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 if 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) containes 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.

```
CImg<float> Edgedetection::canny_edge_detection()
{

  // First we show the picture we start with
  m_show.display(m_picture);


  // Load convolution matrix for noise damping
  // This matrix will be convolved with the image to smooth away artifacts and 'ruined' pixels
  load_matrix();
  cout << "\nFinished loading convolution matrix for noise filtering" << endl;



  // Reduce image to greyscale (intensity scale)
  // This is necessary for the following analysis, since we are interested in the intensity of the pixels and not their colour
  //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)))
  //{
    greyfy();
    cout << "\nFinished turning image to greyscale" << endl;
  //}
  //else
  //  cout << "\nImage is already in greyscale" << endl;

  // Display the image after turning to grayscale
  m_show.display(m_picture);



  // Apply convolution with preset matrix
  picture_convolution();

  m_show.display(m_picture);

  cout << "\nFinished noise filtering" << endl;



  // Calculate gradients associated with each pixel
  // Refer to the 'Canny Edge Detector' article on Wikipedia to know why this is necessary
  // Here we have implemented two possibilities for the gradient calculation
  // See the specific methods within the class for further details
  //simple_gradient();
```

```
sobel_gradient();

cout << "\nFinished calculating gradient for the pixels" << endl;



// Decide whether each pixel is on the edge or not
// This is done following the Canny Edge Detection method of minor and major edges
// See method for details
edge_decision();

// Assign different grey tonalities to major edges, minor edges and non-edges respectively
for (int i=0; i<m_picture.width(); i++) {
  for (int j=0; j<m_picture.height(); j++)
  {

     if (m_edge[i][j] == 2)
     {
        m_picture(i,j,0,0) = 255;
        m_picture(i,j,0,1) = 255;
        m_picture(i,j,0,2) = 255;
     }
     else if (m_edge[i][j] == 1)
     {
        m_picture(i,j,0,0) = 130;
        m_picture(i,j,0,1) = 130;
        m_picture(i,j,0,2) = 130;
     }
     else
     {
        m_picture(i,j,0,0) = 0;
        m_picture(i,j,0,1) = 0;
        m_picture(i,j,0,2) = 0;
     }

  }
}

// Display edges
m_show.display(m_picture);

cout << "\nFinished establishing edges" << endl;

m_show.wait(2000);



// Select edges that are REAL edges
// 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
// See method for details
edge_selection();

// Assign 'white' to real edges, and black to non-edges
for (int i=0; i<m_picture.width(); i++) {
  for (int j=0; j<m_picture.height(); j++)
  {

     if (m_edge[i][j] == 2)
     {
        m_picture(i,j,0,0) = 255;
```

```
            m_picture(i,j,0,1) = 255;
            m_picture(i,j,0,2) = 255;
          }
          else
          {
            m_picture(i,j,0,0) = 0;
            m_picture(i,j,0,1) = 0;
            m_picture(i,j,0,2) = 0;
          }

        }
    }

    // Display final result
    m_show.display(m_picture);

    cout << "\nFinished selecting edges" << endl << endl;

    // Save final result if so required by the user, cropping away the black fringes that
       result from the edge detection work
    if (m_save == 'y')
    {
      crop_and_save();
      m_show.wait(2000);
    }
    else
      m_show.wait();

    // Return the picture result, in case one may want to do more with it (we don't here,
       but you never know)
    return m_picture;

}
```

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.

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

```

Image Analysis

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

```
float Autofocus::algorithm()
{

  float intensity = 0.0;
  float mean_intensity = 0.0;
  float intensity_squared_sum = 0.0;
  float focusing = 0.0;

  greyfy();

  // Calculate intensity of a matrix of pixels
  for (int x=0; x<m_picture.width(); x++) {
    for (int y=0; y<m_picture.height(); y++)
    {
      intensity += (float)(m_picture(x,y,0,0));
    }
  }

  mean_intensity = intensity/(float)(m_picture.width()*m_picture.height());

  // Make sure that no value of mean_intensity is effectively 0.0
  // to prevent division by 0, even though very unlikely
  if (mean_intensity == 0.0) mean_intensity = 1E-10;

  // Apply focusing algorithm of choice, in this case a normalised deviation based
  for (int i=0; i<m_picture.width(); i++) {
    for (int j=0; j<m_picture.height(); j++)
    {
      intensity_squared_sum += ((float)m_picture(i,j,0,0) - mean_intensity)*((float)
        m_picture(i,j,0,0) - mean_intensity);
    }
  }

  focusing = intensity_squared_sum/((float)m_picture.width()*(float)m_picture.height()*
      mean_intensity);

  return focusing;

}
```

autofocus_class.h

This is used to compute a focusing value for each picture taken with 'raspistill'. 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.

