){

    QImage ret = QImage(img.width() + nCol - 1, img.height() + nRow - 1,
img.format());

    for(int i = 0; i < ret.width(); i++){
        for(int j = 0; j < ret.height(); j++){

            // 填充左右侧
            if(i < nCol/2 && j >= nRow/2 && j < ret.height() - nRow/2){
                // 最左侧中间
                ret.setPixel(i, j , img.pixel(0, j - nRow/2));
            }else if(i >= ret.width() - nCol/2 && j >= nRow/2 && j <
ret.height() - nRow/2){
                // 最右侧中间
                ret.setPixel(i, j, img.pixel(img.width() - 1, j - nRow/2));
            }

            // 填充上下侧
            if(j < nRow/2){
```

```
                // 上侧
                if(i >= nCol/2 && i < ret.width() - nCol/2){
                    ret.setPixel(i, j , img.pixel(i - nCol/2, 0));
                }else if(i < nCol/2){
                    ret.setPixel(i, j , img.pixel(0, 0));
                }else if(i >= ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(img.width() - 1, 0));
                }
            }else if(j >= ret.height() - nRow/2){
                // 下侧
                if(i >= nCol/2 && i < ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(i - nCol/2, img.height() -
1));
                }else if(i < nCol/2){
                    ret.setPixel(i, j , img.pixel(0, img.height() - 1));
                }else if(i >= ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(img.width() - 1,
img.height() - 1));
                }
            }

            // 中间
            if(i >= nCol/2 && i < ret.width() - nCol/2 && j >= nRow/2 && j <
ret.height() - nRow/2){
                ret.setPixel(i, j, img.pixel(i - nCol/2, j - nRow/2));
            }
        }
    }

    return ret;
}
```

镜像

```
/**
 * mirror padding
 * 这个镜像真乃神来之笔
 * 打死我也不会改了
 * 这么完美的代码估计也就我能写出来了
 * 要注意不等号
 */
QImage ImageProcessing::mirrorPadding(const QImage & img, const int nCol,
const int nRow){

    // 滤波器的大小不能任性
    if(nCol/2 > img.width() || nRow/2 > img.height()){
        qDebug() << "size error!";
        return img;
    }
```

```
    QImage ret = QImage(img.width() + nCol - 1, img.height() + nRow - 1,
img.format());

    for(int i = 0; i < ret.width(); i++){
        for(int j = 0; j < ret.height(); j++){

            // 填充左右侧
            if(i < nCol/2 && j >= nRow/2 && j < ret.height() - nRow/2){
                // 最左侧中间
                ret.setPixel(i, j , img.pixel(nCol/2 - i, j - nRow/2));
            }else if(i >= ret.width() - nCol/2 && j >= nRow/2 && j <
ret.height() - nRow/2){
                // 最右侧中间
                // 这里的代码，一个礼拜后估计我自己也读不懂了，再过一个月，连上帝都读不
懂了
                ret.setPixel(i, j, img.pixel(2 * ret.width() - 3 * nCol/2 -
i - 1, j - nRow/2));
            }

            // 填充上下侧
            if(j < nRow/2){
                // 上侧
                if(i >= nCol/2 && i < ret.width() - nCol/2){
                    ret.setPixel(i, j , img.pixel(i - nCol/2, nRow/2 - j));
                }else if(i < nCol/2){
                    // r/2 - 1 - i = x - r/2
                    ret.setPixel(i, j , img.pixel(nCol/2 - i - 1, nRow/2 - j
- 1));
                }else if(i >= ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(2 * ret.width() - 3 *
nCol/2 - i - 1, nRow/2 - j - 1));
                }
            }else if(j >= ret.height() - nRow/2){
                // 下侧
                if(i >= nCol/2 && i < ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(i - nCol/2, 2 *
ret.height() - 3 * nCol/2 - j - 1));
                }else if(i < nCol/2){
                    ret.setPixel(i, j , img.pixel(nCol/2 - i - 1, 2 *
ret.height() - 3 * nCol/2 - j - 1));
                }else if(i >= ret.width() - nCol/2){
                    ret.setPixel(i, j, img.pixel(2 * ret.width() - 3 *
nCol/2 - i - 1, 2 * ret.height() - 3 * nCol/2 - j - 1));
                }
            }

            // 中间
```

```
            if(i >= nCol/2 && i < ret.width() - nCol/2 && j >= nRow/2 && j <
ret.height() - nRow/2){
                ret.setPixel(i, j, img.pixel(i - nCol/2, j - nRow/2));
            }
        }
    }

    return ret;
}
```

滤波

图像填充完以后进行滤波，封装为一个静态函数`filterImage()`作为一个通用的滤波函数，参数**为填充后的图像、滤波器、滤波模式**，定义如下：

```
/**
 * filter
 */
QImage ImageProcessing::filterImage(const QImage & img, const Matrix<double>
filter, const QString patten){

//    QImage ret = QImage(img.width() - nCol + 1, img.height() - nRow + 1,
img.format());

//    // int normalRatio = ImageProcessing::filterNormalization(vec, nCol);

//    for(int i = nCol/2; i < img.width() - nCol/2 ; i++){
//        for(int j = nRow/2; j < img.height() - nRow/2; j++){

//            if(patten == "RGB"){
//                // cumpute every block
//                int blockR = ImageProcessing::getBlockResult(img, i, j,
vec, nCol, nRow, 'r');
//                int blockG = ImageProcessing::getBlockResult(img, i, j,
vec, nCol, nRow, 'g');
//                int blockB = ImageProcessing::getBlockResult(img, i, j,
vec, nCol, nRow, 'b');

//                ret.setPixel(i - nCol/2, j - nRow/2, qRgb(blockR, blockG,
blockB));
//            }else if(patten == "Gray"){
//                int blockY = ImageProcessing::getBlockResult(img, i, j,
vec, nCol, nRow, 'y');

//                ret.setPixel(i - nCol/2, j - nRow/2, qRgb(blockY, blockY,
blockY));
//            }else{
//                qDebug() << "patten error!";
//            }
```

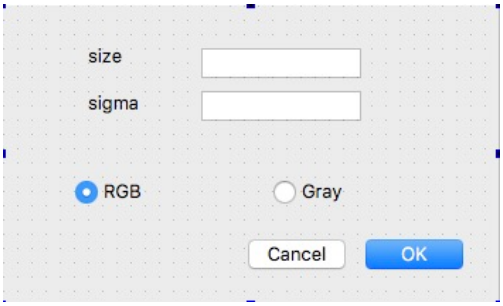
```
//         }
//     }

//     return ret;


    // convolution & normalization
    if(patten == "RGB"){
        Matrix<double> tempMatR =
Matrix<double>::convolution(Matrix<double>::fromQImage(img, 'r'), filter);
        Matrix<double> tempMatG =
Matrix<double>::convolution(Matrix<double>::fromQImage(img, 'g'), filter);
        Matrix<double> tempMatB =
Matrix<double>::convolution(Matrix<double>::fromQImage(img, 'b'), filter);
        Matrix<int> R = Matrix<int>::normalization(tempMatR);
        Matrix<int> G = Matrix<int>::normalization(tempMatG);
        Matrix<int> B = Matrix<int>::normalization(tempMatB);
        return Matrix<int>::toQImage(R, G, B);
    }else if(patten == "Gray"){
        Matrix<double> tempMat =
Matrix<double>::convolution(Matrix<double>::fromQImage(img, 'h'), filter);
        Matrix<int> H = Matrix<int>::normalization(tempMat);
        return Matrix<int>::toQImage(H);
    }
}
```

最初的版本使用的是逐块滤波（每一次取出滤波器大小的图像块进行处理的到结果），后来引入了`Matrix`以后直接将图像转为矩阵然后与滤波器进行**卷积操作**，然后将卷积的到的结果转位图像，*涉及到`Matrix`类中的方法在后面会详细说明*。

高斯滤波

高斯滤波本质上就是线性滤波的一个特殊情况，所以我们借助线性滤波的方法即可。

首先，设计师界面如下：



只要输入滤波器大小和方差即可。

该类关键代码如下：

```
void GaussBlurDialog::on_buttonBox_accepted()
{
    bool sizeOK;
    bool sigmaOK;

    int size = (int)(ui->sizeInputEdit->text().toDouble(&sizeOK));
    double sigma = ui->sigmaInputEdit->text().toDouble(&sigmaOK);

    if(sizeOK && sigmaOK){

        if(ui->rBtnRGB->isChecked()){
            patten = "RGB";
        }else if(ui->rBtnGray->isChecked()){
            patten = "Gray";
        }
        emit confirmed(size, sigma, patten);
    }
}
```

与之前类似，将参数作为信号发送出去。

`ImageProcessing`类中添加静态函数`gaussBlurFilter()`：

```
/**
 * gauss blur
 */
QImage ImageProcessing::gaussBlurFilter(const QImage & img, const int
filterSize, const double sigma, QString patten){

    // calculate the gauss filter kernel
    Matrix<double> kernel = ImageProcessing::computeGaussFilter(filterSize,
sigma);

    return ImageProcessing::linearSpacialFilter(img, kernel, patten);
}
```

首先调用`computerGaussFilter()`计算高斯核：

```
/**
 * compute the gauss filter paremeter
 */
Matrix<double> ImageProcessing::computeGaussFilter(const int filterSize,
const double sigma){

    Matrix<double> kernel(filterSize, filterSize, 0);
    for(int i = 0; i < filterSize; i++){
        for(int j = 0; j < filterSize; j++){

            double x = (i - filterSize/2) * (i - filterSize/2);
            double y = (j - filterSize/2) * (j - filterSize/2);

            kernel(i, j) = (exp(-((x + y) / 2 / sigma / sigma)));
        }
    }

    double sum = 0;
    for(int i = 0; i < filterSize; i++){
        for(int j = 0; j < filterSize; j++){
            sum += kernel(i, j);
        }
    }

    for(int i = 0; i < filterSize; i++){
        for(int j = 0; j < filterSize; j++){
            kernel(i, j) /= sum;
        }
    }

    return kernel;
}
```

然后通过线性滤波即可。

双边滤波

首先来看双边滤波的公式：

$$k(i,j) = exp(-(\frac{(x_i-x_c)^2}{2\sigma_x^2} + \frac{(y_i-y_c)^2}{2\sigma_y^2})) \times exp(-\frac{(I(x_i,y_i)-I(x_c,y_c))^2}{2\sigma_c^2})$$

双边滤波要求对于每一个图像块单元，都采取不同的滤波器，所以上面的方法不再通用，直接给出核心代码：

```
/**
 * bilateral filter
 */
QImage ImageProcessing::bilateralFilter(const QImage & img, const int
filterSize, const double sigma, const double anotherSigma){

    // calculate the gauss-filter kernel
    Matrix<double> gaussFilter =
ImageProcessing::computeGaussFilter(filterSize, sigma);
    // get the distances of every level of bilateral filter
    Matrix<double> bilateralPar =
ImageProcessing::computeEveryDistance(anotherSigma);

    // padding
    Matrix<int> proImg =
Matrix<int>::fromQImage(ImageProcessing::zeroPadding(img, filterSize,
filterSize), 'h');
    Matrix<int> retImg(img.height(), img.width(), 0);

    for(int i = filterSize/2; i < proImg.getNRow() - filterSize/2; i++){
        for(int j = filterSize/2; j < proImg.getNCol() - filterSize/2; j++){

            Matrix<double> curKernel(filterSize, filterSize, 0);
            for(int ii = -filterSize/2; ii <= filterSize/2; ii++){
                for(int jj = -filterSize/2; jj <= filterSize/2; jj++){

                    curKernel(ii + filterSize/2, jj + filterSize/2) =
gaussFilter(ii + filterSize/2, jj + filterSize/2) *
                            bilateralPar(0, abs(proImg(i, j) - proImg(i+ii,
j+jj)));
                }
            }

            ImageProcessing::filterNormalization(curKernel);
            retImg(i - filterSize/2, j - filterSize/2) =
ImageProcessing::getBlockResult(proImg, i, j, curKernel);
        }
    }
    return Matrix<int>::toQImage(retImg);
}
```

首先计算出高斯滤波器，然后计算出所有亮度值差值的平方（即0-255的平方）并存储起来，使用2重循环，对每一块计算出不同的滤波器，然后再用`getBlockResult()`函数计算出当前块得出的结果，这里开始使用的图片处理，引入`Matrix`类以后直接采用矩阵计算。

`getBlockResult()`函数定义如下；

```
int ImageProcessing::getBlockResult(const Matrix<int> & img, int i, int j,
Matrix<double> filter){

    double sum = 0;

    int nCol = filter.getNCol();
    int nRow = filter.getNRow();

    for(int x = 0; x < nRow; x++){
        for(int y = 0; y < nCol; y++){
            sum += img(i- nRow/2 + x, j- nCol/2 + y) * filter(x, y);
        }
    }

    return (int)sum;
}
```

这样双边滤波完成了。

中值滤波&膨胀&腐蚀

中值滤波&膨胀&腐蚀都放在一段代码里处理，关键处理代码如下；

```
/**
 * median filter
 * expand && corrosion
 */
QImage ImageProcessing::medianFilter(const QImage & img, const int size,
const QString filterPatten, const QString colorPatten){

    QImage padImg = ImageProcessing::zeroPadding(img, size, size);

