Changhong XU – Janvier 2019

# Note d’application

Projet GE5 : Évaluation d’une méthode de planification de chemins par réseaux de neurones

## Traitement de l’image

## Introduction

Ce document a dédié à interpréter en détail une partie de notre travail de ce projet et des difficultés rencontrées pendant la phase de développement.

Le traitement de l’image est une partie essentielle de ce projet. Le résultat a des influences directes sur la trajectoire calculée, par exemple, l’échec de détection des obstacles va transmettre plus tard comme une zone accessible pour le programme de planification de chemins. En puis le risque de collision, ça vient du fait que le programme A* n’a pas pris en compte la taille du robot lors du calcul de chemin. Pour régler ces problématiques, on a employé des différences techniques. La détection d’obstacles a réalisé avec la méthode de soustraction de l’image. Cette méthode est plus fiable que la détection d’objet à partir de la couleur dans l’espace couleur d’HSV, puisque ce dernier est très sensible à l’environnement de la manipulation, et le type d’obstacle est limité.

Changhong XU – Janvier 2019

# Détection d'obstacles

## 1. Capture de l'image

La première étape consiste à photographier l'image à l'aide d'une webcam. Un programme de base a fourni dans le répertoire /tests/Capture_image_from_webcam.

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0); // open the default camera
    if(!cap.isOpened())  // check if we succeeded
        return -1;

    namedWindow("frame",1);
    for(;;)
    {
        Mat frame;
        cap >> frame; // get a new frame from camera
        imwrite("frame.png", frame);
        imshow("frame", frame);
        if(waitKey(30) >= 0) break;
    }
    // the camera will be deinitialized automatically in VideoCapture destructor
    return 0;
}
```

Figure 1 : programme capturer l'image à partir d'une webcam

Pour ouvrir la webcam, il est suffisant de définir un constructeur `cv::VideoCapture::VideoCapture ( int index )`. Par défaut, on met 0 (une seule webcam a utilisé) pour l'identité de camera. Pour vérifier l'ouverture de la webcam, utilisez la méthode **`isOpened()`**. `virtual bool cv::VideoCapture::isOpened ( ) const`
La boucle infinie permet de capturer le flux de vidéo consécutivement.

```
bool cv::imwrite ( const String &                filename,
                   InputArray                    img,
                   const std::vector< int > &    params = std::vector< int >()
                 )
```

L'enregistrement de l'image s'est fait en utilisant la fonction `imwrite()`.

**C++:** bool **imwrite**(const String& **filename**, InputArray **img**, const vector<int>& **params**=vector<int>() )

Entrez le nom de fichier, la variable de la matrice d'image comme figure 1. À l'aide de cet exemple, vous pouvez au moins construire un programme par exemple, qui permet de prendre une photo lors d'une touche du clavier.

## 2. Soustraction de l'images

Le principe de cette méthode est simple. Comparer pixels par pixels entre les deux images. S'il y a une différence, on met dans une image binaire un pixel blanc(255), sinon en noir(0). Ce programme a fourni dans le répertoire /tests/detection/

```
cv::Mat backgroundImage = imread("../data/image_no.png");
cv::Mat currentImage = imread("../data/image_o.png");          //Load images
cv::Mat diffImage;
cv::absdiff(backgroundImage, currentImage, diffImage);         //Absolute difference between two images
cv::Mat foregroundMask = cv::Mat::zeros(diffImage.rows, diffImage.cols, CV_8UC1);

float threshold = 30.0f;
float dist;

for(int j=0; j<diffImage.rows; ++j)                            //For all rows
    for(int i=0; i<diffImage.cols; ++i)                        //For all cols
    {
        cv::Vec3b pix = diffImage.at<cv::Vec3b>(j,i);

        dist = (pix[0]*pix[0] + pix[1]*pix[1] + pix[2]*pix[2]);
        dist = sqrt(dist);                                     //Calcul distinction

