appariement.c (partie 1)

```c
#include "appariement.h"


void generer_paires(int pairs[NUM_PAIRS][4]) {
    for (int i = 0; i < NUM_PAIRS; i++) {
        pairs[i][0] = rand() % PATCH_SIZE - PATCH_SIZE / 2;
        pairs[i][1] = rand() % PATCH_SIZE - PATCH_SIZE / 2;
        pairs[i][2] = rand() % PATCH_SIZE - PATCH_SIZE / 2;
        pairs[i][3] = rand() % PATCH_SIZE - PATCH_SIZE / 2;
    }
    printf("Paires BRIEF générées avec succés\n");
}

uint8_t get_pixel(matrice* image, int x, int y) {
    if (x < image->m && y < image->n) {
        return (uint8_t) image->mat[y][x];
    } else {
        printf("erreur: %d<%d ou %d<%d ", x, y, image->m, image->n);
        return (uint8_t) 0;
    }
}

pixel_rgb get_pixel_rgb(matrice* image_r,matrice* image_g, matrice* image_b, int x, int y) {
    pixel_rgb px = {0,0,0};
    if (x < image_r->m && y < image_r->n) {
        px.r = (uint8_t)image_r->mat[y][x];
        px.g = (uint8_t)image_g->mat[y][x];
        px.b = (uint8_t)image_b->mat[y][x];
        return px;
    } else {
```

appariement.c (partie 2)

```c
        printf("erreur: %d<%d ou %d<%d ", x, y, image_r->m, image_r->n);
        return px;
    }
}

uint256_t** init_descriptor(int n) {
    uint256_t** res = malloc(n * sizeof(uint256_t*));
    if (res == NULL) {
        fprintf(stderr, "Erreur d'allocation pour le tableau de descripteurs\n");
        return NULL;
    }
    for (int i = 0; i < n; i++) {
        res[i] = malloc(sizeof(uint256_t));
        res[i]->high1 = 0;
        res[i]->low1 = 0;
        res[i]->high2 = 0;
        res[i]->low2 = 0;
    }
    return res;
}

void free_descriptors(uint256_t** res, int n) {
    if (res != NULL) {
        for (int i = 0; i < n; i++) {
            free(res[i]);
        }
        free(res);
    }
}
```

appariement.c (partie 3)

```
uint256_t** compute_brief(matrice* image, matrice* points, int pairs[NUM_PAIRS][4]) {
    uint256_t** res = init_descriptor(points->n);
    if (res == NULL) {
        return NULL;
    }
    for (int i = 0; i < points->n; i++) {
        if (points->mat == NULL) {
            fprintf(stderr, "Points non valides\n");
            free(res);
            return NULL;
        }
        int x = (int) points->mat[i][0];
        int y = (int) points->mat[i][1];
        for (int j = 0; j < NUM_PAIRS; j++) {
            int x1 = x + pairs[j][0], y1 = y + pairs[j][1];
            int x2 = x + pairs[j][2], y2 = y + pairs[j][3];
            if (x1 < 0 || y1 < 0 || x1 >= image->m || y1 >= image->n ||
                x2 < 0 || y2 < 0 || x2 >= image->m || y2 >= image->n) {
                continue;
            }
            uint8_t p1 = get_pixel(image, x1, y1);
            uint8_t p2 = get_pixel(image, x2, y2);
            if (j < 64) {
                if (p1 < p2) {
                    res[i]->low1 |= (1ULL << j);
                }
            } else {
                if (j < 128) {
                    if (p1 < p2) {
                        res[i]->high1 |= (1ULL << (j - 64));
```

appariement.c (partie 4)

```
                    }
                } else {
                    if (j < 192) {
                        if (p1 < p2) {
                            res[i]->low2 |= (1ULL << (j - 128));
                        }
                    } else {
                        if (p1 < p2) {
                            res[i]->high2 |= (1ULL << (j - 192));
                        }
                    }
                }
            }
        }
    }
    printf("Descripteurs BRIEF calculés\n");
    return res;
}


uint256_t** compute_brief_rgb(matrice* image_r,matrice* image_g,matrice* image_b, matrice* points,
↪   int pairs[NUM_PAIRS][4]) {
    uint256_t** res = init_descriptor(points->n);
    if (res == NULL) {
        return NULL;
    }
    for (int i = 0; i < points->n; i++) {
        if (points->mat == NULL) {
            fprintf(stderr, "Points non valides\n");
            free(res);
            return NULL;
```

appariement.c (partie 5)

```c
        }
        int x = (int) points->mat[i][0];
        int y = (int) points->mat[i][1];
        for (int j = 0; j < NUM_PAIRS; j++) {
            int x1 = x + pairs[j][0], y1 = y + pairs[j][1];
            int x2 = x + pairs[j][2], y2 = y + pairs[j][3];
            if (x1 < 0 || y1 < 0 || x1 >= image_r->m || y1 >= image_r->n ||
                x2 < 0 || y2 < 0 || x2 >= image_r->m || y2 >= image_r->n) {
                continue;
            }
            pixel_rgb p1 = get_pixel_rgb(image_r,image_g,image_b, x1, y1);
            pixel_rgb p2 = get_pixel_rgb(image_r,image_g,image_b, x2, y2);
            if (p1.r < p2.r) res[i]->low1 |= (1ULL << j);
            if (p1.g < p2.g) res[i]->low2 |= (1ULL << j);
            if (p1.b < p2.b) res[i]->high1 |= (1ULL << j);
        }
    }
    printf("Descripteurs BRIEF_rgb calculés\n");
    return res;
}

int one_count(uint64_t x) {
    int count = 0;
    while (x) {
        count += x & 1;
        x >>= 1;
    }
    return count;
}
```

appariement.c (partie 6)

```c
int hamming_distance(uint256_t* d1, uint256_t* d2) {
    uint64_t xor1 = d1->high1 ^ d2->high1;
    uint64_t xor2 = d1->low1 ^ d2->low1;
    uint64_t xor3 = d1->high2 ^ d2->high2;
    uint64_t xor4 = d1->low2 ^ d2->low2;
    return 2*one_count(xor1) + 2*one_count(xor2) + 2*one_count(xor3) + one_count(xor4);
}

matrice* epipolar_line(matrice* F, matrice* X) {
    matrice* res = matrice_nulle(3, 1);
    res = produit(F, X);
    multiplication_scalaire(res, 100);
    return res;
}
```

camera_calibration.c (partie 1)

```c
#include "camera_calibration.h"

// Fonction pour construire la matrice A
matrice* construction_A(double* X, double* Y, double* Z, double* u, double* v, int n) {
    matrice* A = matrice_nulle(2 * n, 12); // 12 colonnes pour les 12 paramètres de P

    for (int i = 0; i < n; i++) {
        int row1 = 2 * i;
        int row2 = 2 * i + 1;
        // lambda_i * u_i = ...
        A->mat[row1][0] = X[i];
        A->mat[row1][1] = Y[i];
        A->mat[row1][2] = Z[i];
        A->mat[row1][3] = 1;
        A->mat[row1][8] = -X[i]* u[i];
        A->mat[row1][9] = -Y[i]* u[i];
        A->mat[row1][10] = -Z[i]* u[i];
        A->mat[row1][11] = -u[i];

        // lambda_i * v_i = ...
        A->mat[row2][4] = X[i];
        A->mat[row2][5] = Y[i];
        A->mat[row2][6] = Z[i];
        A->mat[row2][7] = 1;
        A->mat[row2][8] = -X[i]* v[i];
        A->mat[row2][9] = -Y[i]* v[i];
        A->mat[row2][10] = -Z[i]* v[i];
        A->mat[row2][11] = -v[i];

    }
```

camera_calibration.c (partie 2)

```
    return A;
}

// Calcule de la matrice de projection et des matrices de paramètres intrinsèques
void camera_calibration_resolution(matrice* P, matrice* A, matrice* K, matrice* R, matrice* T) {
    // Étape 1 : Décomposition SVD de A
    matrice* S = matrice_nulle(A->n, A->m);
    matrice* V = matrice_nulle(A->m, A->m);
    matrice* U = matrice_nulle(A->n, A->n);
    qr_algorithm_SVD(A, U, S, V);

    // Étape 2 : Extraire le vecteur p (solution homogène de Ap=0)
    int index_min = S->n - 1;
    while (S->mat[index_min][index_min] < EPSILON) {
        index_min--;
        assert(index_min >= 0);
    }
    matrice *p = matrice_nulle(V->m, 1);
    for (int i = 0; i < V->m; i++) {
        p->mat[i][0] = V->mat[i][index_min];
    }

    // Étape 3 : Construire la matrice de projection P (3*4)
    for (int i = 0; i < 12; i++) {
        P->mat[i / 4][i % 4] = p->mat[i][0];
    }

    free_matrice(U);
    free_matrice(S);
    free_matrice(V);
```

camera_calibration.c (partie 3)

```c
free_matrice(p);

// Étape 4 : Extraire le noyau droit de P pour obtenir le centre C*
matrice* U2 = matrice_nulle(P->n, P->n);
matrice* S2 = matrice_nulle(P->n, P->m);
matrice* V2 = matrice_nulle(P->m, P->m);
qr_algorithm_SVD(P, U2, S2, V2);
index_min = fmin(P->n - 1, P->m - 1);
matrice *C_star = matrice_nulle(V2->m, 1);
for (int i = 0; i < V2->m; i++) {
    C_star->mat[i][0] = V2->mat[i][index_min];
}
// Homogénéisation
double scale = C_star->mat[C_star->n - 1][0];
for (int i = 0; i < C_star->n; i++) {
    C_star->mat[i][0] /= scale;
}
// Étape 5 : QR de M = P[0:3, 0:3]-1
matrice* M = matrice_nulle(3, 3);
for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j)
        M->mat[i][j] = P->mat[i][j];

matrice* M_inv = inverser_matrice(M);
matrice* Q = matrice_nulle(3, 3);
matrice* R_temp = matrice_nulle(3, 3);
decomposition_QR_householder(M_inv, Q, R_temp);
copie_matrice(R_temp, R);
// Étape 6 : Extraire R et K
copie_matrice(R_temp, R); // R temporaire devient R
```

camera_calibration.c (partie 4)

```
    matrice* R_inv = inverser_matrice(R);
    matrice* K_temp = produit(R_inv, M); // K = R-1*M
    multiplication_scalaire(K_temp, 1.0 / K_temp->mat[2][2]); // normalisation
    copie_matrice(K_temp, K);

    // Étape 7 : Calcul de T = -R*C*
    matrice* C_euclidienne = matrice_nulle(3, 1);
    for (int i = 0; i < 3; ++i) {
        C_euclidienne->mat[i][0] = C_star->mat[i][0];
    }
    matrice* RC = produit(R, C_euclidienne);
    multiplication_scalaire(RC, -1.0);
    for (int i = 0; i < 3; ++i)
        T->mat[i][0] = RC->mat[i][0];

    free_matrice(U2);
    free_matrice(S2);
    free_matrice(V2);
    free_matrice(C_star);
    free_matrice(M);
    free_matrice(M_inv);
    free_matrice(Q);
    free_matrice(R_temp);
    free_matrice(R_inv);
    free_matrice(K_temp);
    free_matrice(RC);
}

//Calcule la matrice fondamental F associé aux images
matrice* compute_F(matrice* K1 ,matrice* R1, matrice* T1, matrice* K2, matrice* R2, matrice* T2){
```

camera_calibration.c (partie 5)

```c
    matrice* KR1 = produit(K1, R1);
    matrice* KR1_inv = inverser_matrice(KR1);
    matrice* temp = produit(K2, R2);
    matrice* M = produit(temp, KR1_inv);
    free_matrice(temp);
    matrice* K1T1 = produit(K1, T1);
    matrice* inter = produit(KR1_inv, K1T1);
    matrice* R2_K1T1 = produit(R2, inter);
    matrice* neg_term = produit(K2, R2_K1T1);
    multiplication_scalaire(neg_term, -1);
    matrice* K2T2 = produit(K2, T2);
    matrice* v = somme(neg_term, K2T2);
    return produit_vectoriel(v, M);
}
```

