

Automation Tutorial 2: Code Description

Marco Selvi

8th September 2013

Aims

In this tutorial we will describe in more details what the code written for the microscope's automation actually does. All programs have been written in C++, with the use of different libraries for image processing and serial port communication. In the next section we list all packages and libraries necessary for the correct compilation and execution of the programs themselves.

Remember however that, while this program will compile and run on any platform, it is meant to work on the Raspberry Pi. This is in particular obvious from the restriction for image capture using the program RaspiStill, available only on the Pi. If run on any other platform that does not have RaspiStill as a program, the routine will run correctly until the first invocation of 'raspistill', and then crash.

Libraries and Packages

Here we list all packages necessary for the correct execution of the program, with a short description for each. Terminal commands for obtaining all packages are at the end of this section.

Imaging: CoolImage Library

For the programs that follow we will use the Cool Image Library, obtainable from this website. This is a template library, contained all in one header file. It is very easy to use and efficient at run-time, but takes quite a while to compile on the Raspberry Pi (it contains approximately 45,000 lines of code). To use it, simply include the "CImg.h" file downloadable from the website in the heading of your code and it will work at compile time. See any example of programs in the next sections to check where to include it.

We suggest creating a dummy .cpp file containing the library itself and the definition of the components used, so that it can be compiled separately and then linked as an object. A header file of similar type might be useful also. We have not considered this possibility until the writing of this tutorial because software development was done on a computer that could handle compilation of the CImg.h file easily, but if you plan to do software development on the Pi this is definitely an option to explore.

Serial Communication: Boost Asio Library

Communication over the serial port is done using the Boost Asio library (see here for documentation). This library provides rather simple to implement (if hard to devise) communication with the Arduino. Opening a serial port with this library allows the user to use the serial communication as a stream not dissimilar from the 'iostream' available from the C++ inbuilt libraries. Commands can be written to the stream and outputs can be read from it. However...

A NOTE of caution: the non-asynchronous functions used to read from the serial port block until the buffer used to read into is full (so they must read at least one character, if the buffer used is of such size). If you are developing your own software using this library, be sure that whenever a 'read' function is called on the serial port stream there will be some output to read. Otherwise, your program will block forever. Alternatively, try using the asynchronous read functions provided in the boost libraries. If you can figure out how.

Terminal Commands to Install Libraries

You will need the 'imagemagick' package to handle window display from the CoolImage library through the 'CImgDisplay' class.

You will also need to make sure that the X11 video libraries are installed. These are generally already installed in Raspbian, but checking can't hurt.

As we said, the CoolImage library is used by simply including the header file, so no package is installed for it.

Just to be on the safe side, or if you want to do some testing, you may install the Python imaging library as well. It is not as efficient as the CoolImage library, so since our programs are rather speed-sensitive we opted for the latter, but it is a good first-stage development tool.

So the commands to use to install everything are

```
sudo apt-get update sudo apt-get install libx11-dev imagemagick
python-imaging libboost-all-dev
```

And you are done installing stuff. Now for the real programming...

Edge Recognition

Here we report an implementation of the Canny Edge Recognition algorithm as an example of how to do image processing with the CoolImage library. The relevant code files are found here.

As it can be seen from a quick analysis, the 'edges.cpp' file is a very basic user interface, while all the computation is done in the header file 'edgedetection_class.h'. We will therefore skip a detailed analysis of the former, and concentrate on what the latter does.

edgedetection_class.h

This file handles all the processes needed to detect edges in an image using the Canny Edge Detection algorithm mentioned above. We use '.jpg' files because they have only one layer and are therefore more easily analysed. The

CoolImage library can of course work with multi-layer images such as ‘.bmp’ as well, but the code will need heavy remodeling to work with more complex images. To make your life easier, just convert images to ‘.jpg’. Edges don’t need multi-layering anyway.

The file ‘edgedetection_class.h’ (as the name suggests) contains a class whose methods do everything in the edge detection algorithm. The method invoked by ‘edges.cpp’ is ‘canny_edge_detection()’ that takes no arguments. This method then invokes all other methods in the class. Check all other methods in the complete header file at the link mentioned above. We don’t report them here to avoid clutter.

