【int ABHB_bmp_read_write()】

```
// ファイル名：char* file_nameを読み込み、画像情報をgRed[][], gGreen[][], gBlue[][]に格納して終了します
// 読み込みファイルをオープンします
fp = fopen(file_name, "rb");
// image_bufferのメモリ確保
Int byte_size = (*width_pixel + 3) * *height_pixel * 3 + 54;          // 横ピクセル数が4で割り切れない時の最大バイト数とします
Unsigned char* image_buffer = (unsigned char*)malloc(byte_size);
// 画像情報をimage_bufferに読み込みます
fread(image_buffer, sizeof(unsigned char), byte_size, fp);
fclose(fp);
// BITMAP 認識文字 "BM" をチェックします
// 画像データまでのoffsetを読み込みます
// ヘッダサイズを確認します
// widthを読み込みます
// heightを読み込みます
// 1画素当たりのビット数を読み込みます
Int color_bit = image_buffer[28] + image_buffer[29] * 256;
if (color_bit == 24) {
        // ピクセルごとのRGB値を読み込みます
        for (i = 0; i < *height_pixel; i++) {
                pointer = offset + line * (*height_pixel - (i + 1));
                for (j = 0; j < *width_pixel; j++) {
                        gBlue[i][j] = image_buffer[pointer++];
                        gGreen[i][j] = image_buffer[pointer++];
                        gRed[i][j] = image_buffer[pointer++];
                }
        }
}
if (color_bit == 8) {
        // ピクセルごとの値を読み込みます
        for (i = 0; i < *height_pixel; i++) {
                pointer = offset + line * (*height_pixel - (i + 1));
                for (j = 0; j < *width_pixel; j++) {
                        gBlue[i][j] = image_buffer[pointer];
```

```
                        gGreen[i][j] = image_buffer[pointer];
                        gRed[i][j] = image_buffer[pointer++];
                }
        }
}

if (color_bit == 32) {
        // ピクセルごとの値を読み込みます
        for (i = 0; i < *height_pixel; i++) {
                pointer = offset + line * (*height_pixel - (i + 1));
                for (j = 0; j < *width_pixel; j++) {
                        gBlue[i][j] = image_buffer[pointer++];
                        gGreen[i][j] = image_buffer[pointer++];
                        gRed[i][j] = image_buffer[pointer++];
                        pointer++;
                }
        }
}
// メモリ開放
free(image_buffer);
```

【void ABHB_noise_reduction()】

// gRed[][], gGreen[][]], gBlue[][]の画像情報にノイズリダクションを実行、
// 結果を gBuff_red[][], gBuff_green[][], gBuff_blue[][]に格納して終了します

```
offset[0][0] = -1;          offset[0][1] = -1;          // 左上
offset[1][0] = -1;          offset[1][1] = 0;           // 上
offset[2][0] = -1;          offset[2][1] = 1;           // 右上
offset[3][0] = 0;           offset[3][1] = -1;          // 左
offset[4][0] = 0;           offset[4][1] = 1;           // 右
offset[5][0] = 1;           offset[5][1] = -1;          // 左下
offset[6][0] = 1;           offset[6][1] = 0;           // 下
offset[7][0] = 1;           offset[7][1] = 1;           // 右下
for (i = 1; i < height_pixel - 1; i++) {
        for (j = 1; j < width_pixel - 1; j++) {
                value = 0;
                for (k = 0; k < 8; k++) {
                        value = value + gRed[i + offset[k][0]][j + offset[k][1]];
                }
                gBuff_red[i][j] = (value + gRed[i][j] * 2) / 10;
                value = 0;
                for (k = 0; k < 8; k++) {
                        value = value + gGreen[i + offset[k][0]][j + offset[k][1]];
                }
                gBuff_green[i][j] = (value + gGreen[i][j] * 2) / 10;
                value = 0;
                for (k = 0; k < 8; k++) {
                        value = value + gBlue[i + offset[k][0]][j + offset[k][1]];
                }
                gBuff_blue[i][j] = (value + gBlue[i][j] * 2) / 10;
        }
}
```

