SCHOOL OF ELECTRONICS ENGINEERING
VIT - CHENNAI CAMPUS

CONVOLUTIONAL CODING AND VITERBI DECODER
ECE305 - DIGITAL COMMUNICATION

Project Submitted by:
Shraddha Agrawal – 13BEC1138
Prithvish V N – 13BEC1100
Aashish Alex - 12BEC10
Sujeet -12BEC11

Guided by:
Dr. Velmathi G.
SENSE – VIT Chennai

INDEX

ABSTRACT

The project is a replication of vending machines used in Metro localities such as Delhi, Mumbai, Bangalore and other cities abroad. Vending machine is an automated machine that dispenses products in exchange of money that is put into the system. In other words it's a money operated automated machine for selling merchandise. The task comprises of three stages – finite state machine, followed by sensing hardware and then finally product dispensing mechanism. The project is majorly based on the finite state machine to maneuver the system. The system senses two currency notes (Rs.10 and Rs.20) using SIFT algorithm. The system is efficient in dispensing products more like wafers and chocolates. As per the design the machine will dispense a product only for amount exactly Rs.30. The machine additionally consists of a cancelling mechanism which cancels the order and returns the amount back to the customer.

INTRODUCTION

Vending Machine:

The earliest known reference to a vending machine is in the work of Hero of Alexandria, a first-century AD Greek engineer and mathematician. His machine accepted a coin and then dispensed holy water. When the coin was deposited, it fell upon a pan attached to a lever. The lever opened a valve which let some water flow out. The pan continued to tilt with the weight of the coin until it fell off, at which point a counterweight snapped the lever up and turned off the valve.

The first modern coin-operated vending machines were introduced in London, England in the early 1880s, dispensing postcards. The machine was invented by Percival Everitt in 1883 and soon became a widespread feature at railway stations and post offices, dispensing envelopes, postcards, and notepaper. The Sweetmeat Automatic Delivery Company was founded in 1887 in England as the first company to deal primarily with the installation and maintenance of vending machines.

CHAPTER 1
Basic Model – State machine

1.1 Introduction

Finite State machine:-

A finite-state machine (FSM) is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are vending machines, which dispense products when the proper combination of coins is deposited, elevators, which drop riders off at upper floors before going down, traffic lights, which change sequence when cars are waiting, and combination locks, which require the input of combination numbers in the proper order.

1.2 BLOCK DIAGRAM

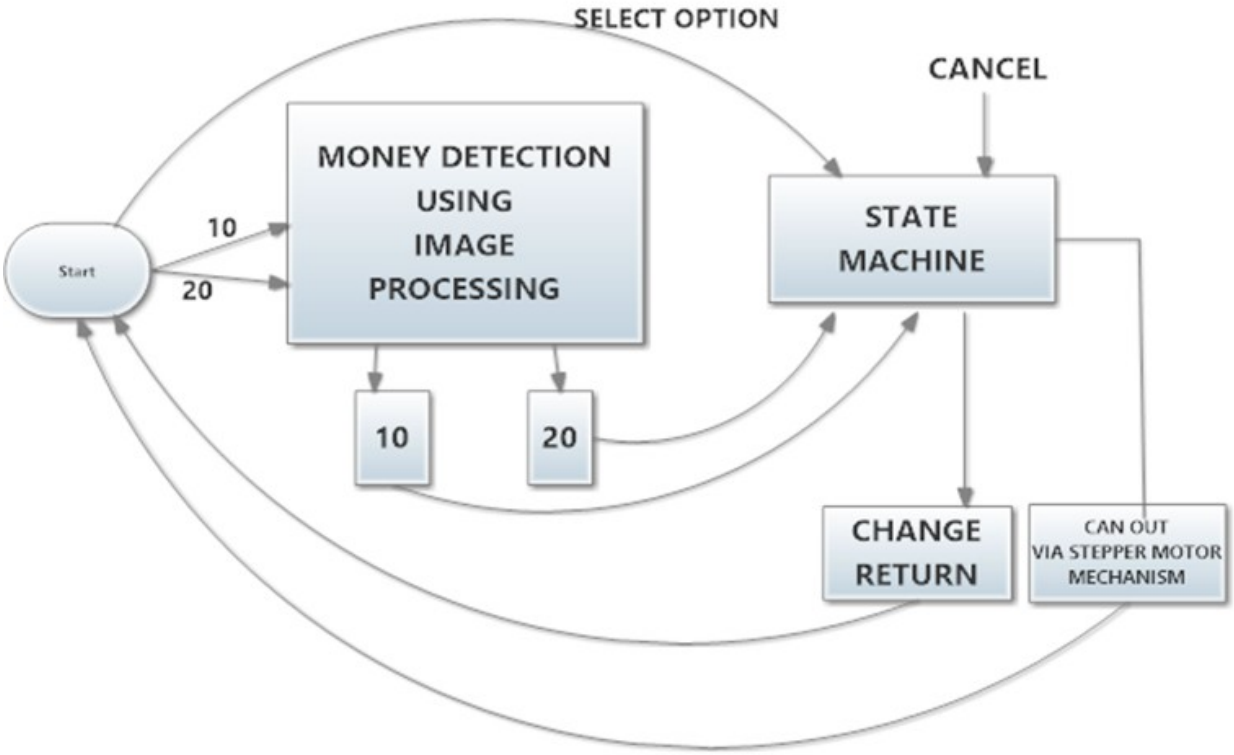


Fig 1.1

NOTE: The Machine takes in 10 Rs , 20 Rs , Cancel and choice as input and gives out coke/pepsi/limca cans and change return as output

