ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS
DEPARTMENT OF INFORMATICS

# GuarddoG Project

***THIS DRAFT USES EXTENSIVE COPY PASTING FROM WIKIPEDIA :P INSTEAD OF A LOREM IPSUM TEXT AND AS PLACE HOLDERS :P***

***WORK IN PROGRESS***

**Author : Ammar Qammaz**
**Supervisor : Georgios Papaioannou**

Athens , November 2011

# Introduction and motivation

## A few opening remarks

Humans increased their physical power during the industrial revolution using machines. They were able to create giant dams , factories , cars , airplanes and skyscrapers to make their everyday life easier . Technology has continued to improve exponentially and in the current age of informatics mental capabilities where multiplied. Merging the following two revolutions we can finally partly replace ourselves from dull and repetitive tasks of day to day life that will gradually stop to trouble the human kind leading to a more pleasant life. The GuarddoG project is about making machines that can see and act as a futuristic private guard .

The process of creating an autonomous robot that can perceive its environment and react and interact whith it took nature millions of years. From the first bacteria to multi cell organisms , the wolf then the dog and the human , enormous evolutionary differences created beings of immense complexity and perfection. For someone to build something that took such a great amount of time in even a quarter of a lifetime is over-ambitious. An extra observation that is thought provoking is that while humans in complex decision making such as chess playing or tactic games with a limited set of rules have been surpassed by computers. In contrast in simple things for humans such as perceiving space , time , and "natural logic" every human has an innate superiority a result of the millions years of natural selection with these characteristics as a basis.

That beeing said GuarddoG does not attempt to create a dog ( with everything a dog implies ) , because this is practically impossible. Its goal is replacing a specific function of a dog as a guardian. I am very optimistic that with time robots will eventually be improved enough to be able to perform a multitude of tasks approaching something that will surely be different than a real dog , better at some things , and worse at some others.

Even though the future will offer even more tools , even now thanks to the marvelous technology and work of all the scientists , mathematicians , physicists , chemists , engineers and computer scientists ( we are litterally standing on the shoulders of giants ) I was able to construct something very close to my original target , spending a fraction of the money and time that would be required before say 10 years for something like it.

## Project Goal

### The goal of the “Guard Dog” Project

The goal of the Guard Dog Project is to build a robotics platform that can act as a guard and traverse a known path and fend off intruders. In case of a security breach it would signal the alarm and begin to follow the perpetrator and after a set distance will resume its previous path.

Robotics and computer vision are not a new domain of computer science and electrical engineering. It was especially shocking for me to see video footage of experiments in the AI Lab of Stanford  ( for example Les earnest and Lou paul and the Rancho Arm ) circa 1971 that perform object detection , complex decision making and that actually use more or less the same algorithms as current robotics projects do. The major difference is not so much about the methods used , but the exponential improvement on computer hardware , popularly coined as Moore`s Law.

We are living in times where many high-end mobile phones actually have more complex processors than the satellites of the first mission to the moon and that experiments such as those that required equipment that cost millions of dollars in 1971 and could only be done in universities or government research centers can be reproduced with consumer electronics readily available everywhere. Unfortunately the consistent computation of the world around a robot is still a very difficult and expensive task with a generic CPU and no specialized hardware , but yet it seems almost feasible when you achieve even something that can work 10 times slower than a human.
Of course all these using cheap building blocks but not passive sensors as modern security systems do but instead a semi-intelligent agent that can do this job the way humans would do it. It is an exploration of the possibilities and limits of current technologies along with software that can leverage them to achieve an almost working result.

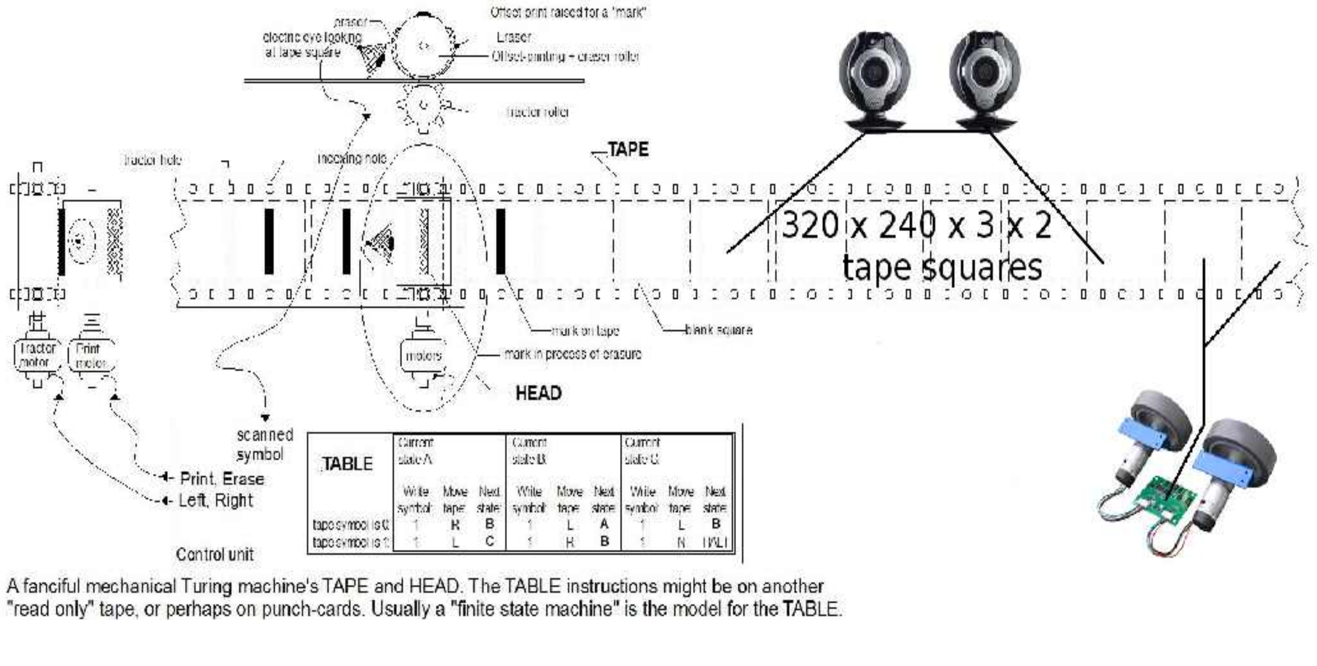
It is also interesting to note that the same computer vision libraries can with small adjustments be fitted for tasks like driving cars in city streets to helping blind people find their way or any task that involves using optical information of ones surroundings to achieve a related goal.

*

# 1 - Mathematical Model

## 1.1.0 Overview

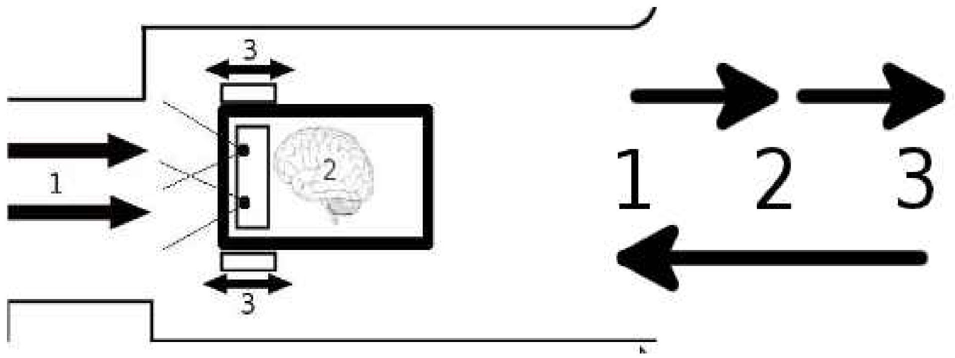
The ease with wich humans sense the world with sight makes the problem of computer vision sound “easy” to solve. In fact the way we see is so natural and persistent that even expert scientists in the field made over-optimistic predictions about this field of computer science. The fact is that despite the explosion in computational speed , and although there is a very big market that could certainly use vision algorithms to automate tasks , there is still no defacto algorithm that can compare to what human vision performs. Moreover from simple reflexes as maintaining focus and coordinating ones gaze , reading text , to tracking your position in an unknown city , vision seems to be “AI-Complete” , since understanding and combining what is seen is an altogether different task than the small builing blocks which are presented here.
A robot that can see and interact with the world , is basically a Turing machine on wheels. Therefore the whole model presented here is an adaptation of different mathematical concepts and a fusion of them together. The strip of tape in this Turing machine is constantly filled with symbols of light intensity as the light gets reflected and activates the camera sensor elements. When the control algorithm decides that the robot has to move it writes it to the according tape elements and the motors move , producing a new view of the world.



| TABLE | Current state A | | | Current state B | | | Current state C | | |
|---|---|---|---|---|---|---|---|---|---|
| | Write symbol | Move tape | Next state | Write symbol | Move tape | Next state | Write symbol | Move tape | Next state |
| tape symbol is 0 | 1 | R | B | 1 | L | A | 1 | L | B |
| tape symbol is 1 | 1 | L | C | 1 | R | B | 1 | N | HALT |

A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

The first thing to take into consideration beginning to approach this problem , is how the physical world is being represented by the cameras. They are after all the means with which the GuarddoG/RoboVision algorithm , a “meta”physical entity can take a peek into reality. The data acquired must then be filtered to remove deformations and distortions that may corrupt the whole process. These steps are described in the Camera Model , Camera Calibration and Image Rectification parts of this document. Once two consistent images of the projection of the world on the camera sensors are aquired , they are examined for optical cues that reveal the details of the world in three dimensions and also the robots position. This is also discussed extensively , and the Disparity Mapping algorithm used by GuarddoG is a new implemetation. When all these steps are finished , the next one is tracking the position of the robot ( LK Optical Flow , RANSAC Homography ) and the combination of the successive 3d Views together ( SLAM , Obstacle Detection ).
The final piece of the algorithm is a knowledge base that will set its goals and keep the state of the world , and steer the robot towards achieving them. For  GuarddoG , its goal is the traversal of a standard path , and raising the alarm if a breach is found.



In beggining to make a system that sees , one can make many choices about the way with which to gather input. As nature teaches us , and by bringing to mind various insects and animals that have been optimized throught a process of millions of years to see one might use anything from ultrasonic sounds , to millions small eyes of insects up to human stereoscopy. With the world represented through the camera beeing so chaotic , and as this project does not deal with a fixed environment in which to be operated , while also having economic restrictions applied the best choice was a human like stereoscopic camera input. It is true that commercial RGB+depth cameras such as Microsoft Kinect can bypass a very big portion of the computational complexity of this project , but they still have their own short comings. The stereoscopic setup wasn`t chosen by accident by nature , and the nature of a robot that uses stereoscopic vision makes it closer to the human experience as a mode of viewing the world.

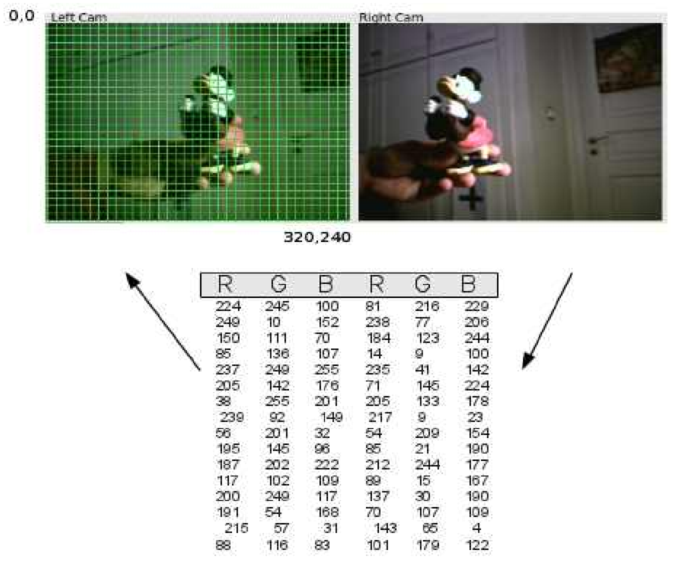
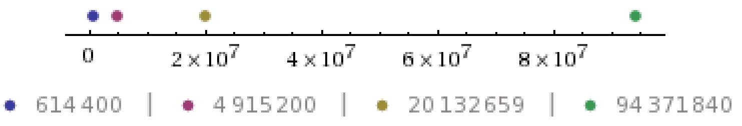


*Illustration 1: What computers see*

| R | G | B | R | G | B |
|---|---|---|---|---|---|
| 224 | 245 | 100 | 81 | 216 | 229 |
| 249 | 10 | 152 | 238 | 77 | 206 |
| 150 | 111 | 70 | 184 | 123 | 244 |
| 85 | 136 | 107 | 14 | 9 | 100 |
| 237 | 249 | 255 | 235 | 41 | 142 |
| 205 | 142 | 176 | 71 | 145 | 224 |
| 38 | 255 | 201 | 205 | 133 | 178 |
| 239 | 92 | 149 | 217 | 9 | 23 |
| 56 | 201 | 32 | 54 | 209 | 154 |
| 195 | 145 | 96 | 85 | 21 | 190 |
| 187 | 202 | 222 | 212 | 244 | 177 |
| 117 | 102 | 109 | 89 | 15 | 167 |
| 200 | 249 | 117 | 137 | 30 | 190 |
| 191 | 54 | 168 | 70 | 107 | 109 |
| 215 | 57 | 31 | 143 | 65 | 4 |
| 88 | 116 | 83 | 101 | 179 | 122 |

Trying to approach the computational limit of a dense stereoscopic method for two frames sized 320x240 pixels in order for a full search from an image patch sized 40x40 pixels on the left eye to all the possible matching patches along the epipolar line on the right eye , we have to make 320 x 320 x 240 / 40 = 24576000 /40 = 614400 operations in the worst case each time we get a depth map. In order to achieve a “human like” response time from the vision system this has to be done at a rate of 25 frames per second , or with a delay of 40 milliseconds per scan.. The number of operations per second increases exponentially as the image size becomes larger
QVGA 320 x 320 x 240 / 40 = 24576000 /40 = 614,400 operations / ms
VGA 640 x 640 x 480 / 40 = 196608000 /40 = 4,915,200 operations / ms
XGA 1024 x 1024 x 768 / 40 = 805306368 / 40 = 20,132,659 operations / ms
...
...
WUXGA 1920 x 1920 x 1024 / 40 = 3774873600 / 40 = 94,371,840 operations / ms



In order to increase computational efficiency and reduce errors , the difficulty of manufacturing a physical stereo rig for the experiments , and the mathematical ambiguity that has a direct effect on the calculations. The relative position of the cameras is supposed to be constant and the two cameras to have a coplanar alignment. The cameras are also never allowed to change their focus ( nor could change it as they do not have manual focus control ).

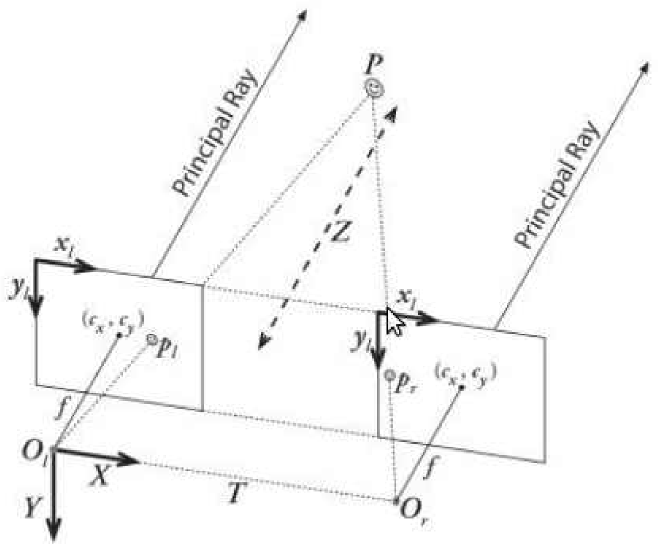


*Illustration 2: Camera rig set-up*

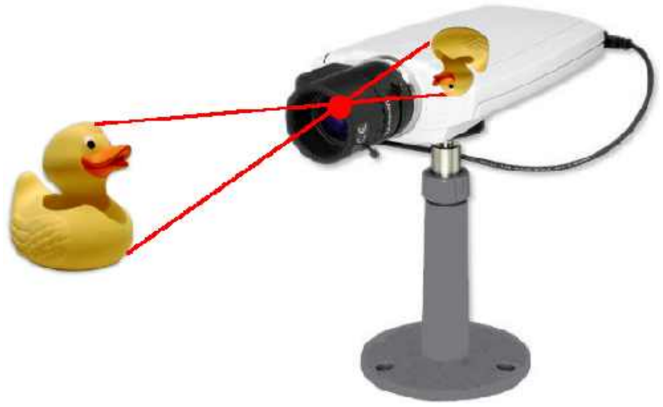
The first issue to be addressed when receiving images is that the pictures being retrieved are actually distorted due to camera optics and lens imperfections.  The way to compensate for these distortions is called camera resectioning ( calibration or rectification) and has been documented extensively. Learning OpenCV Pages 386-394
GuarddoG uses the checkboard OpenCV default method for calibration as a preliminary step for each pair of cameras , done only one time during the initial set up of the robot. The method was conceived by Zhang [Zhang99; Zhang00] and Sturm [Sturm99].