        if(dist>threshold)                                     //If there's difference
        {
            foregroundMask.at<unsigned char>(j,i) = 255;       //Put blanc pixel
        }
    }
```

Figure 2 : Programme soustraction d'image

**C++:** Mat **imread**(const String& **filename**, int **flags**=IMREAD_COLOR ) La fonction imread() permets de charger une image à partir d'un fichier. Ici, on charge deux images, une image qui contient seulement la zone et le robot, et une autre qui couvre la zone, le robot et des obstacles. On définie une matrice diffImage, et puis calculer la différence absolue entre ces deux images grâce à une méthode absdiff(src,src,des). absdiff() prend deux images en entrée, et à la sortie qu'on récupère la différence. Avec deux boucles for, on peut balayer par ligne et par colonne des pixels d'une image. Alors, pour bien faire la distinction entre les pixels, on doit obliger de calculer la valeur distinction des couleurs pour tous les pixels,

RGB dont 3 canal de couleur. Si la distinction est supérieure à 30, c’est-à-dire ce pixel est complètement différent de d’un et d’autre.

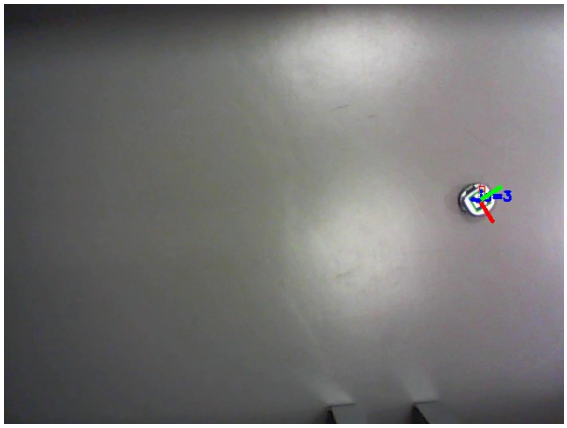


Figure 3 : photo sans obstacles