constante.c (partie 1)

```c
#include "constante.h"

//Appariement

int Distance_seuil = 5;
int Hamming_seuil = 17;

//Detection
int Window = 4;
int Seuil_moravec = 500;


//Ransac
int RANSAC_ITER = 0;
int DIST_THRESHOLD=2.0;
int MIN_INLIERS=5;

//Trouve coin
int Seuil_tc = 40;
int Dist_tc = 10;

//Triangle
int Seuil_triangle=1e-3;
```

detection.c (partie 1)

```
#include "detection.h"

//distance du point à la droite
float point_line_distance(matrice* line, matrice* point) {
  float a = line->mat[0][0];
  float b = line->mat[1][0];
  float c = line->mat[2][0];
  float x = point->mat[0][0];
  float y = point->mat[1][0];
  return fabs(a * x + b * y + c) / sqrt(a * a + b * b);
}

//Transforme la matrice pbm image de n points en une liste de points
matrice* bit_image_to_points (matrice* image, int nb_points){
    matrice* res=matrice_nulle(nb_points,2);
    int c=0;
    for (int i = 0; i < image->n; i++){
        for (int j=0; j<image->m; j++){
            if ((image->mat[i][j])==1){
                res->mat[c][0]=j;
                res->mat[c][1]=i;
                c++;
            }
        }
    }
    return res;

}

matrice* selection_moravec(char* filename, int* nbp, matrice* input){
```

detection.c (partie 2)

```c
    matrice* output=matrice_nulle(input->n,input->m);
    int nb_points=moravec(input, output);
    *nbp=nb_points;
    char output_name[128];
    char parametre[256];
    snprintf(parametre, sizeof(parametre), "fichier:%s, seuil:%d, fenetre:%d, param:%d", filename, Seuil_moravec, Window, PARAM);
    snprintf(output_name, sizeof(output_name), "points_%s.txt", filename);
    save_matrice_to_file(output, filename);
    matrice* points=bit_image_to_points(output, *nbp);
    return points;
}

void init_img_moravec(matrice** img1, matrice** img2, char* filename1, char* filename2, matrice* input1, matrice* input2){
  char points_file1[MAX_FILENAME], points_file2[MAX_FILENAME];
  snprintf(points_file1, sizeof(points_file1), "points_%s.txt", filename1);
  snprintf(points_file2, sizeof(points_file2), "points_%s.txt", filename2);
  char command[256];
  //moravec
  int nbp1;
  int nbp2;
  *img1 =selection_moravec(filename1, &nbp1, input1);
  *img2 = selection_moravec(filename2, &nbp2, input2);
  save_matrice_to_file_dimension(*img1, points_file1);
  snprintf(command, sizeof(command), "python3 plot_detect_un.py %s.jpg points_%s.txt", filename1, filename1);
  //system(command);

  //trouve coin
  int** actif = NULL;
  nbp1= moravec_arr(input1,&actif);
```

detection.c (partie 3)

```
  output1=compute_score(input1,actif,nbp1);
  int nbp1bis = filtre_mat(output1,actif,nbp1);
  for(int i = 0;i<nbp1;i++){
    free(actif[i]);
  }
  free(actif);
  *img1 = bit_image_to_points(output1,nbp1bis);
  free_matrice(output1);
  save_matrice_to_file_dimension(*img1, points_file1);
  //system(command);

  //ransac
  detect_lines_and_extremities(*img1);
  save_matrice_to_file_clean_dimension(*img1, points_file1);
  save_matrice_to_file_dimension(*img2, points_file2);
  //system(command);
}

void init_img_file(matrice** img1,matrice** img2, char* filename1, char* filename2){
  char points_file1[MAX_FILENAME], points_file2[MAX_FILENAME];
  snprintf(points_file1, sizeof(points_file1), "points_%s.txt", filename1);
  snprintf(points_file2, sizeof(points_file2), "points_%s.txt", filename2);
  read_matrice_from_file_dimension(img1, points_file1);
  read_matrice_from_file_dimension(img2, points_file2);
}

//Effecture la mise en correspondance des points
matrice* corresp_color (matrice* img1, matrice* img2, matrice* input1_r,matrice* input1_g,matrice*
↪  input1_b, matrice* input2_r,matrice* input2_g, matrice* input2_b, int nbp1,int nbp2,char*
↪  filename1, char* filename2){
  matrice* retenus = matrice_nulle(nbp1, 2);
  matrice* retenus_dh = matrice_nulle(nbp1, 1);
```

detection.c (partie 4)

```
matrice* F = matrice_nulle(3, 3);
char F_name[100];
snprintf(F_name, sizeof(F_name), "F_%s.txt", filename1);
read_matrice_from_file(F, F_name);
srand(time(NULL));
int pairs[NUM_PAIRS][4];
generer_paires(pairs);
uint256_t** img1_descripteur = compute_brief_rgb(input1_r,input1_g,input1_b,img1, pairs);
uint256_t** img2_descripteur = compute_brief_rgb(input2_r,input2_g,input2_b, img2, pairs);
for (int i = 0; i < nbp1; i++) {
  matrice* X1 = coo_vect(img1->mat[i][0], img1->mat[i][1]);
  matrice* l = epipolar_line(F, X1);
  int h_min = Hamming_seuil;
  for (int j = 0; j < nbp2; j++) {
    matrice* X2 = coo_vect(img2->mat[j][0], img2->mat[j][1]);
    if (point_line_distance(l, X2) < Distance_seuil) {
      int h = hamming_distance(img2_descripteur[j], img1_descripteur[i]);
      if (h < h_min) {
        retenus->mat[i][0] = img2->mat[j][0];
        retenus->mat[i][1] = img2->mat[j][1];
        retenus_dh->mat[i][0] = h;
        h_min = h;
      }
    }
    free_matrice(X2);
  }
  free_matrice(X1);
  free_matrice(l);
}
// Suppression des correspondances aberrantes
```

detection.c (partie 5)

```
  for (int i = 0; i < nbp1; i++) {
    if (retenus->mat[i][0]==0) {
        retenus->mat[i][0] = -1;
        retenus->mat[i][1] = -1;
        img1->mat[i][0] = -1;
        img1->mat[i][1] = -1;
      }
  }
  return retenus;
}
```

manipulation_fichier.c (partie 1)

```c
#include "manipulation_fichier.h"

void nom_fichier(char* filename, char* matrix_name, char* image_name) {
    snprintf(filename, 256, "%s-%s.txt", matrix_name, image_name);
}

bool file_exists(const char *filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE *file = fopen(complete_fn, "r");
    if (file) {
        fclose(file);
        return true;
    }
    return false;
}
void read_matrice_from_file_dimension(matrice** mtx, char* filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE* fichier = fopen(complete_fn, "r");
    assert(fichier != NULL);
    int n, m;
    fscanf(fichier, "%d %d", &n, &m);
    *mtx = matrice_nulle(n, m);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            fscanf(fichier, "%lf", &((*mtx)->mat[i][j]));
        }
    }
    fclose(fichier);
```

manipulation_fichier.c (partie 2)

```c
    printf("Matrice lue depuis %s\n", filename);
}


void save_matrice_to_file_dimension(matrice* matrix, char* filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE* file = fopen(complete_fn, "w");
    assert(file != NULL);

    fprintf(file, "%d %d", matrix->n, matrix->m);
    fprintf(file, "\n");
    for (int i = 0; i < matrix->n; ++i) {
        for (int j = 0; j < matrix->m; ++j) {
            fprintf(file, "%lf ", matrix->mat[i][j]);
        }
        fprintf(file, "\n");
    }

    fclose(file);
    printf("Matrice enregistrée dans %s\n", filename);
}


void save_matrice_pbm(matrice* matrix, char* filename, char* parametre) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/images/%s", filename);
    FILE* file = fopen(complete_fn, "wb");
    assert(file != NULL);
    fprintf(file, "P1\n");
```

manipulation_fichier.c (partie 3)

```
    fprintf(file, "#%s\n", parametre);
    fprintf(file, "%d %d\n", matrix->n, matrix->m);
    for (int i = 0; i < matrix->n; ++i) {
        for (int j = 0; j < matrix->m; ++j) {
            // Arrondir et convertir en entier
            fprintf(file, "%d ", (int)round(matrix->mat[i][j]));
        }
        fprintf(file, "\n");
    }

    fclose(file);
    printf("Matrice enregistrée dans %s\n", filename);
}

void save_matrice_to_file(matrice *A, char* filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE *file = fopen(complete_fn, "w");
    assert(file != NULL);

    for (int i = 0; i < A->n; i++) {
        for (int j = 0; j < A->m; j++) {
            fprintf(file, "%lf ", A->mat[i][j]);
        }
        fprintf(file, "\n");
    }

    fclose(file);
    printf("Matrice enregistrée dans %s\n", filename);
}
```

manipulation_fichier.c (partie 4)

```
int save_matrice_to_file_clean(matrice *A, char* filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE *file = fopen(complete_fn, "w");
    assert(file != NULL);
    int nb_points=0;
    for (int i = 0; i < A->n; i++) {
        if (A->mat[i][0]!=-1){
            nb_points++;
            for (int j = 0; j < A->m; j++) {
                fprintf(file, "%lf ", A->mat[i][j]);
            }
            fprintf(file, "\n");
        }
    }

    fclose(file);
    printf("Matrice enregistrée dans %s\n", filename);
    return nb_points;
}


int save_matrice_to_file_clean_dimension(matrice *A, char* filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE* file = fopen(complete_fn, "w");
    assert(file != NULL);
    fprintf(file, "%d %d", A->n, A->m);
    fprintf(file, "\n");
```

manipulation_fichier.c (partie 5)

```
    int nb_points=0;
    for (int i = 0; i < A->n; i++) {
        if (A->mat[i][0]!=-1){
            nb_points++;
            for (int j = 0; j < A->m; j++) {
                fprintf(file, "%lf ", A->mat[i][j]);
            }
            fprintf(file, "\n");
        }
    }

    fclose(file);
    printf("Matrice enregistrée dans %s\n", filename);
    return nb_points;
}

void read_matrice_from_file(matrice* A, const char *filename) {
    char complete_fn[256];
    snprintf(complete_fn, 256, "points/donnees/%s", filename);
    FILE *file = fopen(complete_fn, "r");
    assert(file != NULL);

    for (int i = 0; i < A->n; i++) {
        for (int j = 0; j < A->m; j++) {
            fscanf(file, "%lf", &A->mat[i][j]);
        }
    }
    fclose(file);
    printf("Matrice lues depuis %s\n", filename);
}
```

manipulation_fichier.c (partie 6)

```
void load_all_points_images(const char* filename, double* u, double*v, int n) {
    FILE *file = fopen(filename, "r");
    double ui, vi;
    for (int i = 0; i < n; i++) {
        fscanf(file, "%lf %lf", &ui, &vi);
        u[i]=ui;
        v[i]=vi;
    }
    fclose(file);
}


void load_all_points_reels(const char* filename, double* X, double* Y, double* Z, int n) {
    FILE *file = fopen(filename, "r");
    assert(file != NULL);
    double Xi, Yi, Zi;
    for (int i = 0; i < n; i++) {
        fscanf(file, "%lf %lf %lf", &Xi, &Yi, &Zi);
        X[i]=Xi;
        Y[i]=Yi;
        Z[i]=Zi;
    }
    fclose(file);
}
matrice* read_jpg(char* filename){
    char input_name[32];
    snprintf(input_name, sizeof(input_name), "%s.txt", filename);
    if (!file_exists(input_name)){
```