This is the code for the principal method, with relevant comments. The comments should be self explanatory.

```

194 | CImg<float> Edgedetection::canny_edge_detection()
195 | {
196 |
197 |     // First we show the picture we start with
198 |     m_show.display(m_picture);
199 |
200 |
201 |     // Load convolution matrix for noise damping
202 |     // This matrix will be convolved with the image to smooth away artifacts and 'ruined'
       pixels
203 |     load_matrix();
204 |     cout << "\nFinished loading convolution matrix for noise filtering" << endl;
205 |
206 |
207 |
208 |     // Reduce image to greyscale (intensity scale)
209 |     // This is necessary for the following analysis, since we are interested in the
       intensity of the pixels and not their colour
210 |     //if ((m_picture(0,0,0,0) != m_picture(0,0,0,1)) || (m_picture(0,0,0,0) != m_picture
       (0,0,0,2)))
211 |     //{
212 |         greyfy();
213 |         cout << "\nFinished turning image to greyscale" << endl;
214 |     //}
215 |     //else
216 |     // cout << "\nImage is already in greyscale" << endl;
217 |
218 |     // Display the image after turning to grayscale
219 |     m_show.display(m_picture);
220 |
221 |
222 |
223 |     // Apply convolution with preset matrix
224 |     picture.convolution();
225 |
226 |     m_show.display(m_picture);
227 |
228 |     cout << "\nFinished noise filtering" << endl;
229 |
230 |
231 |
232 |     // Calculate gradients associated with each pixel
233 |     // Refer to the 'Canny Edge Detector' article on Wikipedia to know why this is
       necessary
234 |     // Here we have implemented two possibilities for the gradient calculation
235 |     // See the specific methods within the class for further details
236 |     //simple_gradient();

```

```

237 | sobel_gradient();
238 |
239 | cout << "\nFinished calculating gradient for the pixels" << endl;
240 |
241 |
242 |
243 | // Decide whether each pixel is on the edge or not
244 | // This is done following the Canny Edge Detection method of minor and major edges
245 | // See method for details
246 | edge_decision();
247 |
248 | // Assign different grey tonalities to major edges, minor edges and non-edges
    | respectively
249 | for (int i=0; i<m_picture.width(); i++) {
250 |     for (int j=0; j<m_picture.height(); j++)
251 |     {
252 |
253 |         if (m_edge[i][j] == 2)
254 |         {
255 |             m_picture(i,j,0,0) = 255;
256 |             m_picture(i,j,0,1) = 255;
257 |             m_picture(i,j,0,2) = 255;
258 |         }
259 |         else if (m_edge[i][j] == 1)
260 |         {
261 |             m_picture(i,j,0,0) = 130;
262 |             m_picture(i,j,0,1) = 130;
263 |             m_picture(i,j,0,2) = 130;
264 |         }
265 |         else
266 |         {
267 |             m_picture(i,j,0,0) = 0;
268 |             m_picture(i,j,0,1) = 0;
269 |             m_picture(i,j,0,2) = 0;
270 |         }
271 |     }
272 | }
273 |
274 |
275 | // Display edges
276 | m_show.display(m_picture);
277 |
278 | cout << "\nFinished establishing edges" << endl;
279 |
280 | m_show.wait(2000);
281 |
282 |
283 |
284 | // Select edges that are REAL edges
285 | // This is done by checking proximity of minor edges with major edges, or by continuing
    | major edges into minor edges if they are reciprocal continuation
286 | // See method for details
287 | edge_selection();
288 |
289 | // Assign 'white' to real edges, and black to non-edges
290 | for (int i=0; i<m_picture.width(); i++) {
291 |     for (int j=0; j<m_picture.height(); j++)
292 |     {
293 |
294 |         if (m_edge[i][j] == 2)
295 |         {
296 |             m_picture(i,j,0,0) = 255;