To avoid re calculation and use of the CPU for reasons avoidable by a better designed algorithm , the whole vision library uses a pipelined architecture , so that the same image will not have to pass a processing stage twice once it enters and according to the needs of the Robot Hypervisior the different stages will try to be combined , or operations will stay pending for the next frame. These things , along with performance statistics for different hardware are discussed in the software section of this document.

*

# 1 - Mathematical Model

### 1.1.1 Camera Pinhole Model



A **pinhole camera** is a simple camera without a lens and with a single small aperture – effectively a light-proof box with a small hole in one side. Light from a scene passes through this single point and projects an inverted image on the opposite side of the box. The human eye in bright light acts similarly, as do cameras using small apertures.

Up to a certain point, the smaller the hole, the sharper the image, but the dimmer the projected image. Optimally, the size of the aperture should be 1/100 or less of the distance between it and the projected image.

Because a pinhole camera requires a lengthy exposure, its shutter may be manually operated, as with a flap of light-proof material to cover and uncover the pinhole. Typical exposures range from 5 seconds to several hours.

The **pinhole camera model** describes the mathematical relationship between the coordinates of a 3D point and its projection onto the image plane of an *ideal* pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light. The model does not include, for example, geometric distortions or blurring of unfocused objects caused by lenses and finite sized apertures. It also does not take into account that most practical cameras have only discrete image coordinates. This means that the pinhole camera model can only be used as a first order approximation of the mapping from a 3D scene to a 2D image. Its validity depends on the quality of the camera and, in general, decreases from the center of the image to the edges as lens distortion effects increase.
Some of the effects that the pinhole camera model does not take into account can be compensated for, for example by applying suitable coordinate transformations on the image coordinates, and others effects are sufficiently small to be neglected if a high quality camera is used. This means that the pinhole camera model often can be used as a reasonable description of how a camera depicts a 3D scene, for example in computer vision and computer graphics.

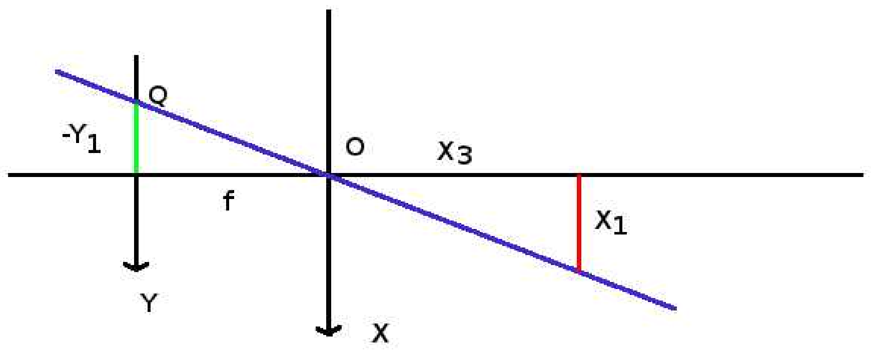
The geometry related to the mapping of a pinhole camera is illustrated in the figure. The figure contains the following basic objects

- A 3D orthogonal coordinate system with its origin at **O**. This is also where the *camera aperture* is located. The three axes of the coordinate system are referred to as X1, X2, X3. Axis X3 is pointing in the viewing direction of the camera and is referred to as the *optical axis*, *principal axis*, or *principal ray*. The 3D plane which intersects with axes X1 and X2 is the front side of the camera, or *principal plane*.
- An image plane where the 3D world is projected through the aperture of the camera. The image plane is parallel to axes X1 and X2 and is located at distance $f$ from the origin **O** in the negative direction of the X3 axis. A practical implementation of a pinhole camera implies that the image plane is located such that it intersects the X3 axis at coordinate $-f$ where $f > 0$. $f$ is also referred to as the *focal length*[*citation needed*] of the pinhole camera.
- A point **R** at the intersection of the optical axis and the image plane. This point is referred to as the *principal point* or *image center*.
- A point **P** somewhere in the world at coordinate $(x_1,x_2,x_3)$ relative to the axes X1,X2,X3.
- The *projection line* of point **P** into the camera. This is the green line which passes through point **P** and the point **O**.
- The projection of point **P** onto the image plane, denoted **Q**. This point is given by the intersection of the projection line (green) and the image plane. In any practical situation we can assume that $x_3 > 0$ which means that the intersection point is well defined.
- There is also a 2D coordinate system in the image plane, with origin at **R** and with axes Y1 and Y2 which are parallel to X1 and X2, respectively. The coordinates of point **Q** relative to this coordinate system is $(y_1,y_2)$.

The *pinhole* aperture of the camera, through which all projection lines must pass, is assumed to be infinitely small, a point. In the literature this point in 3D space is referred to as the *optical (or lens or camera) center*.[*citation needed*]

Next we want to understand how the coordinates $(y_1,y_2)$ of point **Q** depend on the coordinates $(x_1,x_2,x_3)$ of point **P**. This can be done with the help of the following figure which shows the same scene as the previous figure but now from above, looking down in the negative direction of the X2 axis.

Next we want to understand how the coordinates $(y_1,y_2)$ of point **Q** depend on the coordinates $(x_1,x_2,x_3)$ of point **P**. This can be done with the help of the following figure which shows the same scene as the previous figure but now from above, looking down in the negative direction of the X2 axis.



The geometry of a pinhole camera as seen from the X2 axis

In this figure we see two similar triangles, both having parts of the projection line (green) as their hypotenuses. The catheti of the left triangle are $-y_1$ and $f$ and the catheti of the right triangle are $x_1$ and $x_3$. Since the two triangles are similar it follows that which is an expression that describes the relation between the 3D coordinates $(x_1,x_2,x_3)$ of point **P** and its image coordinates $(y_1,y_2)$ given by point **Q** in the image plane.[*citation needed*]

$$\frac{-y_1}{f} = \frac{x_1}{x_3} \text{ or } y_1 = -\frac{f\,x_1}{x_3}$$

$$\frac{-y_2}{f} = \frac{x_2}{x_3} \text{ or } y_2 = -\frac{f\,x_2}{x_3}$$

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{f}{x_3}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

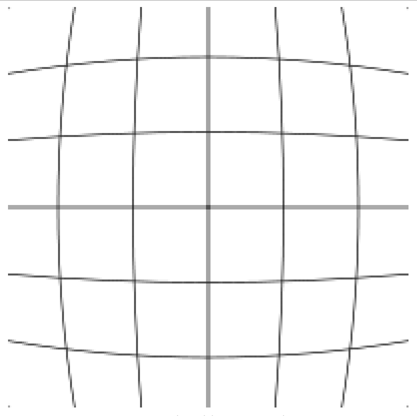
*

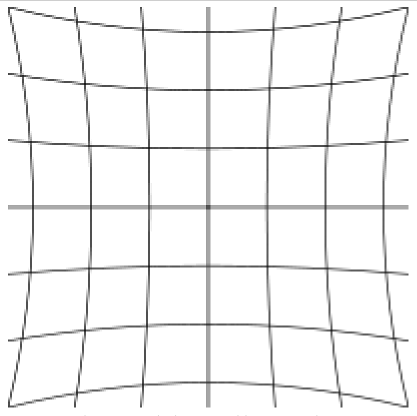
# 1 - Mathematical Model

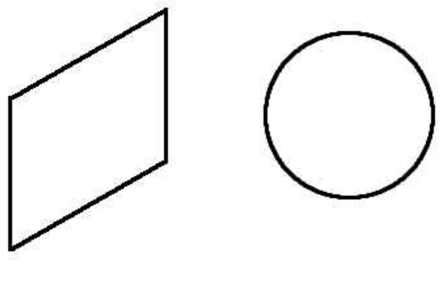
## 1.1.2 Camera Calibration

In mathematics, it is possible to define a lens set that will introduce no distortions in the image captured. In practice, however , and due to manufacturing process inefficiencies two types of distortion occur . Radial distortion that is caused by the shape of lens not being parabolic , and tangential distortions due to the assembly process of the camera in the factory.

Radial distortion causes a characteristic twisting of straight lines as they get closer to the edges of the image and on systems that are heavily based on those images it can have a very detrimental effect on calculations that gets worse as the errors gradually accumulate in time..


Barrel distortion


Pincushion distortion



Tangential Distortion on the other hand is a matter of misplacing the imaging sensor in accordance to the lens ( not a fully parallel placement ) and therefore receiving a slightly skewed image.

is mainly for reasons of manufacturing; it is much easier
to make a “spherical” lens than to make a more mathematically ideal “parabolic” lens. It is also difficult to mechanically align the lens and imager exactly. Here we describe the two main lens distortions and how to model them.* Radial distortions arise as a result of the shape of lens, whereas tangential distortions arise from the assembly process of the camera as a whole.
We start with radial distortion. The lenses of real cameras often noticeably distort the location of pixels near the edges of the imager. This bulging phenomenon is the source of the “barrel” or “fish-eye” effect (see the room-divider lines at the top of Figure 11-12 for a good example). Figure 11-3 gives some intuition as to why radial distortion occurs. With some lenses, rays farther from the center of the lens are bent more than those closer in. A typical inexpensive lens is, in effect, stronger than it ought to be as you get farther from the center. Barrel distortion is particularly noticeable in cheap web cam-eras but less apparent in high-end cameras, where a lot of effort is put into fancy lens systems that minimize radial distortion.
For radial distortions, the distortion is 0 at the (optical) center of the imager and in-creases as we move toward the periphery. In practice, this distortion is small and can be characterized by the first few terms of a Taylor series expansion around $r = 0$.† For cheap web cameras, we generally use the first two such terms; the first of which is convention-ally called k1 and the second k2. For highly distorted cameras such as fish-eye lenses we can use a third radial distortion term k3. In general, the radial location of a point on the imager will be rescaled according to the following equations:
* The approach to modeling lens distortion taken here derives mostly from Brown [Brown71] and earlier
Fryer and Brown [Fryer86].
† If you don’t know what a Taylor series is, don’t worry too much. The Taylor series is a mathematical tech-
nique for expressing a (potentially) complicated function in the form of a polynomial of similar value to
the approximated function in at least a small neighborhood of some particular point (the more terms we
include in the polynomial series, the more accurate the approximation). In our case we want to expand the
distortion function as a polynomial in the neighborhood of $r = 0$. Th is polynomial takes the general form
$f(r) = a 0 + a1r + a2r2+ ...$, but in our case the fact that $f(r) = 0$ at $r = 0$ implies $a 0 = 0$. Similarly, because the
function must be symmetric in r, only the coefficients of even powers of r will be nonzero. For these reasons,
the only parameters that are necessary for characterizing these radial distortions are the

coefficients of
$r^2$, $r^4$, and (sometimes) $r^6$.

Figure 11-3. Radial distortion: rays farther from the center of a simple lens are bent too much com-
pared to rays that pass closer to the center; thus, the sides of a square appear to bow out on the image
plane (this is also known as barrel distortion)

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Here, $(x, y)$ is the original location (on the imager) of the distorted point and $(x_{corrected}, y_{corrected})$ is the new location as a result of the correction. Figure 11-4 shows displacements of a rectangular grid that are due to radial distortion. External points on a front-facing rectangular grid are increasingly displaced inward as the radial distance from the optical center increases.

The second-largest common distortion is tangential distortion. This distortion is due to manufacturing defects resulting from the lens not being exactly parallel to the imaging plane; see Figure 11-5.

Tangential distortion is minimally characterized by two additional parameters, $p_1$ and $p_2$, such that:*

$$x_{corrected} = x + [2 p_1 y + p_2 (r^2 + 2 x^2)]$$
$$y_{corrected} = y + [p_1 (r^2 + 2 y^2) + 2 p_2 x]$$

Thus in total there are five distortion coefficients that we require. Because all five are necessary in most of the OpenCV routines that use them, they are typically bundled into one distortion vector; this is just a 5-by-1 matrix containing $k_1$, $k_2$, $p_1$, $p_2$, and $k_3$ (in that order). Figure 11-6 shows the effects of tangential distortion on a front-facing external rectangular grid of points. The points are displaced elliptically as a function of location and radius.

* The derivation of these equations is beyond the scope of this book, but the interested reader is referred to
the "plumb bob" model; see D. C. Brown, "Decentering Distortion of Lenses", Photometric Engineering 32(3)
(1966), 444–462.

Figure 11-4. Radial distortion plot for a particular camera lens: the arrows show where points on an
external rectangular grid are displaced in a radially distorted image (courtesy of Jean-Yves Bouguet)

Figure 11-5. Tangential distortion results when the lens is not fully parallel to the image plane; in
cheap cameras, this can happen when the imager is glued to the back of the camera (image courtesy

of Sebastian Thrun)
There are many other kinds of distortions that occur in imaging systems, but they typically have lesser effects than radial and tangential distortions. Hence neither we nor OpenCV will deal with them further.

Figure 11-6. Tangential distortion plot for a particular camera lens: the arrows show where points on
an external rectangular grid are displaced in a tangentially distorted image (courtesy of Jean-Yves
Bouguet)
## Calibration
Now that we have some idea of how we'd describe the intrinsic and distortion properties of a camera mathematically, the next question that naturally arises is how we can use OpenCV to compute the intrinsics matrix and the distortion vector.*
OpenCV provides several algorithms to help us compute these intrinsic parameters. The actual calibration is done via cvCalibrateCamera2(). In this routine, the method of calibration is to target the camera on a known structure that has many individual and identifiable points. By viewing this structure from a variety of angles, it is possible to then compute the (relative) location and orientation of the camera at the time of each image as well as the intrinsic parameters of the camera (see Figure 11-9 in the "Chessboards" section). In order to provide multiple views, we rotate and translate the object, so let's pause to learn a little more about rotation and translation.
* For a great online tutorial of camera calibration, see Jean-Yves Bouguet's calibration website (http://www.vision.caltech.edu/bouguetj/calib_doc).

### Rotation Matrix and Translation Vector
For each image the camera takes of a particular object, we can describe the pose of the object relative to the camera coordinate system in terms of a rotation and a translation; see Figure 11-7.
Figure 11-7. Converting from object to camera coordinate systems: the point P on the object is seen
as point p on the image plane; the point p is related to point P by applying a rotation matrix R and a
translation vector t to P
In general, a rotation in any number of dimensions can be described in terms of multiplication of a coordinate vector by a square matrix of the appropriate size. Ultimately, a rotation is equivalent to introducing a new description of a point's location in a different coordinate system. Rotating the coordinate system by an angle θ is equivalent to counterrotating our target point around the origin of that coordinate system by the same angle θ. The representation of a two-dimensional rotation as matrix multiplication is shown in Figure 11-8. Rotation in three dimensions can be decomposed into a two-dimensional rotation around each axis in which the pivot axis measurements remain constant. If we rotate around the x-, y-, and z-axes in sequence* with respective rotation

angles ψ, φ, and θ, the result is a total rotation matrix R that is given by the product of the three matrices R x(ψ), Ry(φ), and Rz(θ), where:
Figure 11-8. Rotating points by θ (in this case, around the Z-axis) is the same as counterrotating the coordinate axis by θ; by simple trigonometry, we can see how rotation changes the coordinates of a point
Thus, R = Rz(θ), Ry(φ), R x(ψ). The rotation matrix R has the property that its inverse is its transpose (we just rotate back); hence we have RTR = RRT = I, where I is the identity matrix consisting of 1s along the diagonal and 0s everywhere else.
The translation vector is how we represent a shift from one coordinate system to another system whose origin is displaced to another location; in other words, the translation vector is just the offset from the origin of the first coordinate system to the origin of the second coordinate system. Thus, to shift from a coordinate system centered on an object to one centered at the camera, the appropriate translation vector is simply T = originobject – origincamera. We then have (with reference to Figure 11-7) that a point in the object (or world) coordinate frame Po has coordinates Pc in the camera coordinate frame:
Pc = R( Po − T )
380 | Chapter 11: Camera Models and Calibration
Combining this equation for Pc above with the camera intrinsic corrections will form the basic system of equations that we will be asking OpenCV to solve. The solution to these equations will be the camera calibration parameters we seek.
We have just seen that a three-dimensional rotation can be specified with three angles and that a three-dimensional translation can be specified with the three parameters (x, y, z); thus we have six parameters so far. The OpenCV intrinsics matrix for a camera has four parameters (fx, fy, cx, and cy), yielding a grand total of ten parameters that must be solved for each view (but note that the camera intrinsic parameters stay the same between views). Using a planar object, we'll soon see that each view fi xes eight parameters. Because the six parameters of rotation and translation change between views, for each view we have constraints on two additional parameters that we use to resolve the camera intrinsic matrix. We'll then need at least two views to solve for all the geometric parameters.
We'll provide more details on the parameters and their constraints later in the chapter, but first we discuss the calibration object. The calibration object used in OpenCV is a flat grid of alternating black and white squares that is usually called a "chessboard" (even though it needn't have eight squares, or even an equal number of squares, in each direction).
*

# 1 - Mathematical Model

### 1.1.3 Image Rectification

Camera resectioning is the process of finding the true parameters of the camera that produced a given photograph or video. Usually, the camera parameters are represented in a 3 × 4 matrix called the camera matrix.