manipulation_fichier.c (partie 7)

```
        char command[128];
        snprintf(command, sizeof(command), "python3 jpg_to_txt.py %s.jpg", filename);
        system(command);
    }
    matrice* input;
    read_matrice_from_file_dimension(&input, input_name);
    return input;
}

matrice* read_jpg_color(char* filename, char* color){
  char input_name[32];
  snprintf(input_name, sizeof(input_name), "%s_%s.txt", filename, color);
  if (!file_exists(input_name)){
      char command[128];
      snprintf(command, sizeof(command), "python3 jpg_to_txt_color.py %s.jpg", filename);
      system(command);
  }
  matrice* input;
  read_matrice_from_file_dimension(&input, input_name);
  return input;
}
```

matrice.c (partie 1)

```
#include "matrice.h"

matrice* matrice_nulle(int n, int m) {
    matrice* M = (matrice*) malloc(sizeof(matrice));
    assert(M != NULL);
    M->n = n;
    M->m = m;
    M->mat = (double**) malloc(n * sizeof(double*));
    assert(M!=NULL);
    for (int i = 0; i < n; i++) {
        M->mat[i] = (double*) calloc(m, sizeof(double));
        assert(M->mat[i]);
    }

    return M;
}

matrice* matrice_identite(int n) {
    matrice * I = matrice_nulle(n, n);
    for (int i = 0; i < n; ++i) {
        I->mat[i][i] = 1;
    }
    return I;
}

void copie_matrice(matrice* old, matrice* new){
    assert(old->n==new->n);
    assert(old->m==new->m);
    for (int i = 0; i < old->n; i++){
        for (int j = 0; j < old->m; j++){
```

matrice.c (partie 2)

```
            new->mat[i][j]=old->mat[i][j];
        }
    }
}

double norme_vecteur(matrice* a, int colonne) {
    double somme = 0;
    for (int i = 0; i < a->n; ++i) {
        somme += a->mat[i][colonne] * a->mat[i][colonne];
    }
    return sqrt(somme);
}

void print_matrice (matrice * a) {
    for(int i = 0; i < a->n; i++){
        for(int j = 0; j < a->m; j++){
            printf("%6.3lf ",a->mat[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

double produit_scalaire(matrice* l, matrice* X2) {
    return l->mat[0][0] * X2->mat[0][0] +
           l->mat[1][0] * X2->mat[1][0] +
           l->mat[2][0] * X2->mat[2][0];
}

bool matrice_egale(matrice* a, matrice* b) {
```

matrice.c (partie 3)

```
    if (a->n != b->n || a->m != b->m) {
        return false;
    }

    for (int i = 0; i < a->n; i++) {
        for (int j = 0; j < a->m; j++) {
            if (fabs(a->mat[i][j] - b->mat[i][j]) > EPSILON) {
                return false;
            }
        }
    }

    return true;
}

void free_matrice(matrice * a){
    for (int i = 0; i < a->n; i++) {
        free(a->mat[i]);
    }
    free(a->mat);
}

matrice * somme(matrice *a, matrice *b) {
    assert(a->n == b->n);
    assert(a->m == b->m);

    matrice * c = matrice_nulle(a->n,a->m);
    for (int i = 0; i < a->n; i++) {
        for (int j = 0; j < a->m; j++) {
            c->mat[i][j] = a->mat[i][j] + b->mat[i][j];
```

matrice.c (partie 4)

```c
        }
    }
    return c;
}


matrice* produit(matrice* a, matrice* b) {
    assert(a->m == b->n);
    matrice * c = matrice_nulle(a->n, b->m);
    for (int i = 0; i < a->n; i++) {
        for (int j = 0; j < b->m; j++) {
            for (int k = 0; k < a->m; k++) {
                c->mat[i][j] += a->mat[i][k] * b->mat[k][j];
            }
        }
    }
    return c;
}


void multiplication_scalaire(matrice* a, double lambda) {
    for (int i = 0; i < a->n; i++) {
        for (int j = 0; j < a->m; j++) {
            a->mat[i][j] = lambda*(a->mat[i][j]);
        }
    }
}

matrice* transposee(matrice* a) {
    matrice* at = matrice_nulle(a->m,a->n);
```

matrice.c (partie 5)

```
    for (int i = 0; i < a->n; i++) {
        for (int j=0; j<a->m; j++){
            at->mat[j][i]=a->mat[i][j];
        }
    }
    return at;
}


matrice* concatenation2(matrice* a, matrice* b) {
    assert(a->n == b->n);
    matrice* result = matrice_nulle(a->n, a->m + b->m);
    for (int i = 0; i < a->n; i++) {
        for (int j = 0; j < a->m; j++) {
            result->mat[i][j] = a->mat[i][j];
        }
    }
    for (int i = 0; i < b->n; i++) {
        for (int j = 0; j < b->m; j++) {
            result->mat[i][a->m + j] = b->mat[i][j];
        }
    }
    return result;
}

matrice* concatenation3(matrice* a, matrice* b, matrice* c) {
    return concatenation2(concatenation2(a,b),c);
}
matrice* concatenationv2(matrice* a, matrice* b){
    return transposee(concatenation2(transposee(a), transposee(b)));
```

matrice.c (partie 6)

```
}

matrice* concatenationv3(matrice* a, matrice*b, matrice* c){
    return concatenationv2(concatenationv2(a,b),c);
}


void assert_ligne_in_range(matrice * a, int i){
    assert(i >= 0 && i < a->n);
}
void assert_colonne_in_range(matrice * a, int j){
    assert(j >= 0 && j < a->m);
}

/*Resolution de système AU=V d inconnue U*/

void echange_ligne(matrice* a, int i, int j) {
    assert_ligne_in_range(a,i);
    assert_colonne_in_range(a,j);

    double * ligne_i =  a->mat[i];
    a->mat[i] = a->mat[j];
    a->mat[j] = ligne_i;
}

void multiplication_ligne(matrice* a, int i, double lambda) {
    assert_ligne_in_range(a,i);
    for (int k = 0; k < a->m; k++) {
        a->mat[i][k] *= lambda;
    }
```

matrice.c (partie 7)

```
}

void ajout_ligne(matrice* a, int i1, int i2, double lambda) {
    assert_ligne_in_range(a,i1);
    assert_ligne_in_range(a,i2);
    for (int k = 0; k < a->m; k++) {
        a->mat[i1][k] = a->mat[i1][k] + lambda * a->mat[i2][k];
    }
}

void set_colonne(matrice *target, int col_idx, matrice* column_vector) {
    assert_colonne_in_range(target, col_idx);

    if (column_vector->n != target->n || column_vector->m != 1) {
        fprintf(stderr, "Dimension mismatch in set_colonne\n");
        return;
    }

    for (int i = 0; i < target->n; i++) {
        target->mat[i][col_idx] = column_vector->mat[i][0];
    }
}

matrice* matrice_colonne(matrice* A, int j) {
    assert_colonne_in_range(A, j);

    // Création de la matrice colonne (n x 1)
    matrice* col = matrice_nulle(A->n, 1);

    // Remplir la matrice colonne avec les éléments de la colonne j de A
```

matrice.c (partie 8)

```
    for (int i = 0; i < A->n; i++) {
        col->mat[i][0] = A->mat[i][j];
    }

    return col;
}

int choix_pivot_partiel(matrice *a, int j) {
    int n = a->n;
    int max_index = j;
    double max_value = fabs(a->mat[j][j]);

    for (int i = j + 1; i < n; i++) {
        if(fabs(a->mat[i][j]) < EPSILON){
            a->mat[i][j] = 0;
        }
        if (fabs(a->mat[i][j]) > max_value) {
            max_value = fabs(a->mat[i][j]);
            max_index = i;
        }
    }
    if (max_value == 0) {
        return -1;  // Retourne -1 si aucun pivot n'est trouvé
    }
    return max_index;
}


matrice* inverser_matrice(matrice* a) {
```

matrice.c (partie 9)

```c
    if (a->n != a->m) {
        printf("Erreur : La matrice n'est pas carrée, donc non inversible.\n");
        exit(EXIT_FAILURE);
    }

    int n = a->n;
    matrice* copie = matrice_nulle(n, n);
    matrice* inv = matrice_nulle(n, n);

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            copie->mat[i][j] = a->mat[i][j];

    // Initialisation de inv à l'identité
    for (int i = 0; i < n; i++)
        inv->mat[i][i] = 1;

    for (int j = 0; j < n; ++j) {
        int pivot_index = choix_pivot_partiel(copie, j);
        if (pivot_index == -1) {
            printf("Erreur : la matrice est singulière, donc non inversible.\n");
            exit(EXIT_FAILURE);
        }

        if (pivot_index != j) {
            echange_ligne(copie, j, pivot_index);
            echange_ligne(inv, j, pivot_index);
        }

        double pivot_value = copie->mat[j][j];
```

matrice.c (partie 10)

```
        if (fabs(pivot_value) < EPSILON) {
            printf("Erreur : Pivot presque nul, la matrice est singulière.\n");
            exit(EXIT_FAILURE);
        }

        multiplication_ligne(copie, j, 1. / pivot_value);
        multiplication_ligne(inv, j, 1. / pivot_value);

        for (int k = 0; k < n; ++k) {
            if (k != j) {
                double lambda = -copie->mat[k][j];
                ajout_ligne(copie, k, j, lambda);
                ajout_ligne(inv, k, j, lambda);
            }
        }
    }

    free_matrice(copie);
    return inv;
}


matrice* produit_vectoriel(matrice* v, matrice* b) {
    assert(v->n == 3 && v->m == 1);
    matrice* ax = matrice_nulle(3, 3);

    double v1 = v->mat[0][0];
    double v2 = v->mat[1][0];
    double v3 = v->mat[2][0];
```

matrice.c (partie 11)

```
    ax->mat[0][1] = -v3;
    ax->mat[0][2] = v2;
    ax->mat[1][0] = v3;
    ax->mat[1][2] = -v1;
    ax->mat[2][0] = -v2;
    ax->mat[2][1] = v1;

    matrice* res = produit(ax, b);

    free_matrice(ax);
    return res;
}


matrice* coo_vect(double x, double y){
    matrice* res=matrice_nulle(3,1);
    res->mat[0][0]=x;
    res->mat[1][0]=y;
    res->mat[2][0]=1;
    return res;
}

matrice* coo_vect_inv(double x, double y){
    matrice* res=matrice_nulle(3,1);
    res->mat[0][0]=y;
    res->mat[1][0]=x;
    res->mat[2][0]=1;
    return res;
}
```

moravec.c (partie 1)

```c
#include "moravec.h"

// Calcul de la variance pour une direction donnée
double variance(matrice* image, int x, int y, int dx, int dy) {
    int count = 0; int sum = 0; int sumSq = 0;
    for (int i = -Window; i <= Window; ++i) {
        int nx = x + i * dx;
        int ny = y + i * dy;
        if (nx >= 0 && nx < image->m && ny >= 0 && ny < image->n) {
            int val = image->mat[ny][nx];
            sum += val;
            sumSq += val * val;
            count++;
        }
    }
    return (count > 0) ? (sumSq - (sum * sum) / count) : 0;
}

int moravec(matrice* image_input, matrice* image_output) {
    int count=0;
    for (int y = 1; y < image_input->n - 1; y++) {
        for (int x = 1; x < image_input->m - 1; x++) {
            int var0 = variance(image_input, x, y, 0, 1);  // Verticale
            int var1 = variance(image_input, x, y, 1, 0);  // Horizontale
            int var2 = variance(image_input, x, y, 1, 1);  // Diagonale bas gauche haut droite
            int var3 = variance(image_input, x, y, 1, -1); // Diagonale haut gauche bas droite

            int vartot = var0;
            if (PARAM){
                if (var1 < vartot) vartot = var1;
```

