# SE101 ICS,Fall 2013
# Lab 8: Code Optimization
# Assigned: October 23
# Due: November 7, 16:00

Xietao (`Foxterran2012@gmail.com`) is the lead person for this Lab.

## 1 Introduction

This Lab deals with optimizing memory intensive code. With the help of optimization techniques mentioned in class, we can improve program's performance a lot, but it's painful, the same as you suffered in Lab 6 and 7.Here I want to give you some tips:

- Start early!!! Read the guide and hand out code carefully.
- Get access to Testbed as soon as possible , and give me your feedback if there is any problem. (This Lab is quite hardware related, so we provide you an uniform testbed. you can find the guide in Section 7)
- Be patient and keep good coding style.
- If you are confused, Email me or publish it on QA site.

The other part of this Lab guide is consist of:

- 2 Hand Out Instructions
- 3 Background: Background information of Image processing
- 4 Implementation Overview: Basic code structure
- 5 Infrastructure: Tools usage which help you test correctness and measure performance
- 6 Assignment Details: coding advice and evaluation details
- 7 Testbed: guide on access to the testbed

You should work **individually** in solving the problems in this lab. Now, let's start this adventure, take it easy and have fun.

## 2 Hand Out Instructions

You should get Lab8 from svn of ICS Course Server.

Start by cd into Lab 8 directory, ls , you will see serveral source code file. **The only file you will be modifying and handing in is** `kernels.c`. The `driver.c` program is a driver program that allows you to evaluate the performance of your solutions. Use the command `make driver` to generate the driver code and run it with the command `./driver`. The `config.h` file contains baseline data, which is also important. Looking at the file `kernels.c` you'll notice a C structure `team` into which you should insert the requested identifying information.**Do this right away so you don't forget.**

## 3 Background

As to Code Optimization,Image processing offers many examples of functions that can benefit from optimization. In this lab, we will consider two image processing operations: `rotate`, which rotates an image counter-clockwise by 90°, and `smooth`, which "smooths" or "blurs" an image.

For this lab, we will consider an image to be represented as a two-dimensional matrix $M$, where $M_{i,j}$ denotes the value of $(i, j)$th pixel of $M$. Pixel values are triples of red, green, and blue (RGB) values. We will only consider square images. Let $N$ denote the number of rows (or columns) of an image. Rows and columns are numbered, in C-style, from 0 to $N - 1$.

Given this representation, the `rotate` operation can be implemented quite simply as the combination of the following two matrix operations:

- *Transpose*: For each $(i, j)$ pair, $M_{i,j}$ and $M_{j,i}$ are interchanged.
- *Exchange rows*: Row $i$ is exchanged with row $N - 1 - i$.

This combination is illustrated in Figure 1.

The `smooth` operation is implemented by replacing every pixel value with the average of all the pixels around it (in a maximum of $3 \times 3$ window centered at that pixel). Consider Figure 2. The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below:

$$\mathtt{M2[1][1]} = \frac{\sum_{\mathtt{i}=0}^{2}\sum_{\mathtt{j}=0}^{2}\mathtt{M1[i][j]}}{9}$$

$$\mathtt{M2[N-1][N-1]} = \frac{\sum_{\mathtt{i}=N-2}^{N-1}\sum_{\mathtt{j}=N-2}^{N-1}\mathtt{M1[i][j]}}{4}$$

## 4 Implementation Overview

### Data Structures

The core data structure deals with image representation. A `pixel` is a struct as shown below:

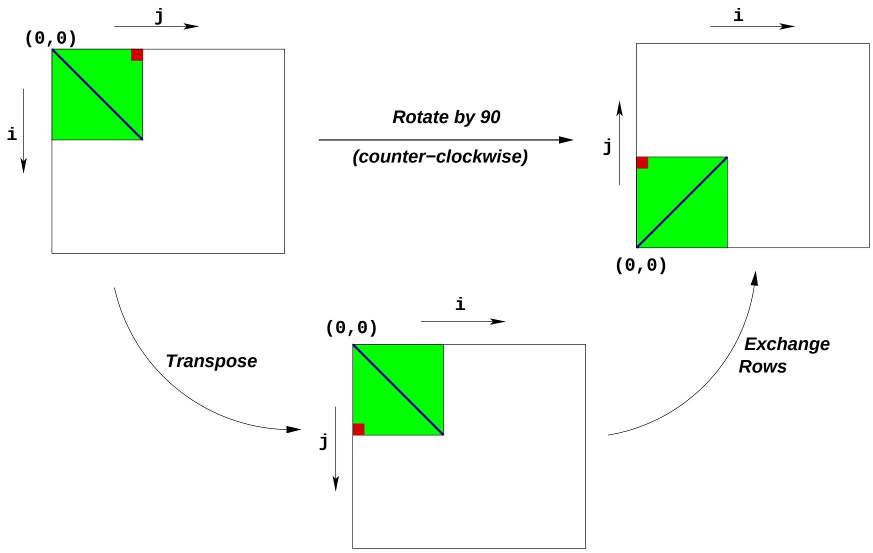


Figure 1: Rotation of an image by $90^{\circ}$ counterclockwise

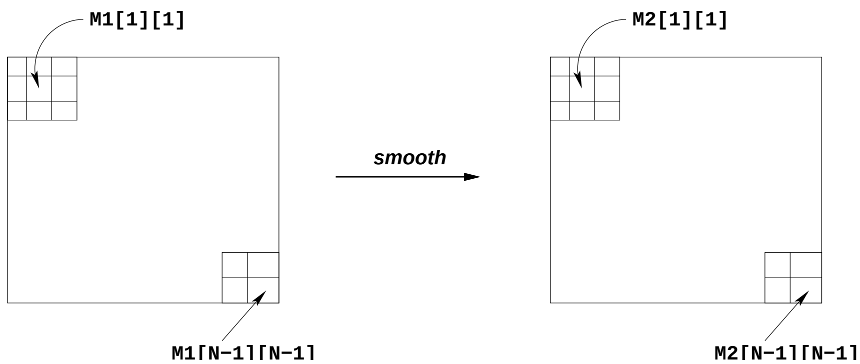


Figure 2: Smoothing an image

```
typedef struct {
   unsigned short red;   /* R value */
   unsigned short green; /* G value */
   unsigned short blue;  /* B value */
} pixel;
```

As can be seen, RGB values have 16-bit representations ("16-bit color"). An image `I` is represented as a one-dimensional array of `pixel`s, where the $(i, j)$th pixel is `I[RIDX(i,j,n)]`. Here `n` is the dimension of the image matrix, and `RIDX` is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

See the file `defs.h` for this code.

## Rotate

The following C function computes the result of rotating the source image `src` by 90° and stores the result in destination image `dst`. `dim` is the dimension of the image.

```
void naive_rotate(int dim, pixel *src, pixel *dst) {
  int i, j;

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(dim-1-j,i,dim)] = src[RIDX(i,j,dim)];

  return;
}
```

The above code scans the rows of the source image matrix, copying to the columns of the destination image matrix. Your task is to rewrite this code to make it run as fast as possible using techniques like code motion, loop unrolling and blocking.

See the file `kernels.c` for this code.

## Smooth

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. Here is part of an implementation,The function `avg` returns the average of all the pixels around the `(i,j)`th pixel. Your task is to optimize `smooth` (and `avg`) to run as fast as possible. (*Note:* The function `avg` is a local function and you can get rid of it altogether to implement `smooth` in some other way.)