This process is often called camera calibration, but "camera calibration" can also mean photometric camera calibration.
[edit] Parameters of camera model

Often, we use [u\,v\,1]^T to represent a 2D point position in Pixel coordinates. [x_w\, y_w\, z_w\,1]^T is used to represent a 3D point position in World coordinates.Note: they were expressed in augmented notation of Homogeneous coordinates which is most common notation in robotics and rigid body transforms. Referring to the pinhole camera model, a camera matrix is used to denote a projective mapping from World coordinates to Pixel coordinates.

The intrinsic matrix containing 5 intrinsic parameters. These parameters encompass focal length, image format, and principal point. The parameters \alpha_{x} = f \cdot m_{x} and \alpha_{y} = f \cdot m_{y} represent focal length in terms of pixels, where mx and my are the scale factors relating pixels to distance. [1] γ represents the skew coefficient between the x and the y axis, and is often 0. u0 and v0 represent the principal point, which would be ideally in the centre of the image.

Nonlinear intrinsic parameters such as lens distortion are also important although they cannot be included in the linear camera model described by the intrinsic parameter matrix. Many modern camera calibration algorithms estimate these intrinsic parameters as well.

Extrinsic parameters

\textbf R,T are the extrinsic parameters which denote the coordinate system transformations from 3D world coordinates to 3D camera coordinates. Equivalently, the extrinsic parameters define the position of the camera center and the camera's heading in world coordinates. T is not the position of the camera. It is the position of the origin of the world coordinate system expressed in coordinates of the camera-centered coordinate system. The positon, C, of the camera expressed in world coordinates is C = − R − 1T = − RTT (since R is a rotation matrix).

When a camera is used, light from the environment is focused on an image plane and captured. This process reduces the dimensions of the data taken in by the camera from three to two (light from a 3D scene is stored on a 2D image). Each pixel on the image plane therefore corresponds to a shaft of light from the original scene. Camera resectioning determines which incoming light is associated with each pixel on the resulting image. In an ideal pinhole camera, a simple projection matrix is enough to do this. With more complex camera systems, errors resulting from misaligned lenses and deformations in their structures can result in more complex distortions in the final image. The camera projection matrix is derived from the intrinsic and extrinsic parameters of the camera, and is often represented by the series of transformations; e.g., a matrix of camera intrinsic parameters, a $3 \times 3$ rotation matrix, and a translation vector. The camera projection matrix can be used to associate points in a camera's image space with locations in 3D world space.

Camera resectioning is often used in the application of stereo vision where the camera projection matrices of two cameras are used to calculate the 3D world coordinates of a point viewed by both cameras.

Some people call this camera calibration, but many restrict the term camera calibration for the estimation of internal or intrinsic parameters only.

*

# 1 - Mathematical Model

### 1.1.4 Image Processing

Digital cameras are devices that capture the light that the universe reflects on their sensor. The general problem most vision algorithms try to solve is guessing what kind of a world reflects the light in that way. The algorithms presented here are building blocks that gradually transform the raw RGB input into more meaningful ( for a computer ) representations .

Convolution is a mathematical operation applied to sets of values that “redistributes” them according to weights. The carrier of the weights is called a convolution matrix and it is typically a 3x3 kernel with one of the values , usually the middle one , called an anchor point.

The values transformed by the convolution matrix are the red , green and blue light intensities of the pixels retrieved from the image sensor. In the following example we assume a 3x3 kernel and a monochrome image sensor that captured 9x6 pixels. The kernel is passed left to right and up to down until all of the elements are changed. GuarddoG uses Gaussian Blur , Sobel and Second Derivative Convolution kernels that follow with example images.

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

**3X3 Convolution Kernel**
**Divisor 9**

As the anchor of the kernel passes from each element of the image array the value ( marked blue ) gets replaced by the addition of the neighboring elements multiplied with the according kernel element.

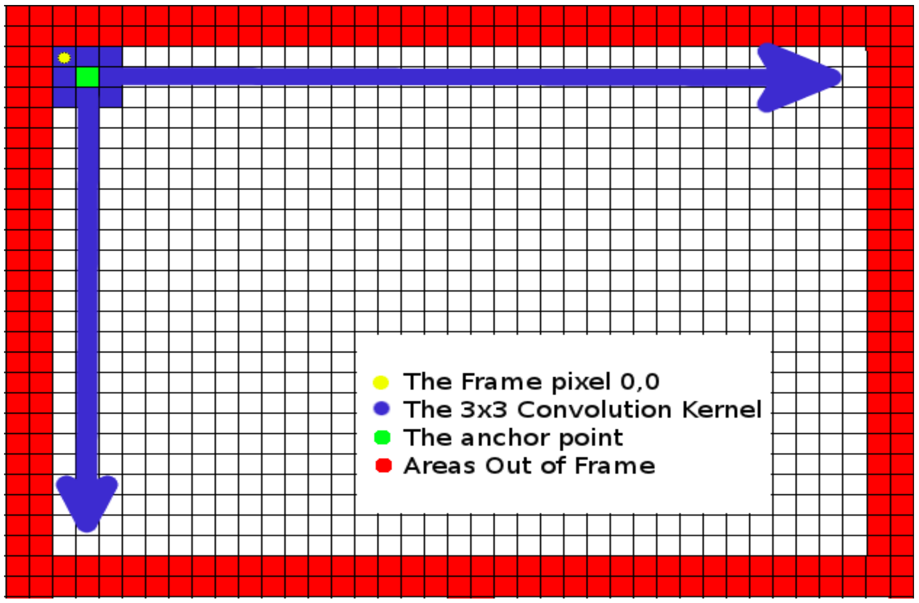
$$H(x,y)=\sum_{i=0}^{M_i-1}\sum_{j=0}^{M_j-1} I(x+i-a_i,y+j-a_j)G(i,j)$$

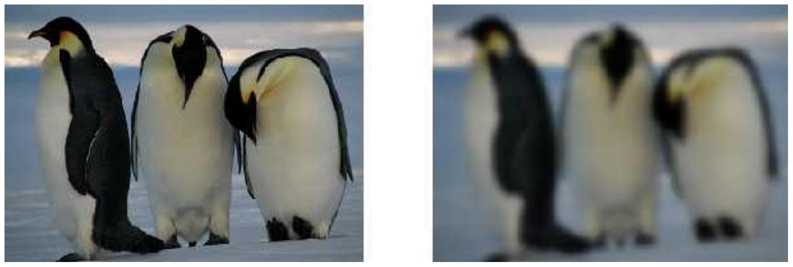
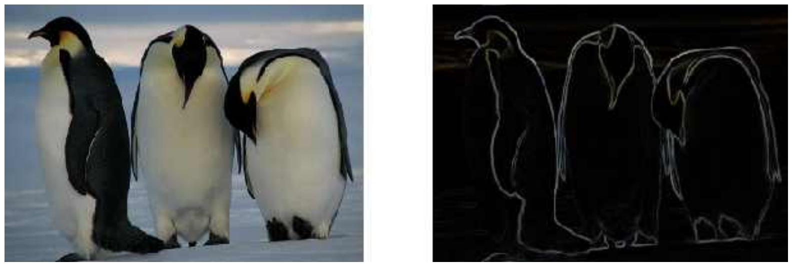
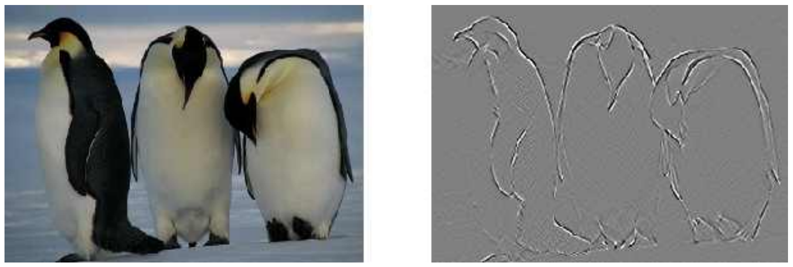
The anchor element on the light intensities array will become
( 1x90+1x80 +1x70+1x90+1*80+1*70+ 1x90 + 1x80 + 1x70 ) / 9 which is 80

**9 x 6 Original Light Intensities Captured**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |
| 90 | 80 | 70 | 90 | 80 | 70 | 90 | 80 | 70 |

An important thing to be noted Is that values on the edges of the array ( marked orange ) can not be correctly calculated as not all neighboring elements exist , common solutions for this is “imagining” that there are zero elements when an element does not exist , using a different divisor to compensate for the missing elements or skipping the elements that can not be calculated correctlly .



| | |
|---|---|
| **GAUSSIAN BLUR**<br><br>1 1 1<br>1 **1** 1<br>1 1 1<br>Divisor 9 |  |
| **SOBEL DERIVATIVE**<br><br>1 -2 1<br>2 **-4** 2<br>1 -2 1<br>Divisor 1 |  |
| **SECOND-ORDER DERIVATIVE**<br><br>-1 0 1<br>0 **0** 0<br>1 0 -1<br>Divisor 3 |  |

A Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination. Gaussian smoothing is also used as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales—see scale-space representation and scale-space implementation.
Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function; this is also known as a two-dimensional Weierstrass transform. By contrast, convolving by a circle (i.e., a circular box blur) would more accurately reproduce the bokeh effect. Since the Fourier transform of a Gaussian is another Gaussian, applying a Gaussian blur has the effect of reducing the image's high-frequency components; a Gaussian blur is thus a low pass filter.

The Sobel operator is used in image processing, particularly within edge detection algorithms. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is either the corresponding gradient vector or the norm of this vector. The Sobel operator is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical direction and is therefore relatively inexpensive in terms of computations. On the other hand, the gradient approximation which it produces is relatively crude, in particular for high frequency variations in the image.

In simple terms, the operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result therefore shows how "abruptly" or "smoothly" the image changes at that point, and therefore how likely it is that that part of the image represents an edge, as well as how that edge is likely to be oriented. In practice, the magnitude (likelihood of an edge) calculation is more reliable and easier to interpret than the direction calculation.

Mathematically, the gradient of a two-variable function (here the image intensity function) is at each image point a 2D vector with the components given by the derivatives in the horizontal and vertical directions. At each image point, the gradient vector points in the direction of largest possible intensity increase, and the length of the gradient vector corresponds to the rate of change in that direction. This implies that the result of the Sobel operator at an image point which is in a region of constant image intensity is a zero vector and at a point on an edge is a vector which points across the edge, from darker to brighter values.

Edge detection is a fundamental tool in image processing and computer vision, particularly in the areas of feature detection and feature extraction, which aim at identifying points in a digital image at which the image brightness changes sharply or, more formally, has discontinuities. The same problem of finding discontinuities in 1D signals is known as step detection.
The purpose of detecting sharp changes in image brightness is to capture important events and

changes in properties of the world. It can be shown that under rather general assumptions for an image formation model, discontinuities in image brightness are likely to correspond to[1][2]:

discontinuities in depth,
discontinuities in surface orientation,
changes in material properties and
variations in scene illumination.

In the ideal case, the result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings as well as curves that correspond to discontinuities in surface orientation. Thus, applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image. If the edge detection step is successful, the subsequent task of interpreting the information contents in the original image may therefore be substantially simplified. However, it is not always possible to obtain such ideal edges from real life images of moderate complexity. Edges extracted from non-trivial images are often hampered by fragmentation, meaning that the edge curves are not connected, missing edge segments as well as false edges not corresponding to interesting phenomena in the image – thus complicating the subsequent task of interpreting the image data.[3]

Edge detection is one of the fundamental steps in image processing, image analysis, image pattern recognition, and computer vision techniques. During recent years, however, substantial (and successful) research has also been made on computer vision methods[which?] that do not explicitly rely on edge detection as a pre-processing step.
To ADD complexity data etc
What information can we get from an image ? And how can we get it ?
Gaussian Blur Operator -> Noise Suppression
Sobel Operator - > Edges
Second Derivative Operator -> Edge maxima
Frame Subtraction -> Movement

Comparing patches/blocks/features

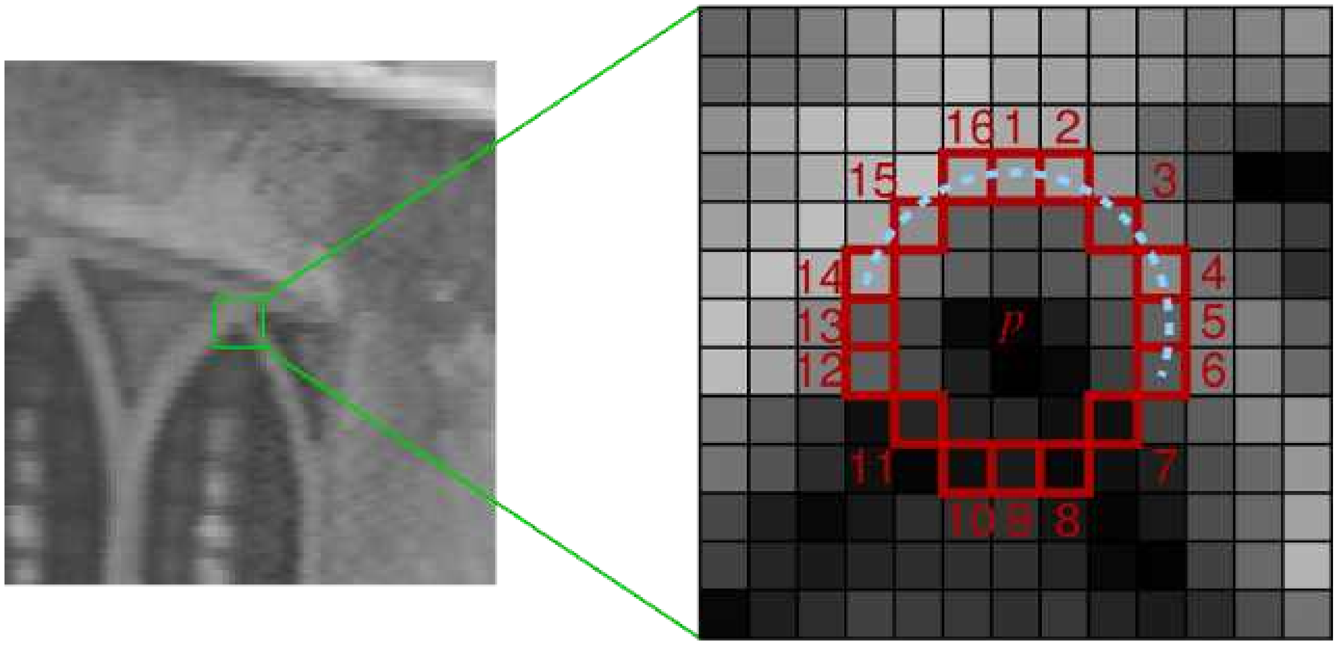
Comparing 2D patches of any image with other 2D patches is the most fundamental algorithmand building block used by the project. A good comparison function is were everything starts from and without it all the following operations diverge more and more to an unusable level.

