Parcel sorting using UR3 robot

Yite Wang
Department of Mechanical Engineering
University of Illinois at Urbana-Champaign
Illinois, U.S.
yitew2@illinois.edu

Angelos Tingrui Guan
Department of Computer Science
University of Illinois at Urbana-Champaign
Illinois, U.S.
tguan2@illinois.edu

***Abstract*—This electronic document is a report containing description of a project which uses UR3 robot to implement parcel sorting in V-REP. Detailed description of methods used in the modeling are also included.**

Keywords—robot, V-REP, UR3, simulation, parcel sorting

NOMENCLATURE

S_i	Screw axis of joint i
θ_i	Joint variable of joint i
$\dot{\theta}_i$	Rate of change of θ_i with respect to time
T	Final pose of the tool
M	Initial pose of the tool
q	Location of the joint
Ad	Adjoint matrix
J	Jacobian matrix
R	Radius or an imaginary ball
P	Point location

I. INTRODUCTION

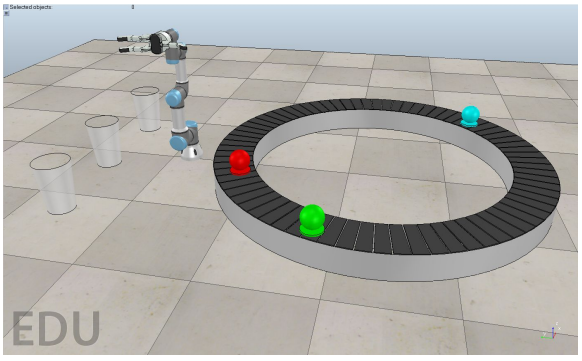

Fig.1 UR3 robot working scene

Parcel sorting is getting increasingly important in daily life because of the rapid development of online shopping. In the present study, researchers are going to use V-REP to simulate the UR3 robot to finish parcel sorting, i.e. using vision sensor to determine the location and the color of the object. Then methods including forward kinematic, inverse kinematic and path planning will be performed to control the robot to sort different kinds of objects.

As Fig.1 shows, gripper "BarrettHand" is connected to the UR3 robot so that the robot is able to grip objects. The conveyor belt is set next to the robot transporting different objects, which is very common in delivery industry. On the left-hand side of the robot, several buckets can be seen. They serve as the container for a specific object.

If the scene is running correctly, a vision sensor will be placed above the running conveyor belt. There will be several objects and markers on the conveyor belt. If the vision sensor detects the marker in the scene, image from the vision sensor will be processed. It will first distinguish different objects with different color in the image. Then location of the object will be calculated using the pixel values in the image. UR3 will get different objects after calculation of inverse kinematic and path planning are done. Then robot will put all the objects with the same color to the same bucket.

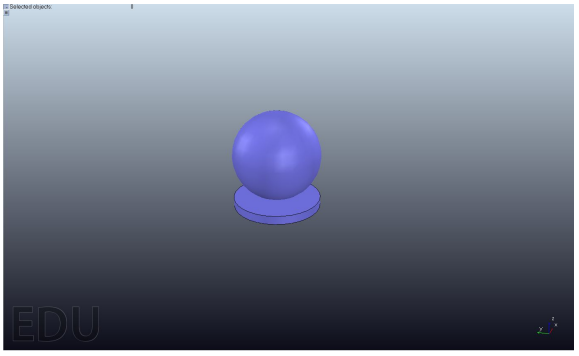

Fig.2 Working object in the scene

As Fig.2 shows, the working objects we are going to sort is a special geometry which has a sphere on the top and

a cylinder on the bottom. They are connected using force censor. The bottom cylinder makes it possible for the object to stand so that it will not roll on the conveyor. The top sphere part makes it easier for the robot to grasp it.

II. Forward Kinematics

A. Schematic of the robot

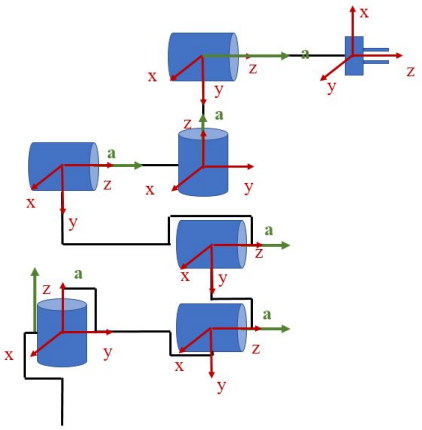

Fig.3 Schematic of the robot in the zero configuration

The schematic of the robot in the zero configuration with base frame, tool frame is shown in Fig.3. All axes are also labelled for different joints in the figure.