    QImage ret(img);

    for(int i = size/2; i < img.width(); i++){
        for(int j = size/2; j < img.height(); j++){

            if(colorPatten == "Gray"){
                std::vector<int> block;
                for(int ii = -size/2; ii <= size/2; ii++){
                    for(int jj = -size/2; jj <= size/2; jj++){
                        block.push_back(qGray(padImg.pixel(i+ii, j+jj)));
                    }
                }
                std::sort(block.begin(), block.end());

                int gray;
                if(filterPatten == "median"){
                    gray = block[block.size()/2];
```

```
                }else if(filterPatten == "expand"){
                    gray = block[block.size() - 1];
                }else if(filterPatten == "corrosion"){
                    gray = block[0];
                }
                ret.setPixel(i - size/2, j - size/2, qRgb(gray, gray,
gray));

            }else if(colorPatten == "RGB"){
                std::vector<int> blockR;
                std::vector<int> blockG;
                std::vector<int> blockB;
                for(int ii = -size/2; ii <= size/2; ii++){
                    for(int jj = -size/2; jj <= size/2; jj++){
                        blockR.push_back(qRed(padImg.pixel(i+ii, j+jj)));
                        blockG.push_back(qGreen(padImg.pixel(i+ii, j+jj)));
                        blockB.push_back(qBlue(padImg.pixel(i+ii, j+jj)));
                    }
                }
                std::sort(blockR.begin(), blockR.end());
                std::sort(blockG.begin(), blockG.end());
                std::sort(blockB.begin(), blockB.end());

                int red;
                int green;
                int blue;
                if(filterPatten == "median"){
                    red = blockR[blockR.size()/2];
                    green = blockG[blockG.size()/2];
                    blue = blockB[blockB.size()/2];
                }else if(filterPatten == "expand"){
                    red = blockR[blockR.size() - 1];
                    green = blockG[blockG.size() - 1];
                    blue = blockB[blockB.size() - 1];
                }else if(filterPatten == "corrosion"){
                    red = blockR[0];
                    green = blockG[0];
                    blue = blockB[0];
                }
                ret.setPixel(i - size/2, j - size/2, qRgb(red, green,
blue));
            }
        }
    }

    return ret;
}
```

三个处理过程一样，区别在于每一次去的是当前块中的不同值：

- 最大值：膨胀
- 最小值：腐蚀
- 中间值：中值

中值滤波滤波器大小需要自己设定，膨胀&腐蚀固定大小为3。

2.4 矩阵类

矩阵类是一个针对**图像处理**、**矩阵操作**、**频域变换**等功能抽象/封装出来的一个模版类，完整代码在文件`matrixTemplate.cpp`中，傅立叶变换包含的代码在`fft.h`和`fft.cpp`中，下面介绍`Matrix<T>`类中常用的一些函数成员。

函数成员

- `T& operator()(size_t r, size_t c) const`：运算符重载，用于获取矩阵指定元素，用法`Matrix<T> a(i, j)`
- `Matrix<T> subMatrix(size_t rs, size_t re, size_t cs, size_t ce) const`：用于父矩阵中获取子矩阵。
- `friend std::ostream& operator<<(std::ostream& out, const Matrix<U>& rhs)`：符号`<<`重载，用于`std::cout`输出。
- `size_t getNRow()/getNCol() const`：获取行数和列数。
- `std::vector<T> getRow(size_t r)/getCol(size_t c) const`：获取某一行或某一列。
- `bool setRow(size_t r, const std::vector<T>& d)/setCol(size_t c, const std::vector<T>& d)`：设置某一行或某一列。
- `Matrix<T>& transpose()`：矩阵转置。

- ```
  // operator +
  Matrix<T> operator+(Matrix<T>& otherMat) const;
  Matrix<T> operator+(T factor) const;

  template<typename U>
  friend Matrix<U> operator+(const U factor, Matrix<U>& mat);

  // operator -
  Matrix<T> operator-(Matrix<T>& otherMat) const;
  Matrix<T> operator-(T factor) const;

  template<typename U>
  friend Matrix<U> operator-(const U factor, Matrix<U> & mat);

  // operator *
  Matrix<T> operator*(Matrix<T>& otherMat) const;
  T mul(vector<T> vec1, vector<T> vec2) const;

  Matrix<T> operator*(T factor) const;

  template<typename U>
  friend Matrix<U> operator*(const U factor, Matrix<U> & mat);

  // operator /
  template<typename U>
  friend Matrix<U> operator/(Matrix<U>& mat, const U factor);
  ```

  与矩阵加减乘除相关的运算符重载。

- `static Matrix<T> fromQImage(const QImage&, char)`：静态方法，将`QImage`转为`Matrix`，实现如下：
  ```

```
// translate between QImage & Matrix
template<typename T>
Matrix<T> Matrix<T>::fromQImage(const QImage& img, char patten)
{
    Matrix<T> ret(img.height(), img.width(), 0);

    for(int i = 0; i < img.height(); i++){
        for(int j = 0; j < img.width(); j++){
            if(patten == 'r'){
                ret(i, j) = qRed(img.pixel(j, i));
            }else if(patten == 'g'){
                ret(i, j) = qGreen(img.pixel(j, i));
            }else if(patten == 'b'){
                ret(i, j) = qBlue(img.pixel(j, i));
            }else if(patten == 'h'){
                ret(i, j) = qGray(img.pixel(j, i));
            }
        }
    }

    return ret;
}
```

根据参数需求，返回不同通道对应的矩阵。

- ```
  static QImage toQImage(const Matrix<int>&);
  static QImage toQImage(const Matrix<double>&);
  static QImage toQImage(const Matrix<int>&, const Matrix<int>&, const
  Matrix<int>&);
  static QImage toQImage(const Matrix<double>&, const Matrix<double>&,
  const  Matrix<double>&);
  ```

将矩阵转为图像，进行了不同的重载，实现如下：

```
// translate between QImage & Matrix
template<typename T>
Matrix<T> Matrix<T>::fromQImage(const QImage& img, char patten)
{
    Matrix<T> ret(img.height(), img.width(), 0);

    for(int i = 0; i < img.height(); i++){
        for(int j = 0; j < img.width(); j++){
            if(patten == 'r'){
                ret(i, j) = qRed(img.pixel(j, i));
            }else if(patten == 'g'){
                ret(i, j) = qGreen(img.pixel(j, i));
            }else if(patten == 'b'){
                ret(i, j) = qBlue(img.pixel(j, i));
```
```

```
            }else if(patten == 'h'){
                ret(i, j) = qGray(img.pixel(j, i));
            }
        }
    }

    return ret;
}

template<typename T>
QImage Matrix<T>::toQImage(const Matrix<int>& gray)
{
    int width = gray.getNCol();
    int height = gray.getNRow();
//    std::cout << width << std::endl;
//    std::cout << height << std::endl;

    QImage img = QImage(width, height, QImage::Format_RGB32);
    for(int i = 0; i < width; i++){
        for(int j =0; j < height; j++){
            int g = gray(j, i);
            img.setPixel(i, j, qRgb(g, g, g));
        }
    }
    return img;
}

template<typename T>
QImage Matrix<T>::toQImage(const Matrix<double> & gray)
{
    return Matrix<int>::toQImage(Matrix<int>::normalization(gray));
}

template<typename T>
QImage Matrix<T>::toQImage(const Matrix<int>& red, const Matrix<int>&
green, const Matrix<int>& blue)
{
    // illegal size
    if(red.getNCol() != blue.getNCol() || red.getNCol() !=
green.getNCol() || green.getNCol() != blue.getNCol() ||
            red.getNRow() != blue.getNRow() || red.getNRow() !=
green.getNRow() || green.getNRow() != blue.getNRow()){
        throw range_error("toQImage :    illegal size of matrix");
    }

    int width = red.getNCol();
    int height = red.getNRow();
    QImage img = QImage(width, height, QImage::Format_RGB32);
    for(int i = 0; i < width; i++){
```

```
        for(int j =0; j < height; j++){
            int r = red(j, i);
            int g = green(j, i);
            int b = blue(j, i);
            img.setPixel(i, j, qRgb(r, g, b));
        }
    }
    return img;
}

template<typename T>
QImage Matrix<T>::toQImage(const Matrix<double>& red, const
Matrix<double>& green, const Matrix<double>& blue){
    return Matrix<int>::toQImage(Matrix<int>::normalization(red),
Matrix<int>::normalization(green), Matrix<int>::normalization(blue));
}
```

主要就是将代表不同类型图像的矩阵转为图像，若输入的参数为1个则为灰度图，3个则为彩色图，输入为`double`需要进行归一化转为整数。

- `static Matrix<double> abs4complex(const Matrix<std::complex<double> > &)`：复数矩阵求绝对值，实现：

```
template<typename T>
Matrix<double> Matrix<T>::abs4complex(const Matrix<std::complex<double>
> & mat)
{
    int width = mat.getNCol();
    int height = mat.getNRow();

    Matrix<double> res(height, width, 0);

    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            res(i, j) = abs(mat(i, j));
        }
    }

    return res;
}
```

对每个单独元素求绝对值即可。

- `static Matrix<T> logtranslate(const Matrix<T> &, T c)`：对每个元素进行**LOG变换**。
- `static void fftshift(Matrix<T> &)`：循环移位，实现：

```
template<typename T>
void Matrix<T>::fftshift(Matrix<T> & mat)
{
    mat.setStartC(mat.getNCol()/2);
    mat.setStartR(mat.getNRow()/2);
}
```

这里的实现是直接修改类中的标记变量，这样在读取元素的时候就可以达到循环移位的效果而且不需要修改本身的数据结构，在`()`重载函数中可以看出：

```
template<typename T>
T& Matrix<T>::operator()(size_t r, size_t c) const
{
    // get element in (r,c)
    // remember check first
    // different of _t has different order
    if(!checkRange(r, c)){
        // out of range
        throw range_error("operator() :    Out Of Range");
    }

    if(_t == 1){
        return _data[(r+_startR)%_nRow][(c+_startC)%_nCol];
    }else{
        return _data[(c+_startC)%_nCol][(r+_startR)%_nRow];
    }
}
```

这里的`startC`和`startR`就是用来标记读取起始点的。

- `static Matrix<int> normalization(const Matrix<double> &)`：矩阵归一化，该函数针对`QImage`的颜色范围进行处理，将本来是`double`类型的数据转为`int`且将范围映射到`[0, 255]`，处理公式为：

$$result = (int)\{\frac{mat(i,j)-min}{max-min} \times 255\}$$

代码实现：

```cpp
template<typename T>
Matrix<int> Matrix<T>::normalization(const Matrix<double> &mat)
{
    int width = mat.getNCol();
    int height = mat.getNRow();

    Matrix<int> res(height, width, 0);

    double max = mat(0, 0), min = mat(0, 0);
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            if(mat(i, j) > max){
                max = mat(i, j);
            }
            if(mat(i, j) < min){
                min = mat(i, j);
            }
        }
    }

    double dis = max - min;
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            res(i, j) = (int) ((mat(i, j) - min)/dis * 255);
        }
    }

    return res;
}
```

- `static Matrix<T> inverseMatrix(const Matrix<T> &, bool *)`：3×3矩阵求逆，主要在仿射变换中使用，实现如下：

```
template<typename T>
Matrix<T> Matrix<T>::inverseMatrix(const Matrix<T> & mat, bool
*isInverse)
{

    // this function just used for matrix with a size of 3*3
    if(mat.getNCol() != 3 || mat.getNRow() != 3){
        *isInverse = false;
        return Matrix(mat);
    }

    T determinant = mat(0, 0)*mat(1, 1)*mat(2, 2) + mat(0, 1)*mat(1,
2)*mat(2, 0) + mat(0, 2)*mat(1, 0)*mat(2, 1) -
            mat(0, 2)*mat(1, 1)*mat(2, 0) - mat(0, 1)*mat(1, 0)*mat(2,
2) - mat(0, 0)*mat(1, 2)*mat(2, 1);
    if(determinant == 0){
        *isInverse = false;
        return Matrix(mat);
    }

    *isInverse = true;
    Matrix<T> inversedMat(3, 3, 0);

    inversedMat(0, 0) = (mat(1, 1)*mat(2, 2) - mat(1, 2)*mat(2,
0))/determinant;
    inversedMat(1, 0) = -1 * (mat(1, 0)*mat(2, 2) - mat(1, 2)*mat(2,
0))/determinant;
    inversedMat(2, 0) = (mat(1, 0)*mat(2, 1) - mat(1, 1)*mat(2,
0))/determinant;
    inversedMat(0, 1) = -1 * (mat(0, 1)*mat(2, 2) - mat(0, 2)*mat(2,
1))/determinant;
    inversedMat(1, 1) = (mat(0, 0)*mat(2, 2) - mat(0, 2)*mat(2,
0))/determinant;
    inversedMat(2, 1) = -1 * (mat(0, 0)*mat(2, 1) - mat(0, 1)*mat(2,
0))/determinant;
    inversedMat(0, 2) = (mat(0, 1)*mat(1, 2) - mat(0, 2)*mat(1,
1))/determinant;
    inversedMat(1, 2) = -1 * (mat(0, 0)*mat(1, 2) - mat(0, 2)*mat(1,
0))/determinant;
    inversedMat(2, 2) = (mat(0, 0)*mat(1, 1) - mat(0, 1)*mat(1,
0))/determinant;

    return inversedMat;
}
```

首先需要判断是否可逆，然后再直接计算，因为这里是简单的矩阵求逆，若是复杂求解，需要使用高斯约旦消元法求解，这里只是为了做3阶矩阵求逆，故没有使用该方法。

- `static void map(const Matrix<double> &, const T, const T, T *, T *)`：矩阵映射，仿照`QMatrix`类进行实现。
- `static Matrix<T> convolution(const Matrix<T> &, const Matrix<T> &)`：卷积，公式：

$$y(n) = \Sigma_{i=-\infty}^{\infty} x(i)h(n-i)$$

实现如下：