```
bool Autofocus::serial_command(string command, int &number, bool couting, string check
    )
{
  stringstream ss;
  istream is(&m_buffering);
  string line;
  bool control = false;
```

```
bool exiting = false;


if (command.compare(m_calibrate) == 0)
{
  ss << command;
  write(m_sp, buffer(ss.str()));
  ss.str("");

  return true;
}
else if (command.compare(m_is_cal) == 0)
{
  ss << command;
  write(m_sp, buffer(ss.str()));
  ss.str("");

  while (exiting == false)
  {
    boost::asio::read_until(m_sp, m_buffering, '\n');
    getline(is, line);
    if (couting)
      cout << line << endl;
    if (atoi(line.c_str()) != 0)
      control = true;
    if (line.compare(check) == 0)
      exiting = true;

    line.clear();
  }
}
else if (command.compare(m_get_z_distance) == 0)
{
  ss << command;
  write(m_sp, buffer(ss.str()));
  ss.str("");

  check = m_endpoint;

  for (int i=0; i<3; i++)
  {
    boost::asio::read_until(m_sp, m_buffering, '\n');
    getline(is, line);
    if (couting)
      cout << line << endl;
    if (line.compare(check) == 0)
      control = true;

    line.clear();
  }
}
// Requires unsigned integer as argument
else if (command.compare(m_move_to) == 0)
{
  string number_in_words = boost::lexical_cast<string>((int)(number));
  ss << command << " " << number_in_words << "\n";
  write(m_sp, buffer(ss.str()));
  ss.str("");

  while (exiting == false)
  {
    boost::asio::read_until(m_sp, m_buffering, '\n');
```

```
        getline(is, line);
        if (couting)
          cout << line << endl;
        if (line.compare(check) == 0)
        {
          exiting = true;
          control = true;
        }

        if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
        {
          exiting = true;
        }
        if (line.compare("ERR: NOT CALIBRATED\r") == 0)
        {
          exiting = true;
        }
        if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
        {
          exiting = true;
        }

        line.clear();
      }
    }
    // Requires signed integer as argument
    else if (command.compare(m_move) == 0)
    {
      string number_in_words = boost::lexical_cast<string>((int)(number));
      ss << command << " " << number_in_words << "\n";
      write(m_sp, buffer(ss.str()));
      ss.str("");

      while (exiting == false)
      {
        boost::asio::read_until(m_sp, m_buffering, '\n');
        getline(is, line);
        if (couting)
          cout << line << endl;
        if (line.compare(check) == 0)
        {
          exiting = true;
          control = true;
        }

        if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
        {
          exiting = true;
        }
        if (line.compare("ERR: NOT CALIBRATED\r") == 0)
        {
          exiting = true;
        }
        if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
        {
          exiting = true;
        }

        line.clear();
      }
    }
    // Requires a number between 0 and 255 as argument
```

```
else if (command.compare(m_set_stage_led_bright) == 0 || command.compare(
    m_set_ring_bright) == 0)
{
  if (number < 0 || number > 255)
    number = 70;

  string number_in_words = boost::lexical_cast<string>((int)(number));
  ss << command << " " << number_in_words << "\n";
  write(m_sp, buffer(ss.str()));
  ss.str("");

  while (exiting == false)
  {
    boost::asio::read_until(m_sp, m_buffering, '\n');
    getline(is, line);
    if (couting)
      cout << line << endl;
    if (line.compare(check) == 0)
    {
      exiting = true;
      control = true;
    }

    if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
    {
      exiting = true;
    }
    if (line.compare("ERR: NOT CALIBRATED\r") == 0)
    {
      exiting = true;
    }
    if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
    {
      exiting = true;
    }

    line.clear();
  }
}
else
{
  ss << command;
  write(m_sp, buffer(ss.str()));
  ss.str("");

  while (exiting == false)
  {
    boost::asio::read_until(m_sp, m_buffering, '\n');
    getline(is, line);
    if (couting)
      cout << line << endl;
    if (atoi(line.c_str()) != 0)
      number = atoi(line.c_str());
    if (line.compare(check) == 0)
    {
      exiting = true;
      control = true;
    }

    if (line.compare("ERR: UNKNOWN COMMAND\r") == 0)
    {
      exiting = true;
```

```
        }
        if (line.compare("ERR: NOT CALIBRATED\r") == 0)
        {
          exiting = true;
        }
        if (line.compare("ERR: POSITION OUT OF RANGE\r") == 0)
        {
          exiting = true;
        }

        line.clear();
      }
    }

  return control;
}
```

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.