【int ABHB_auto_brightness_adjustment()】

```
// 元画像情報を gRed[][], gGreen[][], gBlue[][]として、
// 引数の構造体のパラメータに従って明るさコントラストを最適化して、
// 結果を gBuff_red[][], gBuff_green[][], gBuff_blue[][]に格納して終了します
// グレースケール化した画像をgGray[][]に格納します
// percent指定がマイナスの場合はグレースケール処理のみでリターンします
// パラメータ値の不正をチェックします
// 指定範囲の明るさのヒストグラムを生成します
pxl_counter = 0;
for (i = 0; i < height_pixel; i++) {
        for (j = 0; j < width_pixel; j++) {
                if (abadj_p->range_low <= gGray[i][j] && gGray[i][j] <= abadj_p-
>range_high) {
                        brt_hist[gGray[i][j]]++;
                        pxl_counter++;
                }
        }
}
// 明るい方からhigh_percent%のピクセル数に達した明るさをbrt_max, 暗い方からlow_percent%
のピクセル数に達した明るさをbrt_minに格納、
// それらの平均値をbrt_midとします
counter = 0;
brt_max = 0;
for (n = 255; n > 0; n--) {
        counter = counter + brt_hist[n];
        if (counter > pxl_counter * abadj_p->percent_high / 100) {
                brt_max = n;
                n = 0;
        }
}
counter = 0;
brt_min = 255;
for (n = 0; n < 256; n++) {
        counter = counter + brt_hist[n];
        if (counter > pxl_counter * abadj_p->percent_low / 100) {
                brt_min = n;
```

```
                n = 256;
        }
}
brt_mid = (brt_min + brt_max) / 2;
target_mid = (abadj_p->target_high + abadj_p->target_low) / 2;
// brt_minの明るさをtarget_lowに、brt_maxの明るさをtarget_highになるように引き伸ばします
contrast_ratio = ((double)abadj_p->target_high - (double)abadj_p->target_low) /
((double)brt_max - (double)brt_min);
mid_level = (abadj_p->target_high - abadj_p->target_low) / 2;
for (i = 0; i < height_pixel; i++) {
        for (j = 0; j < width_pixel; j++) {
                value = (int)(target_mid + ((double)gGray[i][j] - (double)brt_mid) *
contrast_ratio);
                if (value > 255)  value = 255;
                if (value < 0)  value = 0;
                gGray[i][j] = value;
                value = (int)(target_mid + ((double)gRed[i][j] - (double)brt_mid) *
contrast_ratio);
                if (value > 255)  value = 255;
                if (value < 0)  value = 0;
                gBuff_red[i][j] = value;
                value = (int)(target_mid + ((double)gGreen[i][j] - (double)brt_mid) *
contrast_ratio);
                if (value > 255)  value = 255;
                if (value < 0)  value = 0;
                gBuff_green[i][j] = value;
                value = (int)(target_mid + ((double)gBlue[i][j] - (double)brt_mid) *
contrast_ratio);
                if (value > 255)  value = 255;
                if (value < 0)  value = 0;
                gBuff_blue[i][j] = value;
        }
}
```