moravec.c (partie 2)

```
                if (var2 < vartot) vartot = var2;
                if (var3 < vartot) vartot = var3;
            }
            else{
                vartot += var1;
                vartot += var2;
                vartot += var3;
                vartot /= 4;
            }
            if (vartot > Seuil_moravec) {
                image_output->mat[y][x] = 1;  // Coin détecté
                count++;
            } else {
                image_output->mat[y][x] = 0;  // Pas de coin
            }
        }
    }
    return count;
}

int moravec_arr(matrice* image_input, int*** detected){
  matrice* tmp = matrice_nulle(image_input->n,image_input->m);
  int size = moravec(image_input,tmp);
  *detected= malloc(sizeof(int*)*size);
  for(int i = 0;i<size;i++){
    (*detected)[i]=malloc(sizeof(int)*2);
  }
  int k = 0;
  for(int i = 0; i < tmp->n; i++){
    for(int j = 0; j < tmp->m; j++){
```

moravec.c (partie 3)

```
      if(tmp->mat[i][j]==1){
        (*detected)[k][0]=i;
        (*detected)[k][1]=j;
        k++;
      }
    }
  }
  return size;
}
```

ransac.c (partie 1)

```
#include"ransac.h"

double point_to_line_dist(double x, double y, double a, double b, double c) {
    return fabs(a*x + b*y + c) / sqrt(a*a + b*b);
}

void detect_lines_and_extremities(matrice* points) {
    srand(time(NULL));

    printf("Démarrage de RANSAC sur %d points\n", points->n);

    for (int iter = 0; iter < RANSAC_ITER; ++iter) {
        int i1 = rand() % points->n;
        int i2 = rand() % points->n;
        if (i1 == i2) continue;

        double x1 = points->mat[i1][0], y1 = points->mat[i1][1];
        double x2 = points->mat[i2][0], y2 = points->mat[i2][1];

        if ((x1 == -1 && y1 == -1) || (x2 == -1 && y2 == -1)) continue;

        double a = y2 - y1;
        double b = x1 - x2;
        double c = x2 * y1 - x1 * y2;

        int inlier_count = 0;
        int* inliers = malloc(sizeof(int) * points->n);

        for (int i = 0; i < points->n; ++i) {
            double x = points->mat[i][0];
```

ransac.c (partie 2)

```
        double y = points->mat[i][1];
        if (x == -1 && y == -1) continue;

        if (point_to_line_dist(x, y, a, b, c) < DIST_THRESHOLD) {
            inliers[inlier_count++] = i;
        }
    }

    if (inlier_count >= MIN_INLIERS) {
        double min_proj = 1e9, max_proj = -1e9;
        double end1[2], end2[2];

        for (int i = 0; i < inlier_count; ++i) {
            int idx = inliers[i];
            double x = points->mat[idx][0];
            double y = points->mat[idx][1];
            double proj = x * a + y * b;

            if (proj < min_proj) {
                min_proj = proj;
                end1[0] = x;
                end1[1] = y;
            }
            if (proj > max_proj) {
                max_proj = proj;
                end2[0] = x;
                end2[1] = y;
            }
        }
        for (int i = 0; i < inlier_count; ++i) {
```

ransac.c (partie 3)

```
                int idx = inliers[i];
                double x = points->mat[idx][0];
                double y = points->mat[idx][1];

                if (!((x == end1[0] && y == end1[1]) || (x == end2[0] && y == end2[1]))) {
                    points->mat[idx][0] = -1;
                    points->mat[idx][1] = -1;
                }
            }

        }
        free(inliers);
    }
}
```

reconstruction.c (partie 1)

```
#include "reconstruction.h"

int reconstruction4(const char* in1, const char* in2,  const char* in3,  const char* in4,  const
↪  char* in5,const char* in6, const char* in7,const char* in8, matrice** mat){
    matrice* mat1;
    matrice* mat2;
    matrice* mat3;
    matrice* mat4;
    int nbp=reconstruction1(in1, in2, &mat1);
    nbp+=reconstruction1(in3, in4, &mat2);
    nbp+=reconstruction1(in5, in6, &mat3);
    nbp+=reconstruction1(in7, in8, &mat4);
    *mat=concatenationv3(concatenationv2(mat1, mat2),mat3,mat4);
    return nbp;
}

int reconstruction1(const  char* image_name1, const char* image_name2,  matrice** mat){
    char file_points1[256], file_points2[256];
    snprintf(file_points1, sizeof(file_points1), "points/donnees/points_ap_%s.txt", image_name1);
    snprintf(file_points2, sizeof(file_points2), "points/donnees/points_ap_%s.txt", image_name2);

    char p_file1[256], p_file2[256];
    snprintf(p_file1, sizeof(p_file1), "P-%s.txt", image_name1);
    snprintf(p_file2, sizeof(p_file2), "P-%s.txt", image_name2);

    matrice* P1 = matrice_nulle(3, 4);
    matrice* P2 = matrice_nulle(3, 4);
    read_matrice_from_file(P1, p_file1);
    read_matrice_from_file(P2, p_file2);
    int nb_points =reconstruction1_aux(P1,P2, file_points1,file_points2, mat, image_name1);
    return nb_points;
```

reconstruction.c (partie 2)

```
    free_matrice(P1);
    free_matrice(P2);
}

bool filtre (matrice* p){
    if
↪   ((p->mat[0][0]>1.5)||(p->mat[1][0]>1.5)||(p->mat[0][0]<-0.4)||(p->mat[1][0]<-0.4)||(p->mat[2][0]>3)){
        return false;
    }
    return true;
}


int reconstruction1_aux(matrice* P1, matrice* P2,const char* file_points1, const char* file_points2,
↪   matrice** mat, const char* image_name1) {
    int points_ecrits=0;
    FILE* f1 = fopen(file_points1, "r");
    FILE* f2 = fopen(file_points2, "r");
    if (!f1 || !f2) {
        fprintf(stderr, "Erreur ouverture fichiers points.\n");
        return 1;
    }
    char output_file[256];
    snprintf(output_file, sizeof(output_file), "points/donnees/points_3d_%s.txt", image_name1);
    FILE* output = fopen(output_file, "w");
    assert(output != NULL);


    printf("Reconstruction des points 3D ...\n");
    double u1, v1, u2, v2;
    while (fscanf(f1, "%lf %lf", &u1, &v1) == 2 && fscanf(f2, "%lf %lf", &u2, &v2) == 2) {
        matrice* A = matrice_nulle(4, 4);
```

reconstruction.c (partie 3)

```
        for (int j = 0; j < 4; ++j) {
            A->mat[0][j] = u1 * P1->mat[2][j] - P1->mat[0][j];
            A->mat[1][j] = v1 * P1->mat[2][j] - P1->mat[1][j];
            A->mat[2][j] = u2 * P2->mat[2][j] - P2->mat[0][j];
            A->mat[3][j] = v2 * P2->mat[2][j] - P2->mat[1][j];
        }

        matrice* S = matrice_nulle(A->m, A->n);
        matrice* V = matrice_nulle(A->n, A->n);
        matrice* U = matrice_nulle(A->m, A->m);
        qr_algorithm_SVD(A, U, S, V);

        int index_min = S->n - 1;
        matrice* p = matrice_nulle(V->m, 1);
        for (int i = 0; i < V->m; i++) {
            p->mat[i][0] = V->mat[i][index_min];
        }

        for (int i = 0; i < 3; ++i) {
            p->mat[i][0] /= p->mat[3][0];
        }

        if (((p->mat[2][0])>1.2)&&(filtre(p))){
            fprintf(output, "%f %f %f\n", p->mat[0][0], p->mat[1][0], p->mat[2][0]);
            points_ecrits++;
        }

        // Libérer les matrices temporaires
        free_matrice(A); free_matrice(S); free_matrice(V); free_matrice(U); free_matrice(p);
    }
```

reconstruction.c (partie 4)

```
    fclose(output);
    *mat=matrice_nulle(points_ecrits, 3);
    snprintf(output_file, sizeof(output_file), "points_3d_%s.txt", image_name1);
    read_matrice_from_file(*mat,output_file);
    fclose(f1);
    fclose(f2);
    return points_ecrits;
}
```

SVD.c (partie 1)

```c
#include "SVD.h"

double norme_vecteur_colonne(matrice a, int colonne) {
    double somme = 0.0;
    for (int i = 0; i < a.n; i++) {
        somme += a.mat[i][colonne] * a.mat[i][colonne];
    }
    return sqrt(somme);
}

void normaliser_colonne(matrice* u) {
    double norme = 0.0;
    for (int i = 0; i < u->n; i++) {
        norme += u->mat[i][0] * u->mat[i][0];
    }
    norme = sqrt(norme);
    if (norme > precision) {
        for (int i = 0; i < u->n; i++) {
            u->mat[i][0] /= norme;
        }
    }
}

int verifier_orthogonalite(matrice* M) {
    matrice* Mt = transposee(M);
    matrice* identite = produit(Mt, M);
    int ok = matrice_egale(identite, matrice_identite(M->m));
    free_matrice(Mt);
    free_matrice(identite);
    return ok;
```

SVD.c (partie 2)

```
}

void decomposition_QR(matrice* A, matrice* Q, matrice* R) {
    int n=A->n;
    // Gram-Schmidt
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < n; i++) {
            Q->mat[i][j] = A->mat[i][j];
        }

        // Orthogonalisation
        for (int k = 0; k < j; k++) {
            long double dot_product = 0.0;
            for (int i = 0; i < n; i++) {
                dot_product += Q->mat[i][k] * A->mat[i][j];
            }
            R->mat[k][j] = dot_product;
            for (int i = 0; i < n; i++) {
                Q->mat[i][j] = Q->mat[i][j] - (R->mat[k][j] * Q->mat[i][k]);
            }
        }

        // Normalisation
        R->mat[j][j] = norme_vecteur_colonne(*Q, j);
        for (int i = 0; i < n; i++) {
            if (R->mat[j][j]>precision){
                Q->mat[i][j] /= R->mat[j][j];
            }
        }
```

SVD.c (partie 3)

```
    assert(verifier_orthogonalite(Q));
    }
}
void decomposition_QR_householder(matrice* A, matrice* Q, matrice* R) {
    int n = A->n;
    int m = A->m;

    // Initialisation
    matrice* Q_temp = matrice_identite(n);
    matrice* A_copy=matrice_nulle(A->n,A->n);
    copie_matrice(A,A_copy);

    for (int k = 0; k < n && k < m; ++k) {
        // Calcul de la norme de la colonne à partir de l'indice k
        long double norm = 0.0;
        for (int i = k; i < n; ++i) {
            norm += A_copy->mat[i][k] * A_copy->mat[i][k];
        }
        norm = sqrtl(norm);

        if (fabsl(norm) < precision) continue;

        // Création du vecteur de Householder
        long double v[n];
        for (int i = 0; i < n; ++i) v[i] = 0.0;

        for (int i = k; i < n; ++i) {
            v[i] = A_copy->mat[i][k];
        }
        v[k] += (v[k] >= 0 ? norm : -norm); // signe pour éviter annulation numérique
```

SVD.c (partie 4)

```
// Normalisation
long double v_norm = 0.0;
for (int i = k; i < n; ++i) {
    v_norm += v[i] * v[i];
}
v_norm = sqrtl(v_norm);
for (int i = k; i < n; ++i) {
    v[i] /= v_norm;
}

// Appliquer H = I - 2vv^T à A_copy
for (int j = k; j < m; ++j) {
    long double dot = 0.0;
    for (int i = k; i < n; ++i) {
        dot += v[i] * A_copy->mat[i][j];
    }
    for (int i = k; i < n; ++i) {
        A_copy->mat[i][j] -= 2 * v[i] * dot;
    }
}

// Appliquer H à Q_temp
for (int j = 0; j < n; ++j) {
    long double dot = 0.0;
    for (int i = k; i < n; ++i) {
        dot += v[i] * Q_temp->mat[i][j];
    }
    for (int i = k; i < n; ++i) {
        Q_temp->mat[i][j] -= 2 * v[i] * dot;
```