```

```

297     m_picture(i,j,0,1) = 255;
298     m_picture(i,j,0,2) = 255;
299 }
300 else
301 {
302     m_picture(i,j,0,0) = 0;
303     m_picture(i,j,0,1) = 0;
304     m_picture(i,j,0,2) = 0;
305 }
306
307 }
308 }
309
310 // Display final result
311 m_show.display(m_picture);
312
313 cout << "\nFinished selecting edges" << endl << endl;
314
315 // Save final result if so required by the user, cropping away the black fringes that
result from the edge detection work
316 if (m_save == 'y')
317 {
318     crop_and_save();
319     m_show.wait(2000);
320 }
321 else
322     m_show.wait();
323
324 // Return the picture result, in case one may want to do more with it (we don't here,
but you never know)
325 return m_picture;
326
327 }

```

edgedetection_class.h

Microscope Control

Now we can start tackling more complex tasks, useful for the control of the microscope. In particular, we need to make all functions and methods ‘cooperate’ efficiently and as fast as possible. The algorithm used in this program is available in the Tutorial 1 PDF. Here we analyse the methods used in the steps of the algorithm.

Image Acquisition

To acquire images to be used in the focusing algorithm, we use the inbuilt ‘raspistill’ program available with the Raspberry Pi camera libraries. To do so, we simply call a ‘system()’ function, which allows to call terminal programs from within a C++ program.

```

1 | system(" raspistill -n -w 480 -h 360 -o test.jpg");
2 |

```

Image Analysis

We use the CImg library to open the images, and then use the focusing formula contained in this method to analyse them.

```
472 float Autofocus::algorithm()
473 {
474     float intensity = 0.0;
475     float mean_intensity = 0.0;
476     float intensity_squared_sum = 0.0;
477     float focusing = 0.0;
478
479     greyfy();
480
481     // Calculate intensity of a matrix of pixels
482     for (int x=0; x<m_picture.width(); x++) {
483         for (int y=0; y<m_picture.height(); y++) {
484             {
485                 intensity += (float)m_picture(x,y,0,0);
486             }
487         }
488     }
489
490     mean_intensity = intensity/((float)m_picture.width()*m_picture.height());
491
492     // Make sure that no value of mean_intensity is effectively 0.0
493     // to prevent division by 0, even though very unlikely
494     if (mean_intensity == 0.0) mean_intensity = 1E-10;
495
496     // Apply focusing algorithm of choice, in this case a normalised deviation based
497     for (int i=0; i<m_picture.width(); i++) {
498         for (int j=0; j<m_picture.height(); j++) {
499             {
500                 intensity_squared_sum += ((float)m_picture(i,j,0,0) - mean_intensity)*((float)
501                     m_picture(i,j,0,0) - mean_intensity);
502             }
503         }
504     }
505
506     focusing = intensity_squared_sum/(((float)m_picture.width()*m_picture.height()*
507         mean_intensity);
508
509     return focusing;
510 }
```

autofocus.class.h

This is used to compute a focusing value for each picture taken with ‘raspis-till’. Then the program uses these values in the previous algorithm to decide what to do.