【void ABHB_resize_small()】

// 元画像情報を gRed[][],gGreen[][],gBlue[][]として、引数の resize_ratio に従ってリサイズ後、結果情報を gBuff_red[][], gBuff_green[][], gBuff_blue[][]に格納して終了します

```
// リサイズ後の縦横ピクセル数を求めます
int resize_pixel = resize_ratio;
*resize_height_pixel = (int)(height_pixel / resize_pixel);
*resize_width_pixel = (int)(width_pixel / resize_pixel);
// 端数がある場合、周辺画像を無視します
start_i = (height_pixel % *resize_height_pixel) / 2;
start_j = (width_pixel % *resize_width_pixel) / 2;
i = start_i;
j = start_j;
// リサイズ後の値をgBuff_xxx[][]に書き込みます
for (resize_i = 0; resize_i < *resize_height_pixel; resize_i++) {
        for (resize_j = 0; resize_j < *resize_width_pixel; resize_j++) {
                value_r = 0;
                value_g = 0;
                value_b = 0;
                for (n = 0; n < resize_pixel; n++) {
                        for (m = 0; m < resize_pixel; m++) {
                                value_r = value_r + (long)gRed[i + n][j + m];
                                value_g = value_g + (long)gGreen[i + n][j + m];
                                value_b = value_b + (long)gBlue[i + n][j + m];
                        }
                }
                gBuff_red[resize_i][resize_j] = (unsigned char)(value_r / (resize_pixel
* resize_pixel));
                gBuff_green[resize_i][resize_j] = (unsigned char)(value_g /
(resize_pixel * resize_pixel));
                gBuff_blue[resize_i][resize_j] = (unsigned char)(value_b /
(resize_pixel * resize_pixel));
                j = j + resize_pixel;
        }
        j = start_j;
        i = i + resize_pixel;
}
```

【int ABHB_rgb_to_hsl()】

```
// RGB の値から HSL の値に変換します
// hueを計算します
if (*green > *red && *red >= *blue) {
        hue = 60 * (*green - *red) / (*green - *blue) + 60;
}
else if (*green >= *blue && *blue > *red) {
        hue = 60 * (*blue - *red) / (*green - *red) + 120;
}
else if (*blue > *green && *green >= *red) {
        hue = 60 * (*blue - *green) / (*blue - *red) + 180;
}
else if (*blue >= *red && *red > *green) {
        hue = 60 * (*red - *green) / (*blue - *green) + 240;
}
else if (*red > *blue && *blue >= *green) {
        hue = 60 * (*red - *blue) / (*red - *green) + 300;
}
else if (*red >= *green && *green > *blue) {
        hue = 60 * (*green - *blue) / (*red - *blue);
}
else {
        hue = 0;          // *red=*green=*blueの場合は0とします
}
if (hue >= 360) {
        hue = 359;        // 計算上360になる場合があるのでその場合は359とします
}
*Hue = hue;
// saturationを計算します
if (*red >= *green && *red >= *blue) {
        Bmax = *red;
}
else if (*green >= *blue) {
        Bmax = *green;
}
else {
```

```
        Bmax = *blue;
}
if (*red <= *green && *red <= *blue) {
        Bmin = *red;
}
else if (*green <= *blue) {
        Bmin = *green;
}
else {
        Bmin = *blue;
}
if (Bmax == 0 || (256 - abs(Bmax + Bmin - 256)) == 0) {
        saturation = 0;
}
else {
        saturation = Bmax - Bmin;          // 双円錐モデル
        if (saturation > 255) {
                saturation = 255;
        }

}
*Saturation = saturation;
// brightnessを計算します
*Brightness = (Bmax + Bmin) / 2;           // HSL円柱/双円錐モデル
// xyzに変換します　(x, y)=(256, 0) をhue=0°として、時計回りに赤→緑→青とします
*x = double(saturation) * cos(hue * PAI / 180.0);
*y = -double(saturation) * sin(hue * PAI / 180.0);
*z = *Brightness;
```