The typical algorithms used for patch comparison are the following
Histogram : Correlation method Histogram , Chi square , intersection ,
Sum of Squared Differences ,
Minimum Absolute Difference

Maximum Absolute Pixel Count
Features :
Physical build
Pyramid of Patch sizes

## Mathematical Model
Patch Comparison and Integral Images

## Mathematical Model
FAST Corner Detection



## Mathematical Model
HAAR Wavelet Face Detection

Haar-like features are digital image features used in object recognition. They owe their name to their intuitive similarity with Haar wavelets and were used in the first real-time face detector.

Historically, working with only image intensities (i.e., the RGB pixel values at each and every pixel of image) made the task of feature calculation computationally expensive. A publication by Papageorgiou et al.[1] discussed working with an alternate feature set based on Haar wavelets instead of the usual image intensities. Viola and Jones[2] adapted the idea of using Haar wavelets and developed the so called Haar-like features. A Haar-like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in these regions and calculates the difference between them. This difference is then used to categorize subsections of an image. For example, let us say we have an image database with human faces. It is a common observation that among all faces the region of the eyes is darker than the region of the cheeks. Therefore a common haar feature for face detection is a set of two adjacent rectangles that lie above the eye and the cheek region. The position of these rectangles is defined relative to a detection window that acts like a bounding box to the target object (the face in this case).

In the detection phase of the Viola–Jones object detection framework, a window of the target size is moved over the input image, and for each subsection of the image the Haar-like feature is calculated. This difference is then compared to a learned threshold that separates non-objects from objects. Because such a Haar-like feature is only a weak learner or classifier (its detection quality is slightly better than random guessing) a large number of Haar-like features are necessary to describe an object with sufficient accuracy. In the Viola–Jones object detection framework, the Haar-like features are therefore organized in something called a classifier cascade to form a strong learner or classifier.

The key advantage of a Haar-like feature over most other features is its calculation speed. Due to the use of integral images, a Haar-like feature of any size can be calculated in constant time (approximately 60 microprocessor instructions for a 2-rectangle feature).

Rectangular Haar-like features

A simple rectangular Haar-like feature can be defined as the difference of the sum of pixels of areas inside the rectangle, which can be at any position and scale within the original image. This modified feature set is called 2-rectangle feature. Viola and Jones also defined 3-rectangle features and 4-rectangle features. The values indicate certain characteristics of a particular area of the image. Each feature type can indicate the existence (or absence) of certain characteristics in the image, such as edges or changes in texture. For example, a 2-rectangle feature can indicate where the border lies between a dark region and a light region.
[edit] Fast computation of Haar-like features

One of the contributions of Viola and Jones was to use summed area tables[3], which they called integral images. Integral images can be defined as two-dimensional lookup tables in the form of a matrix with the same size of the original image. Each element of the integral image contains the sum of all pixels located on the up-left region of the original image (in relation to the element's position). This allows to compute sum of rectangular areas in the image, at any position or scale, using only four lookups:

    \text{sum} = pt_4 - pt_3 - pt_2 + pt_1. \,

where points ptn belong to the integral image (include a figure).

Each Haar-like feature may need more than four lookups, depending on how it was defined. Viola and Jones's 2-rectangle features need six lookups, 3-rectangle features need eight lookups, and 4-rectangle features need nine lookups.
[edit] Tilted Haar-like features

Lienhart and Maydt[4] introduced the concept of a tilted (45°) Haar-like feature. This was used to increase the dimensionality of the set of features in an attempt to improve the detection of objects in images. This was successful, as some of these features are able to describe the object in a better way. For example, a 2-rectangle tilted Haar-like feature can indicate the existence of an edge at 45°.

Messom and Barczak[5] extended the idea to a generic rotated Haar-like feature. Although the idea sounds mathematically sound, practical problems prevented the use of Haar-like features at any angle. In order to be fast, detection algorithms use low resolution images, causing rounding errors. For this reason, rotated Haar-like features are not commonly used.

The Viola–Jones object detection framework is the first object detection framework to provide competitive object detection rates in real-time proposed in 2001 by Paul Viola and Michael Jones[1][2]. Although it can be trained to detect a variety of object classes, it was motivated primarily by the problem of face detection. This algorithm is implemented in OpenCV as cvHaarDetectObjects().

The features employed by the detection framework universally involve the sums of image pixels within rectangular areas. As such, they bear some resemblance to Haar basis functions, which have been used previously in the realm of image-based object detection[3]. However, since the features used by Viola and Jones all rely on more than one rectangular area, they are generally more complex. The figure at right illustrates the four different types of features used in the framework. The value of any given feature is always simply the sum of the pixels within clear rectangles subtracted from the sum of the pixels within shaded rectangles. As is to be expected, rectangular features of this sort are rather primitive when compared to alternatives such as steerable filters. Although they are sensitive to vertical and horizontal features, their feedback is considerably coarser. However, with the use of an image representation called the integral image, rectangular features can be evaluated in constant time, which gives them a considerable speed advantage over their more sophisticated relatives. Because each rectangular area in a feature is always adjacent to at least one other rectangle, it follows that any two-rectangle feature can be computed in six array references, any three-rectangle feature in eight, and any four-rectangle feature in just nine.
[edit] Learning algorithm

The speed with which features may be evaluated does not adequately compensate for their number, however. For example, in a standard 24x24 pixel sub-window, there are a total of 45,396 possible features, and it would be prohibitively expensive to evaluate them all. Thus, the object detection framework employs a variant of the learning algorithm AdaBoost to both select the best features and to train classifiers that use them.
Cascade Architecture
[edit] Cascade architecture

The evaluation of the strong classifiers generated by the learning process can be done quickly, but it isn’t fast enough to run in real-time. For this reason, the strong classifiers are arranged in a cascade in order of complexity, where each successive classifier is trained only on those selected samples which pass through the preceding classifiers. If at any stage in the cascade a classifier rejects the sub-window under inspection, no further processing is performed and continue on searching the next sub-window (see figure at right). The cascade therefore has the form of a degenerate tree. In the case of faces, the first classifier in the cascade – called the attentional operator – uses only two features to achieve a false negative rate of approximately 0% and a false positive rate of 40%.[4] The effect of this single classifier is to reduce by roughly half the number of times the entire cascade is evaluated.

The cascade architecture has interesting implications for the performance of the individual classifiers. Because the activation of each classifier depends entirely on the behavior of its predecessor, the false positive rate for an entire cascade is:

F = \prod_{i=1}^K f_i.

Similarly, the detection rate is:

D = \prod_{i=1}^K d_i.

Thus, to match the false positive rates typically achieved by other detectors, each classifier can get away with having surprisingly poor performance. For example, for a 32-stage cascade to achieve a false positive rate of 10 − 6, each classifier need only achieve a false positive rate of about 65%. At the same time, however, each classifier needs to be exceptionally capable if it is to achieve adequate detection rates. For example, to achieve a detection rate of about 90%, each classifier in the aforementioned cascade needs to achieve a detection rate of approximately 99.7%.

## *Mathematical Model*

### *World Coordinate System*

## *Mathematical Model*

*Epipolar Geometry*
is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. These relations are derived based on the assumption that the cameras can be approximated by the pinhole camera model.

The figure below depicts two pinhole cameras looking at point X. In real cameras, the image plane is actually behind the focal point, and produces a rotated image. Here, however, the projection problem is simplified by placing a virtual image plane in front of the focal point of each camera to produce an unrotated image. OL and OR represent the focal points of the two cameras. X represents the point of interest in both cameras. Points xL and xR are the projections of point X onto the image planes.

Each camera captures a 2D image of the 3D world. This conversion from 3D to 2D is referred to as a perspective projection and is described by the pinhole camera model. It is common to model this projection operation by rays that emanate from the camera, passing through its focal point. Note that each emanating ray corresponds to a single point in the image.
[edit] Epipole or epipolar point

Since the two focal points of the cameras are distinct, each focal point projects onto a distinct point into the other camera's image plane. These two image points are denoted by eL and eR and are called epipoles or epipolar points. Both epipoles eL and eR in their respective image planes and both focal points OL and OR lie on a single 3D line.
[edit] Epipolar line

The line OL–X is seen by the left camera as a point because it is directly in line with that camera's focal point. However, the right camera sees this line as a line in its image plane. That line (eR–xR) in the right camera is called an epipolar line. Symmetrically, the line OR–X seen by the right camera as a point is seen as epipolar line eL–xLby the left camera.

An epipolar line is a function of the 3D point X, i.e. there is a set of epipolar lines in both images if we allow X to vary over all 3D points. Since the 3D line OL–X passes through camera focal point OL, the corresponding epipolar line in the right image must pass through the epipole eR (and correspondingly for epipolar lines in the left image). This means that all epipolar lines in one image must intersect the epipolar point of that image. In fact, any line which intersects with the epipolar point is an epipolar line since it can be derived from some 3D point X.
[edit] Epipolar plane

As an alternative visualization, consider the points X, OL & OR that form a plane called the epipolar plane. The epipolar plane intersects each camera's image plane where it forms lines—the

epipolar lines. All epipolar planes and epipolar lines intersect the epipole regardless of where X is located.
[edit] Epipolar constraint and triangulation
Epipolar geometry

If the relative translation and rotation of the two cameras is known, the corresponding epipolar geometry leads to two important observations

If the projection point xL is known, then the epipolar line eR–xR is known and the point X projects into the right image, on a point xR which must lie on this particular epipolar line. This means that for each point observed in one image the same point must be observed in the other image on a known epipolar line. This provides an epipolar constraint which corresponding image points must satisfy and it means that it is possible to test if two points really correspond to the same 3D point. Epipolar constraints can also be described by the essential matrix or the fundamental matrix between the two cameras.

If the points xL and xR are known, their projection lines are also known. If the two image points correspond to the same 3D point X the projection lines must intersect precisely at X. This means that X can be calculated from the coordinates of the two image points, a process called triangulation.

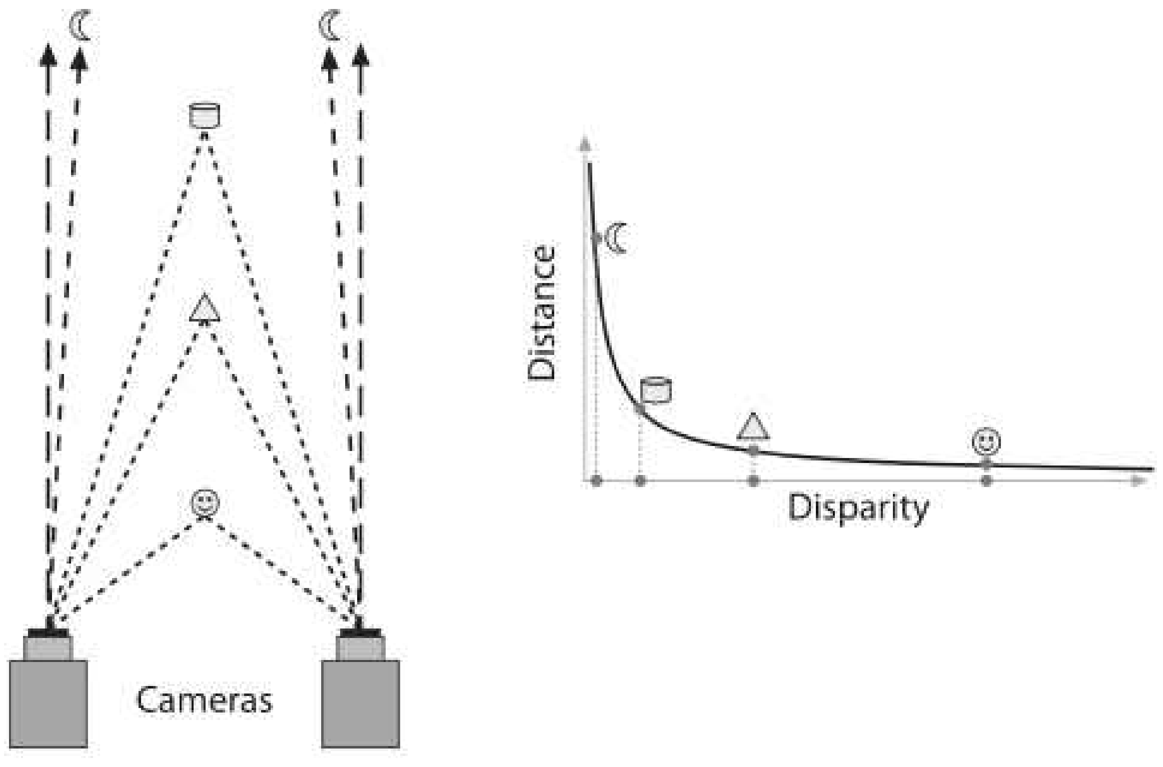
[edit] Simplified cases
Example of epipolar geometry. Two cameras, with their respective focal points OL and OR, observe a point P. The projection of P onto each of the image planes is denoted pL and pR. Points EL and ER are the epipoles.

The epipolar geometry is simplified if the two camera image planes coincide. In this case, the epipolar lines also coincide (EL–PL = ER–PR). Furthermore, the epipolar lines are parallel to the line OL–OR between the focal points, and can in practice be aligned with the horizontal axes of the two images. This means that for each point in one image, its corresponding point in the other image can be found by looking only along a horizontal line. If the cameras cannot be positioned in this way, the image coordinates from the cameras may be transformed to emulate having a common image plane. This process is called image rectification.
[edit] Epipolar geometry of pushbroom sensor

In contrast to the conventional frame camera which uses a two-dimensional CCD, pushbroom camera adopts an array of one-dimensional CCDs to produce long continuous image strip which is called "image carpet". Epipolar geometry of this sensor is quite different from that of frame cameras. First, the epipolar line of pushbroom sensor is not straight, but hyperbola-like curve. Second, epipolar 'curve' pair does not exist.[1]

## *Mathematical Model*

Disparity Mapping



Binocular disparity refers to the difference in image location of an object seen by the left and right eyes, resulting from the eyes' horizontal separation. The brain uses binocular disparity to extract depth information from the two-dimensional retinal images in stereopsis. In computer vision, binocular disparity refers to the difference in coordinates of similar features within two stereo images.

A similar disparity can be used in rangefinding by a coincidence rangefinder to determine distance and/or altitude to a target. In astronomy, the disparity between different locations on the Earth can be used to determine various celestial parallax, and Earth's orbit can be used for stellar parallax.

Human eyes are horizontally separated by about 50–75 mm (interpupillary distance) depending on each individual. Thus, each eye has a slightly different view of the world. This can be easily seen when alternately closing one eye while looking at a vertical edge. The binocular disparity can be observed from apparent horizontal shift of the vertical edge between both views.

At any given moment, the line of sight of the two eyes meet at a point in space. This point in space projects to the same location (i.e. the center) on the retinae of the two eyes. Because of the different viewpoints observed by the left and right eye however, many other points in space do not fall on corresponding retinal locations. Visual binocular disparity is defined as the difference between the point of projection in the two eyes and is usually expressed in degrees as the visual angle.[1]

Figure 1: The full black circle is the point of fixation. The blue object lies nearer to the observer. Therefore it has a "near" disparity dn. Objects lying more far away (green) correspondingly have a "far" disparity df. Binocular disparity is the angle between two lines of projection in one eye. One of which is the real projection from the object to the actual point of projection. The other one is the imaginary projection running through the focal point of the lens of the one eye to the point corresponding to the actual point of projection in the other eye. For simplicity reasons here both objects lie on the line of fixation for one eye such that the imaginary projection ends directly on the fovea of the other eye, but in general the fovea acts at most as a reference. Note that far disparities are smaller than near disparities for objects having the same distance from the fixation point.

In computer vision, binocular disparity is calculated from stereo images taken from a set of stereo cameras. The variable distance between these cameras, called the baseline, can affect the disparity of a specific point on their respective image plane. As the baseline increases, the disparity increases due to the greater angle needed to align the sight on the point. However, in computer vision, binocular disparity is referenced as coordinate differences of the point between the right and left images instead of a visual angle. The units are usually measured in pixels.
[edit] Tricking neurons with 2D images
Figure 2. Simulation of disparity from depth in the plane. (relates to Figure 1)

Brain cells (neurons) in a part of the brain responsible for processing visual information coming from the retinae (primary visual cortex) can detect the existence of disparity in their input from the eyes. Specifically, these neurons will be active, if an object with "their" special disparity lies within the part of the visual field to which they have access (receptive field).[2]

Researchers investigating precise properties of these neurons with respect to disparity present visual stimuli with different disparities to the cells and look whether they are active or not. One possibility to present stimuli with different disparities is to place objects in varying depth in front of the eyes. However, the drawback to this method may not be precise enough for objects placed further away as they possess smaller disparities while objects closer will have greater disparities. Instead, neuroscientists use an alternate method as schematised in Figure 2.