This code (and an implementation of `avg`) is in the file `kernels.c`.

```
void naive_smooth(int dim, pixel *src, pixel *dst) {
  int i, j;

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
```

| Test case | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Method N | 64 | 128 | 256 | 512 | 1024 | Geom. Mean |
| Naive `rotate` (CPE) | 14.7 | 40.1 | 46.4 | 65.9 | 94.5 | |
| Optimized `rotate` (CPE) | 8.0 | 8.6 | 14.8 | 22.1 | 25.3 | |
| Speedup (naive/opt) | 1.8 | 4.7 | 3.1 | 3.0 | 3.7 | 3.1 |
| Method N | 32 | 64 | 128 | 256 | 512 | Geom. Mean |
| Naive `smooth` (CPE) | 695 | 698 | 702 | 717 | 722 | |
| Optimized `smooth` (CPE) | 41.5 | 41.6 | 41.2 | 53.5 | 56.4 | |
| Speedup (naive/opt) | 16.8 | 16.8 | 17.0 | 13.4 | 12.8 | 15.2 |

Table 1: CPEs and Ratios for Optimized vs. Naive Implementations

```
      dst[RIDX(i,j,dim)] = avg(dim, i, j, src); /* Smooth the (i,j)th pixel */

  return;
}
```

### Performance measures

Our main performance measure is *CPE* or *Cycles per Element*. If a function takes $C$ cycles to run for an image of size $N \times N$, the CPE value is $C/N^2$. Table 1 summarizes the performance of the naive implementations shown above and compares it against an optimized implementation. Performance is shown for for 5 different values of $N$. All measurements were made on the Pentium III Xeon Fish machines.(just an example, final measure will be done on Testbed)

The ratios (speedups) of the optimized implementation over the naive one will constitute a *score* of your implementation. To summarize the overall effect over different values of $N$, we will compute the *geometric mean* of the results for these 5 values. That is, if the measured speedups for $N = \{32, 64, 128, 256, 512\}$ are $R_{32}$, $R_{64}$, $R_{128}$, $R_{256}$, and $R_{512}$ then we compute the overall performance as

$$R = \sqrt[5]{R_{32} \times R_{64} \times R_{128} \times R_{256} \times R_{512}}$$

### Assumptions

To make life easier, you can assume that $N$ is a multiple of 32. Your code must run correctly for all such values of $N$, but we will measure its performance only for the 5 values shown in Table 1.

## 5 Infrastructure

We have provided support code to help you test the correctness of your implementations and measure their performance. This section describes how to use this infrastructure. The exact details of each part of the assignment is described in the following section.

**Note:** The only source file you will be modifying is `kernels.c`.

## Versioning

You will be writing many versions of the rotate and smooth routines. To help you compare the performance of all the different versions you've written, we provide a way of "registering" functions.

For example, the file kernels.c that we have provided you contains the following function:

```
void register_rotate_functions() {
   add_rotate_function(&rotate, rotate_descr);
}
```

This function contains one or more calls to add_rotate_function. In the above example, add_rotate_function registers the function rotate along with a string rotate_descr which is an ASCII description of what the function does. See the file kernels.c to see how to create the string descriptions. This string can be at most 256 characters long.

A similar function for your smooth kernels is provided in the file kernels.c.

## Driver

The source code you will write will be linked with object code that we supply into a driver binary. To create this binary, you will need to execute the command

```
unix> make driver
```

You will need to re-make driver each time you change the code in kernels.c. To test your implementations, you can then run the command:

```
unix> ./driver
```

The driver can be run in four different modes:

- *Default mode*, in which all versions of your implementation are run.
- *Autograder mode*, in which only the rotate() and smooth() functions are run. This is the mode we will run in when we use the driver to grade your handin.
- *File mode*, in which only versions that are mentioned in an input file are run.
- *Dump mode*, in which a one-line description of each version is dumped to a text file. You can then edit this text file to keep only those versions that you'd like to test using the *file mode*. You can specify whether to quit after dumping the file or if your implementations are to be run.

If run without any arguments, driver will run all of your versions (*default mode*). Other modes and options can be specified by command-line arguments to driver, as listed below:

-g : Run only rotate() and smooth() functions (*autograder mode*).