【int ABHB_roughness_analysis()】

// 元画像情報を gRed[][], gGreen[][], gBlue[][]として、引数のパラメータ構造体に従ってザラザラ（ツルツル）感を数値化します

// 指定された範囲のvalueが0以上の時はgBuff_blue[][]=valueのピクセル、valueが負の値の時は全ピクセルを対象に

// グレースケールヒストグラムを生成します

// stepで指定された数値を辺の長さとする正方形内のピクセル平均値で算出します

```
int counter = 0;
for (i = top; i <= bottom - step; i = i + step) {
        for (j = left; j <= right - step; j = j + step) {
                if (gBuff_blue[i][j] == value || value < 0) {
                        int red_ave = 0;
                        int green_ave = 0;
                        int blue_ave = 0;
                        int ave_counter = 0;
                        for (ii = i; ii < i + step; ii++) {
                                for (jj = j; jj < j + step; jj++) {
                                        if (gBuff_blue[ii][jj] == value || value < 0)
{
                                                red_ave = red_ave + gRed[ii][jj];
                                                green_ave = green_ave +
gGreen[ii][jj];
                                                blue_ave = blue_ave + gBlue[ii][jj];
                                                ave_counter++;
                                        }
                                }
                        }
                        if (0 < ave_counter) {
                                red_ave = red_ave / ave_counter;
                                green_ave = green_ave / ave_counter;
                                blue_ave = blue_ave / ave_counter;
                                hist[(red_ave + green_ave + blue_ave) / 3]++;
                                counter++;
                        }
                }
        }
```

```
}
// 構造体で指定された比率のlow_brt, mid_brt, high_brtを構造体に書き入れます
int k;
if (0 < counter) {
        for (k = 0; k < 256; k++) {
                hist[k] = hist[k] * 1000 / counter;
        }
}
int low_value = roughness_p->low_percent * 10;
int mid_value = roughness_p->mid_percent * 10;
int hist_value = 0;
for (k = 0; k < 256; k++) {
        hist_value = hist_value + hist[k];
        if (roughness_p->low_brt == -1 && low_value <= hist_value) {
                roughness_p->low_brt = k;
        }
        if (roughness_p->mid_brt == -1 && mid_value <= hist_value) {
                roughness_p->mid_brt = k;
        }
}
int high_value = (100 - roughness_p->high_percent) * 10;
hist_value = 0;
for (k = 255; 0 <= k; k--) {
        hist_value = hist_value + hist[k];
        if (roughness_p->high_brt == -1 && high_value <= hist_value) {
                roughness_p->high_brt = k;
        }
}
```

【int ABHB_edge_mapping()】

// 元画像情報を gGray[][]として、引数で指定した diff_distance, brt_diff_sh に基づいて縦方向および横方向それぞれ全ピクセルの 2 階微分を行い、

// エッジ部分のピクセルの gBuff_blue[][]=value にして終了します

```
for (i = diff_distance; i < height_pixel - diff_distance; i++) {
        for (j = diff_distance; j < width_pixel - diff_distance; j++) {
                int diff = abs((gGray[i - diff_distance][j] - gGray[i][j]) -
(gGray[i][j] - gGray[i + diff_distance][j]));
                if (brt_diff_sh <= diff) {
                        gBuff_blue[i][j] = value;
                }
                diff = abs((gGray[i][j - diff_distance] - gGray[i][j]) - (gGray[i][j] -
gGray[i][j + diff_distance]));
                if (brt_diff_sh <= diff) {
                        gBuff_blue[i][j] = value;
                }
        }
}
```