1.3 TRANSITION TABLE

		INPUTS						OUTPUTS			
Present state		10 Rs	20 RS	Selectio -n	Canc -el	Next State		Peps i	Lim -ca	Return 10	Return 20
0	0	0	1	0	0	1	0	0	0	0	0
		0	1	1	0	1	0	0	0	0	0
		1	0	0	0	0	1	0	0	0	0
		1	0	1	0	0	1	0	0	0	0
0	1	0	0	0	1	0	0	0	0	1	0
		0	0	1	1	0	0	0	0	1	0
		1	0	0	0	1	0	0	0	0	0
		1	0	1	0	1	0	0	0	0	0
		0	1	0	0	0	0	1	0	0	0
		0	1	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	0	1
		0	0	1	1	0	0	0	0	0	1
		1	0	0	0	0	0	1	0	0	0
		1	0	1	0	0	0	0	1	0	0
		0	1	0	0	0	0	1	0	1	0
		0	1	1	0	0	0	0	1	1	0

1.4 EQUATIONS

R2 = q0 q1’ I1’ I2’ C
R1 = q0 q1’ I1’ I2 C + q0’ q1 I1’ I2’ C’
L = q0’ q1 I1’ I2 S C’ + q0 q1’ I1 I2’ S C’ + q0 q1’ I1’ I2 S C’
P = q0’ q1 I1’ I2 S’ C’ + q0 q1’ I1 I2’ S’ C’ + q0 q1’ I1’ I2 S’ C’
d1 = q0’ q1’ I1 I2’ C’
d0 = q0’ q1’ I1’ I2 C’ + q0’ q1 I1 I2’ C’

Where,

R1 => Return 10 Rs
R2 => Return 20 Rs
L => Limca
P => Pepsi
I1 => Input 10 Rs
I2 => Input 20 Rs
q0 => Present state bit 0
q1 => Present state bit 1
d0 => Next state bit 0
d1 => Next state bit 1