Figure 2: The disparity of an object with different depth than the fixation point can alternatively be produced by presenting an image of the object to one eye and a laterally shifted version of the same image to the other eye. The full black circle is the point of fixation. Objects in varying depths are placed along the line of fixation of the left eye. The same disparity produced from a shift in depth of an object (filled coloured circles) can also be produced by laterally shifting the object in constant depth in the picture one eye sees (black circles with coloured margin). Note that for near disparities the lateral shift has to be larger to correspond to the same depth compared with far disparities. This is what neuroscientists usually do with random dot stimuli to study disparity selectivity of neurons since the lateral distance required to test disparities is less than the distances required using depth tests. This principle has also been applied in autostereogram illusions.
[edit] Computing disparity using digital stereo images

The disparity of features between two stereo images are usually computed as a shift to the left of an image feature when viewed in the right image.[3] For example, a single point that appears at the x coordinate t (measured in pixels) in the left image may be present at the x coordinate t - 3 in the right image. In this case, the disparity at that location in the right image would be 3 pixels.

Stereo images may not always be correctly aligned to allow for quick disparity calculation. For example, the set of cameras may be slightly rotated off level. Through a process known as image rectification, both images are rotated to allow for disparities in only the horizontal direction (i.e. there is no disparity in the y image coordinates).[3] This is a property that can also be achieved by precise alignment of the stereo cameras before image capture.

## [edit] Computer algorithm

After rectification, the correspondence problem can be solved using an algorithm that scans both the left and right images for matching image features. A common approach to this problem is to form a smaller image patch around every pixel in the left image. These image patches are compared to all possible disparities in the right image by comparing their corresponding image patches. For example, for a disparity of 1, the patch in the left image would be compared to a similar-sized patch in the right, shifted to the left by one pixel. The comparison between these two patches can be made by attaining a computational measure from one of the following equations that compares each of the pixels in the patches. For all of the following equations, L and R refer to the right and left columns while r and c refer to the current row and column of either images being examined. "d" refers to the disparity of the right image.

Normalized correlation: \frac{\sum{\sum{ L(r,c) \cdot R(r,c) }}}{\sqrt{(\sum{\sum{ L(r,c)^2 }}) \cdot (\sum{\sum{ R(r,c)^2 }})}}

Sum of squared differences: \sum{\sum{ (L(r,c) - R(r,c-d))^2 }}

Sum of absolute differences: \sum{\sum{ \left | L(r,c) - R(r,c-d) \right \vert }}

The disparity with the lowest computed value using one of the above methods is considered the disparity for the image feature. This lowest score indicates that the algorithm has found the best match of corresponding features in both images.

The method described above is a brute-force search algorithm. With large patch and/or image sizes, this technique can be very time consuming as pixels are constantly being re-examined to find the lowest correlation score. However, this technique also involves unnecessary repetition as many pixels overlap. A more efficient algorithm involves remembering all values from the previous pixel. An even more efficient algorithm involves remembering column sums from the previous row (in addition to remembering all values from the previous pixel). Techniques that save previous information can greatly increase the algorithmic efficiency of this image analyzing process.

## [edit] Uses of disparity from images

Knowledge of disparity can be used in further extraction of information from stereo images. One case that disparity is most useful is for depth/distance calculation. Disparity and distance from the cameras are negatively correlated. As the distance from the cameras increases, the disparity decreases. This allows for depth perception in stereo images. Using geometry and algebra, the points that appear in the 2D stereo images can be mapped as coordinates in 3D space.

This concept is particularly useful for navigation. For example, the Mars Exploration Rover uses a similar method for scanning the terrain for obstacles.[4] The rover captures a pair of images with its stereoscopic navigation cameras and disparity calculations are performed in order to detect elevated objects (such as boulders).[5] Additionally, location and speed data can be extracted from subsequent stereo images by measuring the displacement of objects relative to the rover. In some cases, this is the best source of this type of information as the sensors in the wheels may be inaccurate due to slippage.

## *Mathematical Model*

RANSAC

RANSAC is an abbreviation for "RANdom SAmple Consensus". It is an iterative method to estimate parameters of a mathematical model from a set of observed data which contains outliers. It is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iterations are allowed. The algorithm was first published by Fischler and Bolles in 1981.

A basic assumption is that the data consists of "inliers", i.e., data whose distribution can be explained by some set of model parameters, and "outliers" which are data that do not fit the model. In addition to this, the data can be subject to noise. The outliers can come, e.g., from extreme values of the noise or from erroneous measurements or incorrect hypotheses about the interpretation of data. RANSAC also assumes that, given a (usually small) set of inliers, there exists a procedure which can estimate the parameters of a model that optimally explains or fits this data.

A simple example is fitting of a 2D line to a set of observations. Assuming that this set contains both inliers, i.e., points which approximately can be fitted to a line, and outliers, points which cannot be fitted to this line, a simple least squares method for line fitting will in general produce a line with a bad fit to the inliers. The reason is that it is optimally fitted to all points, including the outliers. RANSAC, on the other hand, can produce a model which is only computed from the inliers, provided that the probability of choosing only inliers in the selection of data is sufficiently high. There is no guarantee for this situation, however, and there are a number of algorithm parameters which must be carefully chosen to keep the level of probability reasonably high.

A data set with many outliers for which a line has to be fitted.

Fitted line with RANSAC, outliers have no influence on the result.

[edit] Overview

The input to the RANSAC algorithm is a set of observed data values, a parameterized model which can explain or be fitted to the observations, and some confidence parameters.

RANSAC achieves its goal by iteratively selecting a random subset of the original data. These data are hypothetical inliers and this hypothesis is then tested as follows:

A model is fitted to the hypothetical inliers, i.e. all free parameters of the model are reconstructed from the inliers.
All other data are then tested against the fitted model and, if a point fits well to the estimated

model, also considered as a hypothetical inlier.
    The estimated model is reasonably good if sufficiently many points have been classified as hypothetical inliers.
    The model is reestimated from all hypothetical inliers, because it has only been estimated from the initial set of hypothetical inliers.
    Finally, the model is evaluated by estimating the error of the inliers relative to the model.

This procedure is repeated a fixed number of times, each time producing either a model which is rejected because too few points are classified as inliers or a refined model together with a corresponding error measure. In the latter case, we keep the refined model if its error is lower than the last saved model.
[edit] The algorithm

The generic RANSAC algorithm, in pseudocode, works as follows:

```
input:
    data - a set of observations
    model - a model that can be fitted to data
    n - the minimum number of data required to fit the model
    k - the number of iterations performed by the algorithm
    t - a threshold value for determining when a datum fits a model
    d - the number of close data values required to assert that a model fits well to data
output:
    best_model - model parameters which best fit the data (or nil if no good model is found)
    best_consensus_set - data points from which this model has been estimated
    best_error - the error of this model relative to the data

iterations := 0
best_model := nil
best_consensus_set := nil
best_error := infinity
while iterations < k
    maybe_inliers := n randomly selected values from data
    maybe_model := model parameters fitted to maybe_inliers
    consensus_set := maybe_inliers

    for every point in data not in maybe_inliers
        if point fits maybe_model with an error smaller than t
            add point to consensus_set

    if the number of elements in consensus_set is > d
        (this implies that we may have found a good model,
        now test how good it is)
        this_model := model parameters fitted to all points in consensus_set
        this_error := a measure of how well this_model fits these points
        if this_error < best_error
            (we have found a model which is better than any of the previous ones,
            keep it until a better one is found)
            best_model := this_model
            best_consensus_set := consensus_set
            best_error := this_error

    increment iterations

return best_model, best_consensus_set, best_error
```

Possible variants of the RANSAC algorithm includes

Break the main loop if a sufficiently good model has been found, that is, one with sufficiently small error. May save some computation time at the expense of an additional parameter.
Compute this_error directly from maybe_model without re-estimating a model from the consensus set. May save some time at the expense of comparing errors related to models which are estimated from a small number of points and therefore more sensitive to noise.

## [edit] The parameters

The values of parameters t and d have to be determined from specific requirements related to the application and the data set, possibly based on experimental evaluation. The parameter k (the number of iterations), however, can be determined from a theoretical result. Let p be the probability that the RANSAC algorithm in some iteration selects only inliers from the input data set when it chooses the n points from which the model parameters are estimated. When this happens, the resulting model is likely to be useful so p gives the probability that the algorithm produces a useful result. Let w be the probability of choosing an inlier each time a single point is selected, that is,

w = number of inliers in data / number of points in data

A common case is that w is not well known beforehand, but some rough value can be given. Assuming that the n points needed for estimating a model are selected independently, wn is the probability that all n points are inliers and 1 − wn is the probability that at least one of the n points is an outlier, a case which implies that a bad model will be estimated from this point set. That probability to the power of k is the probability that the algorithm never selects a set of n points which all are inliers and this must be the same as 1 − p. Consequently,

$$1 - p = (1 - wn)k$$

which, after taking the logarithm of both sides, leads to

$$k = \frac{\log(1 - p)}{\log(1 - w^n)}$$

This result assumes that the n data points are selected independently, that is, a point which has been selected once is replaced and can be selected again in the same iteration. This is often not a reasonable approach and the derived value for k should be taken as an upper limit in the case that the points are selected without replacement. For example, in the case of finding a line which fits the data set illustrated in the above figure, the RANSAC algorithm typically chooses 2 points in each iteration and computes maybe_model as the line between the points and it is then critical that the two points are distinct.

To gain additional confidence, the standard deviation or multiples thereof can be added to k. The standard deviation of k is defined as

$$SD(k) = \frac{\sqrt{1 - w^n}}{w^n}$$

## [edit] Advantages and disadvantages

An advantage of RANSAC is its ability to do robust estimation of the model parameters, i.e., it can estimate the parameters with a high degree of accuracy even when a significant number of outliers are present in the data set. A disadvantage of RANSAC is that there is no upper bound on the time it takes to compute these parameters. When the number of iterations computed is limited the solution obtained may not be optimal, and it may not even be one that fits the data in a good way. In this way RANSAC offers a trade-off; by computing a greater number of iterations the probability of a reasonable model being produced is increased. Another disadvantage of RANSAC is that it requires the setting of problem-specific thresholds.

RANSAC can only estimate one model for a particular data set. As for any one-model approach when two (or more) model instances exist, RANSAC may fail to find either one. The Hough transform is an alternative robust estimation technique that may be useful when more than one model instance is present.

## ***Mathematical Model***

### Lukas Kanade Optical Flow

In computer vision, the Lucas–Kanade method is a widely used differential method for optical flow estimation developed by Bruce D. Lucas and Takeo Kanade. It assumes that the flow is essentially constant in a local neighbourhood of the pixel under consideration, and solves the basic optical flow equations for all the pixels in that neighbourhood, by the least squares criterion.[1][2]

By combining information from several nearby pixels, the Lucas-Kanade method can often resolve the inherent ambiguity of the optical flow equation. It is also less sensitive to image noise than point-wise methods. On the other hand, since it is a purely local method, it cannot provide flow information in the interior of uniform regions of the image.

The Lucas-Kanade method assumes that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point p under consideration. Thus the optical flow equation can be assumed to hold for all pixels within a window centered at p. Namely, the local image flow (velocity) vector (Vx,Vy) must satisfy

Ix(q1)Vx + Iy(q1)Vy = − It(q1)

Ix(q2)Vx + Iy(q2)Vy = − It(q2)

\vdots

Ix(qn)Vx + Iy(qn)Vy = − It(qn)

where q_1,q_2,\dots,q_n are the pixels inside the window, and Ix(qi),Iy(qi),It(qi) are the partial derivatives of the image I with respect to position x, y and time t, evaluated at the point qi and at the current time.

These equations can be written in matrix form Av = b, where

A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\[10pt] I_x(q_2) & I_y(q_2) \\[10pt] \vdots & \vdots \\[10pt] I_x(q_n) & I_y(q_n) \end{bmatrix}, \quad\quad v = \begin{bmatrix} V_x\\ [10pt] V_y \end{bmatrix}, \quad \mbox{and}\quad b = \begin{bmatrix} -I_t(q_1)\\ [10pt] -I_t(q_2)\\ [10pt] \vdots \\[10pt] -I_t(q_n) \end{bmatrix}

This system has more equations than unknowns and thus it is usually over-determined. The Lucas-Kanade method obtains a compromise solution by the least squares principle. Namely, it solves the 2×2 system

ATAv = ATb or
v = (ATA) − 1ATb

where AT is the transpose of matrix A. That is, it computes

$$\begin{bmatrix} V_x\\[10pt] V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i)I_y(q_i) \\[10pt] \sum_i I_x(q_i)I_y(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i)I_t(q_i) \\[10pt] -\sum_i I_y(q_i)I_t(q_i) \end{bmatrix}$$

with the sums running from i=1 to n.

The matrix ATA is often called the structure tensor of the image at the point p.
[edit] Weighted window

The plain least squares solution above gives the same importance to all n pixels qi in the window. In practice it is usually better to give more weight to the pixels that are closer to the central pixel p. For that, one uses the weighted version of the least squares equation,

ATWAv = ATWb

or

v = (ATWA) − 1ATWb

where W is an n×n diagonal matrix containing the weights Wii = wi to be assigned to the equation of pixel qi. That is, it computes

$$\begin{bmatrix} V_x\\[10pt] V_y \end{bmatrix} = \begin{bmatrix} \sum_i w_i I_x(q_i)^2 & \sum_i w_i I_x(q_i)I_y(q_i) \\[10pt] \sum_i w_i I_x(q_i)I_y(q_i) & \sum_i w_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i w_i I_x(q_i)I_t(q_i) \\[10pt] -\sum_i w_i I_y(q_i)I_t(q_i) \end{bmatrix}$$

The weight wi is usually set to a Gaussian function of the distance between qi and p.
[edit] Improvements and extensions

The least-squares approach implicitly assumes that the errors in the image data have a Gaussian distribution with zero mean. If one expects the window to contain a certain percentage of "outliers" (grossly wrong data values, that do not follow the "ordinary" Gaussian error distribution), one may use statistical analysis to detect them, and reduce their weight accordingly.

The Lucas-Kanade method per se can be used only when the image flow vector Vx,Vy between the two frames is small enough for the differential equation of the optical flow to hold, which is often less than the pixel spacing. When the flow vector may exceed this limit, such as in stereo matching or warped document registration, the Lucas-Kanade method may still be used to refine some coarse estimate of the same, obtained by other means; for example, by extrapolating the flow vectors computed for previous frames, or by running the Lucas-Kanade algorithm on reduced-scale versions of the images. Indeed, the latter method is the basis of the popular Kanade-Lucas-Tomasi (KLT) feature matching algorithm.

A similar technique can be used compute differential affine deformations of the image contents.

## ***Mathematical Model***

### Homography

Homography is a concept in the mathematical science of geometry. A homography is an invertible transformation from a projective space (for example, the real projective plane) to itself that maps straight lines to straight lines. Synonyms are collineation, projective transformation, and projectivity,[1] though "collineation" is also used more generally.

Formally, a projective transformation in a plane is a transformation used in projective geometry: it is the composition of a pair of perspective projections. It describes what happens to the perceived positions of observed objects when the point of view of the observer changes. Projective transformations do not preserve sizes or angles but do preserve incidence and cross-ratio: two properties which are important in projective geometry. Projectivities form a group.[1]

For more general projective spaces – of different dimensions or over different fields – "homography" means a projective linear transformation (an invertible transformation induced by a linear transformation of the associated vector space), while "collineation" (meaning "maps lines to lines") is more general, and includes both homographies and automorphic collineations (collineations induced by a field automorphism), as well as combinations of these.

## Computer vision applications

In the field of computer vision, any two images of the same planar surface in space are related by a homography (assuming a pinhole camera model). This has many practical applications, such as image rectification, image registration, or computation of camera motion—rotation and translation—between two images. Once camera rotation and translation have been extracted from an estimated homography matrix, this information may be used for navigation, or to insert models of 3D objects into an image or video, so that they are rendered with the correct perspective and appear to have been part of the original scene (see Augmented Reality).

## [edit] 3D plane to plane equation

We have two cameras a and b, looking at points Pi in a plane.