-f <funcfile> : Execute only those versions specified in <funcfile> (*file mode*).

-d <dumpfile> : Dump the names of all versions to a dump file called <dumpfile>, *one line* to a version (*dump mode*).

-q : Quit after dumping version names to a dump file. To be used in tandem with -d. For example, to quit immediately after printing the dump file, type ./driver -qd dumpfile.

-h : Print the command line usage.

## 6 Assignment Details

### Optimizing Rotate (50 points)

In this part, you will optimize `rotate` to achieve as low a CPE as possible. You should compile `driver` and then run it with the appropriate arguments to test your implementations.

For example, running driver with the supplied naive version (for `rotate`) generates the output shown below:

```
unix> ./driver
Teamname: hello
Member 1: world
Email 1: se@test.sjtu.edu.cn

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       14.6    40.9    46.8    63.5    90.9
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

**The Baseline CPEs is stored in `config.h`, you can modify and change it by the CPEs on your own machine showed by first execution. The same to Smooth.**

### Optimizing Smooth (50 points)

In this part, you will optimize `smooth` to achieve as low a CPE as possible.

For example, running driver with the supplied naive version (for `smooth`) generates the output shown below:

```
unix> ./driver

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       695.8   698.5   703.8   720.3   722.7
Baseline CPEs   695.0   698.0   702.0   717.0   722.0
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

**Some advice.** Look at the assembly code generated for the `rotate` and `smooth`. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class. The `smooth` is more compute-intensive and less memory-sensitive than the `rotate` function, so the optimizations are of somewhat different flavors.

### Coding Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.

You can only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in these files. bf Please explain your implementation by comments in `kernels.c`!

## Evaluation

Your solutions for `rotate` and `smooth` will each count for 50% of your grade. The score for each will be based on the following:

- Correctness: You will get NO CREDIT for buggy code that causes the driver to complain! This includes code that correctly operates on the test sizes, but incorrectly on image matrices of other sizes. As mentioned earlier, you may assume that the image dimension is a multiple of 32.
- CPE: You will get full credit for your implementations of `rotate` and `smooth` if they are correct and achieve mean CPEs above thresholds $S_r$ and $S_s$ respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.Here the thresholds $S_r$ and $S_s$ are respectively set to 2.3 and 2.7
- Grading:You should be able to achieve the mean of Speedup more than 1.0.
  **The following is score standard**

```
Mean of speedup(rotate($s))          Sore
Top1                                 +5 bonus(tie means no top1)
>= 2.3                               50
2.2                                  47
2.1                                  46
1.0 - 2.0                            45.0*(($s-1.0)/(2.0-1.0))
<= 1.0                               0


Mean of speedup(smooth($s))          Sore
Top1                                 +5 bonus(tie means no top1)
>= 2.7                               50
2.6                                  48
2.5                                  46
1.0 - 2.4                            45.0*(($s-1.0)/(2.4-1.0))
<= 1.0                               0
```

# 7 Testbed

This Lab is quite hardware related. you can complete it on your PC, but it's hard to compare CPE. Testbed is something like a linux server, you can get access to server via SSH, then you will get a linux CLI(Command Line Interface).The server's IP address is `218.193.187.233` , port number is `22`. Account is your student ID(for example: 5120379001) default Password is 123456 (which you should change it after your first login)

## SSH Basic

**Secure Shell (SSH)** is a cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers that connects, via a secure channel over an insecure network, a server and a client (running SSH server and SSH client programs, respectively).

### Windows Platform Guide

This time, you can do the lab on Windows Platform.Just 2 tools needed:Putty and WinSCP.