1.5 STATE DIAGRAM

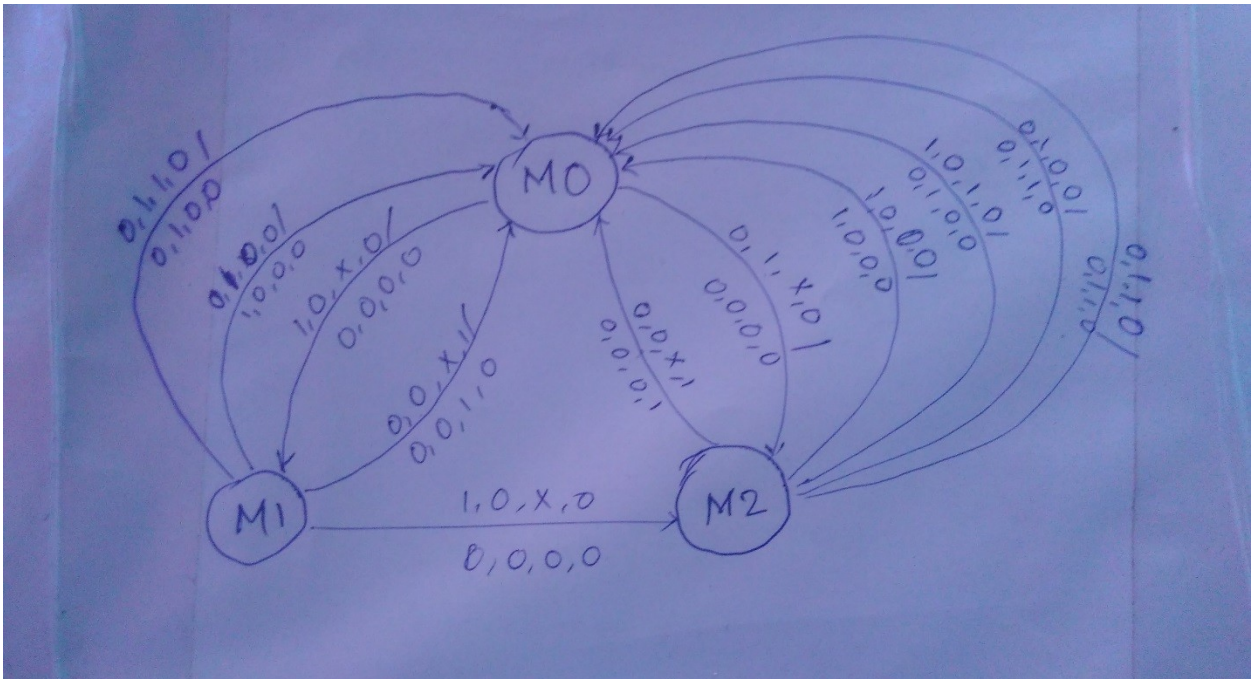


Fig 1.2

1.5 CIRCUIT DIAGRAM

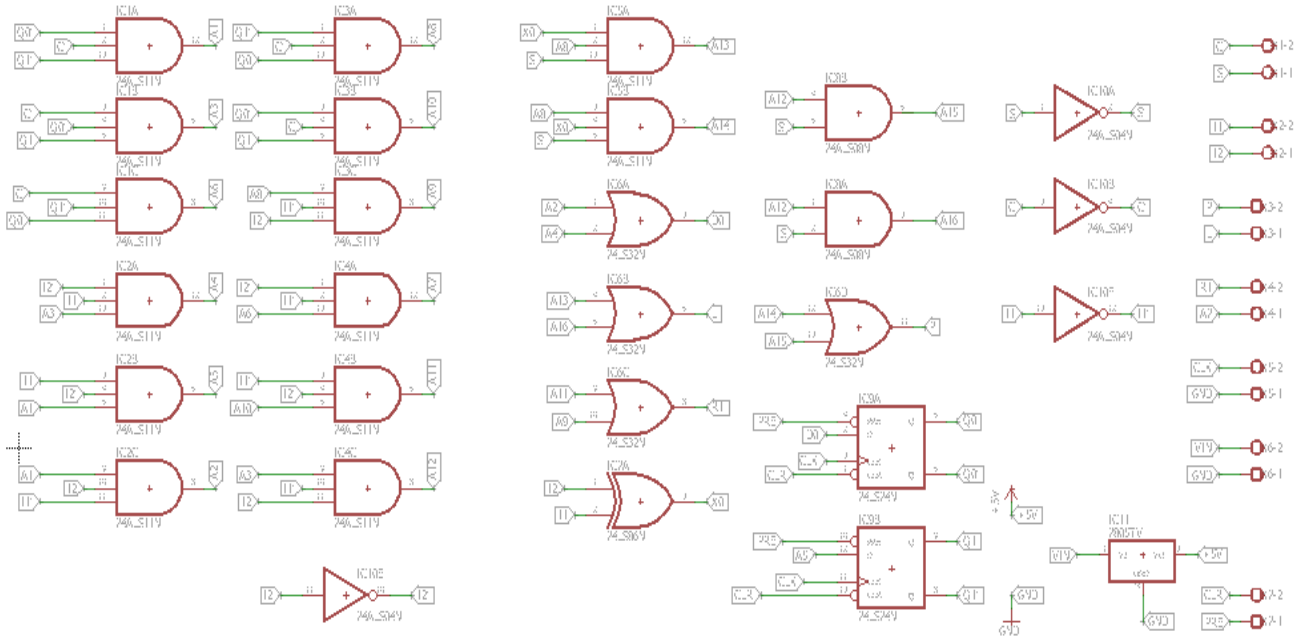


Fig 1.3

CHAPTER 2

SENSING MECHANISM

2.1 BLOCK DIAGRAM

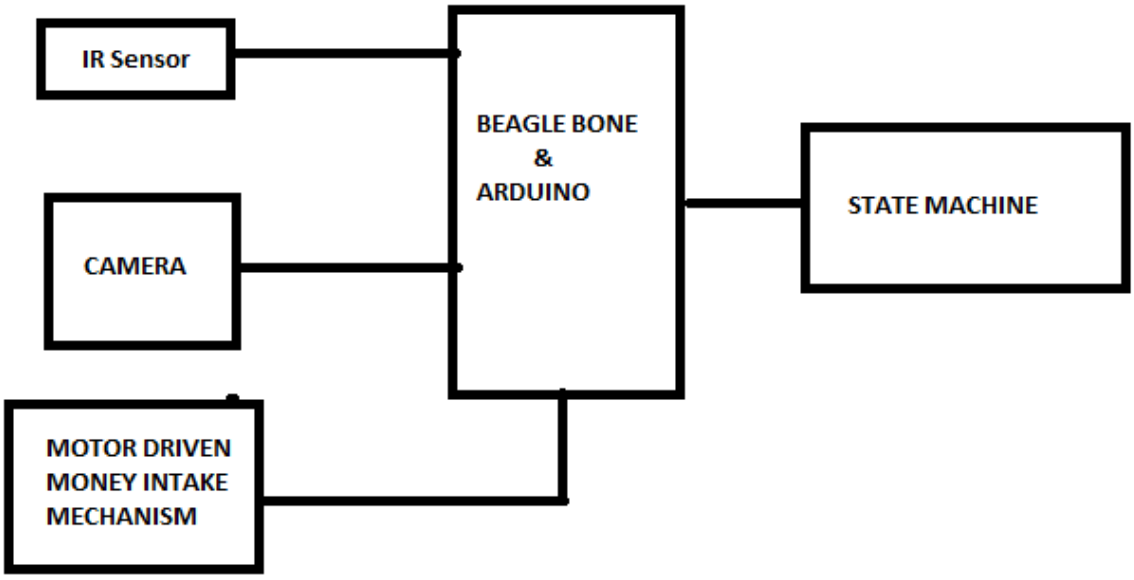


Fig 2.1

2.2 SIFT ALGORITHM

Matching features across different images in a common problem in computer vision. When all images are similar in nature (same scale, orientation, etc) simple corner detectors can work. But when you have images of different scales and rotations, the tool used in computer vision is Scale Invariant Feature Transform.

The algorithm

- **Constructing a scale space** Create internal representations of the original image to ensure scale invariance. The following is done by generating a "scale space".
- **LoG Approximation** The Laplacian of Gaussian is great mathematical tool for finding interesting points (or key points) in an image. But it's computationally expensive. So we cheat and approximate it using the representation created earlier.
- **Finding key points** With the super fast approximation, we now try to find key points. These are maxima and minima in the Difference of Gaussian image we calculate in step 2
- **Get rid of bad key points** Edges and low contrast regions are bad key points. Eliminating these makes the algorithm efficient and robust. A technique similar to the Harris Corner Detector is used here.
- **Assigning an orientation to the key points** An orientation is calculated for each key point. Any further calculations are done relative to this orientation. This effectively cancels out the effect of orientation, making it rotation invariant.
- **Generate SIFT features** finally, with scale and rotation invariance in place, one more representation is generated. This helps uniquely identify features.

Scale spaces

In this process , one gets rid of some detail from the image i.e if one has to focus on a tree then details in a tree such as the leaves, twigs, etc intentionally blured**.** While getting rid of these details, one must ensure that one does not introduce new false details. The only way to do that is with the Gaussian Blur