Passing the projections of Pi from bpi in b to a point api in a:

$${}^ap_i = K_a \cdot H_{ba} \cdot K_b^{-1} \cdot {}^bp_i$$

where Hba is

$$H_{ba} = R - \frac{t n^T}{d}$$

R is the rotation matrix by which b is rotated in relation to a; t is the translation vector from a to b; n and d are the normal vector of the plane and the distance to the plane respectively.

Ka and Kb are the cameras' intrinsic parameter matrices.

Homography-transl-bold.svg

The figure shows camera b looking at the plane at distance d.

Note: From above figure, nTPi is the projection of vector Pi into nT, and equal to d. So $\frac{t n^T}{d} P_i = t$. And we have HbaPi = RPi − t.

## [edit] Mathematical definition

In the complex plane, a Mobius transformation is frequently called a homography. These linear-fractional transformations are expressions of projective transformations on the complex projective line, an extension of the complex plane.

In higher dimensions Homogeneous coordinates are used to represent projective transformations by means of matrix multiplications. With Cartesian coordinates matrix multiplication cannot perform the division required for perspective projection. In other words, with Cartesian coordinates a perspective projection is a non-linear transformation.

Given:

$$p_a = \begin{bmatrix} x_a\\y_a\\1\end{bmatrix}, p^\prime_b = \begin{bmatrix} w^{\prime}x_b\\w^{\prime}y_{b}\\w^{\prime}\end{bmatrix}, \mathbf{H}_{ab} = \begin{bmatrix} h_{11}&h_{12}&h_{13}\\h_{21}&h_{22}&h_{23}\\h_{31}&h_{32}&h_{33} \end{bmatrix}$$

Then:

$$p^\prime_b = \mathbf{H}_{ab}p_a \,$$

where:

$$\mathbf{H}_{ba} = \mathbf{H}_{ab}^{-1}.$$

Also:

$$p_b = p^\prime_b/w^\prime = \begin{bmatrix} x_b\\y_b\\ 1\end{bmatrix}$$

[edit] Affine homography

When the image region in which the homography is computed is small or the image has been acquired with a large focal length, an affine homography is a more appropriate model of image displacements. An affine homography is a special type of a general homography whose last row is fixed to

## ***Mathematical Model***

### Simultaneous localization and mapping

**Simultaneous localization and mapping** (**SLAM**) is a technique used by robots and autonomous vehicles to build up a map within an unknown environment (without *a priori* knowledge), or to update a map within a known environment (with *a priori* knowledge from a given map), while at the same time keeping track of their current location.

Operational definition

Maps are used to determine a location within an environment and to depict an environment for planning and navigation; they support the assessment of actual location by recording information obtained from a form of perception and comparing it to a current set of perceptions. The benefit of a map in aiding the assessment of a location increases as the precision and quality of the current perceptions decrease. Maps generally represent the state at the time that the map is drawn; this is not necessarily consistent with the state of the environment at the time the map is used.

The complexity of the technical processes of locating and mapping under conditions of errors and noise do not allow for a coherent solution of both tasks. Simultaneous localization and mapping (SLAM) is a concept that binds these processes in a loop and therefore supports the continuity of both aspects in separated processes; iterative feedback from one process to the other enhances the results of both consecutive steps.

Mapping is the problem of integrating the information gathered by a set of sensors into a consistent model and depicting that information as a given representation. It can be described by the first characteristic question, What does the world look like? Central aspects in mapping are the representation of the environment and the interpretation of sensor data.

In contrast to this, localization is the problem of estimating the place (and pose) of the robot relative to a map; in other words, the robot has to answer the second characteristic question, Where am I? Typically, solutions comprise tracking, where the initial place of the robot is known, and global localization, in which no or just some a priori knowledge of the environmental characteristics of the starting position is given.

SLAM is therefore defined as the problem of building a model leading to a new map, or repetitively improving an existing map, while at the same time localizing the robot within that map. In practice, the answers to the two characteristic questions cannot be delivered independently of each other.

Before a robot can contribute to answering the question of what the environment looks like, given a set of observations, it needs to know e.g.:

- the robot's own kinematics,
- which qualities the autonomous acquisition of information has, and,
- from which sources additional supporting observations have been made.

It is a complex task to estimate the robot's current location without a map or without a directional reference.[1] Here, the location is just the position of the robot or might include, as well, its orientation.

## [edit] Technical problems

SLAM can be thought of as a chicken or egg problem: An unbiased map is needed for localization while an accurate pose estimate is needed to build that map. This is the starting condition for iterative mathematical solution strategies. In comparison, the atomic orbital model may be seen as a classic approach of how to communicate sufficient results under conditions of imprecise observation.

Beyond, the answering of the two characteristic questions is not as straightforward as it might sound due to inherent uncertainties in discerning the robot's relative movement from its various sensors. Generally, due to the budget of noise in a technical environment, SLAM is not served with just compact solutions, but with a bunch of physical concepts contributing to results.

If at the next iteration of map building the measured distance and direction traveled has a budget of inaccuracies, driven by limited inherent precision of sensors and additional ambient noise, then any features being added to the map will contain corresponding errors. Over time and motion, locating and mapping errors build cumulatively, grossly distorting the map and therefore the robot's ability to determine (know) its actual location and heading with sufficient accuracy.

There are various techniques to compensate for errors, such as recognizing features that it has come across previously, and re-skewing recent parts of the map to make sure the two instances of that feature become one. Some of the statistical techniques used in SLAM include Kalman filters, particle filters (aka. Monte Carlo methods) and scan matching of range data.

## [edit] Mapping

SLAM in the mobile robotics community generally refers to the process of creating geometrically consistent maps of the environment. Topological maps are a method of environment representation which capture the connectivity (i.e., topology) of the environment rather than creating a geometrically accurate map. As a result, algorithms that create topological maps are not referred to as SLAM.

SLAM is tailored to the available resources, hence not aimed at perfection, but at operational compliance. The published approaches are employed in unmanned aerial vehicles, autonomous underwater vehicles, planetary rovers, newly emerging domestic robots and even inside the human body.[2]

It is generally considered that "solving" the SLAM problem has been one of the notable achievements of the robotics research in the past decades.[3] The related problems of data association and computational complexity are amongst the problems yet to be fully resolved.

A significant recent advance in the feature based SLAM literature involved the re-examination the probabilistic foundation for Simultaneous Localisation and Mapping (SLAM) where it was posed in terms of multi-object Bayesian filtering with random finite sets that provide superior performance to leading feature-based SLAM algorithms in challenging measurement scenarios with high false alarm rates and high missed detection rates without the need for data association[4]. Algorithms were developed based on Probability Hypothesis Density (PHD) filtering techniques[5].

[edit] Sensing

SLAM will always use several different types of sensors to acquire data with statistically independent errors. Statistical independence is the mandatory requirement to cope with metric bias and with noise in measures.

Such optical sensors may be one dimensional (single beam) or 2D- (sweeping) laser rangefinders, 3D Flash LIDAR, 2D or 3D sonar sensors and one or more 2D cameras.

Recent approaches apply quasi-optical wireless ranging for multi-lateration (RTLS) or multi-angulation in conjunction with SLAM as a tribute to erratic wireless measures.

A special kind of SLAM for human pedestrians uses a shoe mounted inertial measurement unit as the main sensor and relies on the fact that pedestrians are able to avoid walls. This approach called FootSLAM can be used to automatically build floor plans of buildings that can then be used by an indoor positioning system.[6]

[edit] Locating

The results from sensing will feed the algorithms for locating. According to propositions of geometry, any sensing must include at least one lateration and (n+1) determining equations for an n-dimensional problem. In addition, there must be some additional a priori knowledge about orienting the results versus absolute or relative systems of coordinates with rotation and mirroring.

[edit] Modeling

Contribution to mapping may work in 2D modeling and respective representation or in 3D modeling and 2D projective representation as well. As a part of the model, the kinematics of the robot is included, to improve estimates of sensing under conditions of inherent and ambient noise. The dynamic model balances the contributions from various sensors, various partial error models and finally comprises in a sharp virtual depiction as a map with the location and heading of the robot as some cloud of probability. Mapping is the final depicting of such model, the map is either such depiction or the abstract term for the model.

[edit] Literature

A seminal work in SLAM is the research of R.C. Smith and P. Cheeseman on the representation and estimation of spatial uncertainty in 1986.[7][8] Other pioneering work in this field was conducted by the research group of Hugh F. Durrant-Whyte in the early 1990s.[9]

## ***Mathematical Model***

A* Path Finding

In computer science, A* (pronounced "A star" ( listen)) is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. Peter Hart, Nils Nilsson and Bertram Raphael first described the algorithm in 1968.[1] It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance (with respect to time) by using heuristics. However, when the heuristics are monotone, it is actually equivalent to running Dijkstra.

Description

A* uses a best-first search and finds the least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

the path-cost function, which is the cost from the starting node to the current node (usually denoted g(x))

and an admissible "heuristic estimate" of the distance to the goal (usually denoted h(x)).

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

If the heuristic h satisfies the additional condition h(x) \le d(x,y) + h(y) for every edge x, y of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once (see closed set below)—and A* is equivalent to running Dijkstra's algorithm with the reduced cost d'(x,y): = d(x,y) − h(x) + h(y).

Note that A* has been generalized into a bidirectional heuristic search algorithm; see bidirectional search.
[edit] History

In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A*. [2]
[edit]

As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.
[edit] Process

Like all informed search algorithms, it first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account; the g(x) part of the heuristic is the cost from the starting point, not simply the local cost from the previously expanded node.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set. The lower f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and h values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node. Additionally, if the heuristic is monotonic (or consistent, see below), a closed set of nodes already traversed may be used to make the search more efficient.

Implementation details

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths.

When a path is required at the end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. If these references are being kept then it can be important that the same node doesn't appear in the priority queue more than once (each entry corresponding to a different path to the node, and each with a different cost). A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path. When finding a node in a queue to perform this check, many standard implementations of a min-heap require O(n) time. Augmenting the heap with a hash table can reduce this to constant time.
[edit] Admissibility and optimality

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

Here is the main idea of the proof:

When A* terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

Suppose now that some other search algorithm B terminates its search with a path whose actual cost is not less than the estimated cost of a path through some open node. Based on the heuristic information it has, Algorithm B cannot rule out the possibility that a path through that node has a lower cost. So while B might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm.

This is only true if both:

A* uses an admissible heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic. See (Generalized best-first search strategies and the optimality of A*, Rina Dechter and Judea Pearl, 1985[3])

A* solves only one search problem rather than a series of similar search problems. Otherwise, A* is not guaranteed to expand fewer nodes than incremental heuristic search algorithms. See (Incremental heuristic search in artificial intelligence, Sven Koenig, Maxim Likhachev, Yaxin Liu and David Furcy, 2004[4])

Weighted A* is a variant of the A* algorithm that lets one speed-up a search at the expense of optimality. The basic idea is to "inflate" an admissible heuristic to speed up the search, and yet have a bound on the sub-optimality. If h_a(n) \,\! is an admissible heuristic function, in the weighted version of the A* search one uses h_w(n) = \epsilon h_a(n), \epsilon > 1 as the heuristic function, and perform the A* search as usual (which eventually happens faster than using h_a \,\! since fewer nodes are expanded). The path hence found by the search algorithm can have a cost of at most \epsilon \,\! times that of the least cost path in the graph.[5]
[edit] Complexity

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$| h(x) - h * (x) | = O(\log h * (x))$$

where h * is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the “perfect heuristic” h * that returns the true distance from x to the goal (see Pearl 1984[6] and also Russell and Norvig 2003, p. 101[7])

## *Mathematical Model*

### First-order Logic ***for a Wumpus Like World***

First-order logic is a formal logical system used in mathematics, philosophy, linguistics, and computer science. It goes by many names, including: first-order predicate calculus, the lower predicate calculus, quantification theory, and predicate logic (a less precise term). First-order logic is distinguished from propositional logic by its use of quantifiers; each interpretation of first-order logic includes a domain of discourse over which the quantifiers range. The adjective "first-order" is used to distinguish first-order theories from higher-order theories in which there are predicates having other predicates or functions as arguments or in which predicate quantifiers or function quantifiers are permitted or both.[1] In interpretations of first-order theories, predicates are associated with sets. In interpretations of higher order theories, they may be also associated with sets of sets.

There are many deductive systems for first-order logic that are sound (only deriving correct results) and complete (able to derive any logically valid implication). Although the logical consequence relation is only semidecidable, much progress has been made in automated theorem proving in first-order logic. First-order logic also satisfies several metalogical theorems that make it amenable to analysis in proof theory, such as the Löwenheim–Skolem theorem and the compactness theorem.

First-order logic is of great importance to the foundations of mathematics, where it has become the standard formal logic for axiomatic systems. It has sufficient expressive power to formalize two important mathematical theories: Zermelo–Fraenkel set theory (ZF) and first-order Peano arithmetic. However, no axiom system in first order logic is strong enough to fully (categorically) describe infinite structures such as the natural numbers or the real line. Categorical axiom systems for these structures can be obtained in stronger logics such as second-order logic.

While propositional logic deals with simple declarative propositions, first-order logic additionally covers predicates and quantification.

A predicate resembles a function that returns either True or False. Consider the following sentences: "Socrates is a philosopher", "Plato is a philosopher". In propositional logic these are treated as two unrelated propositions, denoted for example by p and q. In first-order logic, however, the sentences can be expressed in a more parallel manner using the predicate Phil(a), which asserts that the object represented by a is a philosopher. Thus if a represents Socrates then Phil(a) asserts the first proposition, p; if a instead represents Plato then Phil(a) asserts the second proposition, q. A key aspect of first-order logic is visible here: the string "Phil" is a syntactic entity which is given semantic meaning by declaring that Phil(a) holds exactly when a is a philosopher. An assignment of semantic meaning is called an interpretation.

First-order logic allows reasoning about properties that are shared by many objects, through the use of variables. For example, let Phil(a) assert that a is a philosopher and let Schol(a) assert that a is a scholar. Then the formula

$$\text{Phil}(a)\to \text{Schol}(a) \,$$

asserts that if a is a philosopher then a is a scholar. The symbol $\to$ is used to denote a conditional (if/then) statement. The hypothesis lies to the left of the arrow and the conclusion to the right. The truth of this formula depends on which object is denoted by a, and on the interpretations of "Phil" and "Schol".

Assertions of the form "for every a, if a is a philosopher then a is a scholar" require both the use of variables and the use of a quantifier. Again, let Phil(a) assert a is a philosopher and let Schol(a) assert that a is a scholar. Then the first-order sentence

$$\forall a ( \text{Phil}(a) \to \text{Schol}(a))$$

asserts that no matter what a represents, if a is a philosopher then a is scholar. Here $\forall$, the universal quantifier, expresses the idea that the claim in parentheses holds for all choices of a.

To show that the claim "If a is a philosopher then a is a scholar" is false, one would show there is some philosopher who is not a scholar. This counterclaim can be expressed with the existential quantifier $\exists$:

$$\exists a ( \text{Phil}(a) \land \lnot \text{Schol}(a)).$$

Here:

$\lnot$ is the negation operator: $\lnot \text{Schol}(a)$ is true if and only if $\text{Schol}(a) \,$ is false, in other words if and only if a is not a scholar.
$\land$ is the conjunction operator: $\text{Phil}(a) \land \lnot \text{Schol}(a)$ asserts that a is a philosopher and also not a scholar.

The predicates Phil(a) and Schol(a) take only one parameter each. First-order logic can also express predicates with more than one parameter. For example, "there is someone who can be fooled every time" can be expressed as:

\exists x (\mbox{Person}(x) \and \forall y (\mbox{Time}(y) \rightarrow \mbox{Canfool}(x,y))).

Here Person(x) is interpreted to mean x is a person, Time(y) to mean that y is a moment of time, and Canfool(x,y) to mean that (person) x can be fooled at (time) y. For clarity, this statement asserts that there is at least one person who can be fooled at all times, which is stronger than asserting that at all times at least one person exists who can be fooled. Asserting the latter (that there is always at least one foolable person) does not signify whether this foolable person is always the same for all moments of time.

