```
/*****************/
/* main_process  */
/*****************/

// gRed, gGreen, gBlueの2次元配列にクエリ画像のRGB情報が格納されています
// 同じサイズのgBuff_red, gBuff_green, gBuff_blueに結果画像情報を格納してリターンします

#define AREA_INFO_MAX 1000    // area_info_counterの最大値
#define AREA_INFO_CONTENTS 13  // area_info[][]の項目数

// エリアトレース時の有効リアの条件を指定します
// 縦横双方とも20ピクセル以上のエリアを有効（異物）とします
#define OBJECT_WH_MIN 20
int condition_func_object(int top, int bottom, int left, int right, int trace_counter,
int pxl_counter) {
  if (OBJECT_WH_MIN <= bottom - top + 1 && OBJECT_WH_MIN <= right - left + 1)  return
1;
  else return 0;
}

// X線画像の異物を認識します
int main_process(unsigned char** gRed, unsigned char** gGreen, unsigned char** gBlue,
  unsigned char** gBuff_red, unsigned char** gBuff_green, unsigned char** gBuff_blue,
  int width_pixel, int height_pixel) {

  // gGray[][]メモリ確保
  int i;
  unsigned char** gGray = (unsigned char**)malloc(sizeof(unsigned char*) *
height_pixel);
  if (gGray == NULL)  return -99;
  unsigned char* gGray_p = (unsigned char*)malloc(sizeof(unsigned char) * height_pixel
* width_pixel);
  if (gGray_p == NULL) {
    free(gGray);
    return -99;
  }
  for (i = 0; i < height_pixel; i++) {
    gGray[i] = gGray_p + (long long)i * (long long)width_pixel;
  }

  // グレースケールを生成します
  int j;
  for (i = 0; i < height_pixel; i++) {
    for (j = 0; j < width_pixel; j++) {
      gGray[i][j] = (gRed[i][j] + gGreen[i][j] + gBlue[i][j]) / 3;
    }
  }

  // area_info[AREA_INFO_MAX][AREA_INFO_CONTENTS]メモリを確保します
  int n;
  int** area_info;
  int* area_info_p;
  area_info = (int**)malloc(sizeof(int*) * AREA_INFO_MAX);
  if (area_info == NULL) {
    return -99;
  }
  area_info_p = (int*)malloc(sizeof(int) * AREA_INFO_MAX * AREA_INFO_CONTENTS);
  if (area_info_p == NULL) {
    free(area_info);
    return -99;
  }
  for (n = 0; n < AREA_INFO_MAX; n++) {
    area_info[n] = area_info_p + (long long)n * (long long)AREA_INFO_CONTENTS;
```

```
  }


  // 構造体にパラメータを設定します
  scratch_t scratch;
  scratch.mode = 11;      // 縦横双方を処理（1の位＝1：横方向（縦傷）　10の位＝1の場合：縦方向（横傷）を実行）
  scratch.target_brt_min = 0;     // 傷部分明るさ最小値
  scratch.target_brt_max = 120;     // 傷部分明るさ最大値
  scratch.side_brt_min = 120;     // 傷隣接部分明るさ最小値
  scratch.side_brt_max = 255;     // 傷隣接部分明るさ最大値
  scratch.distance = 10;       // 傷中心から隣接部分までの距離（ピクセル）
  scratch.offset_c = 1;       // 傷部分明るさを平均する際の左右（上下）pixel数（=1の場合左右１pixel計3pixelの平均）
  scratch.offset_s = 4;       // 隣接部分明るさを平均する際の左右（上下）pixel数（=2の場合左右2pixel計5pixelの平均）
  scratch.brt_diff_sh = -40;     // 傷部分明るさ平均と隣接部分明るさ平均の差最小値（傷ー隣接の値：黒い傷の場合は負の値）
  scratch.side_diff_sh = 60;     // 両サイドの隣接部分平均の差最大値

  // 黒いラインを検出します
  ABHB_scratch_mapping(gGray, gBuff_blue, width_pixel, height_pixel, 255, scratch); //①

  // gBuff_blue[][]=255のピクセル間の距離が5ピクセル以下のピクセルの間隙を255で埋めます
  ABHB_joint_near_pixel(11, gBuff_blue, width_pixel, height_pixel, 255, 5); //②

  // トレースします
  int area_info_counter = 0;
  ABHB_area_trace(AREA_INFO_MAX, &area_info_counter, area_info, width_pixel, height_pixel,
    gBuff_red, gBuff_green, gBuff_blue, condition_func_object);

  // カレントディレクトリにtrace.bmpをダンプします
  char dump_file_name[256];
  sprintf(dump_file_name, "trace.bmp");
  ABHB_bmp_read_write(1, dump_file_name, gBuff_red, gBuff_green, gBuff_blue, &width_pixel, &height_pixel);

  // クエリ画像情報をgBuff_xxxにコピーします
  for (i = 0; i < height_pixel; i++) {
    for (j = 0; j < width_pixel; j++) {
      gBuff_red[i][j] = gRed[i][j];
      gBuff_green[i][j] = gGreen[i][j];
      gBuff_blue[i][j] = gBlue[i][j];
    }
  }

  // 結果画像としてクエリ画像に異物を囲む赤の矩形を合成します
  int offset = 20;
  int bold = 4;
  int k;
  for (n = 0; n < area_info_counter; n++) {
    for (i = area_info[n][4] - offset; i <= area_info[n][5] + offset; i++) {
      j = area_info[n][6] - offset;
      for (k = 0; k < bold; k++) {
        if (0 <= i && i < height_pixel && 0 <= j + k && j + k < width_pixel) {
          gBuff_red[i][j + k] = 255;
          gBuff_green[i][j + k] = 0;
          gBuff_blue[i][j + k] = 0;
        }
      }
      j = area_info[n][7] + offset;
```

```
      for (k = 0; k < bold; k++) {
        if (0 <= i && i < height_pixel && 0 <= j - k && j - k < width_pixel) {
          gBuff_red[i][j - k] = 255;
          gBuff_green[i][j - k] = 0;
          gBuff_blue[i][j - k] = 0;
        }
      }
    }
    for (j = area_info[n][6] - offset; j <= area_info[n][7] + offset; j++) {
      i = area_info[n][4] - offset;
      for (k = 0; k < bold; k++) {
        if (0 <= i + k && i + k < height_pixel && 0 <= j && j < width_pixel) {
          gBuff_red[i + k][j] = 255;
          gBuff_green[i + k][j] = 0;
          gBuff_blue[i + k][j] = 0;
        }
      }
      i = area_info[n][5] + offset;
      for (k = 0; k < bold; k++) {
        if (0 <= i - k && i - k < height_pixel && 0 <= j && j < width_pixel) {
          gBuff_red[i - k][j] = 255;
          gBuff_green[i - k][j] = 0;
          gBuff_blue[i - k][j] = 0;
        }
      }
    }
  }


  free(area_info_p);
  free(area_info);
  free(gGray_p);
  free(gGray);

  return 0;
}
```