SVD.c (partie 5)

```
            }
        }
    }
    assert(verifier_orthogonalite(Q_temp));

    // Transposer Q_temp pour obtenir Q
    matrice* Q_final = transposee(Q_temp);
    copie_matrice(Q_final, Q);
    copie_matrice(A_copy, R);

    // Libérations
    free_matrice(Q_temp);
    free_matrice(Q_final);
    free_matrice(A_copy);
}


void qr_algorithm(matrice *A, matrice* S) {
    matrice* At = transposee(A);
    matrice* AtA = produit(At, A);
    matrice* Q_accum = matrice_identite(AtA->n);
    matrice* B = AtA;
    int iters = 0;
    long double diff = 1e9;
    while ((diff > precision)&&(iters<1e3)) {
        matrice *Q, *R;
        decomposition_QR_householder(B, Q, R);
        Q_accum = produit(Q_accum, Q);
        matrice* B_new = produit(R, Q);
        diff = 0.0;
```

SVD.c (partie 6)

```
        for (int i = 0; i < B_new->n; i++) {
            diff += fabs(B_new->mat[i][i] - B->mat[i][i]);
        }
        free_matrice(B);
        B = B_new;
        iters++;
    }
    S = matrice_nulle(B->n, B->m);
    for (int i = 0; i < B->n; i++) {
        if (B->mat[i][i] > 0){
            S->mat[i][i] = sqrt(B->mat[i][i]);
        }
    }
    printf("Convergence atteinte après %d itérations.\n", iters);
    free_matrice(At);
    free_matrice(Q_accum);
    free_matrice(B);
}

void eigen_decomposition(matrice* AtA, matrice* S, matrice* V) {
    int n = AtA->n;
    matrice* B = matrice_nulle(n, n);
    copie_matrice(AtA, B);
    matrice* Q_accum = matrice_identite(n);

    double epsilon = 1e-12;
    double diff = 1.0;
    int max_iter = 1000;
    int iters = 0;
```

SVD.c (partie 7)

```c
while (diff > 1e-9 && iters < max_iter) {
    matrice* Q = matrice_nulle(n, n);
    matrice* R = matrice_nulle(n, n);
    decomposition_QR_householder(B, Q, R);

    matrice* B_new = produit(R, Q);
    matrice* Q_accum_new = produit(Q_accum, Q);

    diff = 0.0;
    for (int i = 0; i < n; i++) {
        diff += fabs(B_new->mat[i][i] - B->mat[i][i]);
    }

    free_matrice(B);
    free_matrice(Q);
    free_matrice(R);
    free_matrice(Q_accum);

    B = B_new;
    Q_accum = Q_accum_new;
    iters++;
}

for (int i = 0; i < n && i < S->m && i < S->n; i++) {
    double lambda = B->mat[i][i];
    S->mat[i][i] = (lambda > epsilon) ? lambda : 0.0;
}

copie_matrice(Q_accum, V);
```

SVD.c (partie 8)

```
    free_matrice(B);
    free_matrice(Q_accum);
}

void qr_algorithm_SVD(matrice* A, matrice* U, matrice* S, matrice* V) {
    int n = A->n;
    matrice* At = transposee(A);
    matrice* AtA = produit(At, A);
    eigen_decomposition(AtA, S, V);
    for (int i = 0; i < n; i++) {
        double sigma2 = S->mat[i][i];
        if (sigma2 < 1e-12) continue;

        double sigma = sqrt(sigma2);
        S->mat[i][i] = sigma;

        matrice* v_i = matrice_colonne(V, i);
        matrice* u_i = produit(A, v_i);
        multiplication_scalaire(u_i, 1.0 / sigma);
        normaliser_colonne(u_i);
        set_colonne(U, i, u_i);

        free_matrice(v_i);
        free_matrice(u_i);
    }

    free_matrice(At);
    free_matrice(AtA);
}
```

SVD.c (partie 9)

```
double min_eig(matrice* A) {
    matrice* S;
    qr_algorithm(A, S);
    long double seuil = precision;
    long double min_sigma = 0.0;
    int i = S->n - 1;
    while ((i >= 0)&&fabs(min_sigma<seuil)) {
        if (fabs(S->mat[i][i]) > seuil) {
            min_sigma = S->mat[i][i];
        }
        i--;
    }
    free_matrice(S);
    return min_sigma;
}
```

test_camera_calibration.c (partie 1)

```c
#include "camera_calibration.h"

void calibration_un(char* image_name, matrice* P, matrice* K, matrice* R, matrice* T, double* X, double* Y, double* Z){
    char points_image_file[256];
    double* u = calloc(N, sizeof(double));
    double* v = calloc(N, sizeof(double));
    snprintf(points_image_file, sizeof(points_image_file), "points/donnees/points_calibrage_%s.txt", image_name);
    load_all_points_images(points_image_file, u, v, N);
    matrice* A = construction_A(X, Y, Z, u, v, N);
    char fn[100];
    nom_fichier(fn, "A", image_name);
    save_matrice_to_file(A, fn);
    camera_calibration_resolution(P, A, K, R, T);
    nom_fichier(fn, "P", image_name);
    save_matrice_to_file(P, fn);
    free(u);
    free(v);
    free_matrice(A);
}

int main(int argc, char* argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <nom_de_l'image1> <nom_de_l'image2> <couleurs> [select]\n", argv[0]);
        return 1;
    }
    double* X = calloc(N, sizeof(double));
    double* Y = calloc(N, sizeof(double));
    double* Z = calloc(N, sizeof(double));
    char* cl =argv[3];
    char* image_name1 =argv[1];
```

test_camera_calibration.c (partie 2)

```
    char* image_name2 =argv[2];
    char points_reel_file[256];
    char export_char[32];
    char command[128];
    if(argc>4){
        snprintf(command, sizeof(command), "python3 select_deux.py %s.jpg %s.jpg", image_name1,
↪   image_name2);
        system(command);
    }
    else{
        printf("lecture des fichiers...\n");
    }
    snprintf(points_reel_file, sizeof(points_reel_file), "points/donnees/points_reels_%s.txt", cl);
    load_all_points_reels(points_reel_file, X, Y, Z, N);
    matrice* P1 = matrice_nulle(3, 4); matrice* P2 = matrice_nulle(3, 4);
    matrice* K1 = matrice_nulle(3, 3); matrice* K2 = matrice_nulle(3, 3);
    matrice* R1 = matrice_nulle(3, 3); matrice* R2 = matrice_nulle(3, 3);
    matrice* T1 = matrice_nulle(3, 1); matrice* T2 = matrice_nulle(3, 1);
    calibration_un(image_name1, P1, K1, R1, T1, X, Y, Z);
    calibration_un(image_name2, P2, K2, R2, T2, X, Y, Z);
    matrice* F= compute_F(K1 ,R1, T1, K2, R2, T2);
    snprintf(export_char, sizeof(export_char),"F_%s", image_name1);
    save_matrice_to_file(F, export_char);
    free(X);
    free(Y);
    free(Z);
    return 0;
}
```

test_detection.c (partie 1)

```
#include "detection.h"

int main(int argc, char* argv[]) {
  char command[500];
  if (argc < 3) {
      fprintf(stderr, "Usage: %s <nom_image1> <nom_image2>\n", argv[0]);
      return 1;
  }
  char export_char[64];
  char* filename1 = argv[1];
  char* filename2 = argv[2];

  matrice* input1 = read_jpg(filename1);
  matrice* input2 = read_jpg(filename2);
  matrice* input1_r = read_jpg_color(filename1, "r");
  matrice* input1_g = read_jpg_color(filename1, "g");
  matrice* input1_b = read_jpg_color(filename1, "b");
  matrice* input2_r = read_jpg_color(filename2, "r");
  matrice* input2_g = read_jpg_color(filename2, "g");
  matrice* input2_b = read_jpg_color(filename2, "b");

  matrice *img1, *img2;
  init_img_moravec(&img1,&img2,filename1,filename2,input1,input2);

  int nbp1 = img1->n;
  int nbp2 = img2->n;
  matrice* retenus = corresp_color (img1,img2,input1_r, input1_g, input1_b, input2_r, input2_g,
↪  input2_b, nbp1,nbp2,filename1,filename2);
  snprintf(export_char, sizeof(export_char), "points_ap_%s.txt", filename1);
  int n1=save_matrice_to_file_clean(img1, export_char);
  if (n1==0||n1==1){
```

test_detection.c (partie 2)

```
    fprintf(stderr, "Erreur : pas de points détéctés.\n");
    exit(EXIT_FAILURE);
  }
  snprintf(export_char, sizeof(export_char), "points_ap_%s.txt", filename2);
  int n2=save_matrice_to_file_clean(retenus, export_char);
  assert(n1==n2);
  snprintf(command, sizeof(command),
↪   "python3 plot_points_ap.py %s.jpg %s.jpg points_ap_%s.txt points_ap_%s.txt", filename1,
↪   filename2, filename1, filename2);
  system(command);
  free_matrice(retenus);
  return 0;
}
```

test_moravec.c (partie 1)

```
#include"moravec.h"

matrice* bit_image_to_points (matrice* image, int nb_points){
    matrice* res=matrice_nulle(nb_points,2);
    int c=0;
    for (int i = 0; i < image->n; i++){
        for (int j=0; j<image->m; j++){
            if ((image->mat[i][j])==1){
                res->mat[c][0]=i;
                res->mat[c][1]=j;
                c++;
            }
        }
    }
    return res;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <nom_image>\n", argv[0]);
        return 1;
    }
    char* filename=argv[1];
    char input_name[32];
    char image_name[32];
    snprintf(input_name, sizeof(input_name), "%s.txt", filename);
    snprintf(image_name, sizeof(input_name), "%s.jpg", filename);
    if (!file_exists(input_name)){
        char command[128];
        snprintf(command, sizeof(command), "python3 jpg_to_txt.py %s.jpg", filename);
```

test_moravec.c (partie 2)

```
        system(command);
    }
    matrice* input;
    printf("%s", input_name);
    read_matrice_from_file_dimension(&input, input_name);
    matrice* output=matrice_nulle(input->n,input->m);
    printf("%d, %d\n", output->n, output->m);
    int nb_points=moravec(input, output);
    char output_name[128];
    char parametre[256];
    snprintf(parametre, sizeof(parametre), "fichier:%s, seuil:%d, fenetre:%d, param:%d", filename,
↪   Seuil_moravec, Window, PARAM);
    snprintf(output_name, sizeof(output_name), "%s-mv-%d-%d-%d.pbm", filename, Seuil_moravec, Window,
↪   PARAM);
    save_matrice_pbm(output, output_name, parametre);
    save_matrice_to_file(output, filename);
    matrice* points=bit_image_to_points(output, nb_points);
    char command[500];
    char export_char[32];
    snprintf(export_char, 32, "points_ap_%s.txt",filename);
    save_matrice_to_file(points, export_char);
    snprintf(command, sizeof(command), "python3 plot_detect_un.py %s %s",image_name, export_char);
    //system(command);
    printf("nb_points : %d",nb_points);
    free_matrice(input);
    free_matrice(output);
    return 0;
}
```

test_reconstruction_mult.c (partie 1)

```c
#include "reconstruction.h"

int main(int argc, char* argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <nom_image>*8\n", argv[0]);
        return 1;
    }
    char* image_name1 = argv[1];
    char* image_name2 = argv[2];
    char* image_name3 = argv[3];
    char* image_name4 = argv[4];
    char* image_name5 = argv[5];
    char* image_name6 = argv[6];
    char* image_name7 = argv[7];
    char* image_name8 = argv[8];

    matrice* matrice_output;
    int nb_points =reconstruction4(image_name1, image_name2,image_name3, image_name4,image_name5,
↪   image_name6,image_name7, image_name8, &matrice_output);
    assert(nb_points>0);
    char export[128];
    snprintf(export, sizeof(export), "points_3d_%s_all.txt", image_name1);
    save_matrice_to_file(matrice_output,export );
    char command[256];
    snprintf(command, sizeof(command),
↪   "python3 plot_points_3D.py points_3d_%s points_3d_%s points_3d_%s points_3d_%s", image_name1,
↪   image_name3, image_name5, image_name7);
    system(command);
    return 0;
}
```