Figure 4 : photo avec obstacles

Sans opération post-processing, on obtient une différence comme suite :

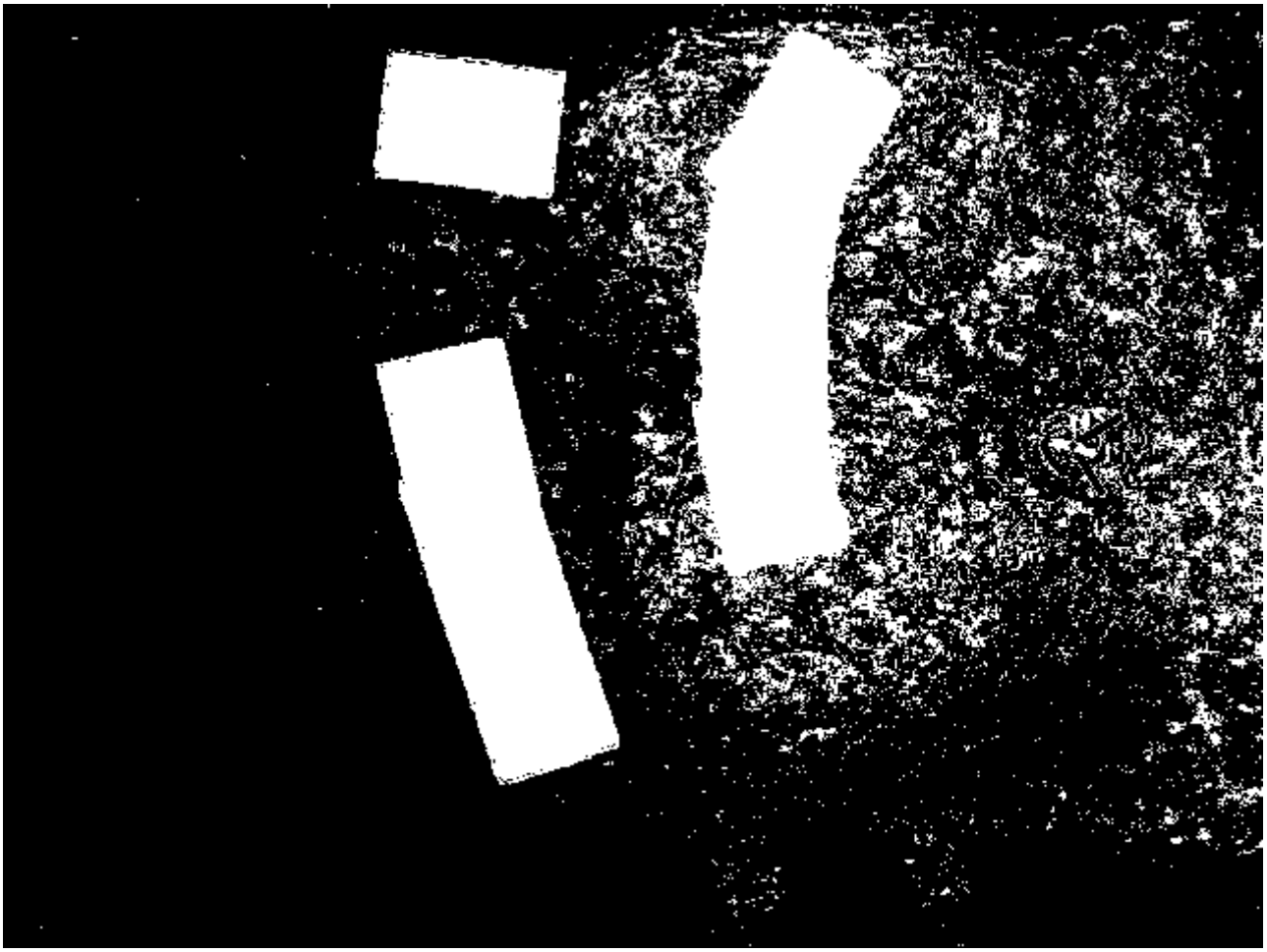


Figure 5 : Photo « soustraction »

On peut constater que l'image de sortie contient des bruits. Une fois la soustraction d'image est finie, on passe à l'étape post-processing.

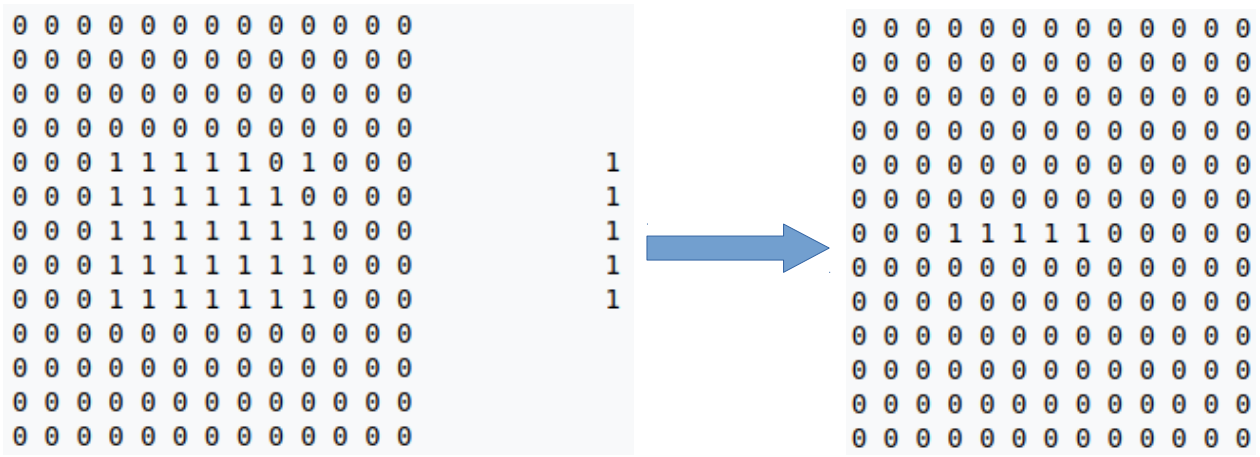
## 3. Post-processing

La méthode emploie ici est la transformation morphologie. La morphologie mathématique est une théorie et technique mathématique et informatique d'analyse de structure qui est liée avec l'algèbre, la théorie des treillis, la topologie et les probabilités.

Le développement de la morphologie mathématique est inspiré des problèmes de traitement d'images, domaine qui constitue son principal champ d'application. Elle fournit en particulier des outils de filtrage, segmentation, quantification en modélisation d'images. Elle est également utilisable en traitement du signal, par exemple pour filtrer les variations d'une mesure(physique, biologie) au cours du temps.

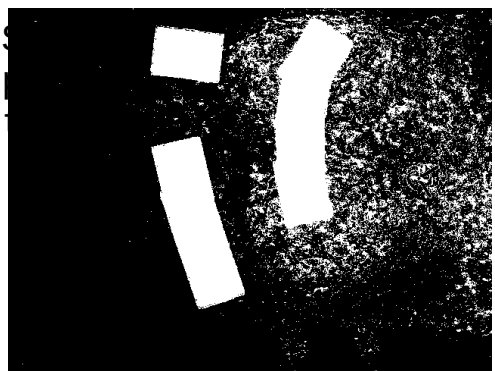
L'érosion est une opération fondamentale du traitement d'image morphologique.