Arduino Control

To control the Arduino, we use the following method.

```
1150 bool Autofocus::serial_command(string command, int &number, bool couting, string check
1151 )
1152 {
1153     stringstream ss;
1154     istream is(&m_buffering);
1155     string line;
1156     bool control = false;
```

```

1156 | bool exiting = false;
1157 |
1158 |
1159 | if (command.compare(m_calibrate) == 0)
1160 | {
1161 |     ss << command;
1162 |     write(m_sp, buffer(ss.str()));
1163 |     ss.str("");
1164 |
1165 |     return true;
1166 | }
1167 | else if (command.compare(m_is_cal) == 0)
1168 | {
1169 |     ss << command;
1170 |     write(m_sp, buffer(ss.str()));
1171 |     ss.str("");
1172 |
1173 |     while (exiting == false)
1174 |     {
1175 |         boost::asio::read_until(m_sp, m_buffering, '\n');
1176 |         getline(is, line);
1177 |         if (couting)
1178 |             cout << line << endl;
1179 |         if (atoi(line.c_str()) != 0)
1180 |             control = true;
1181 |         if (line.compare(check) == 0)
1182 |             exiting = true;
1183 |
1184 |         line.clear();
1185 |     }
1186 | }
1187 | else if (command.compare(m_get_z_distance) == 0)
1188 | {
1189 |     ss << command;
1190 |     write(m_sp, buffer(ss.str()));
1191 |     ss.str("");
1192 |
1193 |     check = m_endpoint;
1194 |
1195 |     for (int i=0; i<3; i++)
1196 |     {
1197 |         boost::asio::read_until(m_sp, m_buffering, '\n');
1198 |         getline(is, line);
1199 |         if (couting)
1200 |             cout << line << endl;
1201 |         if (line.compare(check) == 0)
1202 |             control = true;
1203 |
1204 |         line.clear();
1205 |     }
1206 | }
1207 | // Requires unsigned integer as argument
1208 | else if (command.compare(m_move_to) == 0)
1209 | {
1210 |     string number_in_words = boost::lexical_cast<string>((int)(number));
1211 |     ss << command << " " << number_in_words << "\n";
1212 |     write(m_sp, buffer(ss.str()));
1213 |     ss.str("");
1214 |
1215 |     while (exiting == false)
1216 |     {
1217 |         boost::asio::read_until(m_sp, m_buffering, '\n');

```

```

1218 |     getline(is, line);
1219 |     if (couting)
1220 |         cout << line << endl;
1221 |     if (line.compare(check) == 0)
1222 |     {
1223 |         exiting = true;
1224 |         control = true;
1225 |     }
1226 |
1227 |     if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
1228 |     {
1229 |         exiting = true;
1230 |     }
1231 |     if (line.compare("ERR: NOT CALIBRATED\r") == 0)
1232 |     {
1233 |         exiting = true;
1234 |     }
1235 |     if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
1236 |     {
1237 |         exiting = true;
1238 |     }
1239 |
1240 |     line.clear();
1241 | }
1242 | }
1243 | // Requires signed integer as argument
1244 | else if (command.compare(m_move) == 0)
1245 | {
1246 |     string number_in_words = boost::lexical_cast<string>((int)(number));
1247 |     ss << command << " " << number_in_words << "\n";
1248 |     write(m_sp, buffer(ss.str()));
1249 |     ss.str("");
1250 |
1251 |     while (exiting == false)
1252 |     {
1253 |         boost::asio::read_until(m_sp, m_buffering, '\n');
1254 |         getline(is, line);
1255 |         if (couting)
1256 |             cout << line << endl;
1257 |         if (line.compare(check) == 0)
1258 |         {
1259 |             exiting = true;
1260 |             control = true;
1261 |         }
1262 |
1263 |         if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
1264 |         {
1265 |             exiting = true;
1266 |         }
1267 |         if (line.compare("ERR: NOT CALIBRATED\r") == 0)
1268 |         {
1269 |             exiting = true;
1270 |         }
1271 |         if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
1272 |         {
1273 |             exiting = true;
1274 |         }
1275 |
1276 |         line.clear();
1277 |     }
1278 | }
1279 | // Requires a number between 0 and 255 as argument

```



```

1280 else if (command.compare(m_set_stage_led_bright) == 0 || command.compare(
1281     m_set_ring_bright) == 0)
1282 {
1283     if (number < 0 || number > 255)
1284         number = 70;
1285
1286     string number_in_words = boost::lexical_cast<string>((int)(number));
1287     ss << command << " " << number_in_words << "\n";
1288     write(m_sp, buffer(ss.str()));
1289     ss.str("");
1290
1291     while (exiting == false)
1292     {
1293         boost::asio::read_until(m_sp, m_buffering, '\n');
1294         getline(is, line);
1295         if (couting)
1296             cout << line << endl;
1297         if (line.compare(check) == 0)
1298         {
1299             exiting = true;
1300             control = true;
1301         }
1302
1303         if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
1304         {
1305             exiting = true;
1306         }
1307         if (line.compare("ERR: NOT CALIBRATED\r") == 0)
1308         {
1309             exiting = true;
1310         }
1311         if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
1312         {
1313             exiting = true;
1314         }
1315
1316         line.clear();
1317     }
1318 else
1319 {
1320     ss << command;
1321     write(m_sp, buffer(ss.str()));
1322     ss.str("");
1323
1324     while (exiting == false)
1325     {
1326         boost::asio::read_until(m_sp, m_buffering, '\n');
1327         getline(is, line);
1328         if (couting)
1329             cout << line << endl;
1330         if (atoi(line.c_str()) != 0)
1331             number = atoi(line.c_str());
1332         if (line.compare(check) == 0)
1333         {
1334             exiting = true;
1335             control = true;
1336         }
1337
1338         if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
1339         {
1340             exiting = true;