SIFT takes scale spaces to the next level. The original image is taken progressively blurred out images are generated. Resize the original image to half size. And then again blurred out images are generated and this keeps repeating.

Technical approach towards scale spaces

The number of octaves and scale depends on the size of the original image. While programming SIFT, the octaves and scales can be varied according to the user. However, the creator of SIFT suggests that 4 octaves and 5 blur levels are ideal for the algorithm.

The first octave

If the original image is doubled in size and antialiased a bit (by blurring it) then the algorithm produces more four times more key points. The more the key points, the better

Blurring

Mathematically, "blurring" is referred to as the convolution of the Gaussian operator and the image. Gaussian blur has a particular expression or "operator" that is applied to each pixel. What results is the blurred image.

- L is a blurred image
- G is the Gaussian Blur operator
- I is an image
- x,y are the location coordinates
- σ is the "scale" parameter. It can be termed as the amount of blur. Greater the value, greater the blur.
- The * is the convolution operation in x and y. It "applies" Gaussian blur G onto the image I.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

This is the actual Gaussian Blur operator.

Amount of blurring

The amount of blurring in each image is important. It goes like this. Assume the amount of blur in a particular image is σ. Then, the amount of blur in the next image will be k*σ. Here k is whatever constant you choose.

	scale →				
octave	0.707107	1.000000	1.414214	2.000000	2.828427
	1.414214	2.000000	2.828427	4.000000	5.656854
	2.828427	4.000000	5.656854	8.000000	11.313708
	5.656854	8.000000	11.313708	16.000000	22.627417

This is a table of σ's for my current example. See how each σ differs by a factor sqrt(2) from the previous one.