B. Calculation of the forward kinematics

The basic idea of forward kinematics is to predict the final pose of the tool frame with a set of given joint variables for different joints. To predict the final pose, several equations will be used.

Firstly, screw axis $\boldsymbol{S}$ need to be calculated for each joint for further calculation. It can be determined by the following equation.

$$S = \{[a \ -[a]q\ ]\ ,\ \textit{if revolute joint}\ [0\ a\ ]\ ,\ \textit{if prismatic joint} \tag{1}$$

where, $\boldsymbol{q}$ stands for location of the joint, $[\boldsymbol{a}]$ represents the skew-symmetric matrix of $\boldsymbol{a}$ and is given by

$$[a] = [0\ -a_3\ a_2\ a_3\ 0 - a_1 - a_2\ a_1\ 0\ ] \tag{2}$$

where, a_i represents the i^{th} value of a matrix.

All the $\boldsymbol{a}$ values are vectors and are clearly shown in Fig.3.

For the present case, the $\boldsymbol{S}$ matrix value is shown below.

$$S = \begin{bmatrix} & & & 0 & 0 & 0 & 0 & 0 & 0 & \\ & & & 0 & 1 & 1 & 1 & 0 & 1 & \\ & & & 1 & 0 & 0 & 0 & 1 & 0 & \\ 0 & -0.00435 & -0.248 & & -0.461 & & 0.1126 & & -0.5466 & \\ 0 & 0 & 0 & & 0 & & -4.581 * 10^{-4} & & 0 & \\ 0 & -3.118 * 10^{-5} & 1.616 * 10^{-4} & & 2.2659 * 10^{-4} & & 0 & & 4.614610^{-4} & \end{bmatrix} \tag{3}$$

where the column i in S matrix represents the screw axis of joint i.

With the calculated screw axes, we can easily calculate all the $[S_i]$ using the following equation.

$$[S_i] = [[w_i]\ v_i\ 0\ 0\ ] \tag{4}$$

where w_i is the first three numbers in crew axis $\boldsymbol{S}$ and v_i is the last three numbers in screw axis $\boldsymbol{S}$, and $[w_i]$ is the skew-symmetric matrix of w_i.

Other than calculating the twists, initial pose should also be calculated. Several important functions are used in python to get necessary variables. ***'simxGetObjectPosition'*** and ***'simxGetObjectQuaternion'*** will return two arrays, i.e. ***[x,y,z]*** and ***[qx,qy,qz,qw]***. Initial pose can be calculated using the following equation.

$$M = \begin{bmatrix} 1 - 2 * qy^2 & 2 * qx * qy - 2 * qz * qw & 2 * qx * qz + 2 * qy * qw & x \\ 2 * qx * qy + 2 * qz * qw & 1 - 2 * qx^2 - 2 * qx * qz & 2 * qy * qz - 2 * qx * qw & y \\ 2 * qx * qz - 2 * qy * qw & 2 * qy * qz + 2 * qx * qw & 1 - 2 * qx^2 - 2 * qy^2 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Thus, we are able to calculate M value:

$$M = \begin{bmatrix} 2.751 * 10^{-3} & 1 & 1.633 * 10^{-4} & 4.750 * 10^{-4} \\ 5.148 * 10^{-4} & -1.647 * 10^{-4} & 1 & 1.944 * 10^{-1} \\ 1 & -2.751 * 10^{-3} & -5.153 * 10^{-4} & 5.466 - 5.153 * 10^{-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Now it is possible to derive the final tool pose using the initial pose M, all spatial twists and all the joint variables. The final pose is given by the following equation.

$$T = e^{[S_1]\theta_1} \ldots e^{[S_6]\theta_6} M \tag{7}$$

Equation (7) is called the product of exponentials formula (POEs formula) in the robot forward kinematics. More detailed explanation can be found in [1] and [2]. It should also be noticed that $[S_i]\theta_i$ is also called spatial twist in the present document.