【int ABHB_area_trace()】

```
// gBuff_blue[][]=255 の塊の最上部左端のピクセル座標(x,y)を(j_start,i_start)とした場合、
// その塊の外周を順次トレースします
offset[0][0] = -1;          offset[0][1] = 0;          // 上
offset[1][0] = 0;           offset[1][1] = 1;          // 右
offset[2][0] = 1;           offset[2][1] = 0;          // 下
offset[3][0] = 0;           offset[3][1] = -1;         // 左
offset[4][0] = -1;          offset[4][1] = 0;          // 上
offset[5][0] = 0;           offset[5][1] = 1;          // 右
offset[6][0] = 1;           offset[6][1] = 0;          // 下
offset[7][0] = 0;           offset[7][1] = -1;         // 左
// エリア領域の外周のピクセルをトレースします
Int continue_flag = 1;
while (continue_flag == 1) {
        // トレースしたピクセルのgArea_redを255にセットします
        gBuff_red[i_trace][j_trace] = 255;
        // トレース前の位置から時計回りの上下左右のピクセルで、ブルーのピクセルを探します
        m = m_before;
        find_n = 0;
        for (n = 1; n < 5; n++) {
                // 画像範囲内のみ、探します
                if (0 <= i_trace + offset[m + n][0] && i_trace + offset[m + n][0] < gHeight && 0 <= j_trace + offset[m + n][1] && j_trace + offset[m + n][1] < gWidth) {
                        // ブルーのピクセルが見つかったらその位置を記憶してループを抜けます
                        if (gBuff_blue[i_trace + offset[m + n][0]][j_trace + offset[m + n][1]] == 255) {
                                find_n = n;
                                n = 5;
                        }
                }
        }
        // ブルーのピクセルが見つかった場合、
        if (0 < find_n) {
                // そのピクセルがスタートピクセルに一致した場合、
```

```
                if (i_trace + offset[m + find_n][0] == i_start && j_trace + offset[m +
find_n][1] == j_start) {
                        // そのトレースピクセルと内部のピクセル数を数えます
                        // 縦横の長さとエリア内のピクセル数が条件に合った場合のみ有効
にします
                        if (condition_func(top, bottom, left, right, trace_counter,
pxl_counter) == 1) {
                                area_info[*area_info_counter][0] = 0;
                                area_info[*area_info_counter][1] = trace_counter;
                                area_info[*area_info_counter][2] = i_start;
                                area_info[*area_info_counter][3] = j_start;
                                area_info[*area_info_counter][4] = top;
                                area_info[*area_info_counter][5] = bottom;
                                area_info[*area_info_counter][6] = left;
                                area_info[*area_info_counter][7] = right;
                                (*area_info_counter)++;
                                If (*area_info_counter >= area_info_counter_max)
*area_info_counter = area_info_counter_max - 1;
                        }
                        // トレースを終了します
                        continue_flag = 0;
                }
                // そのピクセルがスタートピクセルと一致しない場合
                else {
                        // 次のトレースの為の設定を行います
                        i_trace = i_trace + offset[m + find_n][0];
                        j_trace = j_trace + offset[m + find_n][1];
                        m_before = (m + find_n + 2) % 4;
                }
        }
        // 該当領域のピクセルが見つからなかった場合、トレースを中止します
        else {
                continue_flag = 0;
        }
}
```

【int ABHB_sort_effective_area_info()】

// area_info[][0]が負の値のエリア情報をクリアして、有効なエリア情報だけを残して０から採番しなおします

```
for (n = 0; n < *area_info_counter; n++) {
        if (area_info[n][0] < 0) {
                for (m = n + 1; m < *area_info_counter; m++) {
                        for (k = 0; k < 13; k++) {
                                area_info[m - 1][k] = area_info[m][k];
                        }
                }
                n--;
                (*area_info_counter)--;
        }
}
```