- **Putty**: Putty is a tiny tool to execute SSH login and remote command execution. you can download it here `http://www.chiark.greenend.org.uk/ sgtatham/putty/download.html`
- **WinSCP**: WinSCP is a SSH file transfer tool. you can download it here `http://winscp.net/eng/download.php`

Install these two tools just by click ”Next”. After Installation, you can pin it to the taskbar.The following is Putty usage step by step:

- **Step 1 Session Configure**: As showed in Figure 3.Execute putty, Click Session Category, in Host Name, type the server's IP address, default port number is 22. in Saved Sessions, type ”perflab”, Then Click Save Button.
- **Step 2 Login**: Double Click on session ”perflab”. a black window will come out, maybe with a warning asking whether or not to store the SHA key information, just say yes. As showed in Figure 4 type your student ID. wait a few seconds, comes the password check, type your password(default is 123456, which you should **change it immediately** after login).
- **Step 3 Execute**: After enter your account and password, a linux CLI comes and you get a Terminal just like you used in Ubuntu. try ”vi test.cc”. as showed in Figure 5 and Figure 7
- **Step 4 Encoding Fix**: When you compile the code, you may get some encoding error( words can't read) that's probably caused by encoding font error. fix it by Load the perflab session. Choose Category Window¿Translation, Set the Remote character set to UTF-8 as showed in Figure **??** then Choose Session Category, Click Save Button to save this change to session ”perflab”

WinSCP usage is almost the same as Putty, it's for remote file transfer. Try it by yourself.

### Linux Platform Guide

People who use Linux must be very familiar with CLI(or Shell) like bash. the command to login would be `ssh -l 5120379001 218.193.187.233` Another command maybe used is ”scp”, which means ssh copy. the command would be like `scp kernels.c 5120379001@218.193.187.233`

### Summarize for Testbed section

In this Section, you learn how to get access to a remote SSH Server. Indeed, I suggest using Windows Platform this time. the command to change password is `passwd`.

## 8 Hand In Instructions

When you have completed the lab, you only need to commit the `kernels.c` to svn server.