```

```

1341     }
1342     if (line.compare("ERR: NOT CALIBRATED\r") == 0)
1343     {
1344         exiting = true;
1345     }
1346     if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
1347     {
1348         exiting = true;
1349     }
1350
1351     line.clear();
1352 }
1353 }
1354
1355 return control;
1356 }

```

autofocus.class.h

The commands we pass are prebuilt in the code loaded on the Arduino, available [here](#). What they do is pretty straightforward, and at the link above a full description of all the commands available is provided. Note that we use the ‘boost::asio::read_some()’ and the ‘boost::asio::read_until()’ functions to read the output from the Arduino and save it in strings we can use to interpret it. It is important to make sure that every time we read an output, there is an output to read. Otherwise these functions will block. For a complete documentation of these functions check the links available [here](#). Other formulas are available for the usage of Arduino controls from the main program. Check the full code for details. They all contain the necessary comments to understand their functionality.

Algorithm Implementation

The implementation of the tuning algorithm is in this function.

```

587 bool Autofocus::fine_tune(float f_max, int f_max_pos,
588     bool previous_direction,
589     bool up_or_down,
590     int times_checked)
591 {
592
593     float f_above = 0;
594     float f_below = 0;
595     int f_above_pos = 0;
596     int f_below_pos = 0;
597
598
599     if (!previous_direction && m_steps > m_min_steps)
600     {
601         //cout << "\nReducing number of steps... " << endl;
602         m_steps = (int)(0.5*m_steps);
603     }
604     else if (m_steps <= m_min_steps)
605     {
606         //cout << "\nUsing minimum number of steps: " << m_min_steps << endl;
607         m_steps = m_min_steps;
608     }
609
610
611     // Take picture at starting point

```

```

612     raspistill_save ();
613
614
615     // Compute MIDDLE picture
616     f_max = move_and_capture(0, f_max_pos);
617     cout << "." << flush;
618     m_values << m_ind << "\t" << f_max << endl;
619     m_ind++;
620
621     // Set max values for future use
622     m_f_max = f_max;
623     m_f_max_pos = f_max_pos;
624
625     // Remove picture if necessary
626     if (!m_leave_output)
627         remove_picture();
628
629
630     //cout << "\nStart call at\n" << f_max_pos << "\t" << f_max << endl;
631
632
633
634     if (up_or_down)
635         // Start checking from ABOVE the starting position
636     {
637
638         //cout << "\nChecking UP..." << endl;
639
640         f_above = move_and_capture(m_steps, f_above_pos);
641         //cout << f_above_pos << "\t" << f_above << endl;
642         cout << "." << flush;
643
644         // Remove picture if necessary
645         if (!m_leave_output)
646             remove_picture();
647
648
649
650         if (f_above >= ((2.0-m_precision)*f_max))
651         {
652
653             //cout << "\nStaying UP..." << endl;
654
655             // Keep checking up, maintaining the same number of steps
656             fine_tune(f_above, f_above_pos, true, true);
657
658         }
659         else
660         {
661
662             //cout << "\nChecking DOWN..." << endl;
663
664             f_below = move_and_capture(-2*m_steps, f_below_pos);
665             //cout << f_below_pos << "\t" << f_below << endl;
666             cout << "." << flush;
667
668             // Remove picture if necessary
669             if (!m_leave_output)
670                 remove_picture();
671
672
673