```
template<typename T>
Matrix<T> Matrix<T>::convolution(const Matrix<T> & imageMat, const
Matrix<T> & filterMat)
{
    // input: padded image & unmirrored filter
    Matrix<T> fliter = Matrix(filterMat);
    for(int i = 0; i < fliter.getNRow(); i++){
        for(int j = 0; j < fliter.getNCol(); j++){
            fliter(i, j) = filterMat(filterMat.getNRow()-i-1,
filterMat.getNCol()-j-1);
            fliter(fliter.getNRow()-i-1, fliter.getNCol()-j-1) =
filterMat(i, j);
        }
    }
    // cout << fliter << endl;
    Matrix<T> res(imageMat.getNRow() - fliter.getNRow() + 1,
imageMat.getNCol() - fliter.getNCol() + 1, 0);
    for(int i = 0; i < res.getNRow(); i++){
        for(int j = 0; j < res.getNCol(); j++){
            T temp = 0;
            for(int x = 0; x < fliter.getNRow(); x++){
                for(int y = 0; y < fliter.getNCol(); y++){
                    temp += fliter(x, y) * imageMat(y+i, x+j);
                }
            }
            res(i, j) = temp;
        }
    }
    return res;
}
```

首先将滤波器镜像（其实大多数滤波器本身就是对称的），然后根据卷积公式计算即可。

- ```
  static Matrix<T> multiplication(const Matrix<T> &, const Matrix<T> &);
  static Matrix<std::complex<double> > multiplication(const
  Matrix<std::complex<double> > &, const Matrix<double> &);
  static Matrix<std::complex<double> > multiplication(const Matrix<double>
  &, const Matrix<std::complex<double> > &);
  ```

  点乘，对位相乘即可，有多种形式重载。
  ```

关于矩阵的总结

以上就是矩阵类的常用方法了，当然，本身类的实现远不止这些，就不赘述了。

这个矩阵类其实有一些不太完美，还有一些至今没能弄明白的问题，为了减少运算，所以在`transpose()`和`fftshift()`中都是采用变量进行标记，例如使用`_t`标记转置，用`startC`和`startR`标记循环移位，这种性能提升加上c++这门语言本身过于灵活也带来一些难以解决的问题，经过很多测试和修改`fftshift()`已经解决了，但是转置还有一些问题，但是这些问题又不是完全因为转置带来的，只是这些问题会将转置结果弄得非常麻烦，关于这个问题，我会在**问题和解决方案**中详细说明。

2.5 频域滤波

本次实验，实现了**空间域到频域的转换(FFT)**，并且实现了**频域的高斯低通滤波**。

FFT

这里的FFT主要采用基2时抽取法，通过递归 / 分治实现。

FFT所有功能封装在`fft.cpp`中，主要包含下面几个函数：

- `size_t calcN(size_t length)`：根据`length`计算最接近的2的整次幂，实现如下：

```
size_t calcN(size_t length) {
    // check if length is power of 2
    // if it is, just return length
    // if not, get the correct N and return

    if(0 == (length & (length-1))){
        return length;
    }

    // calc the correct N
    vector<size_t> vec;
    while(length){
        vec.push_back(length);
        length = length >> 1;
    }

    size_t res = vec[0];
    for(int i = 1; i < vec.size(); i++){
        res |= vec[i];
    }
    return res + 1;
}
```

首先根据**如果N和N-1按位与的结果为0，则N是2的整数次幂**判断当前输入是否是2点整次幂，若不是，**将N依次右移位直到全为0，所有结果相或并且加1**求得结果。

- ```
  std::complex<double> wnk(size_t N, int k);
  std::complex<double> w_minusn_k(size_t N, int k);
  ```

  用来计算傅立叶变换中的 W_N^{nk} 和 W_N^{-nk}，主要是使用了欧拉公式，如下：

  ```
  complex<double> wnk(size_t N, int k){
      // calc the W_{N}^{k}
      double p = 2 * M_PI * k /N;
      return complex<double>(cos(p), -1 * sin(p));
  }

  complex<double> w_n_minusk(size_t N, int k){
      // calc the W_{N}^{-k}
      double p = -2 * M_PI * k /N;
      return complex<double>(cos(p), -1 * sin(p));
  }
  ```

- `vector<std::complex<double> > calfft(vector<std::complex<double> > data, size_t N, char patten)`：用来进行FFT计算，所有的FFT和IFFT计算都是基于该函数，采用递归的方式实现，可以计算一维的FFT和IFFT，公式如下：

$$X(k) = X_1(k) + W_N^k X_2(k)$$

$$X(k + N/2) = X_1(k) - W_N^k X_2(k)$$

  实现如下：

  ```
  vector<complex<double> >
  calfft(vector<complex<double> > data, size_t N, char patten) {

      // patten:
      //         'i': ifft
      //         'f': fft

      // change length to make it beign just the power of 2
      N = calcN(N);
      // append 0 if necessary
      while (N > data.size()){
          data.push_back(complex<double>(0.0, 0.0));
      }

      // start fft
      // check if N is 0, 1, 2
      // if N is 0 or 1, just return data
      // if N is 2, do dft on them
      // if N > 2, do fft
      // 1. split input into two part
      // 2. do fft on them seperately
      // 3. construct result from output
  ```
  ```

```
    vector<complex<double> > res;
    if(N == 1 || N == 0){
        return data;
    }else if(N == 2){
        if(patten == 'i'){
            res.push_back(w_n_minusk(2, 0)*data[0] + w_n_minusk(2,
0)*data[1]);
            res.push_back(w_n_minusk(2, 0)*data[0] + w_n_minusk(2,
1)*data[1]);
        }else if(patten == 'f'){
            res.push_back(wnk(2, 0)*data[0] + wnk(2, 0)*data[1]);
            res.push_back(wnk(2, 0)*data[0] + wnk(2, 1)*data[1]);
        }
        return res;
    }else{
        // split
        vector<complex<double> > evenItems;
        vector<complex<double> > oddItems;
        for(int i = 0; i < data.size(); i++){
            if(i % 2 == 0){
                evenItems.push_back(data[i]);
            }else{
                oddItems.push_back(data[i]);
            }
        }

        // fft/ifft seperately
        vector<complex<double> > evenRes = calfft(evenItems, N/2,
patten);
        vector<complex<double> > oddRes = calfft(oddItems, N/2, patten);

        // construct
        for(int i = 0; i < N/2; i++){
            if(patten == 'i'){
                res.push_back(evenRes[i] + w_n_minusk(N, i) *
oddRes[i]);
            }else if(patten == 'f'){
                res.push_back(evenRes[i] + wnk(N, i) * oddRes[i]);
            }
        }
        for(int i = 0; i < N/2; i++){
            if(patten == 'i'){
                res.push_back(evenRes[i] - w_n_minusk(N, i) *
oddRes[i]);
            }else if(patten == 'f'){
                res.push_back(evenRes[i] - wnk(N, i) * oddRes[i]);
            }
        }
    }
```

```
    return res;
}
```

- ```
  // fft of vector<int>
  std::vector<std::complex<double> > fft(std::vector<int> data, size_t
  N=0);
  // fft of vector<double>
  std::vector<std::complex<double> > fft(std::vector<double> data, size_t
  N=0);
  // fft of complex
  std::vector<std::complex<double> > fft(std::vector<std::complex<double>
  > data, size_t N=0);
  ```

  这是外部调用的FFT函数的重载，主要针对输入的参数不同采取不同的方式。

- `std::vector<std::complex<double> > ifft(std::vector<std::complex<double> > data, size_t N=0)`：计算IFFT，只是在调用`calcfft`传入一个不同的`patten`。
- `Matrix<std::complex<double> > expand(const Matrix<std::complex<double> >& mat, size_t row, size_t col)`：在计算二维FFT的时候，需要对整个矩阵 / 图片进行扩展，变成长宽都是2^n的形式。

- ```
  Matrix<std::complex<double> > fft2d(const Matrix<int>& mat, size_t row,
  size_t col);
  Matrix<std::complex<double> > fft2d(const Matrix<double>& mat, size_t
  row, size_t col);
  Matrix<std::complex<double> > fft2d(const Matrix<std::complex<double> >&
  mat, size_t row, size_t col);
  Matrix<std::complex<double> > ifft2d(const Matrix<std::complex<double>
  >& mat, size_t row, size_t col);
  ```

  二维的FFT和IFFT，找一段代码看一下实现方式：

```
Matrix<complex<double> >
fft2d(const Matrix<complex<double> >& mat, size_t row, size_t col){

    // expand
    Matrix<complex<double> > matExpand = expand(mat, row, col);
    return fftRow(fftRow(matExpand, calcN(row), calcN(col),
'f').transpose(), calcN(col), calcN(row), 'f').transpose();
}
```

就是对当前输入的二维矩阵，对每一行做一次FFT，然后转置，转置后再对每一行做FFT，然后转置得到结果。

其中`fftRow()`实现如下：

```
Matrix<complex<double> >
fftRow(const Matrix<std::complex<double> >& mat, size_t row, size_t col,
char patten){

    // calculate every row of the matrix
    Matrix<complex<double> > newMat(row, col, complex<double>(0, 0));

    for(int i = 0; i < row; i++){
        vector<complex<double>> vecRow;
        if(patten == 'f'){
            vecRow = fft(mat.getRow(i), col);
        }else if(patten == 'i'){
            vecRow = ifft(mat.getRow(i), col);
        }
        if(!newMat.setRow(i, vecRow)){
            // fail to set the row of newMat;
            // I don't know how to solve this now;
        };
    }
    return newMat;
}
```

对图片进行FFT

对图片进行FFT主要进行下面几个步骤：

1. 图像转为矩阵，一般转为灰度矩阵
2. 二维傅立叶变换
3. 取模
4. 进行对数变换
5. 做`fftshift`
6. 归一化
7. 得到的矩阵转为`QImage`

在Qt中实现如下：

```
void MainWindow::on_actionFFT_triggered()
{
    QImage img;

    if(getDisplayImage(img)){

        Matrix<int> g = Matrix<int>::fromQImage(img, 'h');
        Matrix<std::complex<double> > ff = fft2d(g, g.getNRow(),
g.getNCol());
        Matrix<double> ffta = Matrix<double>::abs4complex(ff);
        Matrix<double> fftl = Matrix<double>::logtranslate(ffta, 1);
        Matrix<double>::fftshift(fftl);
        Matrix<int> fftres = Matrix<int>::normalization(fftl);
        QImage res = Matrix<int>::toQImage(fftres);
        showImage(res);
    }
}
```

这些静态函数都在前面的`matrixTemplate.cpp`和`fft.cpp`中实现了。

频域高斯滤波

创建新的设计师界面类，进行参数传递，然后在`ImageProcessing`类中实现频域滤波。

频域低通和高通滤波实现上是一样的，只是有细节上的不同，所以写在一起。

频域滤波的Matlab代码如下：

```
im = imread('pictures/test.jpg');
im = rgb2gray(im);
f = fftshift(fft2(im));
% Do your filter here
i = abs(ifft2(fftshift(f)));
figure, imshow(i, []);
```

所以低通滤波分为以下几个步骤：

1. 图像转为灰度矩阵
2. `fft2d`
3. `fftshift`
4. 生成滤波器，滤波
5. `fftshift`
6. `ifft2d`
7. 求绝对值
8. 归一化
9. 转为图片输出

滤波采用高斯滤波，低通滤波器和高通滤波器表达式为：

低通：$H(u,v) = exp(-D^2(u,v)/2\sigma^2), D(u,v) <= D_0$

高通：$H(u,v) = 1 - exp(-D^2(u,v)/2\sigma^2), D(u,v) <= D_0$

代码实现如下：

```
/**
 * gauss low/hight pass filter
 * procssing in frequency
 */
QImage ImageProcessing::gaussLPFilter(const QImage &img, const int d, const
int sigma, const char patten)
{

    /**
     * Code in Matlab:
     *
     *      im = imread('pictures/test.jpg');
     *      im = rgb2gray(im);
     *
     *      f = fftshift(fft2(im));
     *      % Do your filter here
     *      i = abs(ifft2(fftshift(f)));
     *      figure, imshow(i, []);
     *
     *  rgb2gray ==> fft2d ==> fftshift ==> filter ==> fftshift ==> ifft2d
==> abs ==> normalization.
     *
     */

    // fft2
    Matrix<int> im = Matrix<int>::fromQImage(img, 'h');
    Matrix<std::complex<double> > f = fft2d(im, im.getNRow(), im.getNCol());
    Matrix<std::complex<double> >::fftshift(f);

    // gauss filter
    Matrix<double> filter(f.getNRow(), f.getNCol(), 0);
    int width = filter.getNCol();
    int height = filter.getNRow();
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            double dis = (width/2-j)*(width/2-j) + (height/2-i)*(height/2-
i);
            if(dis <= d * d){
                if(patten == 'l'){
                    filter(i, j) = exp((-1 * dis)/(2 * sigma * sigma));
                }else if(patten == 'h'){
                    filter(i, j) = 1- exp((-1 * dis)/(2 * sigma * sigma));
                }
            }else{
                if(patten == 'l'){
                    filter(i, j) = 0;
```

```
                }else if(patten == 'h'){
                    filter(i, j) = 1;
                }
            }
        }
    }

    // multiplication
    Matrix<std::complex<double> > multiRes =
Matrix<double>::multiplication(f, filter);