**Good luck!**

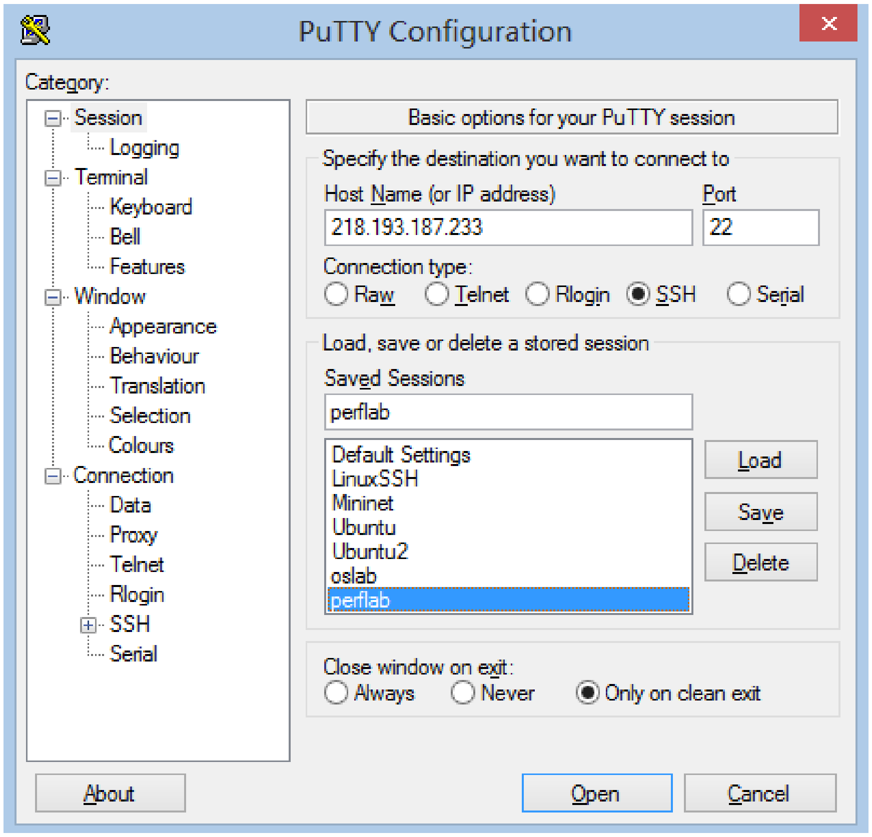


Figure 3: putty init

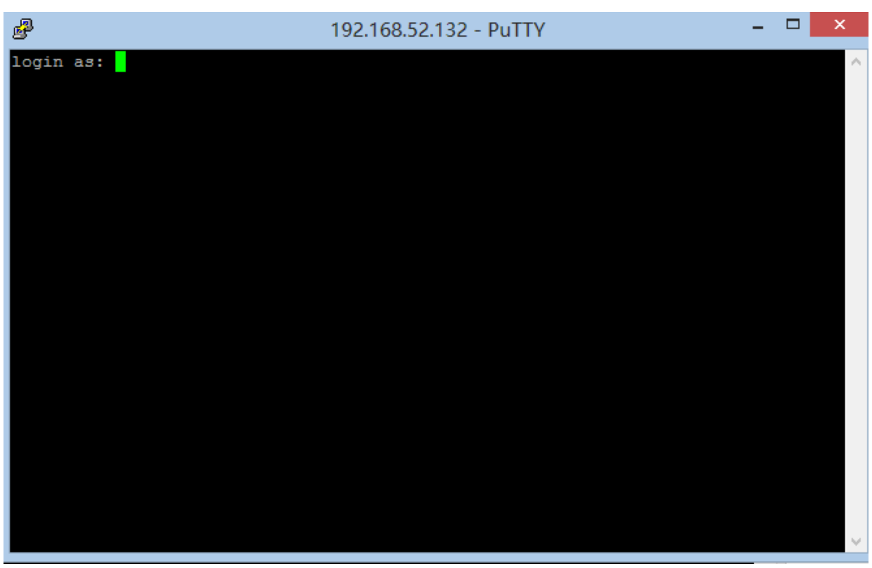


Figure 4: putty login

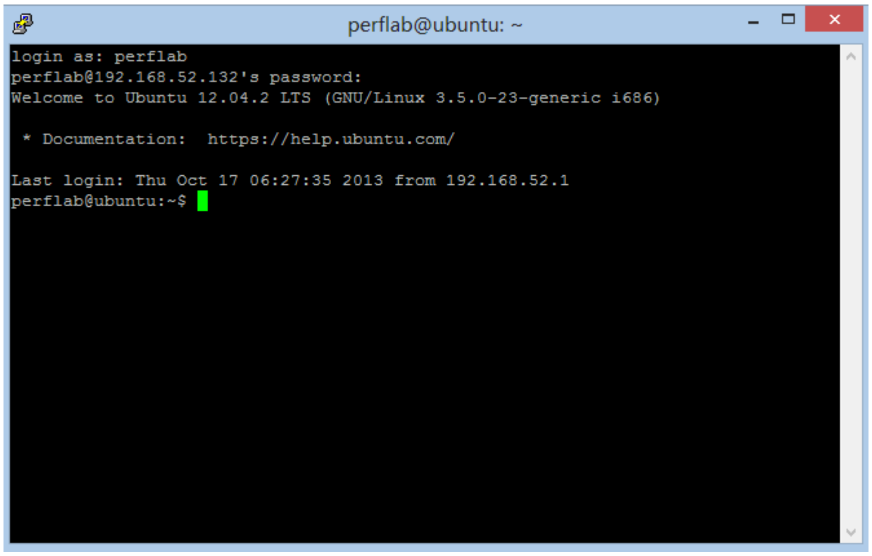