III. Inverse Kinematics

A. Abbreviations and Acronyms

Given a set of screw axis for each joint (denoted S), the initial pose (denoted M), and the goal pose (T_2^0). Use iterative approximation to find a set of thetas (θ) that move the robot from initial pose to goal pose.

B. Calculation

Before starting the iteration, the algorithm requires an initial guess for theta. (In our implementation, we use random values within joint limit for the initial guess.) and an error criterion to determine when to stop the iteration.

We use iterative method to get the approximation of theta. The following steps are wrapped in a while loop:

Firstly, determine a current transpose matrix using equation (7) with current guess of thetas and initial pose.

Secondly, Compute twist $[\boldsymbol{V}]$ with the equation using the following equation.

$$[V] = log(T_2^0 (T_1^0)^{-1}) \tag{8}$$

Where, T_2^0 is the goal pose and T_1^0 is the current pose.

Perform unskew transformation on $[\boldsymbol{V}]$ to get $\boldsymbol{V}$.

In particular, for a skew matrix ***x***, the result matrix after the unskew operation is:

$$Unskew(x) = [x(3,2) \ -x(3,1)\ x(2,1)\ x(1,4)\ x(2,4)\ x(3,4)\ ] \quad (9)$$

If the norm of ***V*** is less than the error criterion, break from the loop and take current theta as the output.

Thirdly, if the norm of ***V*** is higher than the criterion, the computation will continue and the Jacobian matrix will be determined using ***S*** and current guess of thetas.

It should be noticed that each column of Jacobian matrix is calculated by:

$$J[k] = [Ad_{e^{[S_1]\theta_1}\cdots e^{[S_{k-1}]\theta_{k-1}}}]S_k \quad (10)$$

Where the Adjoint matrix ***Ad*** is defined as:

$$[Ad_{T_1^0}] = [R_1^0\ 0\ [P_1^0]R_1^0\ R_1^0\ ] \quad (11)$$

Then, calculate derivative of theta by:

$$\dot{\theta} = (J^T J + \mu I)^{-1} \cdot J^T \cdot V \quad (12)$$

There are various methods to calculate the Jacobian, we choose regulated least norm solution by adding the term because it regulates the output in a certain range and improves the possibility of success of this algorithm.

More details on least square problems can be found in [3].

Lastly, increment theta with derivative of theta times ***dt***.

$$\theta = \theta + \dot{\theta} \cdot dt \quad (13)$$

Then continue to the next iteration.

C. *Edge case Discussion*

- There are certainly cases where no solution exists for given tool poses. For instance, if a given pose is beyond the operation range then there would be no solution for such pose. In our implementation, when difference between the norm of ***V*** in the current iteration and norm of ***V*** in the last iteration is below a criterion which determines if two norms are close to each other, and the norms of ***V*** of both iterations are not in the range of the error criterion, the algorithm will terminate with an error message and ask the user to input a new goal pose.
- For tool poses that have more than one solution, in our implementation, the algorithm will terminate as soon as it finds one viable solution within the norm error criterion.
- Likewise, for tool poses with only one solution, the algorithm will terminate as soon as it finds one solution and neglect other possibilities no matter such possibilities exist or not.

D. *More reference*

Further discussion in inverse kinematics can be found at [4] and [5].

IV. PATH PLANNING

A. *Collision Detection*

a) Collision Between Two Objects

In general, when detecting the collision between the two objects, we approximate them as spheres. For two spheres with given radiuses and positions of the center, we can determine whether the two objects are in collision by the following logical expression.

$$\|P_2 - P_1\| \leq r_1 + r_2 \quad (14)$$

Where P_i denotes coordinate vector of the two objects, r_i denotes radius of two spheres. If above expression is true then two spheres are in collision.

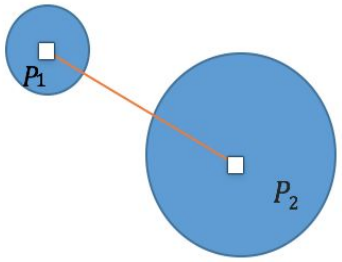

Fig.4 Collision of spheres

b) Collision Detection of a Given Configuration

In our implementation, we first approximate the robot as a set of spheres that covers the whole robot as fit as possible. Each sphere either stay static or moves with certain sets of joints depending on its relative position regarding the robot.