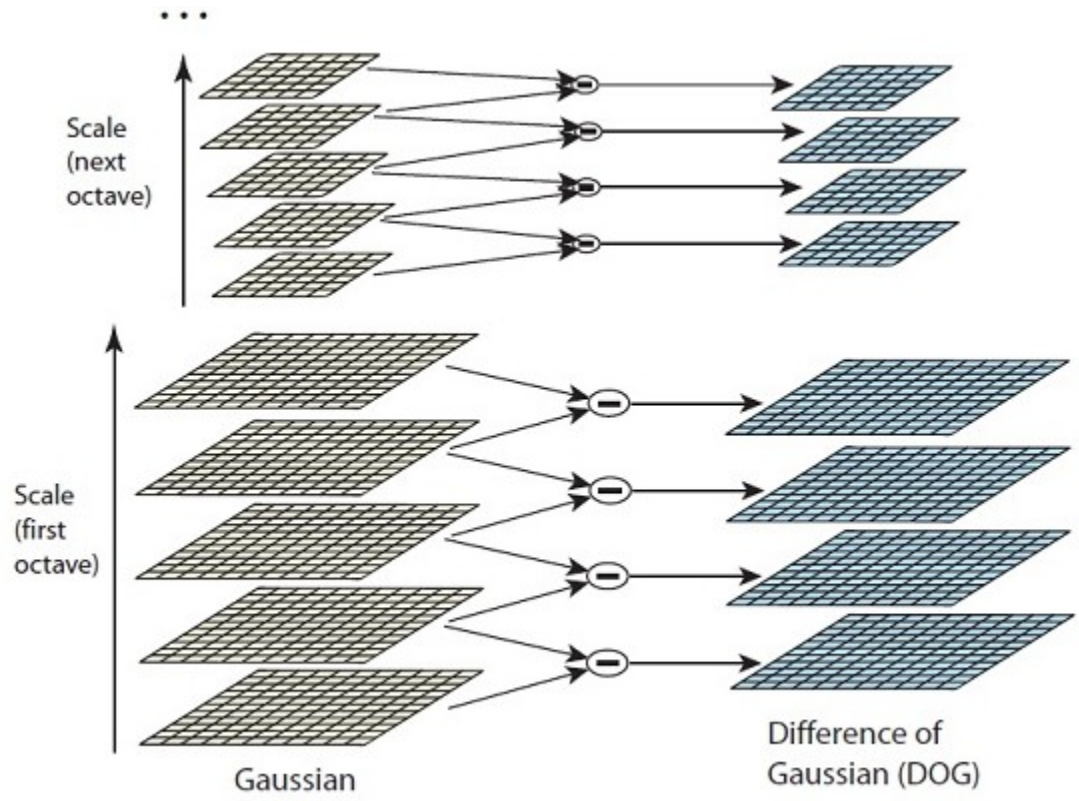
Laplacian of Gaussian

The idea was to blur an image progressively, shrink it, blur the small image progressively and so on. Now those blurred images are used to generate another set of images, the Difference of Gaussians (DoG). These DoG images are a great for finding out interesting key points in the image.

The Laplacian of Gaussian (LoG) operation goes like this. You take an image, and blur it a little. And then, you calculate second order derivatives on it (or, the "laplacian"). This locates edges and corners on the image. These edges and corners are good for finding key points.

But the second order derivative is extremely sensitive to noise. The blur smoothes it out the noise and stabilizes the second order derivative.

The problem is, calculating all those second order derivatives is computationally intensive. So we cheat a bit.



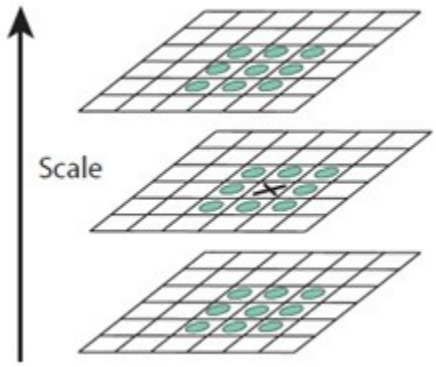
These Difference of Gaussian images are approximately equivalent to the Laplacian of Gaussian. And we've replaced a computationally intensive process with a simple subtraction (fast and efficient)

Finding key points

Finding key points is a two part process

- Locate maxima/minima in DoG images
- Find subpixel maxima/minima

The first step is to coarsely locate the maxima and minima. This can be done by iterating through each pixel and check all its neighbors. The check is done within the current image, and also the one above and below it.

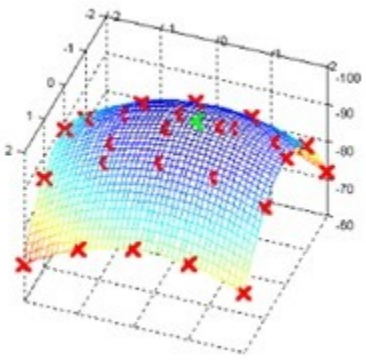


X marks the current pixel. The green circles mark the neighbours. This way, a total of 26 checks are made. X is marked as a "key point" if it is the greatest or least of all 26 neighbours.

Usually, a non-maxima or non-minima position won't have to go through all 26 checks. A few initial checks will usually sufficient to discard it.

Note that key points are not detected in the lowermost and topmost scales. There simply aren't enough neighbours to do the comparison.

Once this is done, the marked points are the approximate maxima and minima. They are "approximate" because the maxima/minima almost never lies exactly on a pixel. It lies somewhere between the pixel. But we simply cannot access data "between" pixels. So, one must mathematically locate the subpixel location.



Using the available pixel data, subpixel values are generated. This is done by the Taylor expansion of the image around the approximate key point.

Mathematically, it's like this:

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

One can easily find the extreme points of this equation (differentiate and equate to zero). On solving, one will get subpixel key point locations. These subpixel values increase chances of matching and stability of the algorithm.

