Rewrite of the Vision Tracking System, To Benefit the Accuracy and Speed of the nVidia Tegra©

Nicholas Burrell
Software Team

Febuary 28, 2016

Contents

1 Introduction

1.1 Before We Begin

Thank you for taking the time to read the documentation for our systems new rewrite. The old system had a ***LOT*** of issues, which we can see below. They range from simple things like inconsistent variable naming schemes and tabbing, to more extreme things like memory management and even neglecting an entire subsystem of the Tegra©, the CUDA© Cores.
Some examples of poor writting include...

```
1 template <class T>
2 T getMin (vector <T> input) {
3    size_t i;
4   T minimum = 0;
5   T minimum_out = 0;
6    if (input.size() == 0)
7      return 360;
8    if (input.size() == 1)
9      return input[0];
10   else{
11     minimum = 360;
12     for (i = 0; i < input.size(); i++) {
13       if (minimum > abs(input[i])) {
14         minimum = abs(input[i]);
15         minimum_out = input[i];
16       }
17     }
18   }
19   return minimum_out;
20 }
```

Listing 1: Poor Coding Part 1

The first issue with this example, is it is written for reusability, while it doesn't need to be. That is *not* and issue, but for the sake of simplicity, it can and *will* be rewritten. The second being the hard coded value. Line 11, to be precise. If we were to write this code to be reusable, we should just have the value be some arbitrarily large number, say, 600 billion. Speaking of bad practices...

```

float angle;
  int numOfBadFrames = 0;
    bool die = false;
  string filename("snapshot");
  string suffix(".png");
  int i_snap(0), iter(0);
  auto index = 0;
    vector < vector <Point> > contours;
    vector<Rect > boundRect;
  vector <KeyPoint> keypoints;
  vector <Point> centers;
  //vector <float > abs_angle;
  Mat frame(Size(480, 360), CV_8UC3, Scalar(0));
  bool bIsConnected = false;
  int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_storage cli_addr; // connector's address information
  socklen_t sin_size;
  struct sigaction sa;
  int yes=1;
  char s[INET6_ADDRSTRLEN];
  int rv;
  char buf[256];
  int numchars = 0;
```

Listing 2: Poor Coding Part 2

The naming convention has *no* convention. It appears as though the code was copied off of the internet. Granted, a small portion of it was. (Mostly the sockets) Along with this, the tabbing is atrocious. This code is *begging* to be rewritten.

1.2 General Improvements

Before we delve into what will be rewritten, lets compare and contrast some of the issues with solutions. The first issue is the writing style. Relatively easy fix. The next issue, is speed. The fact that the camera in the rear of the robot can go at *whatever* framerate we need at the time, means my code should perform that fast as well. Parallel processing would improve the speed *drastically*, as would the GPU. Some of these will require some things like named pipes, or more advanced things such as CUDA© mathematics.

2 New Features

2.1 Improvements (In Greater Detail)

One of the things I have noticed with the system in the past, is that if there is too much ambient IR or other kinds lighting, (e.g. the Towers LEDs), it has caused issues. The way to fix this light correction. This would take some more interesting algorithms to accomplish, such as sequential frame analysis and comparison to find differences in the light, and increasing or decreasing the color threshold to accommodate the current conditions.
Another interesting feature being integrated is a custom made networking API. A client/server socket wrapper. Really simple to use. Simply call the client or server constructor, and then either send or receive, respectively.

2.2 New Features

There are not nearly as many new features as their are improvements and bug fixes. Completely revamped algorithms, boosting mathmatics with CUDA© and assembly (Mostly GAS and NASM) for smaller operations (getMax() and getMin()).

2.3 Improvements Continued

In the mathematics department, we are *completely* redoing some of the algorithms. Examples such as:

```

cv::Point getMean(cv::Point pt1, cv::Point pt2) {
  return Point(((pt1.x + pt2.x)/2), ((pt1.y + pt2.y)/2));
}

int getProxToPoint (cv::Point pt1, int midline) {
  //int midline = (frame.cols / 2);
  return (midline - pt1.x);


}

float fGetAnglefromPixel ( float _fMidLine, int _iFOV, float _fX) {
  cout << _fX << endl;
  float _fRatioOfXToFrame = _fX/(_fMidLine);
  cout << _fRatioOfXToFrame << endl;
  float _fAngle = _fRatioOfXToFrame * (_iFOV/2);
  cout << _fAngle;
  if (_fX < _fMidLine) {
    _fAngle = (((_iFOV/2) - _fAngle)* -1);
  }
  else {
    _fAngle = (_fAngle - (_iFOV/2));
  }
  cout << _fAngle << endl;
  return _fAngle;
}
```

Listing 3: Basic Mathematical Algorithms

These are *very* basic mathematical functions. The last one can be put as simply as

$$|\theta| = \frac{x}{m} * \frac{FOV}{2} \tag{1}$$

where m is the x at the middle of the image, the FOV in our case being ≈ 67. Of course, there is additional logic involved, including how to tell if you are to the left or the right, but that can be done by doing the following.

```
if (x < midline) return -x;
```

```
else return x;
```

Listing 4: Additional Logic Required

The other two are exceptionally easy formulas.
getProxToPoint (where m is the midline of the image)

$$m - x \tag{2}$$

And for getMean (Assuming X and Y are inputs).

$$x = \frac{(x_1 + x_2)}{2} \tag{3}$$

$$y = \left(\frac{y_1 + y_2}{2}\right) \tag{4}$$

Putting the preceding code into assembly would be relatively simple. As inline assembly, the code would look like the following.

```
struct Point{
  uint32 x, //for the sake of being used; is just unsigned int
  uint32 y //same as above
  };
  inline Point getMean(Point pt1, Point pt2) {
    uint32 x3 = 0;
    uint32 y3 = 0;
    Point pt3;

    #ifdef __i386__
    asm ("mov bx, $2\n"
        "add %1, %2\n"
        "mov eax, %1\n"
        "div ax\n"
        "mov %0, %1"
        : "=x"(pt3.x): "x1"(pt1.x), "x2"(pt2.x) : );

    asm ("mov bx, $2\n"
        "add %1, %2\n"
        "mov eax, %1\n"
        "div ax\n"
        "mov %0, %1"
        : "=y3"(pt3.y): "y1"(pt1.y), "y2"(pt2.y) : );

    #endif
    //TODO ARM\textcopyright assembly
    return pt3;

  }
```

Listing 5: Basic Mathematical Algorithms Continued; Inline Assembly

It appears as though this is a sizeable chunk of code; however, if you take into account the speed boost by doing the commands in *raw* (almost) machine code (1:1 correspondence), it is blazingly fast. Doing the math, it would take $\frac{22}{Hz}$ ***SECONDS***! (Hz being CPU speed in GHz) 22 operations, or approximately 44 bytes *per instance*. Reason enough for it to be inline. The issue with the inline assembly, is portability. At the moment, if I were to compile this code on an i686 platform (or x86_64 with *very* little modification) it would most likely run as expected; however, on an ARM© based platform, say a Cortex© A9 (the very same processor inside of the NVidia© Jetson TK1) it would complain about register names. Code should be written to be portable, unless it is *not* explicitly needed. The __i686__ is only there so the user can test the code on a PC, for debugging reasons, *not* release (Possibly and encompassing #ifdef DEBUG).