test_triangulation.c (partie 1)

```c
#include "triangle.h"
#include "reconstruction.h"

#define NBRPOINTS 50

int main(int argc, char* argv[]) {
    if (argc < 9) {
        fprintf(stderr, "Usage: %s <nom_image>*8\n", argv[0]);
        return 1;
    }
    char* image_name1 = argv[1];
    char* image_name2 = argv[2];
    char* image_name3 = argv[3];
    char* image_name4 = argv[4];
    char* image_name5 = argv[5];
    char* image_name6 = argv[6];
    char* image_name7 = argv[7];
    char* image_name8 = argv[8];
    matrice* matrice_output;
    int nb_points =reconstruction4(image_name1, image_name2,  image_name3,  image_name4,
↪   image_name5,  image_name6,image_name7,  image_name8, &matrice_output);
    double** env = mat_to_table (matrice_output,&nb_points);
    unsigned long int n = trois_parmi(nb_points);
    printf("malloc triangle debut\n");
    fflush(stdout);
    triangle* triangle_table = triangles(nb_points);
    printf("malloc triangle fin\n");
    fflush(stdout);
    //char fn_complete[512];
    //snprintf(fn_complete, sizeof(fn_complete), "points/donnees/tri_%s.txt", filename);
    //FILE* file2 = fopen(fn_complete, "w");
```

test_triangulation.c (partie 2)

```c
    bool* garde = keeptrig(triangle_table,n,nb_points,env);
    /*for (int i = 0; i < n; i++) {
        if (garde[i]) {
            for (int j = 0; j < 3; j++) {
                double* point = env[triangle_table[i][j]];
                file_print_vect(point, file2);
            }
            fprintf(file2, "---\n");
        }
    }*/
    stl_generate("test.stl",env,triangle_table,n,garde);
    //fclose(file2);
    free(garde);
    destroy_trigs(triangle_table);
    destroy_points(env, nb_points);

    return EXIT_SUCCESS;
}
```

triangle.c (partie 1)

```c
#include"triangle.h"

// Produit scalaire pour des vecteurs de dimension 3
double scalaire(double* v1, double* v2) {
    double s = 0;
    for (int i = 0; i < 3; i++) {
        s += v1[i] * v2[i];
    }
    return s;
}
double norme(double* v) {
    return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}
// Addition de deux vecteurs dans v3
void add(double* v1, double* v2,double* v3) {
    for (int i = 0; i < 3; i++) {
        v3[i] = v1[i] + v2[i];
    }
}

// Addition in-place de deux vecteurs
void addin(double* v1, double* v2) {
    for (int i = 0; i < 3; i++) {
        v1[i] += v2[i];
    }
}

// Soustraction de deux vecteurs dans v3
void sub(double* v1, double* v2, double* v3) {
    for (int i = 0; i < 3; i++) {
```

triangle.c (partie 2)

```
        v3[i] = v1[i] - v2[i];
    }
}

// Mise à l'échelle d'un vecteur
void scale(double* v1, double s) {
    for (int i = 0; i < 3; i++) {
        v1[i] *= s;
    }
}

// Calcul du barycentre
void barycentre(triangle l, double* bary, double** points) {
    bary[0] = 0;
    bary[1] = 0;
    bary[2] = 0;
    addin(bary, points[l.a]);
    addin(bary, points[l.b]);
    addin(bary, points[l.c]);
    scale(bary, 1.0 / 3);
}

// Produit vectoriel pour des vecteurs de dimension 3
void prod(double* v1, double* v2, double* v3) {
    for (int i = 0; i < 3; i++) {
        v3[i] = v1[(i + 1) % 3] * v2[(i + 2) % 3] - v1[(i + 2) % 3] * v2[(i + 1) % 3];
    }
}

// Test si un vecteur est nul
```

triangle.c (partie 3)

```c
bool nulv(double* v) {
    for (int i = 0; i < 3; i++) {
        if (v[i] != 0) {
            return false;
        }
    }
    return true;
}

// Calcul du nombre de combinaisons de 3 parmi n
unsigned long int trois_parmi(int n) {
    return (unsigned long int) (n <= 2) ? 0 : (unsigned long int)n * (unsigned long int)(n - 1) *
↪   (unsigned long int)(n - 2) / 6;
}

// Génère toutes les combinaisons possibles de triangles
triangle* triangles(int card) {
    unsigned long int n = trois_parmi(card);
    fprintf(stdout, "binomial(%d, 3) = %lu\n", card, n); fflush(stdout);
    triangle* trigs = malloc(n * sizeof(triangle));
    unsigned long int ind = 0;
    for (unsigned long int i = 0; i < card; i++) {
        for (unsigned long int j = i + 1; j < card; j++) {
            for (unsigned long int k = j + 1; k < card; k++) {
                trigs[ind].a = i;
                trigs[ind].b = j;
                trigs[ind].c = k;
                ind++;
            }
        }
    }
```

triangle.c (partie 4)

```
    return trigs;
}

// Impression d'un vecteur dans un fichier
void file_print_vect(double* v, FILE* file) {
    fprintf(file, "%f %f %f\n", v[0], v[1], v[2]);
}

// Impression d'un vecteur sur la console
void print_vect(double* v) {
    printf("(%1.3f, %1.3f, %1.3f)\n", v[0], v[1], v[2]);
}

// Destruction des triangles
void destroy_trigs(triangle* l) {
    free(l);
}

// Destruction des points
void destroy_points(double** l, int n) {
    for (int i = 0; i < n; i++) {
        free(l[i]);
    }
    free(l);
}

bool* keeptrig(triangle* l, unsigned long int ntrig, int size, double** point) {
    printf("malloc bool debut\n");
    fflush(stdout);
    bool* res = malloc(ntrig * sizeof(bool));
```

triangle.c (partie 5)

```
    printf("malloc bool fin\n");
    fflush(stdout);

    double v1[3];
    double v2[3];
    double v[3];
    double n[3];
    double bary[3];

    for (unsigned long int i = 0; i < ntrig; i++) {
        sub(point[l[i].b], point[l[i].a],v1);
        sub(point[l[i].c], point[l[i].a],v2);

        prod(v1, v2,n);

        barycentre(l[i], bary, point);
        res[i] = true;

        int signe = 0;
        for (int j = 0; j < size; j++) {
            sub(bary, point[j], v);
            double s = scalaire(n, v);

            if (fabs(s) < 0.01) {
                s = 0;
            }
            if (s != 0) {
                if (signe == 0) {
                    signe = (s > 0) ? 1 : -1;
                } else if (signe * s < 0) {
```

triangle.c (partie 6)

```
                    res[i] = false;
                    break;
                }
            }
        }
    }
    fflush(stdout);

    return res;
}


// Lecture des points depuis un fichier
double** read_points(char* filename, int count) {
    char complete_fn[256];
    snprintf(complete_fn, sizeof(complete_fn), "points/donnees/%s.txt", filename);
    FILE* file = fopen(complete_fn, "r");
    if (!file) {
        perror("Erreur d'ouverture du fichier de points");
        exit(EXIT_FAILURE);
    }
    double** points = malloc(count * sizeof(double*));
    if (!points) {
        perror("Erreur d'allocation pour les points");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < count; i++) {
        points[i] = malloc(3 * sizeof(double));
        if (!points[i]) {
            perror("Erreur d'allocation pour un point");
```

triangle.c (partie 7)

```
            exit(EXIT_FAILURE);
        }
        fscanf(file, "%lf %lf %lf", &points[i][0], &points[i][1], &points[i][2]);
    }
    fclose(file);
    return points;
}

double** rand_points(int n){
  double ** res = malloc(n*sizeof(double*));
  for(int i = 0; i<n; i++){
    res[i]=malloc(sizeof(double)*3);
    for(int j = 0; j< 3; j++){
      res[i][j] = rand()%1000*0.001;
    }
  }
  return res;
}

void stl_generate(char* filename, double** point, triangle* l, unsigned long int ntrig,bool *garde){
    char complete_fn[256];
    snprintf(complete_fn, 256, "stltest/%s", filename);

    fflush(stdout);
    FILE* file = fopen(complete_fn, "w");
    assert(file != NULL);

    fflush(stdout);
    fprintf(file, "solid \n");
    double v1[3];
```

triangle.c (partie 8)

```c
    double v2[3];
    double n[3];
    int count=0;
    for (unsigned long int i = 0; i < ntrig; i++) {
      if (garde[i]){
        count++;
        sub(point[l[i].b], point[l[i].a],v1);
        sub(point[l[i].c], point[l[i].a],v2);

        prod(v1, v2,n);
        fprintf(file, "    facet normal %lf %lf %lf\n        outer loop\n",n[0],n[1],n[2]);
            fprintf(file, "        vertex %lf %lf %lf\n",
↪   point[l[i].a][0],point[l[i].a][1],point[l[i].a][2]);
            fprintf(file, "        vertex %lf %lf %lf\n",
↪   point[l[i].b][0],point[l[i].b][1],point[l[i].b][2]);
            fprintf(file, "        vertex %lf %lf %lf\n",
↪   point[l[i].c][0],point[l[i].c][1],point[l[i].c][2]);
        fprintf(file, "        endloop\n    endfacet\n");
      }
    }
    printf("nb triangles : %d\n", count);
    fprintf(file, "endsolid \n");
    fclose(file);
    printf("stl generated : %s\n", filename);
    fflush(stdout);
}

double** mat_to_table (matrice* mat,int* n){
  *n = mat->n;
  double** space = malloc(sizeof(double*)*(*n));
  for(int i = 0; i<*n;i++){
    space[i]=malloc(sizeof(double)*3);
    for(int j = 0; i<3;i++){
```

triangle.c (partie 9)

```
      space[i][j]=mat->mat[i][j];
    }
  }
  return space;
}
```

trouve_coin.c (partie 1)

```
#include "trouve_coin.h"

void update_score(matrice* plan, matrice* score, int i, int j,int k, int l){
        double dy = l - j;
        if (k-i==0){
    int y=j;
    int z=l;
    if(j>l){
      y=l;
      z=j;
    }
    for(y=y+1;y<=z;y++){
      score->mat[i][y]++;
    }
        }
        else {
    dy=dy/((double)k-i);
    int x = i;
    double y = j;
    int u = k;
    if(k-i<0){
      x=l;
      y = l;
      u=i;
      dy=dy;
    }
    x+=1;
    y+=1*dy;
    while(x<=u){
      score->mat[x][(int)y]++;
```

trouve_coin.c (partie 2)

```
      score->mat[x][(int)y+1]++;
      x+=1;
      y+=dy;
    }
  }
}

bool au_bord(matrice* plan, int i, int j){
  return (i<20||i>plan->n-10||j<20||j>plan->m-20);

}

int distance(int i,int j,int k , int l){
  return (i-k)*(i-k) + (j-l)*(j-l);
}

matrice* compute_score(matrice* plan, int** actif, int size){ //size est la taille d'actif
                                                              // actif est un tableau de couple d'entier
↪  correspondant au support de plan
  matrice* score = matrice_nulle(plan->n,plan->m);
  for(int i = 0; i < size; i++){
    for(int j = i+1; j < size; j++){
      if(distance(actif[i][0],actif[i][1],actif[j][0],actif[j][1])<Dist_tc*Dist_tc &&
↪  !au_bord(plan,actif[i][0],actif[i][1]) && !au_bord(plan,actif[j][0],actif[j][1])){ //On ne traite
↪  pas de points trop au bord
        update_score(plan,score,actif[i][0],actif[i][1],actif[j][0],actif[j][1]);
      }
    }
  }
  return score;
}

int filtre_mat(matrice* input, int** actif, int size){
```