    // ifft2
    Matrix<std::complex<double> >::fftshift(multiRes);
    Matrix<std::complex<double> > ifftRes = ifft2d(multiRes,
multiRes.getNRow(), multiRes.getNCol());
    Matrix<double> absRes = Matrix<int>::abs4complex(ifftRes);
    Matrix<int> res = Matrix<int>::normalization(absRes);

    // cut
    return Matrix<int>::toQImage(res.subMatrix(0, img.height(), 0,
img.width()));
}
```

在滤波环节，先生成滤波器然后与原图卷积即可。

2.5 仿射变换

图像旋转

仿射变换固定了矩阵中5个参数，所以这里只实现旋转变换。

旋转变换过程如下：

1. 通过变换矩阵，求解图像四个顶点旋转后的位置
2. 根据旋转后的顶点坐标确定整个新图像大小和位置平移量
3. 对新图像的每个点，通过变换矩阵的逆矩阵求的原图的坐标点
4. 对于求的的非整形坐标，通过**最邻近差值 / 双线性插值 / 双三次线性插值**方法找到原图对应的点
5. 将原图点像素拷贝给新图对应点

代码实现如下：

```
/**
 * geometry translate
 * rotation
 */
QImage ImageProcessing::getmetryRotate(const QImage &img, Matrix<double>
matrix)
{
    bool isInversed;
```

```
    Matrix<double> inversedMat = Matrix<double>::inverseMatrix(matrix,
&isInversed);
    if(!isInversed){
        QImage ret(img);
        return ret;
    }

    // calculate the range/size of the new picture
    // the four
    int x1 = 0, y1 = 0;
    int x2 = img.width(), y2 = 0;
    int x3 = 0, y3 = img.height();
    int x4 = img.width(), y4 = img.height();

    // coordinates of four vertex
    int tx1, ty1, tx2, ty2, tx3, ty3, tx4, ty4;
    Matrix<int>::map(matrix, x1, y1, &tx1, &ty1);
    Matrix<int>::map(matrix, x2, y2, &tx2, &ty2);
    Matrix<int>::map(matrix, x3, y3, &tx3, &ty3);
    Matrix<int>::map(matrix, x4, y4, &tx4, &ty4);

//    QMatrix m = QMatrix(0.8, 0.5, -0.5, 0.8, 0, 0);
//    QMatrix im = m.inverted();
//    m.map(x1, y1, &tx1, &ty1);
//    m.map(x2, y2, &tx2, &ty2);
//    m.map(x3, y3, &tx3, &ty3);
//    m.map(x4, y4, &tx4, &ty4);

    // get the range of new picture
    int maxX = tx1>tx2?(tx1>tx3?(tx1>tx4?tx1:tx4):(tx3>tx4?tx3:tx4)):
(tx2>tx3?(tx2>tx4?tx2:tx4):(tx3>tx4?tx3:tx4));
    int minX = tx1<tx2?(tx1<tx3?(tx1<tx4?tx1:tx4):(tx3<tx4?tx3:tx4)):
(tx2<tx3?(tx2<tx4?tx2:tx4):(tx3<tx4?tx3:tx4));
    int maxY = ty1>ty2?(ty1>ty3?(ty1>ty4?ty1:ty4):(ty3>ty4?ty3:ty4)):
(ty2>ty3?(ty2>ty4?ty2:ty4):(ty3>ty4?ty3:ty4));
    int minY = ty1<ty2?(ty1<ty3?(ty1<ty4?ty1:ty4):(ty3<ty4?ty3:ty4)):
(ty2<ty3?(ty2<ty4?ty2:ty4):(ty3<ty4?ty3:ty4));
    // the size of new picture
    int height = maxY - minY;
    int width = maxX - minX;
    // translation(平移量), the picture rotate taking the origin as the
center
    int deltaX = tx1 - minX;
    int deltaY = ty1 - minY;

    QImage ret(width, height, QImage::Format_RGB32);
    for(int i = -deltaX ; i < ret.width()-deltaX; i++){
        for(int j = -deltaY; j < ret.height()-deltaY; j++){
            // (x, y) is the origenal position of current point
```

```
            double x, y;

            // map: new position (i, j) ==> origenal position(x, y)
            // im.map(i, j, &x, &y);
            Matrix<double>::map(inversedMat, i, j, &x, &y);

            if(x >= 0 && x < img.width() - 1 && y >= 0 && y < img.height() -
1){
                ret.setPixel(i + deltaX, j + deltaY,
ImageProcessing::bilinearInterpolation(x, y, img));
                // ret.setPixel(i + deltaX, j + deltaY,
ImageProcessing::nearestInterpolation(x, y, img));
            }else{
                ret.setPixel(i + deltaX, j + deltaY, qRgb(255, 255, 255));
            }
        }
    }

    return ret;
}
```

这里只实现了最邻近插值和双线性插值，代码如下：

```
/**
 * nearest interpolation
 */
QRgb ImageProcessing::nearestInterpolation(double x, double y, const QImage
& img){

    return img.pixel((int)(x + 0.5), (int)(y + 0.5));
}

/**
 * bilinear interpolation
 */
QRgb ImageProcessing::bilinearInterpolation(double x, double y, const QImage
& img){

    int r = 0, g = 0, b = 0;

    // four integer point near (x, y)
    QRgb rgb11 = img.pixel((int)x, (int)y);
    QRgb rgb12 = img.pixel((int)x+1, (int)y);
    QRgb rgb21 = img.pixel((int)x, (int)y+1);
    QRgb rgb22 = img.pixel((int)x+1, (int)y+1);

    /**
     * bilinear interpolation by direct calculate
     *
```

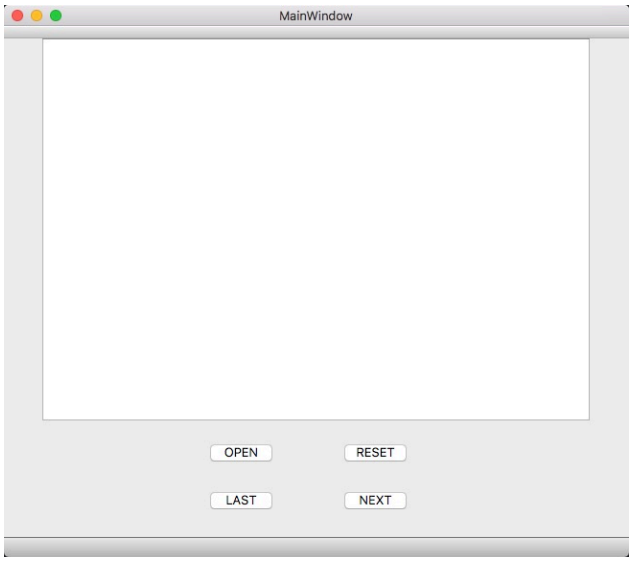
```
     * g(x, y) = ((1 - dx)*f(2, 1) + dx*f(2,2)) * (1 - dy) + ((1 - dx)*f(1,
1) + dx*f(1, 2)) * dy
     * dx = x - (int)x, dy = y - (int)y
     *
     */
    r = ((1 - x + (int)x) * qRed(rgb21) + (x - (int)x) * qRed(rgb22)) * (1 -
y + (int)y) + ((1 - x + (int)x) * qRed(rgb11) + (x - (int)x) * qRed(rgb12))
* (y -(int)y);
    g = ((1 - x + (int)x) * qGreen(rgb21) + (x - (int)x) * qGreen(rgb22)) *
(1 - y + (int)y) + ((1 - x + (int)x) * qGreen(rgb11) + (x - (int)x) *
qGreen(rgb12)) * (y -(int)y);
    b = ((1 - x + (int)x) * qBlue(rgb21) + (x - (int)x) * qBlue(rgb22)) * (1
- y + (int)y) + ((1 - x + (int)x) * qBlue(rgb11) + (x - (int)x) *
qBlue(rgb12)) * (y -(int)y);


    /**
     * bilinear interpolation by matrix map
     *
     * g(x, y) = [1-x  x] 「 f(2, 1)  f(1, 1) 」 「 1 - y 」
     *                    」 f(2, 2)  f(1, 2) 」 」   y   」
     *
     *  this method doesn't work. I dont know why. Maybe there are sth wrong
with my using of matrix.map .
     */
//    Matrix<double> m(3, 3, 0);
//    int res1, res2;
//    m.setMatrix(qRed(rgb21), qRed(rgb11), qRed(rgb22), qRed(rgb12), 0, 0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    r  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

//    m.setMatrix(qGreen(rgb21), qGreen(rgb11), qGreen(rgb22),
qGreen(rgb12), 0, 0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    g  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

//    m.setMatrix(qBlue(rgb21), qBlue(rgb11), qBlue(rgb22), qBlue(rgb12), 0,
0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    b  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