【int ABHB_scratch_mapping()】

```
// ほぼ同等の明るさの平面に囲まれる黒い（白い）ライン状のピクセルをマッピングします
int edge_margin = scratch_p.distance + scratch_p.offset_s + 1;
for (i = edge_margin; i < height_pixel - edge_margin; i++) {
        for (j = edge_margin; j < width_pixel - edge_margin; j++) {
                // 一定の明るさの範囲のピクセルを対象とします
                if (scratch_p.target_brt_min <= gGray[i][j] && gGray[i][j] <=
scratch_p.target_brt_max) {
                        if (scratch_p.mode / 10 == 1) {
                                // 縦方向を処理します
                                int brt_center = 0;
                                for (k = -scratch_p.offset_c; k <= scratch_p.offset_c;
k++) {
                                        ii = i + k;
                                        jj = j;
                                        brt_center = brt_center + gGray[ii][jj];
                                }
                                brt_center = brt_center / (scratch_p.offset_c * 2 +
1);
                                int brt_side1 = 0;
                                for (k = -scratch_p.offset_s; k <= scratch_p.offset_s;
k++) {
                                        ii = i - scratch_p.distance + k;
                                        jj = j;
                                        brt_side1 = brt_side1 + gGray[ii][jj];
                                }
                                brt_side1 = brt_side1 / (scratch_p.offset_s * 2 + 1);
                                int brt_side2 = 0;
                                for (k = -scratch_p.offset_s; k <= scratch_p.offset_s;
k++) {
                                        ii = i + scratch_p.distance + k;
                                        jj = j;
                                        brt_side2 = brt_side2 + gGray[ii][jj];
                                }
                                brt_side2 = brt_side2 / (scratch_p.offset_s * 2 + 1);
```

```
                                if (scratch_p.side_brt_min < brt_side1 &&
scratch_p.side_brt_min < brt_side2 && brt_side1 < scratch_p.side_brt_max && brt_side2 <
scratch_p.side_brt_max) {
                                        if (abs(brt_side1 - brt_side2) <
scratch_p.side_diff_sh) {
                                                // 傷が黒い場合
                                                if (scratch_p.brt_diff_sh < 0) {
                                                        if
(abs(scratch_p.brt_diff_sh) < brt_side1 - brt_center && abs(scratch_p.brt_diff_sh) <
brt_side2 - brt_center) {
                                                                gBuff_blue[i][j] =
value;
                                                        }
                                                }
                                                // 傷が白い場合
                                                else {
                                                        if
(abs(scratch_p.brt_diff_sh) < brt_center - brt_side1 && abs(scratch_p.brt_diff_sh) <
brt_center - brt_side2) {
                                                                gBuff_blue[i][j] =
value;
                                                        }
                                                }
                                        }
                                }
                        }
                        if (scratch_p.mode % 10 == 1) {
                                // 横方向をチェックします
                        }
                }
        }
}
```

【int ABHB_joint_near_pixel()】

```
// 縦横斜め、指定した方向に指定したピクセル数以下で隣接するピクセル同士を繋げます
if (direction % 10 == 1) {
        // 横方向を処理します
        for (i = 0; i < height_pixel; i++) {
                for (j = 1; j < width_pixel; j++) {
                        if (gBuff_blue[i][j - 1] == 0 && gBuff_blue[i][j] == value) {
                                counter = 0;
                                ii = i;
                                jj = j - 1;
                                while (1 <= jj && gBuff_blue[ii][jj] == 0 && counter <
joint_length) {
                                        jj--;
                                        counter++;
                                }
                                if (counter < joint_length) {
                                        for (jj = j - 1; jj > j - 1 - counter; jj--)
{
                                                gBuff_blue[ii][jj] = value;
                                        }
                                }
                        }
                }
        }
}
if ((direction / 10) % 10 == 1) {
        // 縦方向を処理します
}
if ((direction / 100) % 10 == 1) {
        // 右下がり方向を処理します
}
if ((direction / 1000) % 10 == 1) {
        // 右上がり方向を処理します
}
```

【int ABHB_clear_continue_pixel()】

```
// 縦横斜め、指定した方向に指定したピクセル数以下で連続するピクセルをクリアします
if (direction % 10 == 1) {
        // 横方向を処理します
        for (i = 0; i < height_pixel; i++) {
                for (j = 1; j < width_pixel; j++) {
                        if (gBuff_blue[i][j - 1] == value && gBuff_blue[i][j] == 0) {
                                counter = 0;
                                ii = i;
                                jj = j - 1;
                                while (1 <= jj && gBuff_blue[ii][jj] == value &&
counter <= clear_length) {
                                        jj--;
                                        counter++;
                                }
                                if (counter <= clear_length) {
                                        for (jj = j - 1; jj > j - 1 - counter; jj--)
{
                                                gBuff_blue[ii][jj] = 0;
                                        }
                                }
                        }
                }
        }
}
if ((direction / 10) % 10 == 1) {
        // 縦方向を処理します
}
if ((direction / 100) % 10 == 1) {
        // 右下がり方向を処理します
}
if ((direction / 1000) % 10 == 1) {
        // 右上がり方向を処理します
}
```