Un exemple d'application :




L'érosion de A par B

S t les pixels
e opération d
cs.



Érosion

Au contraire, une opération mathématique qui s'appelle dilatation permet d'agrandir les objets blancs de l'image. Le résultat dépend la forme de l'élément structuré, et l'itération d'opération.

Par exemple, on prend l'image précédente et s'applique des opérations dilatation :
Avec élément structuré pour érosion : 8*8
pour dilatation : 3*3

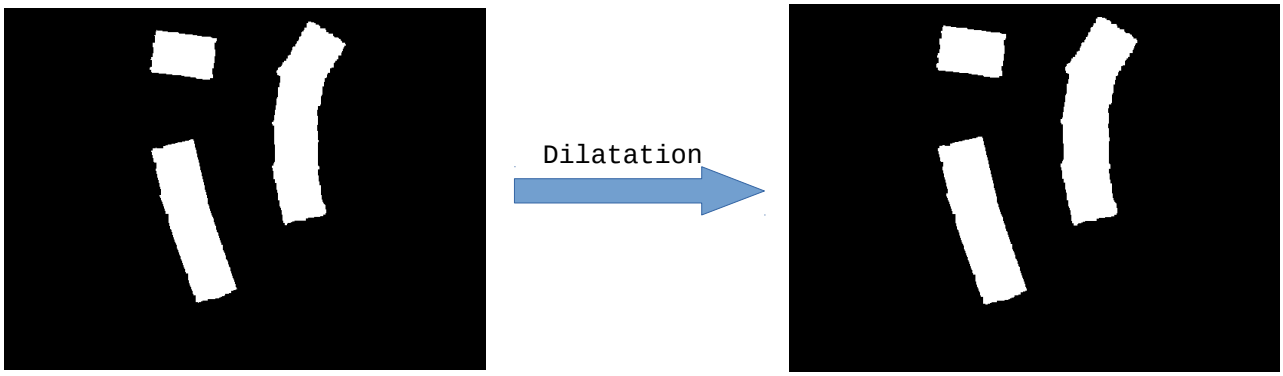



Image suivi érosion

image suivi érosion – dilatation

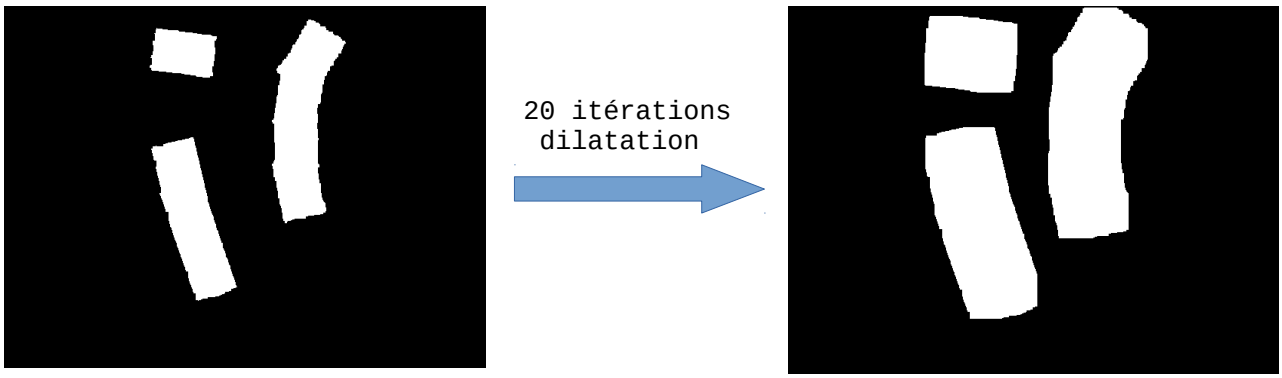



image suivi érosion

image suivi érosion–20xdilatation

La difficulté ici est de définir dynamiquement la forme d'un élément structuré, et le nombre d'itérations afin

d’agrandir les obstacles avec certaine pixels qui permet d’éviter justement la collision entre le robot et des obstacles pour un chemin calculé. Malheureusement on n’a plus du temps à implanter ce fonctionnement. Alors dans notre application, la forme d’élément structuré reste fixée, 8*8 pour l’érosion et 3*3 pour la dilatation, le nombre d’itérations est 20, et peut varier de 0 à 50 depuis l’interface QT pour adapter vos besoins.