    return QRgb(qRgb(r, g, b));
}
```

双线性插值点矩阵算法中，得到的结果有问题，所以采用直接计算的方法，具体会在**问题和解决方案**中说明。

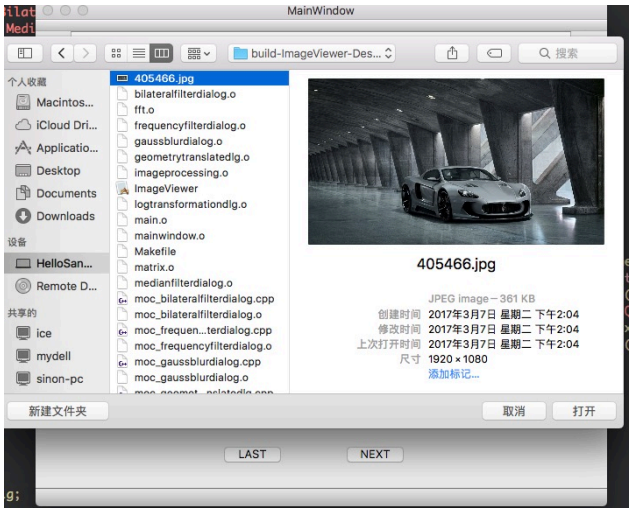
3. 实验结果

3.1基础功能

程序运行窗口如下：

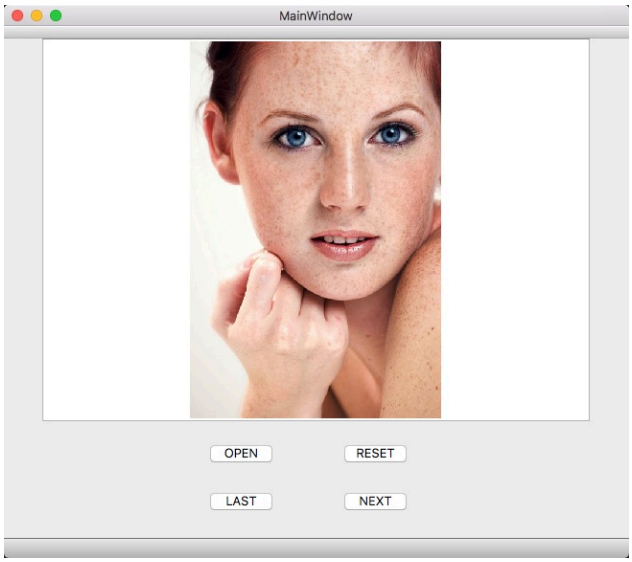


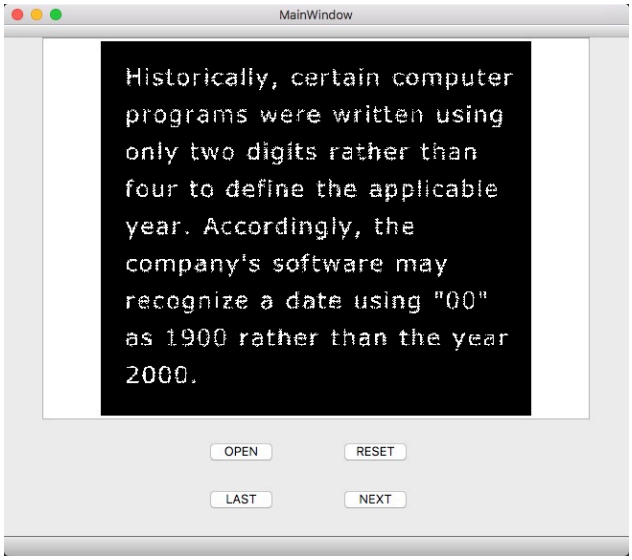
打开图片



上一张&下一张

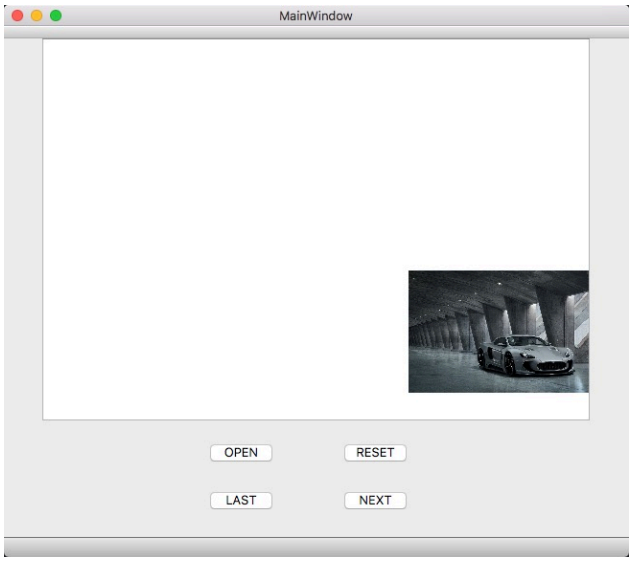
点击`LAST`和`NEXT`按钮的到如下结果：





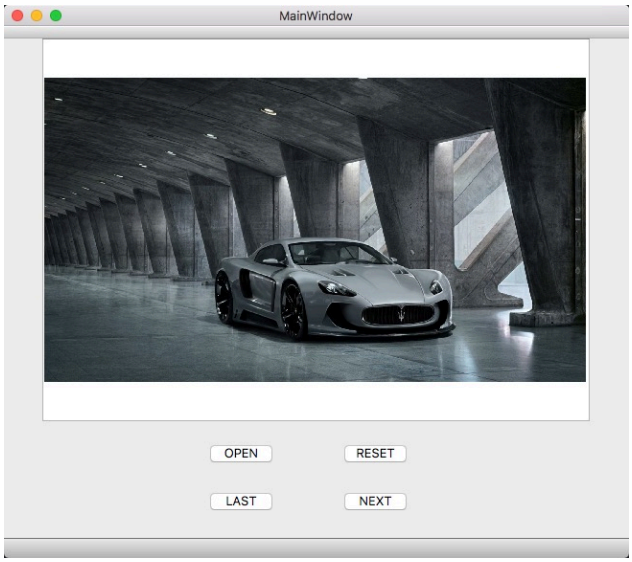
鼠标事件

通过点击拖动和滚轮缩放得到如下结果：



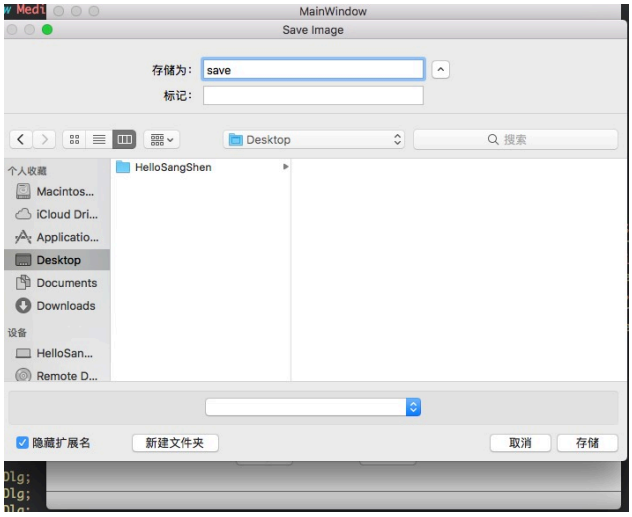
重置图片

点击`RESET`按钮：



保存图片

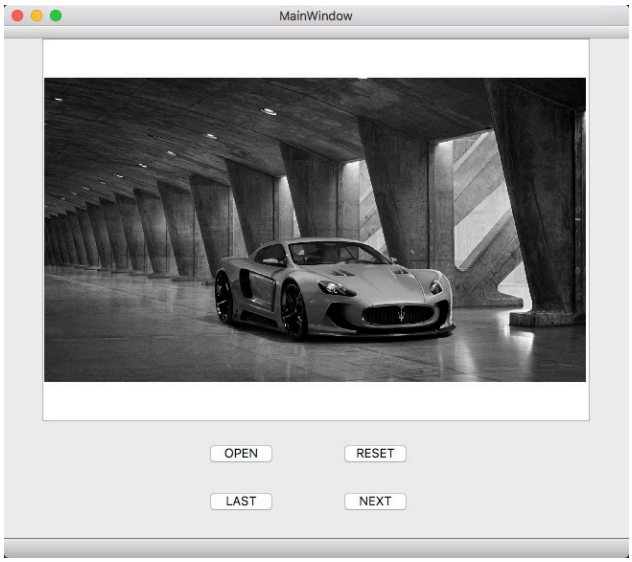
在菜单栏选择`Save`功能，得到保存窗口：



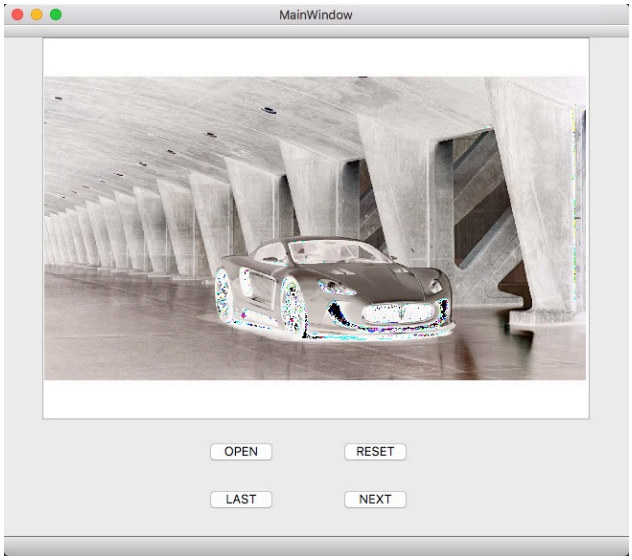
输入文件名，点击储存就可以看到保存下来的图片。

3.2 图像空间变换

RGB转灰度图

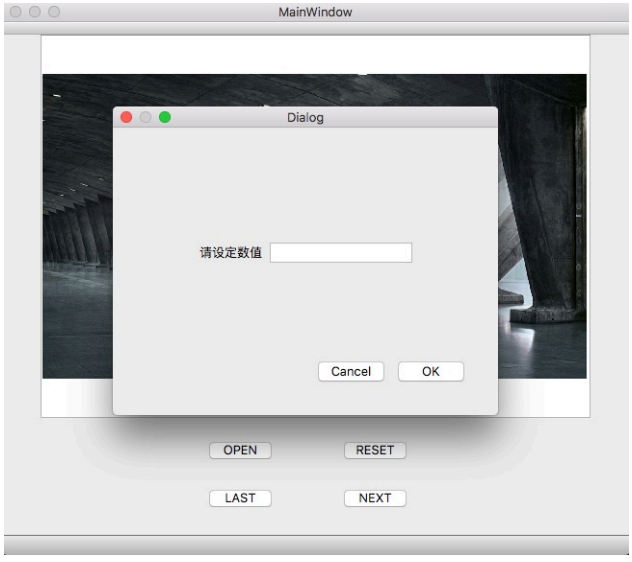


图像反转

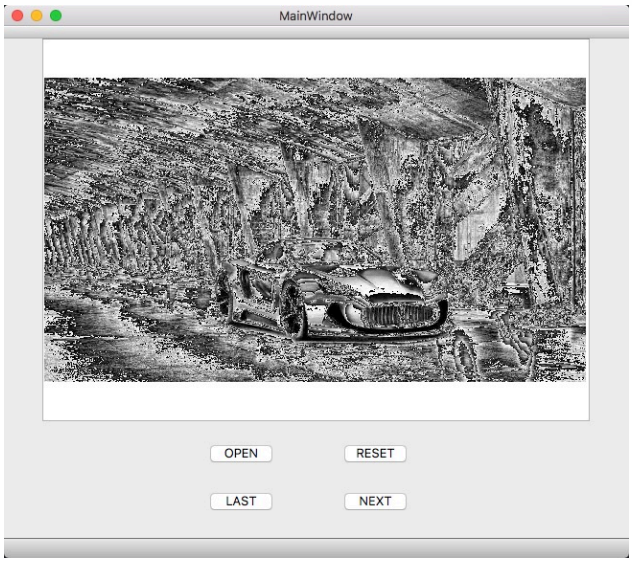


LOG变换

LOG变换输入框口如下：

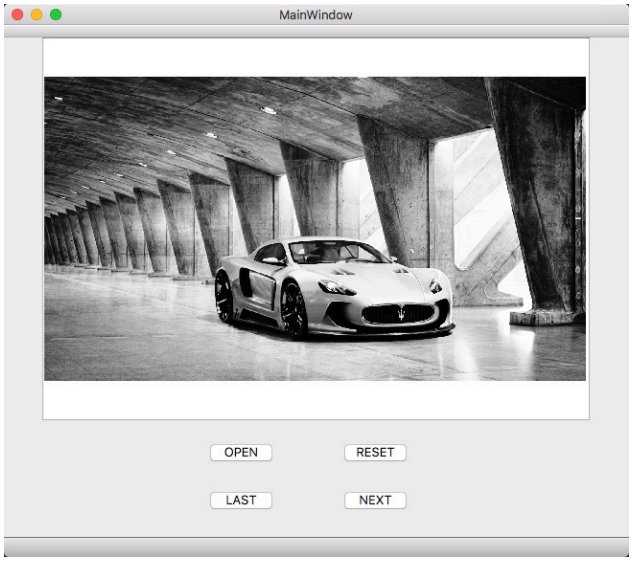


输入数值10，得到结果如下：

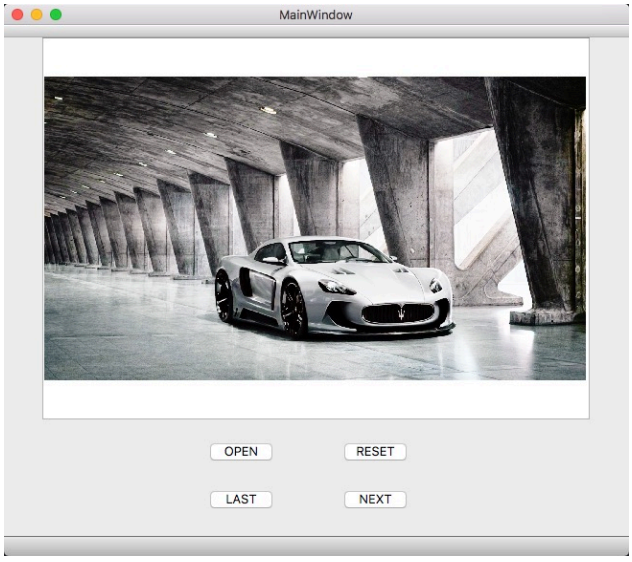


直方图均衡

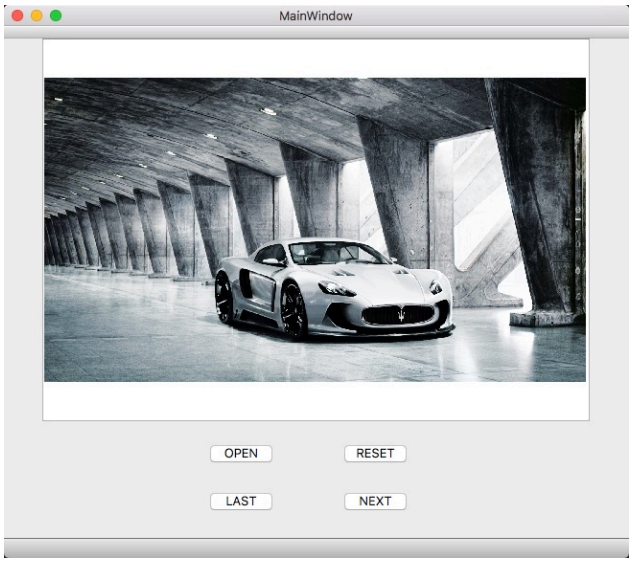
灰度直方图均衡



彩色直方图均衡



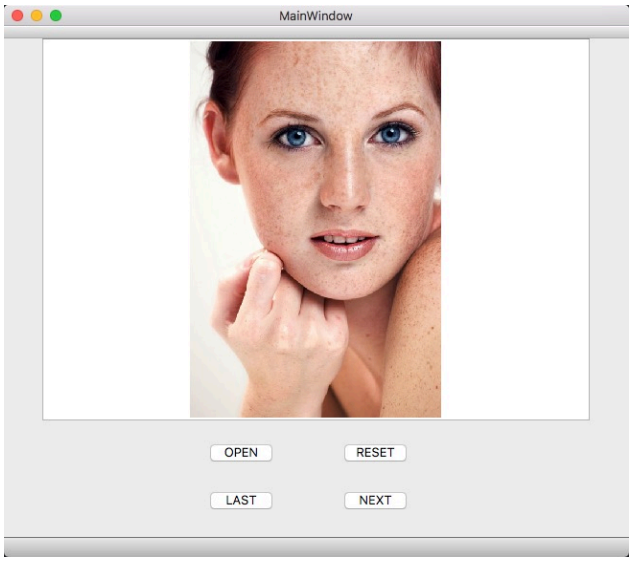
HSI直方图均衡



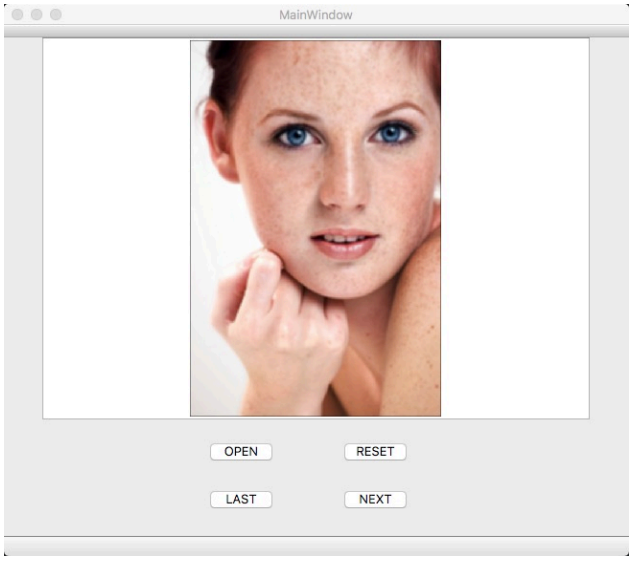
3.2 图像空间变换

线性滤波

首先输入滤波器：

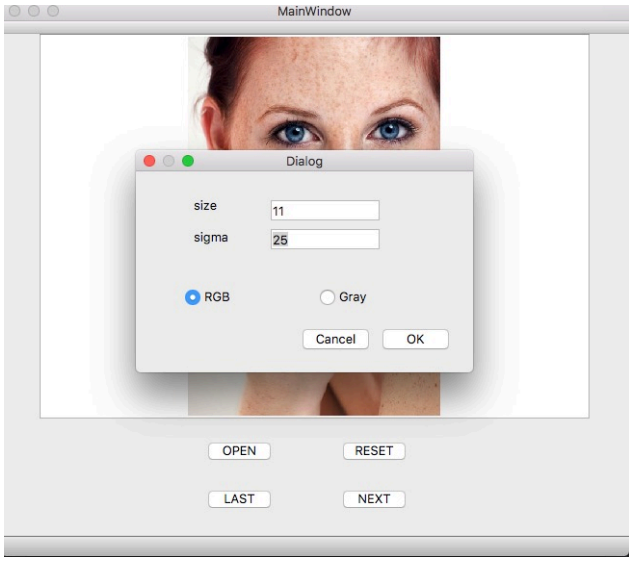