For a given set of joint variables (i.e., a configuration), we first check if any of the spheres that represent the robot collide with environmental obstacles in the scene in its current pose. Then, we check if any of the spheres collide with other spheres attached to the robot to detect self-collision. In addition, our algorithm also checks if the position on the z-axis is higher than the height of the base joint in order to avoid the collision with the ground.

The following is a pseudo-code for the collision detection algorithm given a set of joint twists, a set of thetas and all the initial positions of the spheres attached to the robot. In particular, the spheres are formatted in an array grouped by the last joint that determines the position of each sphere. If a sphere does not move with any joint, it is stored in the first array. For example, if the position of a sphere is determined by the first, the second, and the third joint, the sphere is stored in the fourth array.

check_current_collision:

1) all_positions = find_all_target_position(S, theta, all_initpos)
2) For each sphere in all_positions:
   i) For each obstacle: check current sphere and current obstacle, if they are in collision, break from the loop and return true for collision.

ii) For each sphere from current sphere to the last sphere: Check the sphere in outer loop and sphere in this loop for collision, if they are in collision, break from the loop and return true for collision.

3) If no collision is detected after the loop, return false for no collision.

c) Collision Detection Along a Given Path

Given a set of thetas at the start position θ_a and a set of thetas at the end position θ_b, in order to determine if there is collision along this straight-line path, we need to find segments along the line and apply collision detection of a configuration on these segment points using the method discussed in b).

Let ***s*** vary from 0 to 1 with a chosen step size, the set of theta segments can be calculated using

$$\theta = (1 - s) \cdot \theta_a + s \cdot \theta_b \quad (15)$$

Then check the collision for each theta using the method described in b), if any of these thetas is in collision, then the path is in collision.

B. Collision-free Path Planning

We choose Sampling-based Planner algorithm as our overall approach to the path planning problem. More detailed information on this topic could be found at [6].

The basic step in our implementations is just building on two search trees with the start configuration and the end configuration as parent nodes, using random sampling of new configuration in free space. If a new randomized set of thetas in free space can connect to only one of the trees in a collision free path, add it as the node to that tree. If a random set of thetas in free space can connect to both tree with no collision, find the path by tracing back both tree to their parent nodes. If a set of thetas cannot connect to any of the tree without collision, discard it and generate a new set of thetas.

An example algorithm is given below.

1) Add theta start (θ_{start}) to the forward tree ($T_{forward}$) with parent node NONE.
2) Add theta end (θ_{end}) to the backward tree ($T_{backward}$) with parent node NONE.
3) Check if θ_{start} and θ_{end} can be connected without collision, if so, return the path containing only these two points, if not, go to the following steps.
4) Repeat following steps for a threshold times, if no solution is found after the threshold, return NONE as no solution found.

i) Randomly sample a set of thetas. If this theta is in collision, continue to the next loop and sample a new set of thetas.

ii) Now that the sampled theta is in free space, find the nodes in forward and backward tree that are closest to current theta. (We are not using the RRT approach so the closest node is just the set of thetas that has the shortest distance with the current theta.) Check if there is collision between the closest nodes to the current theta.

If there is no collision between both the closest forward tree node and the current theta and the closest backward tree node and the current theta. Loop through both tree by finding the parent node of the current node and record the thetas as the output path. Note that for the forward tree, the theta of the parent node should present before the child. For the backward tree, the theta of the parent node shows up after the theta of the child node. Break from the loop and return the found path.

If there is no collision between only one of the closest tree node and the current theta. Add current theta as the child node of that node. Mark the parent node as the closest tree node with collision-free path.

iii) Continue the loop and generate new sample thetas until a path is found.

C. Discussion

- There are configurations at which the collision returns a false positive and a false negative. The main reason is that the spheres we used to represent the robot are not perfectly matching the real physical shape of the robot. To improve the performance of our collision detector, we need to find a better set of a sphere that represents the robot more accurately. For example, we can use a lot of small spheres and fill the robot shape with these small spheres. This method would improve the accuracy of the detector algorithm. Alternatively, we can try to use larger spheres to represent the robot first, if no collision is found then there is no collision, if collision is detected then we decrease size of spheres and use more spheres to more accurately approximate the shape of the robot, continue this step until we reached a criterion where we can assert that there is indeed collision in the given configuration of the robot. This method improves the efficiency and part of the speed performance of the algorithm when there is no collision. However, it might take longer time to compute if there is collision.
- There would be such cases that there is no collision-free path exist for the given start and end set of configurations. In our approach, we simply stop computing after a number of iteration and assume that there is no path for the set of input. It is up to the user to decide whether to start the algorithm again or change a set of input.
- The time needed to plan collision-free path depend on the given input and the complexity of the surrounding environment. If there are more obstacles near the input positions, the algorithm would take longer to compute. The calculation time may also depend on the absolute distance between the start and end position for similar reason. Because randomization is used in the algorithm,