```
bool Autofocus::fine_tune(float f_max, int f_max_pos,
    bool previous_direction,
    bool up_or_down,
    int times_checked)
{

  float f_above = 0;
  float f_below = 0;
  int f_above_pos = 0;
  int f_below_pos = 0;


  if (!previous_direction && m_steps > m_min_steps)
  {
    //cout << "\nReducing number of steps... " << endl;
    m_steps = (int)(0.5*m_steps);
  }
  else if (m_steps <= m_min_steps)
  {
    //cout << "\nUsing minimum number of steps: " << m_min_steps << endl;
    m_steps = m_min_steps;
  }


  // Take picture at starting point
```

```
  raspistill_save ();


// Compute MIDDLE picture
f_max = move_and_capture(0, f_max_pos);
cout << "." << flush;
m_values << m_ind << "\t" << f_max << endl;
m_ind++;

// Set max values for future use
m_f_max = f_max;
m_f_max_pos = f_max_pos;

// Remove picture if necessary
if (!m_leave_output)
  remove_picture();


//cout << "\nStart call at\n" << f_max_pos << "\t" << f_max << endl;



if (up_or_down)
// Start checking from ABOVE the starting position
{

  //cout << "\nChecking UP..." << endl;

  f_above = move_and_capture(m_steps, f_above_pos);
  //cout << f_above_pos << "\t" << f_above << endl;
  cout << "." << flush;

  // Remove picture if necessary
  if (!m_leave_output)
    remove_picture();



  if (f_above >= ((2.0-m_precision)*f_max))
  {

    //cout << "\nStaying UP..." << endl;

    // Keep checking up, maintaining the same number of steps
    fine_tune(f_above, f_above_pos, true, true);

  }
  else
  {

    //cout << "\nChecking DOWN..." << endl;

    f_below = move_and_capture(-2*m_steps, f_below_pos);
    //cout << f_below_pos << "\t" << f_below << endl;
    cout << "." << flush;

    // Remove picture if necessary
    if (!m_leave_output)
      remove_picture();
```

```
    if (f_below >= ((2.0-m_precision)*f_max))
    {

      //cout << "\nStaying DOWN..." << endl;

      // Change direction of movement and continue checking down
      fine_tune(f_below, f_below_pos, false, false);

    }
    else
    {

      //cout << "\nI was already there, so I am going back... " << times_checked << endl;

      // Move back to maximum
      serial_command(m_move_to, f_max_pos);
      stop_stage();

      // If the maximum hasn't changed lately (in the last two checks) then we have arrived...
      if (times_checked == m_number_of_times && m_steps > m_min_steps)
      {
        m_steps = m_min_steps;
        fine_tune(f_max, f_max_pos, false, false, ++times_checked);
      }
      else if (times_checked < m_number_of_times && m_steps > m_min_steps)
      {
        fine_tune(f_max, f_max_pos, false, false, ++times_checked);
      }
      else
      {
        return true;
      }

    }

  }

}



else
// Start checking from BELOW the starting position
{

  //cout << "\nChecking DOWN..." << endl;

  f_below = move_and_capture(-m_steps, f_below_pos);
  //cout << f_below_pos << "\t" << f_below << endl;
  cout << "." << flush;

  // Remove picture if necessary
  if (!m_leave_output)
    remove_picture();


  if (f_below > ((2.0-m_precision)*f_max))
  {

    //cout << "\nStaying DOWN..." << endl;
```

```
        // Keep checking down, maintaining the same number of steps
        fine_tune(f_below, f_below_pos, true, false);

      }
      else
      {

        //cout << "\nChecking UP..." << endl;

        f_above = move_and_capture(2*m_steps, f_above_pos);
        //cout << f_above_pos << "\t" << f_above << endl;
        cout << "." << flush;

        // Remove picture if necessary
        if (!m_leave_output)
          remove_picture();


        if (f_above > ((2.0-m_precision)*f_max))
        {

          //cout << "\nStaying UP..." << endl;

          // Invert direction and start checking above
          fine_tune(f_above, f_above_pos, false, true);

        }
        else
        {

          //cout << "\nI was already there, so I am going back... " << times_checked << endl;

          // Move back to maximum
          serial_command(m_move_to, f_max_pos);
          stop_stage();

          // If the maximum hasn't changed lately (in the last two checks) then we have arrived...
          if (times_checked == m_number_of_times && m_steps > m_min_steps)
          {
            m_steps = m_min_steps;
            fine_tune(f_max, f_max_pos, false, true, ++times_checked);
          }
          else if (times_checked < m_number_of_times && m_steps > m_min_steps)
          {
            fine_tune(f_max, f_max_pos, false, true, ++times_checked);
          }
          else
          {
            return true;
          }

        }

      }

    }

    return true;

```

```
}
```

autofocus_class.h

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.