结果如下：

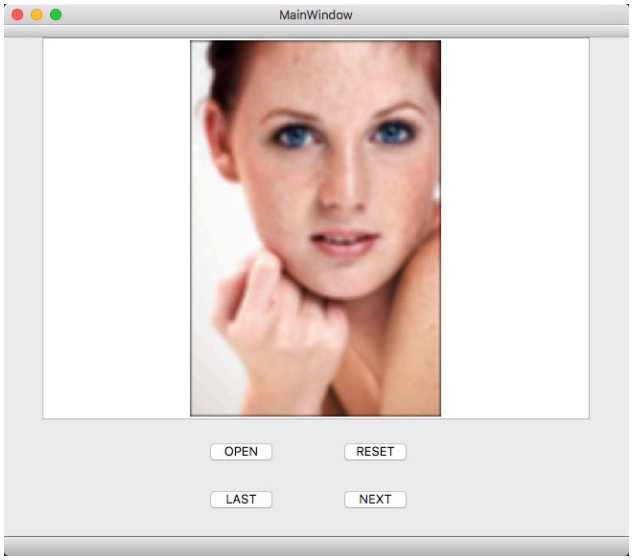


高斯滤波

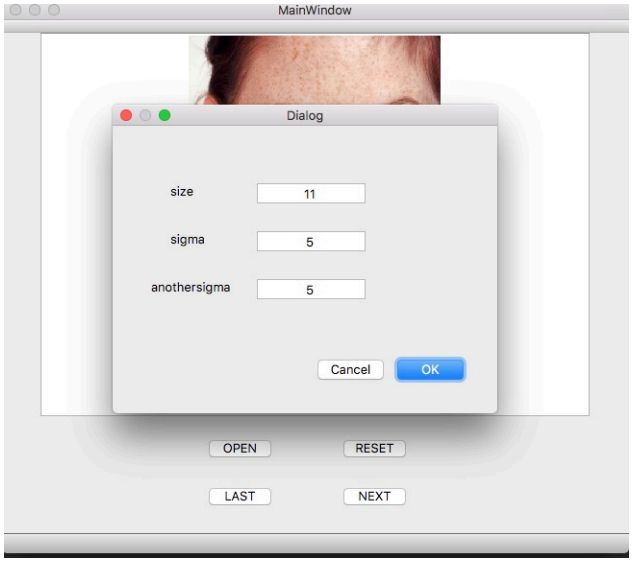
输入框如下：



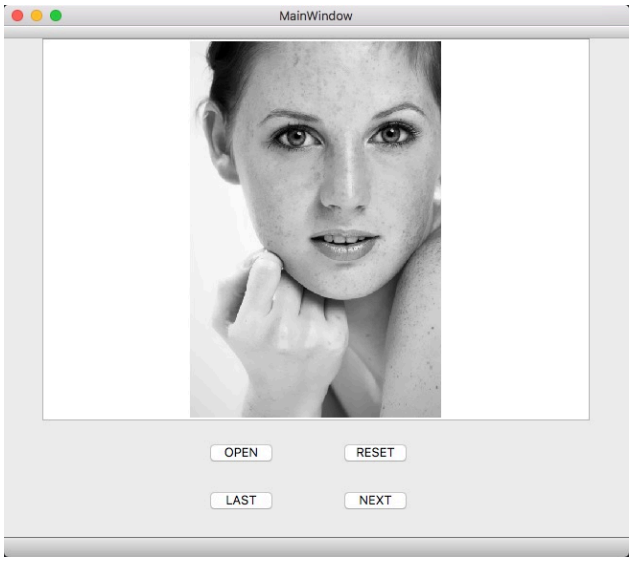
结果如下：



双边滤波



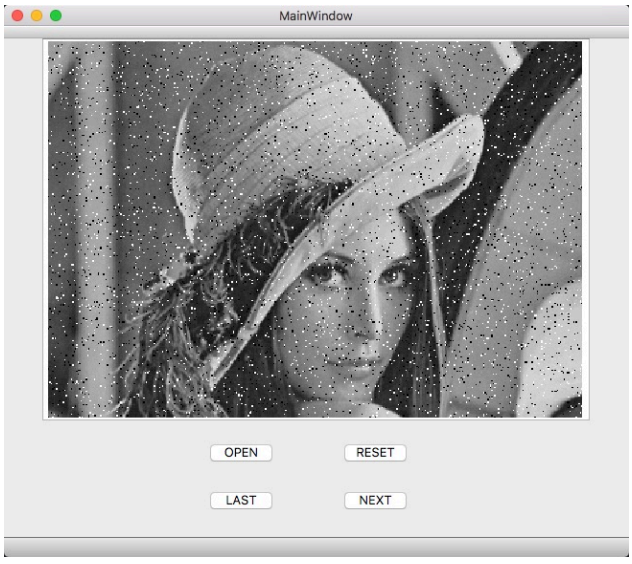
得到结果如下：



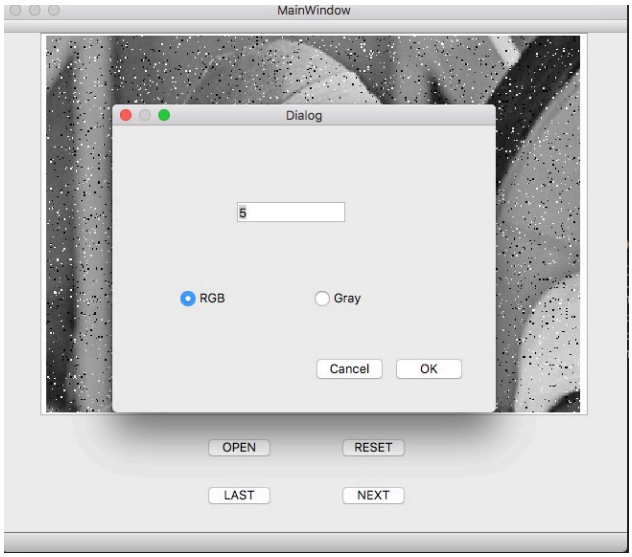
图片中人脸痘痘有轻微改善，调整参数可以得到更好的效果。

中值滤波

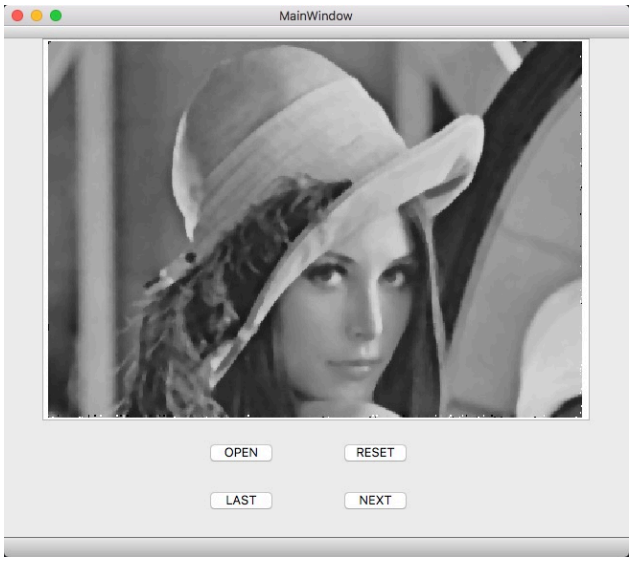
待处理原图：



输入参数：



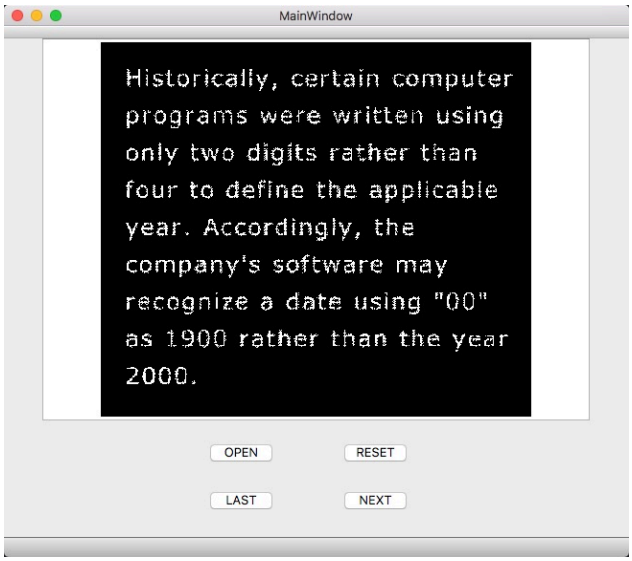
处理结果：



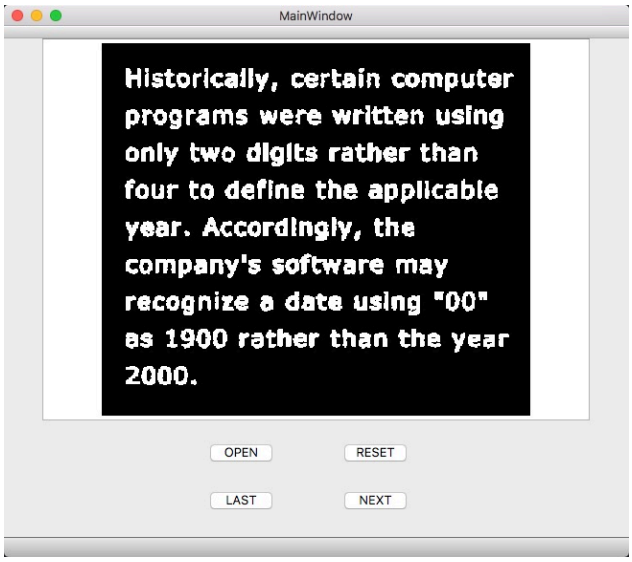
可以看出，噪声已经被较好处理掉了。

膨胀&腐蚀

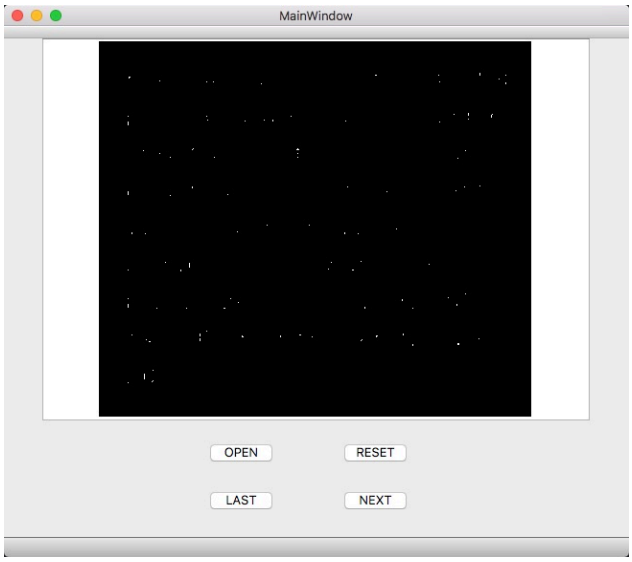
待处理原图：



膨胀操作：



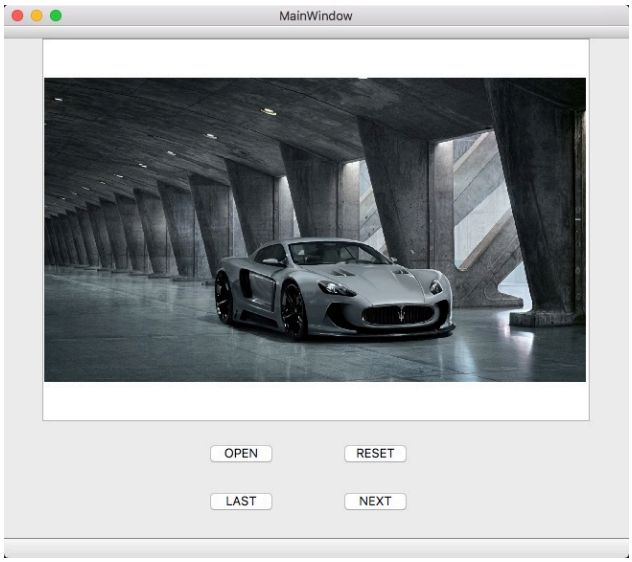
腐蚀操作：



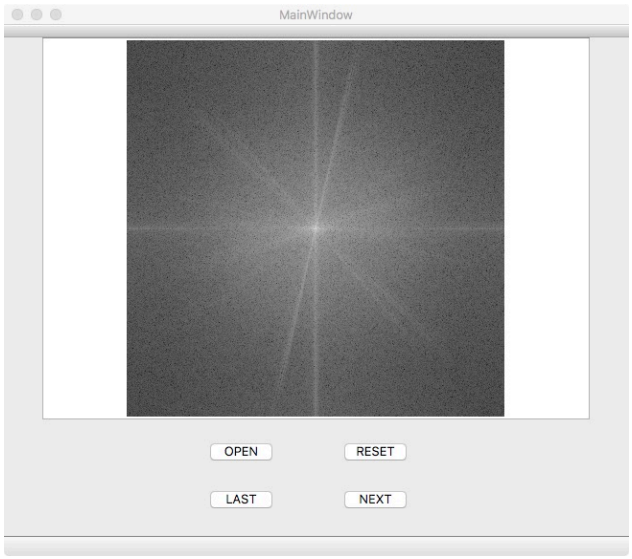
3.3 频域滤波

FFT

原图：

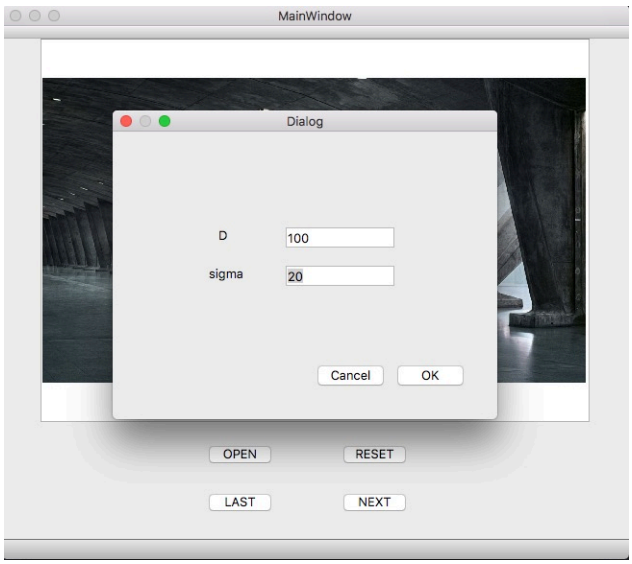


FFT结果：

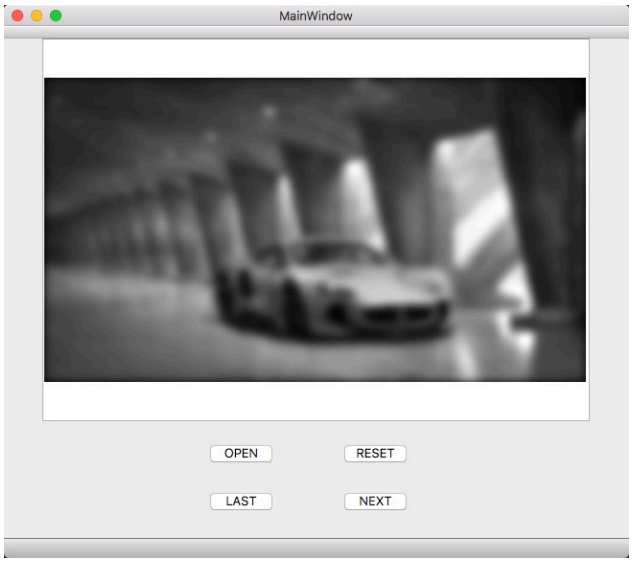


频域高斯低通&高通滤波

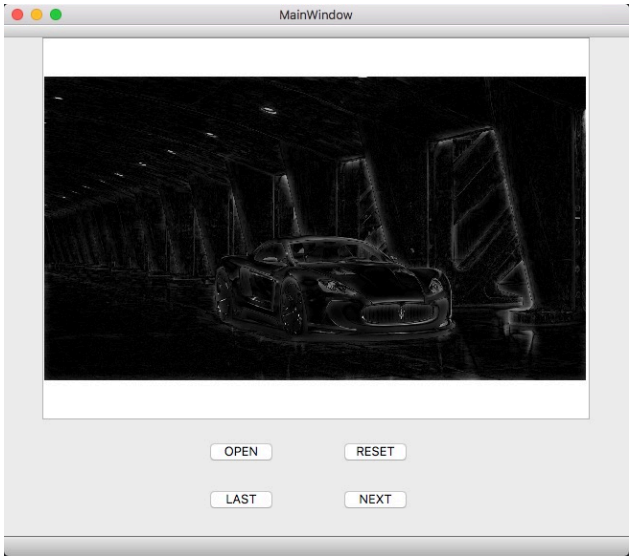
输入低通滤波参数：



低通滤波结果：



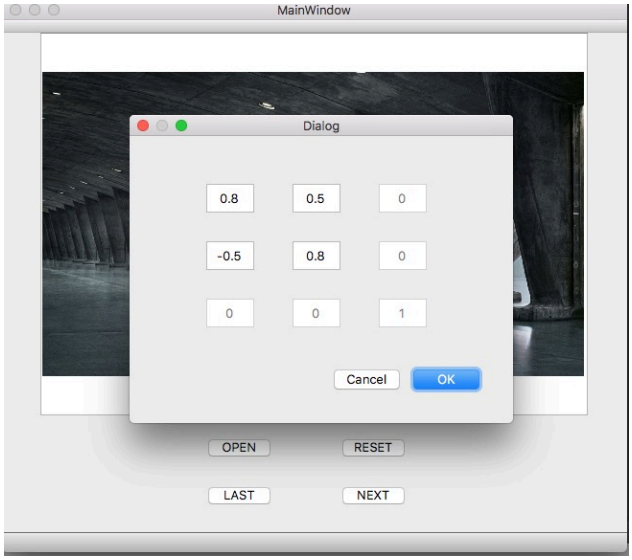
输入同样参数得到高通滤波结果：



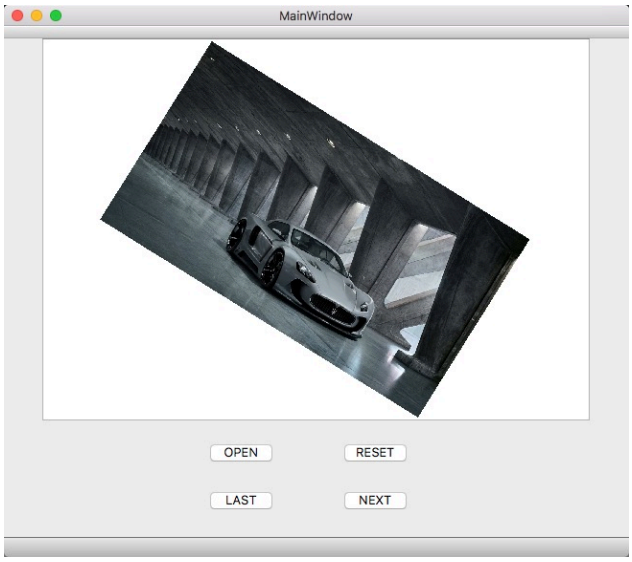
3.4 仿射变换

图像旋转

输入旋转矩阵：



顺时针旋转大约30度结果：



4. 问题和解决方案

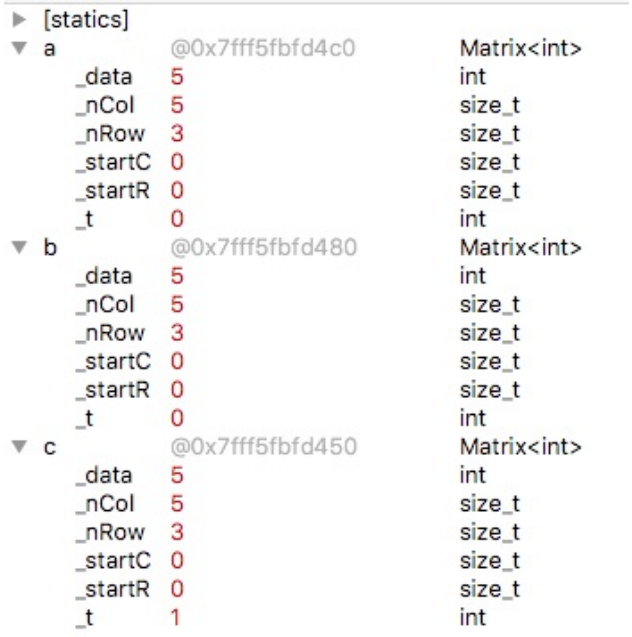
4.1 针对最快的矩阵转置方法的一些测试和思考

看下面一段测试代码，主要就是对于矩阵a，转置后赋值给b，将b再转置后赋值给c：