2.3 Beagle bone Black

The BeagleBoard is a low-power open-source hardware single-board computer produced by Texas Instruments in association with Digi-Key and Newark element14. The BeagleBoard was also designed with open source software development in mind, and as a way of demonstrating the Texas Instrument's OMAP3530 system-on-a-chip.The board was developed by a small team of engineers as an educational board that could be used in colleges around the world to teach open source hardware and software capabilities. It is also sold to the public under the Creative Commons share-alike license. The board was designed using Cadence OrCAD for schematics and Cadence Allegro for PCB manufacturing; no simulation software was used.

Specifications

- Package on Package POP CPU/memory chip.
- Processor TI DM3730 Processor - 1 GHz ARM Cortex-A8 core
- 'HD capable' TMS320C64x+ core (800 MHz up to 720p @30 fps)
- Imagination Technologies PowerVR SGX 2D/3D graphics processor supporting dual independent displays
- 512 MB LPDDR RAM
- 4 GB microSD card supplied with the BeagleBoard-xM and loaded with The Angstrom Distribution
- Peripheral connections
- DVI-D (HDMI connector chosen for size - maximum resolution is 1400x1050)
- S-Video
- USB OTG (mini AB)
- 4 USB ports
- Ethernet port
- MicroSD/MMC card slot
- Stereo in and out jacks
- RS-232 port
- JTAG connector
- Power socket (5 V barrel connector type)
- Camera port
- Expansion port
- Boot code stored on the uSD card
- Boot from uSD/MMC only
- Alternative Boot source button.

- Has been demonstrated using Android,[18] Angstrom Linux,[19] Fedora, Ubuntu, Gentoo,[20] Arch Linux ARM[21] and Maemo Linux distributions,[22] FreeBSD, [30] the Windows CE operating system,[24] and RISC OS.

The following board is loaded with a debian image (a Linux environment similar to Ubuntu fedora available for beagle bone) and then was used to perform the SIFT algorithm using the OPENCV LIBRARY for Computer vision.

2.4 Code

```
"""
Python module for use with David Lowe's SIFT code available at:
http://www.cs.ubc.ca/~lowe/keypoints/
adapted from the matlab code examples.

Jan Erik Solem, 2009-01-30
"""

from __future__ import division
import os
from numpy import *
import pylab
import sift
from PIL import Image

def process_image(imagename, resultname):
    """ process an image and save the results in a .key ascii file"""

    #check if linux or windows
    if os.name == "posix":
        cmmd = "./sift <"+imagename+">"+resultname
    else:
        cmmd = "siftWin32 <"+imagename+">"+resultname

    os.system(cmmd)
    print 'processed', imagename

def read_features_from_file(filename):
    """ read feature properties and return in matrix form"""

    f = open(filename, 'r')
    header = f.readline().split()

    num = int(header[0]) #the number of features
    featlength = int(header[1]) #the length of the descriptor
    if featlength != 128: #should be 128 in this case
        raise RuntimeError, 'Keypoint descriptor length invalid (should be
128).'

    locs = zeros((num, 4))
    descriptors = zeros((num, featlength));

    #parse the .key file
    e =f.read().split() #split the rest into individual elements
    pos = 0
    for point in range(num):
        #row, col, scale, orientation of each feature
        for i in range(4):
            locs[point,i] = float(e[pos+i])
        pos += 4

        #the descriptor values of each feature
        for i in range(featlength):
            descriptors[point,i] = int(e[pos+i])
```

```
        #print descriptors[point]
        pos += 128

        #normalize each input vector to unit length
        descriptors[point] = descriptors[point] /
linalg.norm(descriptors[point])
        #print descriptors[point]

    f.close()

    return locs,descriptors

def match(desc1,desc2):
    """ for each descriptor in the first image, select its match to second
image
        input: desc1 (matrix with descriptors for first image),
        desc2 (same for second image)"""

    dist_ratio = 0.6
    desc1_size = desc1.shape

    matchscores = zeros((desc1_size[0],1))
    desc2t = desc2.T #precompute matrix transpose
    for i in range(desc1_size[0]):
        dotprods = dot(desc1[i,:],desc2t) #vector of dot products
        dotprods = 0.9999*dotprods
        #inverse cosine and sort, return index for features in second image
        indx = argsort(arccos(dotprods))

        #check if nearest neighbor has angle less than dist_ratio times 2nd
        if arccos(dotprods)[indx[0]] < dist_ratio * arccos(dotprods)[indx[1]]:
            matchscores[i] = indx[0]

    return matchscores

def plot_features(im,locs):
    """ show image with features. input: im (image as array),
        locs (row, col, scale, orientation of each feature) """

    pylab.gray()
    pylab.imshow(im)
    pylab.plot([p[1] for p in locs], [p[0] for p in locs], 'ob')
    pylab.axis('off')
    pylab.show()

def appendimages(im1,im2):
    """ return a new image that appends the two images side-by-side."""

    #select the image with the fewest rows and fill in enough empty rows
    rows1 = im1.shape[0]
    rows2 = im2.shape[0]

    if rows1 < rows2:
        im1 = concatenate((im1,zeros((rows2-rows1,im1.shape[1]))), axis=0)
    else:
        im2 = concatenate((im2,zeros((rows1-rows2,im2.shape[1]))), axis=0)
```

```
    return concatenate((im1,im2), axis=1)

def plot_matches(im1,im2,locs1,locs2,matchscores):
    """ show a figure with lines joining the accepted matches in im1 and im2
        input: im1,im2 (images as arrays), locs1,locs2 (location of features),
        matchscores (as output from 'match'). """

    im3 = appendimages(im1,im2)

    pylab.gray()
    pylab.imshow(im3)

    cols1 = im1.shape[1]
    for i in range(len(matchscores)):
        if matchscores[i] > 0:
            pylab.plot([locs1[i,1], locs2[int(matchscores[i]),1]+cols1],
[locs1[i,0], locs2[int(matchscores[i]),0]], 'c')
    pylab.axis('off')
    pylab.show()


def make_score(matches):
    return sum(matches)/len(matches)


sift.process_image('basmati.pgm', 'basmati.key')
l1,d1 = sift.read_features_from_file('basmati.key')
sift.process_image('scene.pgm', 'scene.key')
l2,d2 = sift.read_features_from_file('scene.key')
x=match(d1,d2)
im1 = array(Image.open('basmati.pgm'))
im2 = array(Image.open('scene.pgm'))
plot_matches(im1,im2,l1,l2,x)
```