even for the same set of input, the calculation time may still vary randomly.

V. Image Processing

A. Description

For the purpose of our simulation, the goal of image processing in this project is to detect if the objects to be processed is within the working range of the robot. In order to achieve this goal, a vision sensor is set above the part of conveyor belt that is closest to the robot. A object is considered to be in the working range if its pixel coordinate returned by the vision sensor is within a certain range that confines it to around the center of the image. Then if the object is in the range of operation, convert its coordinate from pixels to world reference frame.

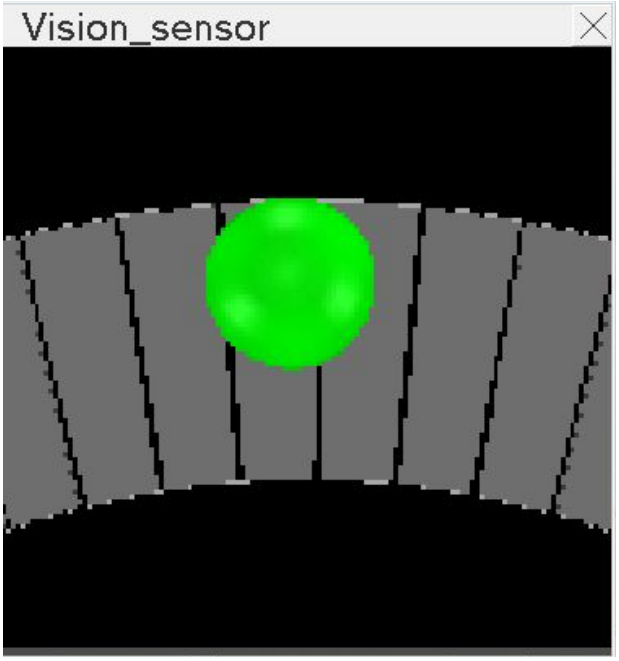

Fig.5 Image from Vision Sensor during simulation

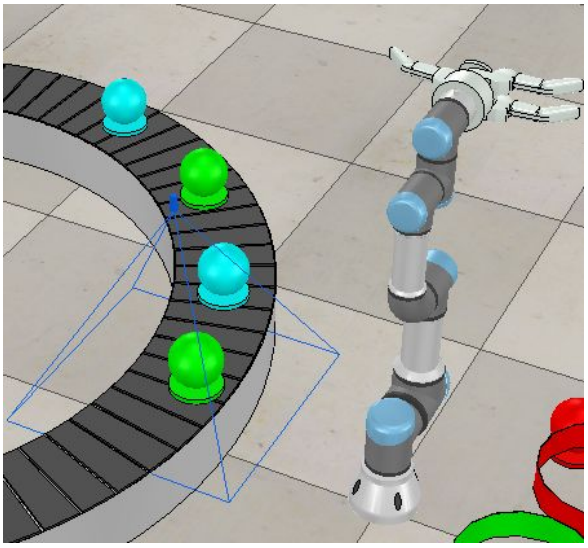

Fig.5 Vision Sensor in the scene

B. Calculation

The color of the object is determined by the RGB value of pixels returned from vision sensor in the api function ***simxGetVisionSensorImage()***. This function also returns the resolution which would be used for coordinate transformation later. For the purpose of this project, by making sure there is enough distance between each object so that there will only be one object in the image, the center of the object in the pixel frame is calculated by adding all the (x,y) coordinates of the pixels of this color and divide the sum by number of such pixels. There are more accurate ways to find the centroid of the object, more detailed discussion on such methods can be find at [7] and [8].