The range of the quantifiers is the set of objects that can be used to satisfy them. (In the informal examples in this section, the range of the quantifiers was left unspecified.) In addition to specifying the meaning of predicate symbols such as Person and Time, an interpretation must specify a nonempty set, known as the domain of discourse or universe, as a range for the quantifiers. Thus a statement of the form \exists a \text{Phil}(a) is said to be true, under a particular interpretation, if there is some object in the domain of discourse of that interpretation that satisfies the predicate that the interpretation uses to assign meaning to the symbol Phil.
[edit] Syntax

There are two key parts of first order logic. The syntax determines which collections of symbols are legal expressions in first-order logic, while the semantics determine the meanings behind these expressions.
[edit] Alphabet

Unlike natural languages, such as English, the language of first-order logic is completely formal, so that it can be mechanically determined whether a given expression is legal. There are two key types of legal expressions: terms, which intuitively represent objects, and formulas, which intuitively express predicates that can be true or false. The terms and formulas of first-order logic are strings of symbols which together form the alphabet of the language. As with all formal languages, the nature of the symbols themselves is outside the scope of formal logic; they are often regarded simply as letters and punctuation symbols.

It is common to divide the symbols of the alphabet into logical symbols, which always have the same meaning, and non-logical symbols, whose meaning varies by interpretation. For example, the logical symbol \land always represents "and"; it is never interpreted as "or". On the other hand, a non-logical predicate symbol such as Phil(x) could be interpreted to mean "x is a philosopher", "x is a man named Philip", or any other unary predicate, depending on the interpretation at hand.
[edit] Logical symbols

There are several logical symbols in the alphabet, which vary by author but usually include:

The quantifier symbols \forall and \exists
The logical connectives: \land for conjunction, \lor for disjunction, \rightarrow for implication, \harr for biconditional, \lnot for negation. Occasionally other logical connective symbols are included. Some authors use \Rightarrow, or Cpq, instead of \rightarrow, and \Harr, or Epq, instead of \harr, especially in contexts where \to is used for other purposes. Moreover, \supset may replace \rightarrow; tilde (~), Np, or Fpq, may replace \lnot; Apq may replace \lor; and &, or Kpq, may replace \land, especially if these symbols are not available for technical reasons.
Parentheses, brackets, and other punctuation symbols. The choice of such symbols varies depending on context.
An infinite set of variables, often denoted by lowercase letters at the end of the alphabet x, y, z, … . Subscripts are often used to distinguish variables: x0, x1, x2, … .

An equality symbol (sometimes, identity symbol) =; see the section on equality below.

It should be noted that not all of these symbols are required - only one of the quantifiers, negation and conjunction, variables, brackets and equality suffice. There are numerous minor variations that may define additional logical symbols:

Sometimes the truth constants T, Vpq, or \top, for "true" and F, Opq, or \bot, for "false" are included. Without any such logical operators of valence 0, these two constants can only be expressed using quantifiers.
Sometimes additional logical connectives are included, such as the Sheffer stroke, Dpq (NAND), and exclusive or, Jpq.

[edit] Non-logical symbols

The non-logical symbols represent predicates (relations), functions and constants on the domain of discourse. It used to be standard practice to use a fixed, infinite set of non-logical symbols for all purposes. A more recent practice is to use different non-logical symbols according to the application one has in mind. Therefore it has become necessary to name the set of all non-logical symbols used in a particular application. This choice is made via a signature.[2]

The traditional approach is to have only one, infinite, set of non-logical symbols (one signature) for all applications. Consequently, under the traditional approach there is only one language of first-order logic.[3] This approach is still common, especially in philosophically oriented books.

For every integer n ≥ 0 there is a collection of n-ary, or n-place, predicate symbols. Because they represent relations between n elements, they are also called relation symbols. For each arity n we have an infinite supply of them:

Pn0, Pn1, Pn2, Pn3, …

For every integer n ≥ 0 there are infinitely many n-ary function symbols:

f n0, f n1, f n2, f n3, …

In contemporary mathematical logic, the signature varies by application. Typical signatures in mathematics are {1, ×} or just {×} for groups, or {0, 1, +, ×, <} for ordered fields. There are no restrictions on the number of non-logical symbols. The signature can be empty, finite, or infinite, even uncountable. Uncountable signatures occur for example in modern proofs of the Löwenheim-Skolem theorem.

In this approach, every non-logical symbol is of one of the following types.

A predicate symbol (or relation symbol) with some valence (or arity, number of arguments) greater than or equal to 0. These which are often denoted by uppercase letters P, Q, R,... .
Relations of valence 0 can be identified with propositional variables. For example, P, which can stand for any statement.
For example, P(x) is a predicate variable of valence 1. One possible interpretation is "x is a man".
Q(x,y) is a predicate variable of valence 2. Possible interpretations include "x is greater than y" and "x is the father of y".
A function symbol, with some valence greater than or equal to 0. These are often denoted by lowercase letters f, g, h,... .
Examples: f(x) may be interpreted as for "the father of x". In arithmetic, it may stand for "-x". In set theory, it may stand for "the power set of x". In arithmetic, g(x,y) may stand for "x+y". In set theory, it may stand for "the union of x and y".

Function symbols of valence 0 are called constant symbols, and are often denoted by lowercase letters at the beginning of the alphabet a, b, c,... . The symbol a may stand for Socrates. In arithmetic, it may stand for 0. In set theory, such a constant may stand for the empty set.

The traditional approach can be recovered in the modern approach by simply specifying the "custom" signature to consist of the traditional sequences of non-logical symbols.
[edit] Formation rules

The formation rules define the terms and formulas of first order logic. When terms and formulas are represented as strings of symbols, these rules can be used to write a formal grammar for terms and formulas. These rules are generally context-free (each production has a single symbol on the left side), except that the set of symbols may be allowed to be infinite and there may be many start symbols, for example the variables in the case of terms.
[edit] Terms

The set of terms is inductively defined by the following rules:

Variables. Any variable is a term.
Functions. Any expression f(t1,...,tn) of n arguments (where each argument ti is a term and f is a function symbol of valence n) is a term. In particular, symbols denoting individual constants are 0-ary function symbols, and are thus terms.

Only expressions which can be obtained by finitely many applications of rules 1 and 2 are terms. For example, no expression involving a predicate symbol is a term.
[edit] Formulas

## ***Hardware***

### ***Camera Sensors***

An active-pixel sensor (APS) is an image sensor consisting of an integrated circuit containing an array of pixel sensors, each pixel containing a photodetector and an active amplifier. There are many types of active pixel sensors including the CMOS APS used most commonly in cell phone cameras, web cameras and in some DSLRs. Such an image sensor is produced by a CMOS process (and is hence also known as a CMOS sensor), and has emerged as an alternative to charge-coupled device (CCD) imager sensors.

History

The term active pixel sensor was coined by Tsutomu Nakamura who worked on the Charge Modulation Device active pixel sensor at Olympus,[4] and more broadly defined by Eric Fossum in a 1993 paper.[5]

Image sensor elements with in-pixel amplifiers were described by Noble in 1968,[6] by Chamberlain in 1969,[7] and by Weimer et al. in 1969,[8] at a time when passive-pixel sensors – that is, pixel sensors without their own amplifiers – were being investigated as a solid-state alternative to vacuum-tube imaging devices. The MOS passive-pixel sensor used just a simple switch in the pixel to read out the photodiode integrated charge.[9] Pixels were arrayed in a two-dimensional structure, with access enable wire shared by pixels in the same row, and output wire shared by column. At the end of each column was an amplifier. Passive-pixel sensors suffered from many limitations, such as high noise, slow readout, and lack of scalability. The addition of an amplifier to each pixel addressed these problems, and resulted in the creation of the active-pixel sensor. Noble in 1968 and Chamberlain in 1969 created sensor arrays with active MOS readout amplifiers per pixel, in essentially the modern three-transistor configuration. The CCD was invented in 1970 at Bell Labs. Because the MOS process was so variable and MOS transistors had characteristics that changed over time (Vt instability), the CCD's charge-domain operation was more manufacturable and quickly eclipsed MOS passive and active pixel sensors. A low-resolution "mostly digital" N-channel MOSFET imager with intra-pixel amplification, for an optical mouse application, was demonstrated in 1981.[10]

Another type of active pixel sensor is the hybrid infrared focal plane array (IRFPA) designed to operate at cryogenic temperatures in the infrared spectrum. The devices are two chips that are put together like a sandwich: one chip contains detector elements made in InGaAs or HgCdTe, and the other chip is typically made of silicon and is used to readout the photodetectors. The exact date of origin of these devices is classified, but by the mid-1980s they were in widespread use.

By the late 1980s and early 1990s, the CMOS process was well established as a well controlled stable process and was the baseline process for almost all logic and microprocessors. There was a resurgence in the use of passive-pixel sensors for low-end imaging applications,[11] and active-pixel sensors for low-resolution high-function applications such as retina simulation[12] and high energy particle detector.[13] However, CCDs continued to have much lower temporal noise and fixed-pattern noise and were the dominant technology for consumer applications such as camcorders as well as for broadcast cameras, where they were displacing video camera tubes.

Eric Fossum, et al., invented the image sensor that used intra-pixel charge transfer along with an in-pixel amplifier to achieve true correlated double sampling (CDS) and low temporal noise operation, and on-chip circuits for fixed-pattern noise reduction, and published the first extensive article[5] predicting the emergence of APS imagers as the commercial successor of CCDs. Between 1993 and 1995, the Jet Propulsion Laboratory developed a number of prototype devices, which validated the key features of the technology. Though primitive, these devices demonstrated good image performance with high readout speed and low power consumption.

In 1995, personnel from JPL founded Photobit Corp., who continued to develop and commercialize APS technology for a number of applications, such as web cams, high speed and motion capture cameras, digital radiography, endoscopy (pill) cameras, DSLRs and of course, camera-phones. Many other small image sensor companies also sprang to life shortly thereafter due to the accessibility of the CMOS process and all quickly adopted the active pixel sensor approach.

The cameras used by GuarddoG are based on the OV7720/OV7221 CMOS VGA (640x480) CAMERACHIP Sensor , and are cheap and easy to find as they are the camera system used by the Playstation 3 Gaming Console

**Camera Sensor Key Specifications**

| | |
|---|---|
| Array Size | 640 x 480 |
| Power Supply Digital Core Voltage | 1.8VDC + 10% |
| Power Supply Analog Voltage | 3.0V to 3.3V |
| Power Supply I/O Voltage | 1.7V to 3.3V |
| Power Requirements - Active | 120 mW typical (60 fps VGA, YUV) |
| Power Requirements - Standby | < 20 µA |
| Temperature Range | -20°C to +70°C |

| | |
|---|---|
| Output Format (8-bit) | • YUV/YCbCr 4:2:2<br>• RGB565/555/444<br>• GRB 4:2:2<br>• Raw RGB Data |
| Lens Size | 1/4" |
| Max Image Transfer Rate | 60 fps for VGA |
| Scan Mode | Progressive |
| Electronic Exposure | Up to 510:1 (for selected fps) |
| Pixel Size | 6.0 µm x 6.0 µm |
| Fixed Pattern Noise | < 0.03% of VPEAK-TO-PEAK |
| Image Area | 3984 µm x 2952 µm |
| Package Dimensions | 5345 µm x 5265 µm |

## *Hardware*

### *Camera Synchronization*

Stereo vision on a mobile robot is a non-trivial problem that traditionally requires expensive hardware-synchronized cameras. Because standard stereo reconstruction assumes that the images from the left and right cameras are captured from a common scene , any motion that occurs between the left and right cameras capturing frames is equivalent to a change in the stereo camera`s baseline. This change in baseline invalidates the system`s extrinsic calibration , causing the quality of the rectification to decrease and the distances to be distorted by the robot`s velocity.

Hardware synchronization , the process of forcing two or more cameras to share a common hardware clock , has been traditionally limited to the professional stereo vision systems such as Point Grey`s Bumblebee product line . Thankfully , the inexpensive Playstation Eye camera is built on the same high-end OmniVision OV7720 chipset that is comparable to those found in many machine vision cameras. These cameras can be hardware-synchronized using the exposed frame clock input (FSIN) and output (VSYNC ) pins . By shorting one camera`s VSYNC pin to the others cameras FSIN pins the cameras are forced to share a common clock . To reduce the risk of a difference in ground potentials damaging the OV7720 delicate circuitry , each camera was also modified to share a common ground .

This hardware synchronization guarantees that all three cameras capture images simultaneously , but does not guarantee that the frames will travel retaining their synchronization on the USB .

Each camera has its own hardware clock and that means that in addition to the small distortion in space ( due to optics ) we have a small distortion in the fourth dimension , the axis of time. To tackle this problem guarddog uses cameras that have a very fast refresh rate of 120fps @ 320x240 pixels with a rewired shutter (FSIN , VSYNC pins ) in order for synchronization on the hardware side of the camera snapshots. A secondary problem is that there is non uniform latency over the USB cable and the USB host controller . This is problem is combated using direct frame grabbing via V4L2 and zero-copy passing by pointer to the beginning of the image pipelining and static linkage of the libraries consisting of the project to reduce delays and overheads.

## *Hardware*

### *USB Host*

A USB system has an asymmetric design, consisting of a host, a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. Additional USB hubs may be included in the tiers, allowing branching into a tree structure with up to five tier levels. A USB host may have multiple host controllers and each host controller may provide one or more USB ports. Up to 127 devices, including hub devices if present, may be connected to a single host controller.

USB devices are linked in series through hubs. There always exists one hub known as the root hub, which is built into the host controller.

A physical USB device may consist of several logical sub-devices that are referred to as device functions. A single device may provide several functions, for example, a webcam (video device function) with a built-in microphone (audio device function). Such a device is called a compound device in which each logical device is assigned a distinctive address by the host and all logical devices are connected to a built-in hub to which the physical USB wire is connected. A host assigns one and only one device address to a function.
Diagram: inside a device are several endpoints, each of which is connected by a logical pipes to a host controller. Data in each pipe flows in one direction, although there are a mixture going to and from the host controller.
USB endpoints actually reside on the connected device: the channels to the host are referred to as pipes

USB device communication is based on pipes (logical channels). A pipe is a connection from the host controller to a logical entity, found on a device, and named an endpoint. Because pipes correspond 1-to-1 to endpoints, the terms are sometimes used interchangeably. A USB device can have up to 32 endpoints: 16 into the host controller and 16 out of the host controller. The USB standard reserves one endpoint of each type, leaving a theoretical maximum of 30 for normal use. USB devices seldom have this many endpoints.

There are two types of pipes: stream and message pipes depending on the type of data transfer.

isochronous transfers: at some guaranteed data rate (often, but not necessarily, as fast as possible) but with possible data loss (e.g., realtime audio or video).
interrupt transfers: devices that need guaranteed quick responses (bounded latency) (e.g., pointing devices and keyboards).
bulk transfers: large sporadic transfers using all remaining available bandwidth, but with no guarantees on bandwidth or latency (e.g., file transfers).
control transfers: typically used for short, simple commands to the device, and a status response, used, for example, by the bus control pipe number 0.

A stream pipe is a uni-directional pipe connected to a uni-directional endpoint that transfers data using an isochronous, interrupt, or bulk transfer. A message pipe is a bi-directional pipe connected to a bi-directional endpoint that is exclusively used for control data flow. An endpoint is built into the USB device by the manufacturer and therefore exists permanently. An endpoint of a pipe is addressable with a tuple (device_address, endpoint_number) as specified in a TOKEN packet that the host sends when it wants to start a data transfer session. If the direction of the data transfer is from the host to the endpoint, an OUT packet (a specialization of a TOKEN packet) having the desired device address and endpoint number is sent by the host. If the direction of the data transfer is from the device to the host, the host sends an IN packet instead. If the destination endpoint is a uni-directional endpoint whose manufacturer's designated direction does not match the TOKEN packet (e.g., the manufacturer's designated direction is IN while the TOKEN packet is an OUT packet), the TOKEN packet will be ignored. Otherwise, it will be accepted and the data transaction can start. A bi-directional endpoint, on the other hand, accepts both IN and OUT packets.

## *Hardware*

### *Embedded System*

V4L2 mmap()
Name
v4l2-mmap -- Map device memory into application address space
Synopsis

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Arguments

start

Map the buffer to this address in the application's address space. When the MAP_FIXED flag is specified, start must be a multiple of the pagesize and mmap will fail when the specified address cannot be used. Use of this option is discouraged; applications should just specify a NULL pointer here.

uATX (6.75 inches by 6.75 inches [171.45 millimeters by 171.45 millimeters]) (ITX compatible)
Integrated Intel® Celeron® 220 processor (1.2 Ghz) with a 533 MHz system bus
One 240-pin DDR2 SDRAM Dual Inline Memory Module (DIMM) sockets
Support for DDR2 677/533/400 MHz DIMMs
Support for up to 1 GB of system memory
SiS* SiS662 Northbridge
SiS* SiS964L Southbridge
ADI* AD1888 audio codec
Integrated SiS Mirage* 1 graphic engine
Winbond* W83627DHG-B based Legacy I/O controller for hardware management, serial, parallel, and PS/2* ports

## *Booting from a USB STICK*

Most current PC firmware permits booting from a USB drive, allowing the launch of an operating system from a bootable flash drive. Such a configuration is known as a Live USB.