```
template<typename T>
void Matrix<T>::test4MatrixTranspose()
{
    std::cout << "--------------------initialize matrix a-------------------
-" << std::endl;
    Matrix<int> a(3, 5, 5);
    std::cout << "a._t:    " << a._t << std::endl << a << endl;

    std::cout << "--------------------a.transpose() ==> b-------------------
-" << std::endl;
    Matrix<int> b = a.transpose();
    std::cout << "a._t:    " << a._t << std::endl << a << endl;
    std::cout << "b._t:    " << b._t << std::endl << b << endl;

    std::cout << "--------------------b.transpose() ==> c-------------------
-" << std::endl;
    Matrix<int> c = b.transpose();
    std::cout << "b._t:    " << b._t << std::endl << b << endl;
    std::cout << "c._t:    " << c._t << std::endl << c << endl;
}
```

我们加上断点以后执行结果如下：

```
--------------------initialize matrix a--------------------
a._t:    1
5        5        5        5        5
5        5        5        5        5
5        5        5        5        5

--------------------a.transpose() ==> b--------------------
a._t:    0
5        5        5
5        5        5
5        5        5
5        5        5
5        5        5

b._t:    1
5        5        5        5        5
5        5        5        5        5
5        5        5        5        5

--------------------b.transpose() ==> c--------------------
b._t:    0
5        5        5
5        5        5
5        5        5
5        5        5
5        5        5

c._t:    1
5        5        5        5        5
5        5        5        5        5
5        5        5        5        5
```



很明显有大问题在里面，至少我们可以看到，当`a`转置完了赋值给`b`，`b._t`的值为默认的1，但是矩阵`b`和`a`初始值竟然一样，`b`经过转置以后才勉为其难将`b._t`变为0而且此时跟`a`转置完的结果才一样，至于`c`，重复上次操作，自己感受吧。

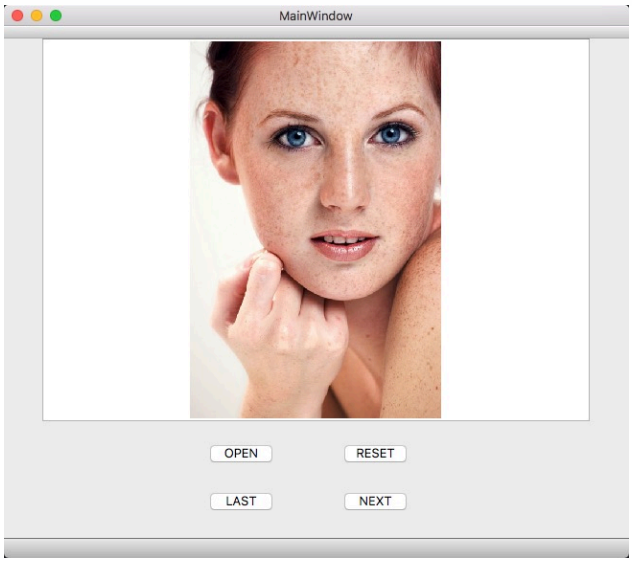
所以之前一直以为转置出现了问题其实不准确，估计是构造函数出了问题。

但是构造函数那么多，我怎么知道该改那个？初步估计可能是那个`swap`函数有问题，**但是怎么改我还没想好**。

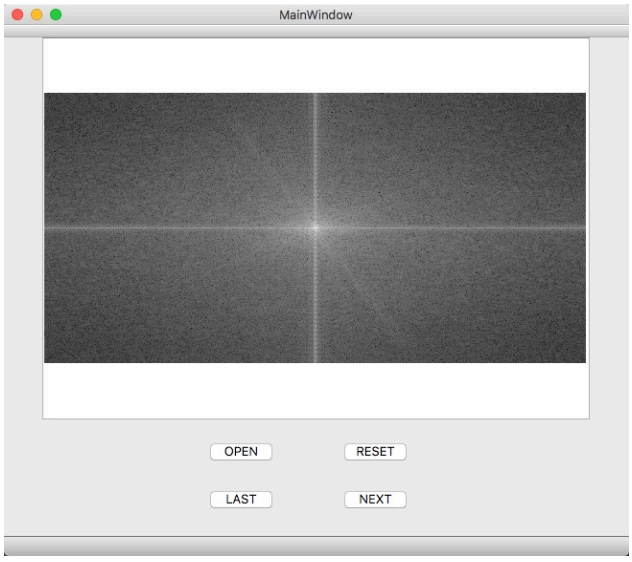
——记录于 *2017/06/08*。

这个转置的问题会导致我们在做FFT的时候出现问题，举例如下：

原图：



FFT变换以后：



问题就出来了，按道理这张图片处理后会是一张512×1024的图片，但是结果是1024×512的图，当然我也尝试过将得到的结果转置以后再输出，可以发现循环移位出现了问题：

总的来说，就是三个标记变量`_t、_startC、_startR`没有达到效果，**要么不能转置回来、要么无法循环移位**。

后来对构造函数进行了长时间的观察，发现一个问题；在执行`=`运算符的时候会调用下面构造函数：