print make_score(x)

CHAPTER 3
Product Dispensary Mechanism

3.1 Design Block

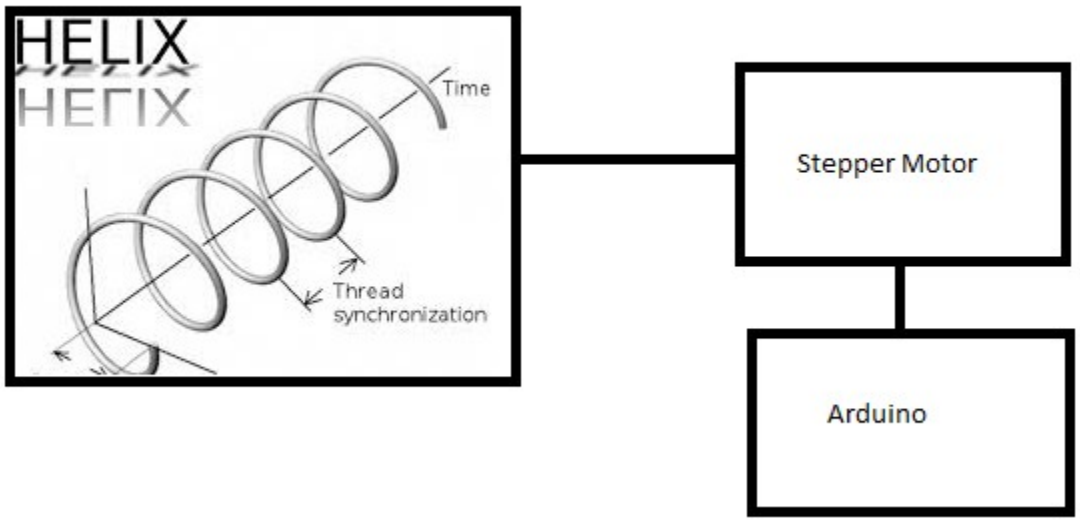


Fig 3.1

Mechanism

The Mechanism is designed in such a way that one end of the strong helix wire is connected to the stepper which is controlled by the Arduino controller and whenever the arduino sends a command to the stepper the stepper rotates a particular amount of rotation with precision and then stops which on the other side pushes a product further and realeases the the product on the end.

3.1 Stepper Motor

A stepper motor or step motor or stepping motor is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any feedback sensor (an open-loop controller), as long as the motor is carefully sized to the application in respect to torque and speed.

Fundamentals of operation

DC brushed motors rotate continuously when DC voltage is applied to their terminals. The stepper motor is known by its property to convert a train of input pulses (typically square wave pulses) into a precisely defined increment in the shaft position. Each pulse moves the shaft through a fixed angle.

Stepper motors effectively have multiple "toothed" electromagnets arranged around a central gear-shaped piece of iron. The electromagnets are energized by an external driver circuit or a microcontroller. To make the motor shaft turn, first, one electromagnet is given power, which magnetically attracts the gear's teeth. When the gear's teeth are aligned to the first electromagnet, they are slightly offset from the next electromagnet. This means that when the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next one. From there the process is repeated. Each of those rotations is called a "step", with an integer number of steps making a full rotation. In that way, the motor can be turned by a precise angle.