Figure 5: putty execute

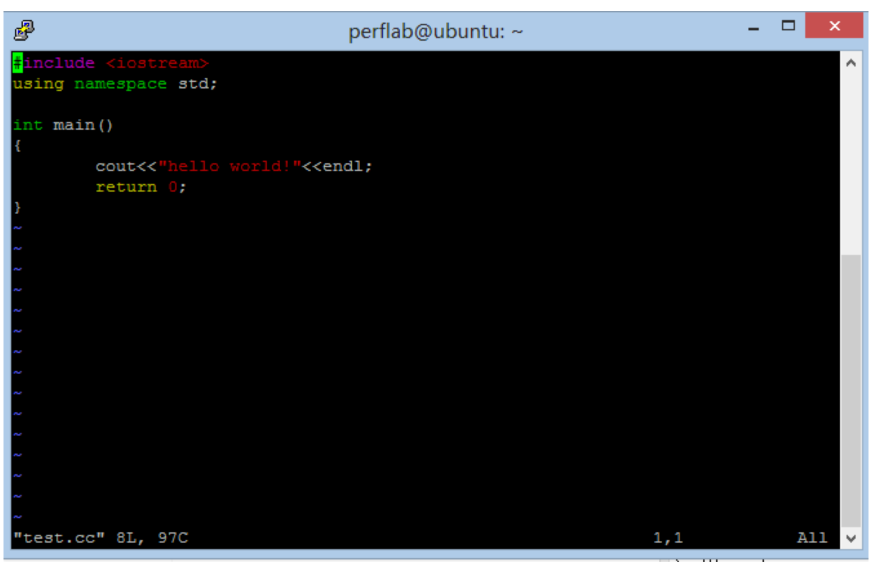


Figure 6: putty vi

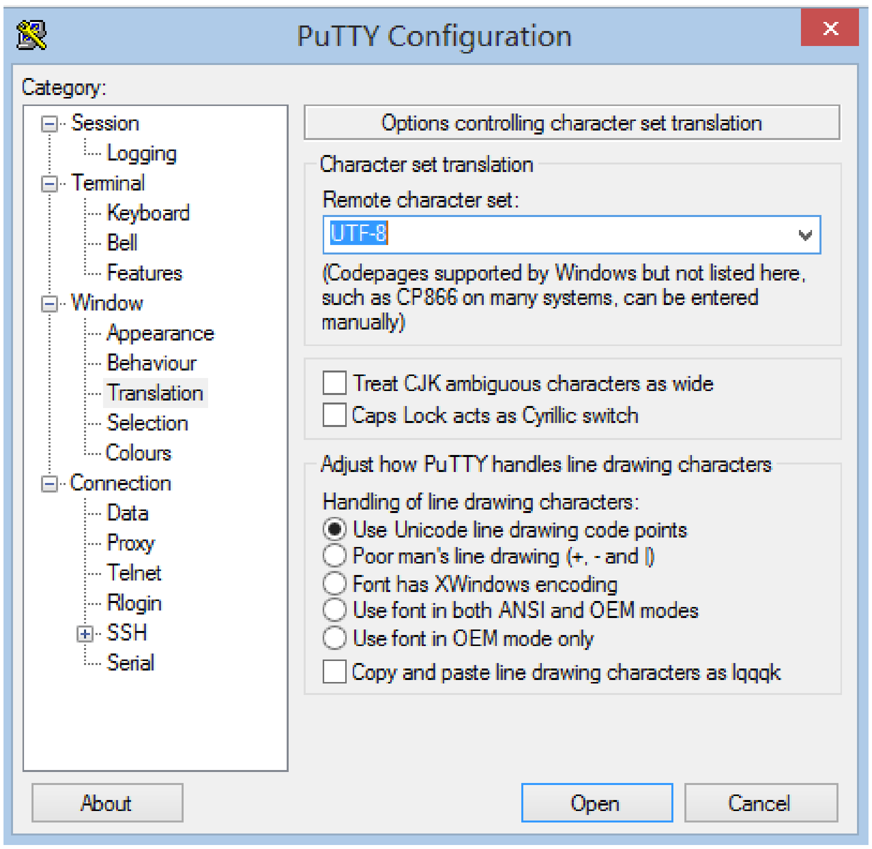


Figure 7: putty encoding