Original flash memory designs had very limited estimated lifetimes. The failure mechanism for flash memory cells is analogous to a metal fatigue mode; the device fails by refusing to write new data to specific cells that have been subject to many read-write cycles over the device's lifetime. Originally, this potential failure mode limited the use of "live USB" system to special purpose applications or temporary tasks, such as:

- Loading a minimal, hardened kernel for embedded applications (e.g. network router, firewall).
- Bootstrapping an operating system install or disk cloning operation, often across a network.

- Maintenance tasks, such as virus scanning or low-level data repair, without the primary host operating system loaded.

As of 2011, newer flash memory designs have much higher estimated lifetimes. Several manufacturers are now offering warranties of 5 years, or more. That should make the device more attractive for more applications. By reducing the probability of the device's premature failure, flash memory devices can now be considered for use where a magnetic disk would normally have been required. Flash drives have also experienced an exponential growth in their storage capacity over time (following the Moore's Law growth curve). As of 2011, single packaged devices with capacities of 64GB are readily available, and devices with 8GB capacity are very economical. Storage capacities in this range have traditionally been considered to offer adequate space, because they allow enough space for both the operating system software and some free space for the user's data.

## Advantages

Data stored on flash drives is impervious to scratches and dust, and flash drives are mechanically very robust making them suitable for transporting data from place to place and keeping it readily at hand. Most personal computers support USB as of 2010.

Flash drives also store data densely compared to many removable media. In mid-2009, 256 GB drives became available, with the ability to hold many times more data than a DVD or even a Blu-ray disc.

Compared to hard drives, flash drives use little power, have no fragile moving parts, and for most capacities are small and light.

Flash drives implement the USB mass storage device class so that most modern operating systems can read and write to them without installing device drivers. The flash drives present a simple block-structured logical unit to the host operating system, hiding the individual complex implementation details of the various underlying flash memory devices. The operating system can use any file system or block addressing scheme. Some computers can boot up from flash drives.

Specially manufactured flash drives are available that have a tough rubber or metal casing designed to be waterproof and virtually "unbreakable". These flash drives retain their memory after being submerged in water, and even through a machine wash. Leaving such a flash drive out to dry completely before allowing current to run through it has been known to result in a working drive with no future problems. Channel Five's Gadget Show cooked one of these flash drives with propane, froze it with dry ice, submerged it in various acidic liquids, ran over it with a jeep and fired it against a wall with a mortar. A company specializing in recovering lost data from computer drives managed to recover all the data on the drive.[38] All data on the other removable storage devices tested, using optical or magnetic technologies, were destroyed.

## [edit] Disadvantages

Main article: Flash memory#Limitations

Like all flash memory devices, flash drives can sustain only a limited number of write and erase cycles before the drive fails.[39][40] This should be a consideration when using a flash drive to run application software or an operating system. To address this, as well as space limitations, some developers have produced special versions of operating systems (such as Linux in Live USB)[41] or commonplace applications (such as Mozilla Firefox) designed to run from flash drives. These are typically optimized for size and configured to place temporary or intermediate files in the computer's main RAM rather than store them temporarily on the flash drive.

Most USB flash drives do not include a write-protect mechanism, although some have a switch on the housing of the drive itself to keep the host computer from writing or modifying data on the drive. Write-protection makes a device suitable for repairing virus-contaminated host computers without risk of infecting the USB flash drive itself.

A drawback to the small size is that they are easily misplaced, left behind, or otherwise lost. This is a particular problem if the data they contain are sensitive (see data security). As a consequence, some manufacturers have added encryption hardware to their drives—although software encryption systems which can be used in conjunction with any mass storage medium achieve the same thing.[citation needed] Most drives can be attached to keychains, necklaces and lanyards. The USB plug is usually fitted with a removable and easily lost protective cap, or is retractable.

USB flash drives are more expensive per unit of storage than large hard drives, but are less expensive in capacities of a few tens of gigabytes as of 2011.[42][43] Maximum available capacity is increasing with time, but is less than larger hard drives. This balance is changing, but the rate of change is slowing.

## *Hardware*

### *The Energy – Weight – Heat – Cost Problem*

## *Hardware*

### *GuarddoG Part List / Specifications*

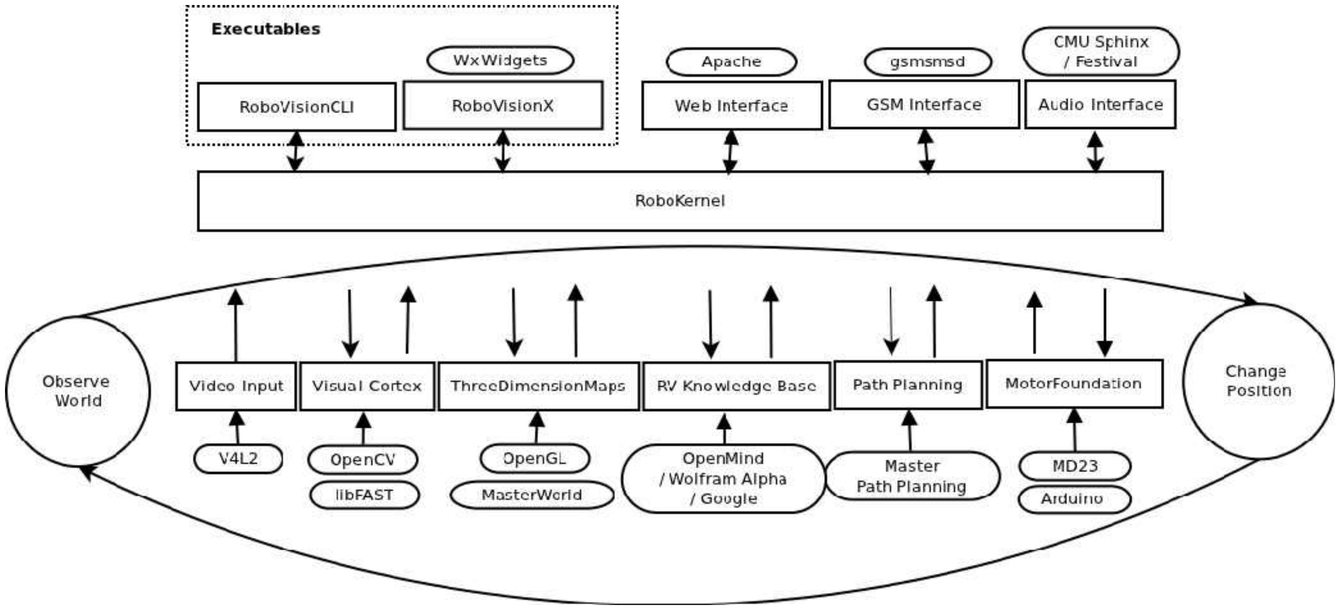
Embedded Electronics
1x Arduino = 25 euro ( Uno )
3x Infrared Led = 3 euro
1x RD-01 ( or RD-02 Devantech motors ) = 130 euro
2x Buttons ( power -on ) = 2 euro
2x Switches ( power supply ) = 2 euro
2x LED HeadLights = 10 euro
2x Ultrasonic Devantech SRF-05 with mounting = 40 euro
1x Dual Axis Accelerometer ( memsic 2125 ) = 30 euro
Total : 252 euro

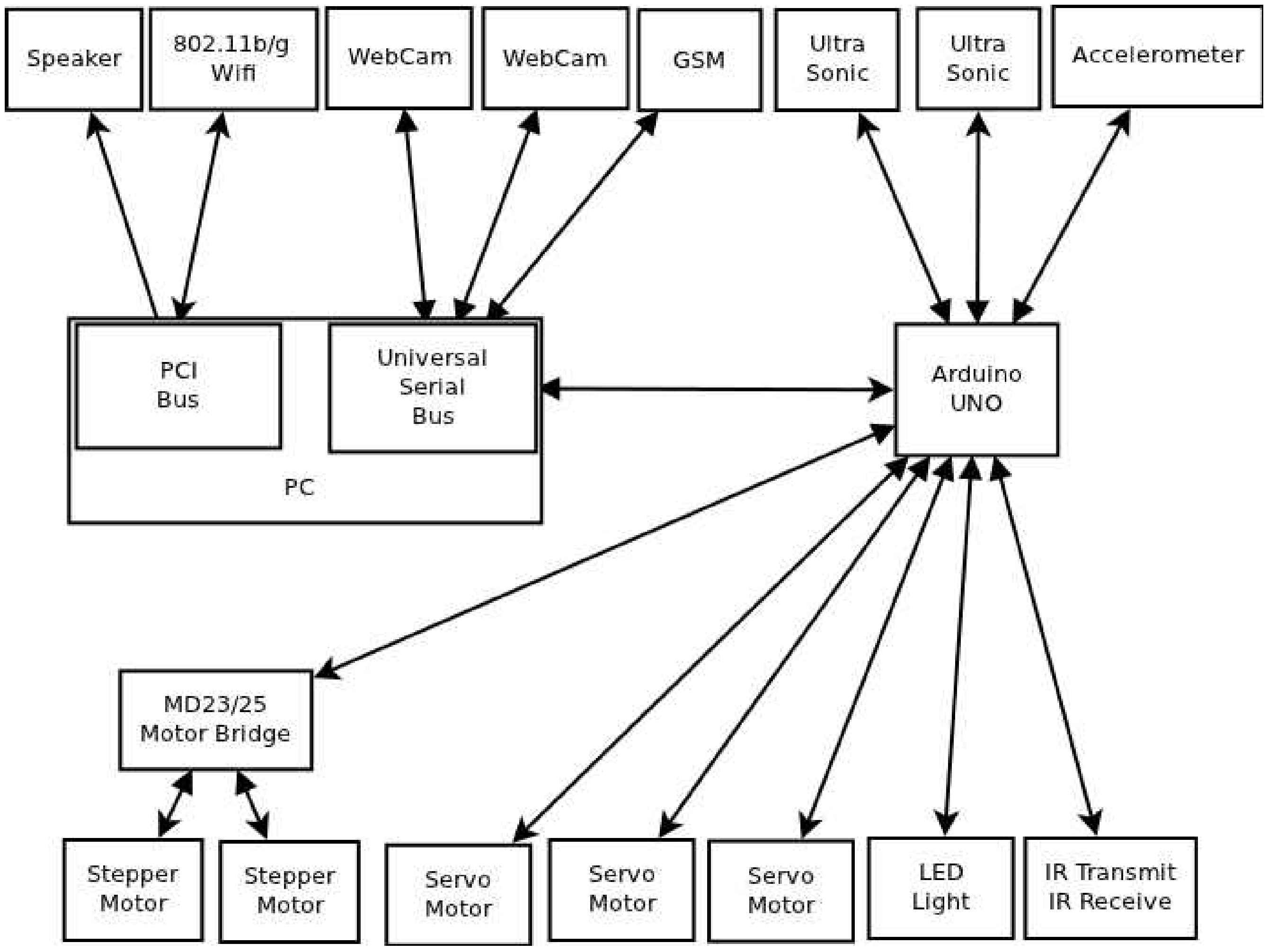
Computer Hardware

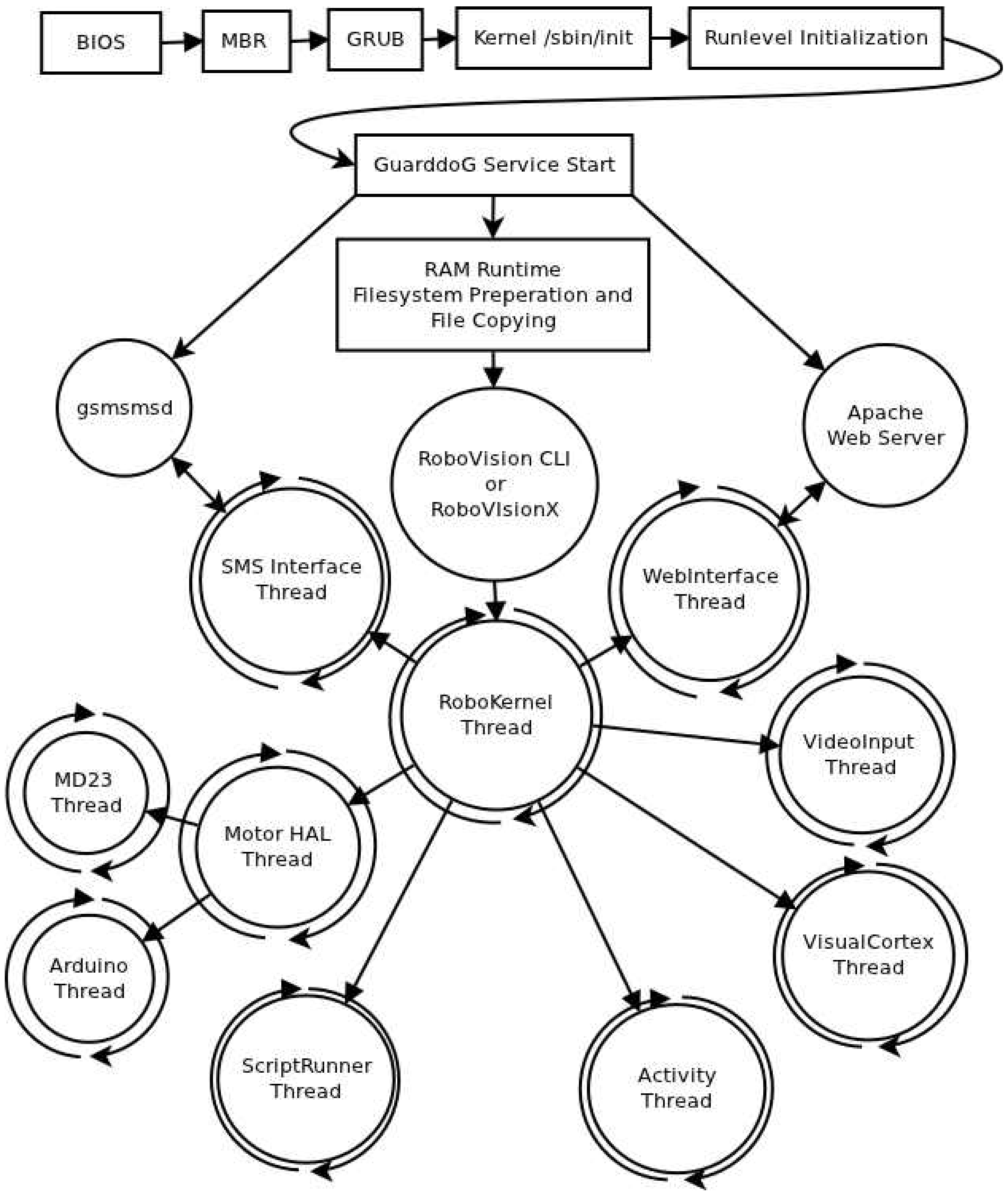
1x Fan = 5 euro
1x Mini-Itx Motherboard = 65-75 euro ( Currently on guarddog Intel D201GLY2 )
1x PicoPSU 90W = 45 euro
1x AC-DC 12 V Converter = 30 euro
2x Webcams ( On guarddog MS VX-6000 ) = 92 euro , LOGITECH C510 HD , PS3 Eyes
1x WIFI PCI card ( WG311T ) = 30 euro
1x USB Flash Drive 8GB + = 20 euro
1x 512-2048MB RAM DIMM ( on guarddog 512MB DDR2 ) = 30 euro
Total : 327 euro

## *Software Stack*

***Pipeline Outline***







***Software Stack***

***Performance Hypervisor***

## ***Software Stack***
***Implementation Framework***

## ***Software Stack***
***Statistics***

## ***The System in Practice***
***Installation***

## ***The System in Practice***
***Data Sets and Test Results***

## ***The System in Practice***
***Commercial Value***

## ***The System in Practice***
***Weaknesses***

## ***Future Work***
***Network Connectivity***

- ***NLP – AI Knowledge Base***
- ***Speech Recognition***
- ***Commercial Robots***
- ***CUDA / VLSI acceleration***

***Acknowledgements***