trouve_coin.c (partie 3)

```
  int s=0;
  matrice* tmp = matrice_nulle(input->n,input->m);
  for(int i = 0;i<size;i++){
    if(!au_bord(input,actif[i][0],actif[i][1])){
      tmp->mat[actif[i][0]][actif[i][1]]=1;
    }
  }
  for(int i = 0;i<input->n;i++){
    for(int j = 0;j<input->m;j++){
                    if(tmp->mat[i][j]==1 && input->mat[i][j]<Seuil_tc){
      s++;
                            input->mat[i][j]=1;
                    }
                    else {
                            input->mat[i][j]=0;
                    }
            }
      }
      free_matrice(tmp);
  return s;
}

int pretty_mat(matrice* input, int** actif, int size){
  int s=0;
  matrice* tmp = matrice_nulle(input->n,input->m);
  for(int i = 0;i<size;i++){
    if(!au_bord(input,actif[i][0],actif[i][1])){
      tmp->mat[actif[i][0]][actif[i][1]]=1;
    }
  }
```

trouve_coin.c (partie 4)

```
for(int i = 0;i<input->n;i++){
  for(int j = 0;j<input->m;j++){
                    if(tmp->mat[i][j]==1 && input->mat[i][j]<Seuil_tc){
      s++;
      tmp->mat[i][j]=1;
                    }
                    else {
      tmp->mat[i][j]=0;
                    }
            }
      }
for(int i = 0;i<input->n;i++){
  for(int j = 0;j<input->m;j++){
    if(tmp->mat[i][j]==1){
      for(int k = -5;k<5;k++){
        for(int l= -5;l<5;l++){
          if(!au_bord(input,i+k,j+l)){
          input->mat[i+k][j+l]=1;
          tmp->mat[i+k][j+l]=2;
          }
        }
      }
    }
    else if (tmp->mat[i][j]==0){
      input->mat[i][j]=0;
    }
  }
}
      free_matrice(tmp);
return s;
```

trouve_coin.c (partie 5)

```
}
```

Makefile (partie 1)

```
WFLAGS := -Wall
CFLAGS := -std=c99

all : test_triangulation test_moravec test_trouve_coin test_detection test_camera_calibration
↪  test_reconstruction_mult test_reconstruction test_SVD

test_camera_calibration : constante.o manipulation_fichier.o matrice.o SVD.o camera_calibration.o
↪  test_camera_calibration.c
        gcc $(CFLAGS) $(WFLAGS) -g constante.o manipulation_fichier.o matrice.o SVD.o
↪  camera_calibration.o  test_camera_calibration.c -lm -o test_camera_calibration

test_moravec : constante.o matrice.o manipulation_fichier.o moravec.o test_moravec.c moravec.h
        gcc $(CFLAGS) $(WFLAGS) -g constante.o manipulation_fichier.o matrice.o moravec.o
↪  test_moravec.c -lm -o test_moravec

test_detection : constante.o ransac.o detection.o manipulation_fichier.o matrice.o moravec.o
↪  appariement.o trouve_coin.o  SVD.o camera_calibration.o test_detection.c
        gcc $(CFLAGS) $(WFLAGS) -g constante.o ransac.o detection.o manipulation_fichier.o matrice.o
↪  moravec.o appariement.o trouve_coin.o  SVD.o camera_calibration.o test_detection.c -lm -o
↪  test_detection

test_triangulation : constante.o triangle.o manipulation_fichier.o matrice.o SVD.o
↪  camera_calibration.o reconstruction.o triangle.h test_triangulation.c
        gcc $(CFLAGS) $(WFLAGS) -g constante.o triangle.o manipulation_fichier.o matrice.o
↪  reconstruction.o SVD.o camera_calibration.o  test_triangulation.c -lm -o test_triangulation

test_reconstruction_mult : constante.o manipulation_fichier.o matrice.o reconstruction.o SVD.o
↪  camera_calibration.o test_reconstruction_mult.c
        gcc $(CFLAGS) $(WFLAGS) -g constante.o manipulation_fichier.o matrice.o reconstruction.o
↪  SVD.o camera_calibration.o  test_reconstruction_mult.c -lm -o test_reconstruction_mult

ransac.o : ransac.c ransac.h
```

Makefile (partie 2)

```
triangle.o : triangle.h triangle.c
	gcc $(CFLAGS) $(WFLAGS) -g -c triangle.c -lm -o triangle.o

manipulation_fichier.o : manipulation_fichier.h manipulation_fichier.c
	gcc $(CFLAGS) $(WFLAGS) -g -c manipulation_fichier.c -lm -o manipulation_fichier.o

matrice.o : matrice.c matrice.h
	gcc $(CFLAGS) $(WFLAGS) -g -c matrice.c -lm -o matrice.o

SVD.o : SVD.c SVD.h
	gcc $(CFLAGS) $(WFLAGS) -g -c SVD.c -lm -o SVD.o

trouve_coin.o : trouve_coin.c trouve_coin.h
	gcc $(CFLAGS) $(WFLAGS) -g -c trouve_coin.c -lm -o trouve_coin.o

detection.o : detection.c detection.h
	gcc $(CFLAGS) $(WFLAGS) -g -c detection.c -lm -o detection.o

constante.o : constante.c constante.h
	gcc $(CFLAGS) $(WFLAGS) -g -c constante.c -lm -o constante.o

clean :
	rm -f *.o
	rm -f *.txt
	rm -f test_detection
	rm -f test_moravec
	rm -f test_camera_calibration
	rm -f test_reconstruction_mult
	rm -f test_triangulation
```

jpg_to_txt_color.py (partie 1)

```
import cv2
import numpy as np
import sys
import os

if len(sys.argv) != 2:
    print("Usage: python script.py <image>.jpg")
    exit(1)

image_path = sys.argv[1]
input_dir = "points/images/"
output_dir = "points/donnees/"
input_path = os.path.join(input_dir, image_path)

image_rgb = cv2.imread(input_path, cv2.IMREAD_COLOR)
image = cv2.cvtColor(image_rgb, cv2.COLOR_BGR2LAB)

if image is None:
    print(f"Error: Image '{input_path}' not found or cannot be read!")
    exit(1)

height, width = image.shape[:2]
image_name = os.path.splitext(os.path.basename(image_path))[0]
os.makedirs(output_dir, exist_ok=True)
output_path = os.path.join(output_dir, f"{image_name}_r.txt")

with open(output_path, 'w') as f:
    f.write(f"{width} {height}\n")
    for row in image:
        row_data = [f"{pixel[2]}" for pixel in row]
```

jpg_to_txt_color.py (partie 2)

```
        f.write(' '.join(row_data) + '\n')

print(f"Image converted successfully to '{output_path}'!")

height, width = image.shape[:2]
image_name = os.path.splitext(os.path.basename(image_path))[0]
os.makedirs(output_dir, exist_ok=True)
output_path = os.path.join(output_dir, f"{image_name}_g.txt")

with open(output_path, 'w') as f:
    f.write(f"{width} {height}\n")
    for row in image:
        row_data = [f"{pixel[1]}" for pixel in row]
        f.write(' '.join(row_data) + '\n')

print(f"Image converted successfully to '{output_path}'!")

height, width = image.shape[:2]
image_name = os.path.splitext(os.path.basename(image_path))[0]
os.makedirs(output_dir, exist_ok=True)
output_path = os.path.join(output_dir, f"{image_name}_b.txt")

with open(output_path, 'w') as f:
    f.write(f"{width} {height}\n")
    for row in image:
        row_data = [f"{pixel[0]}" for pixel in row]
        f.write(' '.join(row_data) + '\n')

print(f"Image converted successfully to '{output_path}'!")
```

jpg_to_txt.py (partie 1)

```
import cv2
import numpy as np
import sys
import os

if len(sys.argv) != 2:
    print("Usage: python script.py <image>.jpg")
    exit(1)

image_path = sys.argv[1]
input_dir = "points/images/"
output_dir = "points/donnees/"
input_path = os.path.join(input_dir, image_path)


image = cv2.imread(input_path, cv2.IMREAD_GRAYSCALE)


if image is None:
    print(f"Error: Image '{input_path}' not found or cannot be read!")
    exit(1)

height, width = image.shape

image_name = os.path.splitext(os.path.basename(image_path))[0]

os.makedirs(output_dir, exist_ok=True)
output_path = os.path.join(output_dir, f"{image_name}.txt")

with open(output_path, 'w') as f:
```

jpg_to_txt.py (partie 2)

```
    f.write(f"{width} {height}\n")  # Écrire les dimensions sur la première ligne
    for row in image:
        row_str = ' '.join(map(str, row))  # Convertir chaque pixel en string
        f.write(row_str + '\n')  # Écrire la ligne dans le fichier

print(f"Image converted successfully to '{output_path}'!")
```

plot_detect.py (partie 1)

```
import matplotlib.pyplot as plt
import numpy as np
import sys


def load_points(filename):
    """
    Charge les coordonnées des points depuis un fichier texte.
    Le fichier doit contenir les coordonnées sous la forme : x y
    """
    points = np.loadtxt(filename)
    return points


def plot_images_with_points(image1, image2, points1, points2):
    """
    Affiche les deux images avec les points respectivement à leurs coordonnées données.
    """
    fig, ax = plt.subplots(1, 2, figsize=(12, 6))

    # Génération des couleurs aléatoires pour chaque point ou chaque correspondance
    colors = np.random.rand(len(points1), 3)

    # Affichage de l'image 1
    ax[0].imshow(image1, cmap='gray')

    # Affichage de l'image 2
    ax[1].imshow(image2, cmap='gray')

    for i in range(1,(min(len(points1), len(points2)))):
```

plot_detect.py (partie 2)

```
        color = np.random.rand(3,)
        ax[0].scatter(points1[i, 0], points1[i, 1], color=color, label='Points Image 1' if i == 0
↪   else "")
        ax[1].scatter(points2[i, 0], points2[i, 1], color=color, label='Points Image 2' if i == 0
↪   else "")


    ax[0].set_title('Image 1')
    ax[1].set_title('Image 2')
    plt.show()


def main(image1_name, image2_name, points1_name, points2_name):
    """
    Fonction principale pour charger les images et les points, puis les afficher.
    """
    # Définition des chemins des fichiers
    image1_path = f"points/images/{image1_name}"
    image2_path = f"points/images/{image2_name}"
    points1_path = f"points/donnees/{points1_name}"
    points2_path = f"points/donnees/{points2_name}"

    # Chargement des images
    image1 = plt.imread(image1_path)
    image2 = plt.imread(image2_path)

    # Chargement des coordonnées des points
    points1 = load_points(points1_path)
    points2 = load_points(points2_path)

    # Affichage des images avec les points
    plot_images_with_points(image1, image2, points1, points2)
```

plot_detect.py (partie 3)

```
if __name__ == '__main__':
    if len(sys.argv) != 5:
        print("Usage: python script.py <image1_name> <image2_name> <points1_name> <points2_name> ")
        sys.exit(1)

    image1_name = sys.argv[1]
    image2_name = sys.argv[2]
    points1_name = sys.argv[3]
    points2_name = sys.argv[4]

    main(image1_name, image2_name, points1_name, points2_name)
```

plot_detect_un.py (partie 1)

```
import matplotlib.pyplot as plt
import numpy as np
import sys


def load_points(filename):
    """
    Charge les coordonnées des points depuis un fichier texte.
    Le fichier doit contenir les coordonnées sous la forme : x y
    """
    points = np.loadtxt(filename)
    return points


def plot_image_with_points(image, points):
    """
    Affiche une image avec des points superposés.
    """
    fig, ax = plt.subplots(figsize=(8, 6))

    # Génération des couleurs aléatoires pour chaque point
    colors = np.random.rand(len(points), 3)

    ax.imshow(image, cmap='gray')

    for i in range(1, (len(points))):
        ax.scatter(points[i, 0], points[i, 1], color=colors[i], label='Points' if i == 0 else "")

    plt.show()
```

plot_detect_un.py (partie 2)

```python
def main(image_name, points_name):
    """
    Fonction principale pour charger l'image et les points, puis les afficher.
    """
    # Définition des chemins des fichiers
    image_path = f"points/images/{image_name}"
    points_path = f"points/donnees/{points_name}"

    # Chargement de l'image
    image = plt.imread(image_path)

    # Chargement des coordonnées des points
    points = load_points(points_path)

    # Affichage de l'image avec les points
    plot_image_with_points(image, points)


if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("Usage: python script.py <image_name> <points_name>")
        sys.exit(1)

    image_name = sys.argv[1]
    points_name = sys.argv[2]

    main(image_name, points_name)
```

plot_points_3D.py (partie 1)