Once the centroid in pixel coordinate is found, next step is to convert it to reference frame used in inverse kinematics. This is simply done by the following steps. First, find difference from the object center to the center of the image, calculated by resolution/2. Next, convert this difference in the unit of world coordinate using the width of the vision sensor range divided by resolution. Finally, add this difference to the world (x,y) coordinate of the vision sensor and add a z-offset to transform it into 3D coordinate. Z-offset need to be high enough to prevent the gripper from colliding with the object and conveyor belt. This new coordinate will be used as input for inverse kinematic function to get a set of joint angles as goal pose.

The following pseudo code is for transforming pixel frame to world frame.

```
ratio = 0.4/resolution
gripper_final_x=camera_x+(pixel_row-resolution/2)*ratio
gripper_final_y=camera_y+(pixel_colume-resolution/2)*rati
o
gripper_final_z = 0.16
```

VI. Results

1. Description of Final Simulation

The basic objectives of our simulation are to distinguish the objects in the image according to their color, to calculate their center points, to determine if the object is within the working range and stop the conveyor belt if object is in working range, to convert the point coordinate from pixel coordinate to our chosen reference frame, to move the object to the corresponding destination according to its color using inverse and forward kinematics, collision detection and path planning and to move the robot back to origin pose and start conveyor again until the next object is detected and processed.

VII. Future Work

A. Finish Minimum Requirement

The first thing we are going to finish is to meet the basic requirement of the project, i.e. using the vision sensor to

detect objects on the conveyor belts of different colors and let UR3 robot to sort them. When the vision sensor detects the marker in the image, the conveyor will stop and robot will start to work. When all the objects are removed from the conveyor, the conveyor will start to run again.

B. Capturing Moving Objects

When the minimum requirement is met, the next step will be trying to make robot grip the object when the conveyor is working, which needs constantly adjusting the input variables from the vision sensor.

C. Moving robot

When the minimum requirement is met, the next step will be trying to make robot move. In the real life, there will be more spaces for storing different deliveries. So, it is vital for robots to move around to get different objects in different locations. It should also be noticed that the working space of UR3 robot is small. That is also one motivation of making the robot moving. The basic idea is installing the UR3 robot on another robot which can move. Thus, making the other robot move will also lead to the movement of UR3 robot.

D. Sorting objects according to shape

If given enough time, we would try to add objects with similar size but in different shapes on the conveyor and figure out an algorithm that allows the robot to determine which objects need to be picked up and which objects are irrelevant to its task and need to be neglected. This goal requires more thought in image processing and less relevant to robot motion, thus we will try to implement this only after we have the main goal accomplished.

ACKNOWLEDGMENT

We would like to thank our instructor Prof. Bretl at University of Illinois at Urbana-Champaign. He was always willing to answer our questions when we were in trouble. We would also like to thank our lab teaching assistant Siwei Tang, without whose help we would not be able to solve difficult problems one by one.

REFERENCES

[1] R. Brockett, “Robotic manipulators and the product of exponentials formula,” in Mathematical Theory of Networks and Systems, P.A Fuhrman, Ed. New York: Springer-Verlag, 1984, pp. 120-129.

[2] R. Murray, Z. Li, and S. Sastry, A Mathematical Introduction to Robotic Manipulation. Boca Raton, FL: CRC, 1994.

[3] M. Kern, “Linear Least Squares Problems - Singular Value Decomposition,” Numerical Methods for Inverse Problems, pp. 45–69, Jan. 2016.

[4] K. Tchoń, J. Karpińska, and M. Janiak, “Approximation of Jacobian inverse kinematics algorithms,” International Journal of Applied Mathematics and Computer Science, vol. 19, no. 4, Jan. 2009.

[5] K. Tchoń, J. Karpińska, and M. Janiak, “Approximation of Jacobian inverse kinematics algorithms,” International Journal of Applied Mathematics and Computer Science, vol. 19, no. 4, Jan. 2009.

[6] Andrew Short, Zengxi Pan, Nathan Larkin, Stephen van Duin, "Recent progress on sampling based dynamic motion planning algorithms", Advanced Intelligent Mechatronics (AIM) 2016 IEEE International Conference on, pp. 1305-1311, 2016.

[7] J. R. Parker, Algorithms for image processing and computer vision. New York: Wiley Computer Pub., 1997.

[8] R. J. Radke, “Optimization Algorithms for Computer Vision,” Computer Vision for Visual Effects, pp. 353–363.