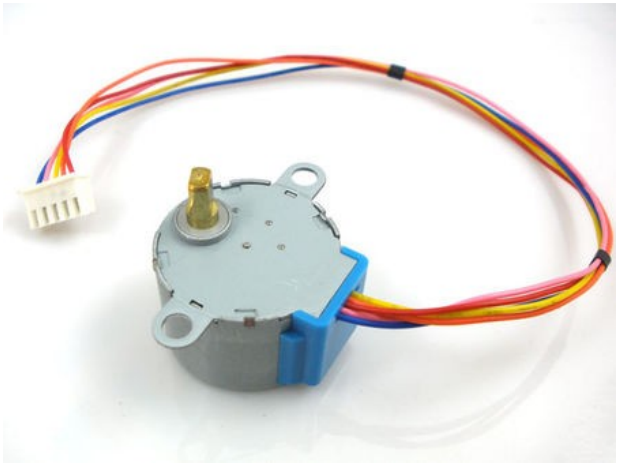


Fig 3.2

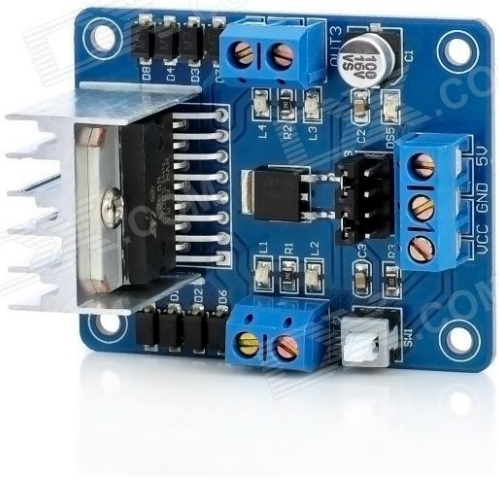


Fig 3.3

3.3 Stepper Code

```
/*
 Stepper Motor Control - one revolution

 This program drives a unipolar or bipolar stepper motor.
 The motor is attached to digital pins 8 - 11 of the Arduino.

 The motor should revolve one revolution in one direction, then
 one revolution in the other direction.

*/

#include <Stepper.h>

const int stepsPerRevolution = 200;  // change this to fit the number of steps per revolution
// for your motor

// initialize the stepper library on pins 8 through 11:
Stepper myStepper(stepsPerRevolution, 8, 9, 10, 11);

void setup() {
  // set the speed at 60 rpm:
  myStepper.setSpeed(60);
  // initialize the serial port:
  Serial.begin(9600);
}

void loop() {
  // step one revolution  in one direction:
  Serial.println("clockwise");
  myStepper.step(stepsPerRevolution);
  delay(500);

  // step one revolution in the other direction:
  Serial.println("counterclockwise");
  myStepper.step(-stepsPerRevolution);
  delay(500);
}
```

SNAPSHOTS

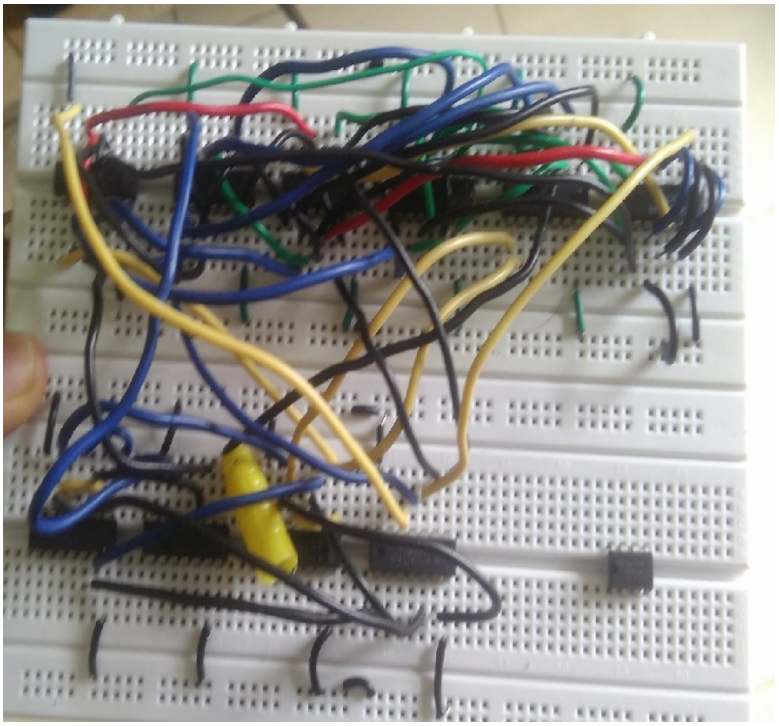


Fig 4

Fig 5

Fig 6

REFERENCES

https://en.wikipedia.org/wiki/BeagleBoard

http://aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoints/

http://opencv-python tutroals.readthedocs.org/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html