```
template<typename T>
Matrix<T>::Matrix(const Matrix<T>& mat):
_nRow(mat._nRow), _nCol(mat._nCol), _t(mat._t)
{
    // new _data
    // new and assign _data[i] within loop

    // std::cout << "constructor 2" << std::endl;

    _t = 1;
    _startR = 0;
    _startC = 0;

    _data = new T* [_nRow];

    for(int i = 0; i < _nRow; i++){
        _data[i] = new T[_nCol];
    }

    for(int i = 0; i < _nRow ; i++){
        for (int j = 0; j < _nCol; j++) {
            _data[i][j] = mat._data[i][j];
        }
    }
}
```

这里面的`_t = 1;_startR = 0;_startC = 0;`是整个问题的罪魁祸首，以前在写构造函数的时候可能没有注意到函数的调用顺序，但注释且进行修改以后，又带来了频域滤波的**部分元素无法访问**，就不一个一个截图了，反正是非常麻烦，但如果将这留着，频域滤波就完全没有问题，所以这是个很**玄学**的问题。

——记录于 *2017/06/19*。

反正经过无数次修改，目前是没找到更合适的方法，而且我对C++的一些特性可能了解不够深入没办法解决，所以暂时构造函数如下：

```
template<typename T>
Matrix<T>::Matrix(const Matrix<T>& mat):
_nRow(mat._nRow), _nCol(mat._nCol), _t(mat._t), _startC(mat._startC),
_startR(mat._startR)
{
    // new _data
    // new and assign _data[i] within loop

    // std::cout << "constructor 2" << std::endl;

    _t = 1;

    _data = new T* [_nRow];

    for(int i = 0; i < _nRow; i++){
        _data[i] = new T[_nCol];
    }

    for(int i = 0; i < _nRow ; i++){
        for (int j = 0; j < _nCol; j++) {
            _data[i][j] = mat._data[i][j];
        }
    }
}
```

这段代码能保证程序目前可以运行且不会在运算的时候崩溃，频域滤波也没有任何问题，只是FFT不太好看而已，但是由于频域结果大多都是中心对称的，所以也不影响分析，暂时先这样，以后有新的办法再修改，会随时跟新GitHub。

——记录于 *2017/06/20*。

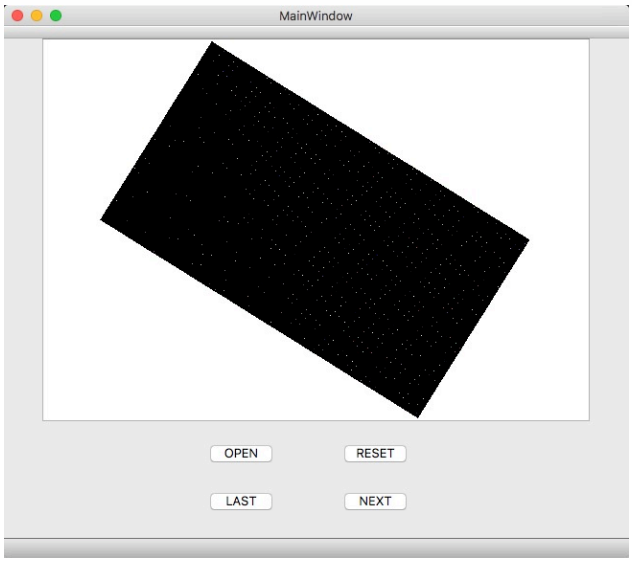
4.2 仿射变换双线性插值的矩阵计算

采用下面算法进行双线性插值：

```
/**
 * bilinear interpolation
 */
QRgb ImageProcessing::bilinearInterpolation(double x, double y, const QImage
& img){

    int r = 0, g = 0, b = 0;

    // four integer point near (x, y)
    QRgb rgb11 = img.pixel((int)x, (int)y);
    QRgb rgb12 = img.pixel((int)x+1, (int)y);
    QRgb rgb21 = img.pixel((int)x, (int)y+1);
    QRgb rgb22 = img.pixel((int)x+1, (int)y+1);
```

```
    /**
     * bilinear interpolation by direct calculate
     *
     * g(x, y) = ((1 - dx)*f(2, 1) + dx*f(2,2)) * (1 - dy) + ((1 - dx)*f(1,
1) + dx*f(1, 2)) * dy
     * dx = x - (int)x, dy = y - (int)y
     *
     */
//    r = ((1 - x + (int)x) * qRed(rgb21) + (x - (int)x) * qRed(rgb22)) * (1
- y + (int)y) + ((1 - x + (int)x) * qRed(rgb11) + (x - (int)x) *
qRed(rgb12)) * (y -(int)y);
//    g = ((1 - x + (int)x) * qGreen(rgb21) + (x - (int)x) * qGreen(rgb22))
* (1 - y + (int)y) + ((1 - x + (int)x) * qGreen(rgb11) + (x - (int)x) *
qGreen(rgb12)) * (y -(int)y);
//    b = ((1 - x + (int)x) * qBlue(rgb21) + (x - (int)x) * qBlue(rgb22)) *
(1 - y + (int)y) + ((1 - x + (int)x) * qBlue(rgb11) + (x - (int)x) *
qBlue(rgb12)) * (y -(int)y);


    /**
     * bilinear interpolation by matrix map
     *
     * g(x, y) = [1-x  x] 「 f(2, 1)  f(1, 1) 」 「 1 - y 」
     *                    」 f(2, 2)  f(1, 2) 」 」   y   」
     *
     *  this method doesn't work. I dont know why. Maybe there are sth wrong
with my using of matrix.map .
     */
    Matrix<double> m(3, 3, 0);
    int res1, res2;
    m.setMatrix(qRed(rgb21), qRed(rgb11), qRed(rgb22), qRed(rgb12), 0, 0);
    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
    r  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

    m.setMatrix(qGreen(rgb21), qGreen(rgb11), qGreen(rgb22), qGreen(rgb12),
0, 0);
    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
    g  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

    m.setMatrix(qBlue(rgb21), qBlue(rgb11), qBlue(rgb22), qBlue(rgb12), 0,
0);
    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
    b  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

    return QRgb(qRgb(r, g, b));
}
```

矩阵计算得到的结果：



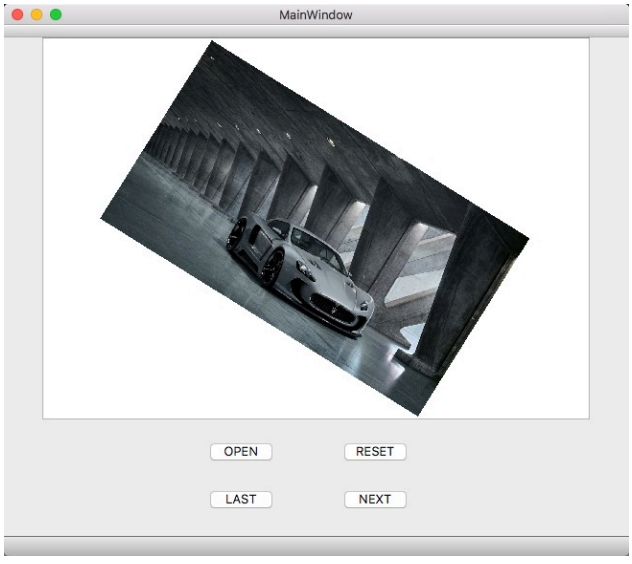
旋转没问题，但是图片已经不是原来的图片了。

直接公式计算：

```
/**
 * bilinear interpolation
 */
QRgb ImageProcessing::bilinearInterpolation(double x, double y, const QImage
& img){

    int r = 0, g = 0, b = 0;

    // four integer point near (x, y)
    QRgb rgb11 = img.pixel((int)x, (int)y);
    QRgb rgb12 = img.pixel((int)x+1, (int)y);
    QRgb rgb21 = img.pixel((int)x, (int)y+1);
    QRgb rgb22 = img.pixel((int)x+1, (int)y+1);

    /**
     * bilinear interpolation by direct calculate
     *
     * g(x, y) = ((1 - dx)*f(2, 1) + dx*f(2,2)) * (1 - dy) + ((1 - dx)*f(1,
1) + dx*f(1, 2)) * dy
     * dx = x - (int)x, dy = y - (int)y
     *
     */
    r = ((1 - x + (int)x) * qRed(rgb21) + (x - (int)x) * qRed(rgb22)) * (1 -
y + (int)y) + ((1 - x + (int)x) * qRed(rgb11) + (x - (int)x) * qRed(rgb12))
* (y -(int)y);
    g = ((1 - x + (int)x) * qGreen(rgb21) + (x - (int)x) * qGreen(rgb22)) *
(1 - y + (int)y) + ((1 - x + (int)x) * qGreen(rgb11) + (x - (int)x) *
qGreen(rgb12)) * (y -(int)y);
    b = ((1 - x + (int)x) * qBlue(rgb21) + (x - (int)x) * qBlue(rgb22)) * (1
- y + (int)y) + ((1 - x + (int)x) * qBlue(rgb11) + (x - (int)x) *
qBlue(rgb12)) * (y -(int)y);
```

```
    /**
     * bilinear interpolation by matrix map
     *
     * g(x, y) = [1-x  x] 「 f(2, 1)  f(1, 1) | 「 1 - y |
     *                    | f(2, 2)  f(1, 2) 」 |   y   」
     *
     *  this method doesn't work. I dont know why. Maybe there are sth wrong
with my using of matrix.map .
     */
//    Matrix<double> m(3, 3, 0);
//    int res1, res2;
//    m.setMatrix(qRed(rgb21), qRed(rgb11), qRed(rgb22), qRed(rgb12), 0, 0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    r  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

//    m.setMatrix(qGreen(rgb21), qGreen(rgb11), qGreen(rgb22),
qGreen(rgb12), 0, 0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    g  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

//    m.setMatrix(qBlue(rgb21), qBlue(rgb11), qBlue(rgb22), qBlue(rgb12), 0,
0);
//    Matrix<int>::map(m, (int)x + 1 - x, x - (int)x, &res1, &res2);
//    b  = res1 * ((int)y + 1 - y) + res2 * (y - (int)x);

    return QRgb(qRgb(r, g, b));
}
```

得到结果：



不仅计算没有问题，速度也比之前矩阵计算稍快，所以采用后者。

5. 写在最后

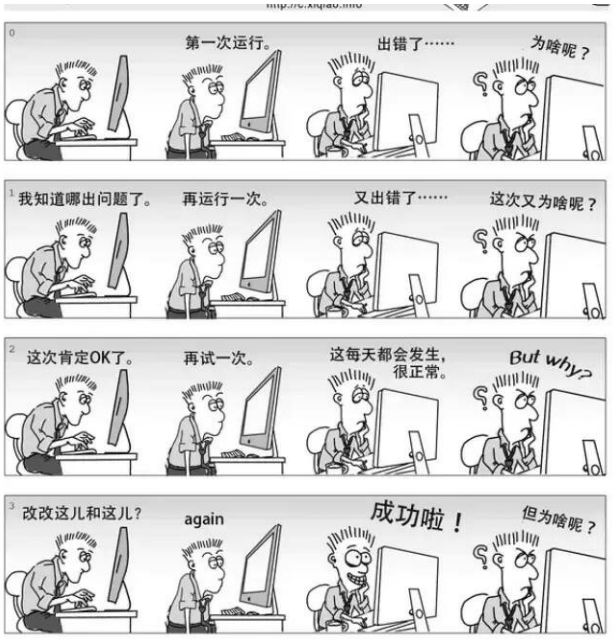
这个项目前前后后写了接近一个学期，就像堆积木，一点一点堆积，但是每堆积一些，压力就更大一些也更害怕会倒，因为没法保证之前的实现一定没问题，尤其是引入Matrix和fft以后，每写一步都要测试好多遍生怕出问题，但是还是没有避免掉前面说的快速转置的问题，这就是为了性能带来的一连串问题，甚至有些问题都非常玄学，其实本质问题出现在构造函数里面，而以前根本没考虑过，到现在时间过去太久，很多细节都记得不是非常清楚，花了很多时间调试也不完美，挺不爽的。

其实还有很多已经实现了的算法没有用上，比如花了非常多时间写出来的重复填充和镜像填充，用上的话效果会更好，但是考虑到时间和精力的问题，也没给用户选择填充的操作，以后再说吧，同时界面也没有花时间优化，因为这个毕竟是无底洞。

这个学期的这个项目让我意识到一个问题：程序永远不会完美，永远会可能不满足需求，甚至在后来会发现很多前期做出来的不能满足新的需求，这时候就让人非常纠结，如果不重构就显得非常不完美，强迫症是非常难受的，所以在后来我还简单重新修改了一遍。

这个小项目加起来大概有3500行（包括注释和空行），报告也都快70页了。程序永远不可能完美，报告也是，尽管花了很多时间让他们变得更符合自己的要求，但是一定会有不足的地方，见谅。

关于玄学的问题，配个图更能表达我的感受：



项目GitHub地址：https://github.com/HelloSangShen/ImageViewer

等到作业验收完成、学期结束以后，会把这个最终版本上传到GitHub（暂时就不传了），以后有新思路会不定期更新，毕竟这个也是可以写入自己简历的小项目，所以还是比较感谢这个小项目，首先是可以作为一个项目经历，另一方面也确实从本质上提升了自己对C++/Qt/DIP/DSP的理解，据一个最简单的例子，以前只是记得FFT的公式和算法，可以计算，但是涉及到8位以上就非常难算了，这次实现了FFT算法，也对这个算法有了非常深入的理解，知道了从递归的角度去解决这个计算问题。

——报告最后修改于 *2017/06/20*。