```

```

674     if (f_below >= ((2.0-m_precision)*f_max))
675     {
676
677         //cout << "\nStaying DOWN..." << endl;
678
679         // Change direction of movement and continue checking down
680         fine_tune(f_below, f_below_pos, false, false);
681
682     }
683     else
684     {
685
686         //cout << "\nI was already there, so I am going back..." << times_checked <<
        endl;
687
688         // Move back to maximum
689         serial_command(m_move_to, f_max_pos);
690         stop_stage();
691
692         // If the maximum hasn't changed lately (in the last two checks) then we have
        arrived...
693         if (times_checked == m_number_of_times && m_steps > m_min_steps)
694         {
695             m_steps = m_min_steps;
696             fine_tune(f_max, f_max_pos, false, false, ++times_checked);
697         }
698         else if (times_checked < m_number_of_times && m_steps > m_min_steps)
699         {
700             fine_tune(f_max, f_max_pos, false, false, ++times_checked);
701         }
702         else
703         {
704             return true;
705         }
706     }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 else
716 // Start checking from BELOW the starting position
717 {
718
719     //cout << "\nChecking DOWN..." << endl;
720
721     f_below = move_and_capture(-m_steps, f_below_pos);
722     //cout << f_below_pos << "\t" << f_below << endl;
723     cout << "." << flush;
724
725     // Remove picture if necessary
726     if (!m_leave_output)
727         remove_picture();
728
729
730     if (f_below > ((2.0-m_precision)*f_max))
731     {
732
733         //cout << "\nStaying DOWN..." << endl;

```

```

734 // Keep checking down, maintaining the same number of steps
735 fine_tune(f_below, f_below_pos, true, false);
736
737 }
738 else
739 {
740
741 //cout << "\nChecking UP..." << endl;
742
743 f_above = move_and_capture(2*m_steps, f_above_pos);
744 //cout << f_above_pos << "\t" << f_above << endl;
745 cout << "." << flush;
746
747 // Remove picture if necessary
748 if (!m_leave_output)
749     remove_picture();
750
751
752 if (f_above > ((2.0-m_precision)*f_max))
753 {
754
755 //cout << "\nStaying UP..." << endl;
756
757 // Invert direction and start checking above
758 fine_tune(f_above, f_above_pos, false, true);
759
760 }
761 else
762 {
763
764 //cout << "\nI was already there, so I am going back... " << times_checked <<
765 endl;
766
767 // Move back to maximum
768 serial_command(m_move_to, f_max_pos);
769 stop_stage();
770
771 // If the maximum hasn't changed lately (in the last two checks) then we have
772 arrived...
773 if (times_checked == m_number_of_times && m_steps > m_min_steps)
774 {
775     m_steps = m_min_steps;
776     fine_tune(f_max, f_max_pos, false, true, ++times_checked);
777 }
778 else if (times_checked < m_number_of_times && m_steps > m_min_steps)
779 {
780     fine_tune(f_max, f_max_pos, false, true, ++times_checked);
781 }
782 else
783 {
784     return true;
785 }
786 }
787 }
788 }
789 }
790
791 return true;
792
793

```

The passages are directly correlated to the algorithm in Tutorial 1. We use a (rather arbitrary) minimum number of steps for each objective, that should give a sufficient resolution to find a focus point while disregarding changes due to noise. We also use an acceptance threshold (default is 99%) on the focus value, to account for noise influence. Code is available for a more automated focusing program in the file `focus_everything.h`. For more details check the comments on the code provided.

Other Files

The file `'focus_everything.h'` implements different usages of the autofocus class. Again check comments for descriptions.

The file `'autofocus_class_initialisation.h'` contains an implementation of instructions to initialise the autofocus class. It also provides the possibility to use a default initialisation, that will assume the objective in use on the microscope is a '4x'.

The file `'focus_everything.cpp'` is the main file that puts together all the header files and provides the user interface through terminal. Through this file it is possible to use all methods provided by the autofocus class recursively, and decide whether to save the output of each operation.

The 'Makefile' provided allows to compile and launch all programs available. The flags implemented here are necessary for the correct usage of the boost libraries and the CImg library, linking all other useful libraries.

All files can be found here, with relevant comments describing all the functionalities.