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import sys

def lire_et_tracer_points(fichiers):
    try:
        # Initialiser la figure 3D
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

        # Couleurs pour différencier les fichiers
        couleurs = ['b', 'g', 'r', 'c', 'm', 'y', 'k']

        # Initialiser les bornes pour garder un repère orthonormé
        x_min, x_max = float('inf'), float('-inf')
        y_min, y_max = float('inf'), float('-inf')
        z_min, z_max = float('inf'), float('-inf')

        for index, fichier in enumerate(fichiers):
            x_coords = []
            y_coords = []
            z_coords = []
            complete_name = f'points/donnees/{fichier}.txt'
            with open(complete_name, 'r') as f:
                for ligne in f:
                    x, y, z = map(float, ligne.strip().split())
                    x_coords.append(x)
                    y_coords.append(y)
                    z_coords.append(z)
```

plot_points_3D.py (partie 2)

```
        # Mettre à jour les bornes
        x_min, x_max = min(x_min, min(x_coords)), max(x_max, max(x_coords))
        y_min, y_max = min(y_min, min(y_coords)), max(y_max, max(y_coords))
        z_min, z_max = min(z_min, min(z_coords)), max(z_max, max(z_coords))

        # Choisir une couleur pour ce fichier
        couleur = couleurs[index % len(couleurs)]
        ax.scatter(x_coords, y_coords, z_coords, c=couleur, marker='o', label=f"Fichier {fichier}")

        # Ajouter des étiquettes pour chaque point
        for i, (x, y, z) in enumerate(zip(x_coords, y_coords, z_coords)):
            ax.text(x, y, z, f"{i}", color=couleur)

    # Définir les mêmes échelles pour chaque axe
    max_range = max(x_max - x_min, y_max - y_min, z_max - z_min) / 2.0
    mid_x = (x_max + x_min) / 2.0
    mid_y = (y_max + y_min) / 2.0
    mid_z = (z_max + z_min) / 2.0
    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)

    # Ajouter des étiquettes des axes
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.title('Points 3D')
    plt.legend()
    plt.show()
```

plot_points_3D.py (partie 3)

```
    except FileNotFoundError as e:
        print(f"Erreur : Le fichier est introuvable : {e}")
    except ValueError as e:
        print(f"Erreur de format dans un des fichiers : {e}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage : plot_points.py <nom_du_fichier1> <nom_du_fichier2> ...")
    else:
        fichiers = sys.argv[1:]
        lire_et_tracer_points(fichiers)
```

plot_points_ap.py (partie 1)

```
import matplotlib.pyplot as plt
import numpy as np
import sys

def load_points(filename):
    """
    Charge les coordonnées des points depuis un fichier texte.
    Le fichier doit contenir les coordonnées sous la forme : x y
    """
    points = np.loadtxt(filename)
    return points

def filter_valid_points(points1, points2):
    """
    Filtre les paires de points où ni les coordonnées de points1 ni de points2 ne contiennent -1.
    """
    mask = np.all(points1 != -1, axis=1) & np.all(points2 != -1, axis=1)
    return points1[mask], points2[mask]

def plot_images_with_points(image1, image2, points1, points2):
    """
    Affiche les deux images concaténées avec les points respectivement à leurs coordonnées données.
    Affiche aussi les numéros des points et trace une ligne entre chaque i-ème point des deux images.
    """
    combined_image = np.hstack((image1, image2))

    fig, ax = plt.subplots(figsize=(12, 6))
    ax.imshow(combined_image, cmap='gray')

    points1, points2 = filter_valid_points(points1, points2)
```

plot_points_ap.py (partie 2)

```
    # Génération des couleurs aléatoires pour chaque correspondance
    colors = np.random.rand(len(points1), 3)

    for i in range(len(points1)):
        color = colors[i]
        # Point image 1
        ax.scatter(points1[i, 0], points1[i, 1], color=color)
        ax.text(points1[i, 0] + 3, points1[i, 1] - 3, str(i), color=color, fontsize=11)

        # Point image 2
        ax.scatter(points2[i, 0] + image1.shape[1], points2[i, 1], color=color)
        ax.text(points2[i, 0] + image1.shape[1] + 3, points2[i, 1] - 3, str(i), color=color,
↪   fontsize=11)

        # Ligne entre les points
        ax.plot([points1[i, 0], points2[i, 0] + image1.shape[1]],
                [points1[i, 1], points2[i, 1]], color=color, linestyle='-', linewidth=1)

    ax.set_title('Points appariés valides')
    plt.axis('off')
    plt.show()

def main(image1_name, image2_name, points1_name, points2_name):
    """
    Fonction principale pour charger les images et les points, puis les afficher.
    """
    image1_path = f"points/images/{image1_name}"
    image2_path = f"points/images/{image2_name}"
    points1_path = f"points/donnees/{points1_name}"
    points2_path = f"points/donnees/{points2_name}"
```

plot_points_ap.py (partie 3)

```
    image1 = plt.imread(image1_path)
    image2 = plt.imread(image2_path)

    points1 = load_points(points1_path)
    points2 = load_points(points2_path)

    plot_images_with_points(image1, image2, points1, points2)

if __name__ == '__main__':
    if len(sys.argv) != 5:

↪   print("Usage: python plot_points_ap.py <image1_name> <image2_name> <points1_name> <points2_name>")
        sys.exit(1)

    image1_name = sys.argv[1]
    image2_name = sys.argv[2]
    points1_name = sys.argv[3]
    points2_name = sys.argv[4]

    main(image1_name, image2_name, points1_name, points2_name)
```

select_deux.py (partie 1)

```
import cv2 as cv
import os
import numpy as np
import argparse

# Listes pour stocker les points correspondants
points_img1 = []
points_img2 = []

# Fonction pour redimensionner l'image en conservant le ratio et avec des dimensions maximales
def resize_image(image, max_height=600, max_width=3000):
    height, width = image.shape[:2]
    height_ratio = max_height / height
    width_ratio = max_width / width
    resize_ratio = min(height_ratio, width_ratio, 1.0)  # Assure que l'image ne sera pas agrandie
    new_width = int(width * resize_ratio)
    new_height = int(height * resize_ratio)
    return cv.resize(image, (new_width, new_height), interpolation=cv.INTER_AREA)

# Sauvegarder les points dans un fichier
def save_points(filename, points):
    with open(filename, 'w') as f:
        f.write(f"{len(points)} 2\n")  # Écrit le nombre de points en première ligne
        np.savetxt(f, points, fmt='%d')

# Sélection des points pour la première image
def select_points_img1(event, x, y, flags, param):
   if event == cv.EVENT_LBUTTONDOWN:
        print("Point cliqué :")
        print(f"Original: x={x}, y={y}")
```

select_deux.py (partie 2)

```
        points_img1.append([x, y])
        cv.circle(param, (x, y), 5, (0, 255, 0), -1)
        cv.imshow('Image 1', param)

# Sélection des points pour la première image
def select_points_img2(event, x, y, flags, param):
   if event == cv.EVENT_LBUTTONDOWN:
        print("Point cliqué :")
        print(f"Original: x={x}, y={y}")
        points_img2.append([x, y])
        cv.circle(param, (x, y), 5, (0, 255, 0), -1)
        cv.imshow('Image 2', param)

# Fonction principale pour charger les images et sélectionner les points
def process_images(image_name1, image_name2):
    # Charger les images
    image_path1 = f'./points/images/{image_name1}'
    image_path2 = f'./points/images/{image_name2}'
    original_img1 = cv.imread(image_path1)
    original_img2 = cv.imread(image_path2)

    # Vérification des images chargées
    if original_img1 is None or original_img2 is None:
        print("Erreur : Une ou plusieurs images n'ont pas pu être chargées.")
        return

    img1 = resize_image(original_img1)
    img2 = resize_image(original_img2)

    # Afficher les images et configurer les callbacks
```

select_deux.py (partie 3)

```python
    cv.imshow('Image 1', img1)
    cv.imshow('Image 2', img2)
    cv.setMouseCallback('Image 1', select_points_img1, param=img1)
    cv.setMouseCallback('Image 2', select_points_img2, param=img2)

    print("Sélectionnez les points pour les images. Appuyez sur une touche pour continuer.")
    cv.waitKey(0)

    # Sauvegarder les points sélectionnés
    base_name1 = os.path.splitext(image_name1)[0]
    base_name2 = os.path.splitext(image_name2)[0]
    save_points(f'points/donnees/points_{base_name1}.txt', points_img1)
    save_points(f'points/donnees/points_{base_name2}.txt', points_img2)
    print(f"Points sauvegardés dans points_{base_name1} et points_{base_name2}")

# Point d'entrée du programme
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Sélectionner des points sur deux images.")
    parser.add_argument("image_name1", type=str, help="Nom de la première image.")
    parser.add_argument("image_name2", type=str, help="Nom de la deuxième image.")
    args = parser.parse_args()

    print("- Cliquez pour sélectionner un point.")
    print("- Appuyez sur une touche pour terminer la sélection.")
    process_images(args.image_name1, args.image_name2)
```

select_un.py (partie 1)

```
import cv2 as cv
import os
import numpy as np
import argparse

# Liste pour stocker les points
points = []

# Redimensionne l'image sans dépasser les dimensions maximales
def resize_image(image, max_height=600, max_width=3000):
    height, width = image.shape[:2]
    height_ratio = max_height / height
    width_ratio = max_width / width
    resize_ratio = min(height_ratio, width_ratio, 1.0)
    new_width = int(width * resize_ratio)
    new_height = int(height * resize_ratio)
    return cv.resize(image, (new_width, new_height), interpolation=cv.INTER_AREA)

# Sauvegarde les points dans un fichier texte
def save_points(filename, points):
    with open(filename, 'w') as f:
        #f.write(f"{len(points)} 2\n")
        np.savetxt(f, points, fmt='%d')

# Callback de sélection de points
def select_points(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        print(f"Point cliqué : x={x}, y={y}")
        points.append([x, y])
        cv.circle(param, (x, y), 5, (0, 255, 0), -1)
```

select_un.py (partie 2)

```
        cv.imshow('Image', param)

# Fonction principale
def process_image(image_name):
    image_path = f'./points/images/{image_name}'
    original_img = cv.imread(image_path)

    if original_img is None:
        print("Erreur : l'image n'a pas pu être chargée.")
        return

    img = resize_image(original_img)

    cv.imshow('Image', img)
    cv.setMouseCallback('Image', select_points, param=img)

    print("Cliquez pour sélectionner les points. Appuyez sur une touche pour terminer.")
    cv.waitKey(0)

    base_name = os.path.splitext(image_name)[0]
    save_points(f'points/donnees/points_calibrage_{base_name}.txt', points)
    print(f"Points sauvegardés dans points_calibrage_{base_name}.txt")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Sélectionner des points sur une seule image.")
    parser.add_argument("image_name", type=str, help="Nom de l'image.")
    args = parser.parse_args()

    process_image(args.image_name)
```