****関数***
ABHB_scratch_mapping()

引数１：unsigned char** gGray　クエリ画像のグレースケール値が格納されている配列のポインタ
引数２：unsigned char** gBuff_blue　　結果をマッピングする配列のポインタ
引数３：int width_pixel　　横方向ピクセル数
引数４：int height_pixel　　縦方向ピクセル数
引数５：int value　　結果をマッピングする値
引数６：scratch_t scratch　　パラメータscratch_mapping構造体

scratch_mapping 構造体定義
typedef struct {
int mode　　：1の位＝1の場合、横方向微分（縦傷）を実行　10の位＝1の場合縦方向微分（横傷）を実行
int target_brt_min　　：傷部分ピクセル明るさ最小値
int target_brt_max　　：傷部分ピクセル明るさ最大値
int side_brt_min　　：傷隣接部分明るさ最小値
int side_brt_max　　：傷隣接部分明るさ最大値
int distance　　：傷中心から隣接部分までの距離（ピクセル）
int offset_c　：傷部分明るさを平均する際の左右（上下）pixel数（=1の場合左右１pixel計3pixelの平均）
int offset_s　　：隣接部分明るさを平均する際の左右（上下）pixel数（=2の場合左右2pixel計5pixelの平均）
int brt_diff_sh：傷部分明るさ平均と隣接部分明るさ平均の差最小値（傷ー隣接の値：黒い傷の場合は負の値）

int side_diff_sh　　: 両サイドの隣接部分平均の差最大値
}scratch_t;

ABHB_joint_near_pixel()

引数１：int direction　　1の位が１：横方向　10の位が１：縦方向
　　100の位が１：右下がり斜め　1000の位が１：右上がり斜め
引数２：unsigned char** gBuff_blue　　結合する配列のポインタ
引数３：int width_pixel　　横方向ピクセル数
引数４：int height_pixel　　縦方向ピクセル数
引数５：int value　　結合する配列の値
引数６：int distance　　結合するピクセル間距離最大値