Project Instructions

1 Introduction

The Swedish Meteorological and Hydrological Institute (SMHI) routinely records the temperature at various locations in Sweden. In some places, this has been going on for hundreds of years. In 2013, the institute published its oldest dataset with average daily temperatures of Uppsala starting from January 12:th 1722 and ending in 2013. More recent datasets for other locations are also available as part of the SMHI OpenData initiative. A collection of SMHI datasets are made available on the course webpage. There is clearly a wealth of information hiding in these datasets! Use your skills as a programmer and as a scientist in order to make some interesting observations regarding the Swedish climate!

2 Project description

Select one of the available SMHI datasets and write a program that extracts information from the Swedish climate data. Use plots in order to visualize your results. Use ROOT to make plots and eventual statistical analysis: a small code skeleton (see Section 4) is provided to get you started. Section 3 contains a few examples of what kind of information you could get from the data. Decide on *at least three* interesting results to produce. At least one of them should be your own idea and should *not* come from Section 3. Use your imagination! Start by looking at the data and understanding the content of the files and the format, pay attention to descriptive metadata. Note that the data within one dataset can be of different quality, so you might consider skipping "bad" records.

In the real world, scientists share code and collaborate. In this project, you will work together with your colleagues as a team. All team members must have a `github.com` account (the same we used for the course) and their contribution must be seen as authorship on github commits. Each group will appoint one team member as the release manager. The release managers will

create a repository on their `github.com` account and define the policies with which the other team members will add their code to the repository.

One policy (let's call it **A**) could follow the way we did during the course, where Florido was the release manager and all students would fork its *upstream* repository into their own *remote origin master* and send *pull requests* to release manager's *remote upstream*. In this way the release manager is the only one to decide what changes go in the main *upstream* repository. This is the best way to keep the code and changes clean.

Alternatively (policy **B**), the release manager can add all the team members as *collaborators* to the created repository, and everyone can *clone*, *pull* and *push* to the same main repository as their *remote origin*. In this case all the team members are responsible to keep the code tidy, and the release manager must be able to rollback changes when these are not acceptable. This can be complicated if there is no deep understanding of git by all team members, so use it only if you know what you're doing.

Either way, make sure you create a *branch* for each big change or contribution you make, and then ask the release manager to *merge* the *branch* to the *upstream master* (policy **A**), or do the merge yourself in your *working directory* before pushing to the *remote origin* (policy **B**). It might be helpful to repeat the interactive tutorial we did during the course to understand these concepts.

Add your code, draft reports and other notes in the `github.com` repository. It is better if those are in separate folders.

We will only accept projects that can be checked by reaching the respective `github` URL. You should provide us with such URL[1]. For example, Florido's repository for the course had such URL:

https://github.com/floridop/MNXB01-2017/

By using `git` you and your team members can work independently, committing the changes to their own fork repository as you go. When working on this kind of project, it is customary to keep a log. The `git` repository should contain a file called `ChangeLog`, where you document your progress. Add a date and a short descriptive comment to this log for each major change that you make to the code. The examiners should be able to understand how did the code develop by reading such a log. As a first step of the project work, every team should prepare a short `Workplan` document containing the planned distribution of sub-tasks that have to be done in order to achieve the goal. Discuss and agree with your colleagues early on what sort of code needs

[1] Send the link by e-mail to Oxana

to be written and agree preliminarily on who should write what in order to avoid unnecessary collisions. Record the task distribution in the `Workplan` document in the `git` repository and, if necessary, update it regularly. If you get stuck or if you realize that the code needs to be restructured, talk to your colleagues. This is a team project!

Naturally, you need to document not only your code but also your results. Use LaTeX to write a scientific report that describes the results that you managed to produce and the approaches that you used, such as short description of classes, functions, methods etc. The report should be complete with plots to back up your claims. Scientists collaborate on their reports in very much the same way as they collaborate on their code. Begin by splitting the report into many `.tex` files that you `\input` from a main file. Use one `.tex` file for each result that you decided to produce. Then put all of the report code in `git`. Even for very big reports, if everyone is working just on their own part of the report using the appropriate `.tex` file, problems with collisions are rare.

3 Example results

This section contains a few examples of what you could do with the data. Use them as guidelines and as inspiration. You don't have to do everything the examples say, but more complete implementations earn you both brownie points and (probably) a better grade. Remember to comment your code!

3.1 The temperature of a given day

Use the data to make a histogram of the temperature of a given day of the year. An example of such a histogram is shown in Figure 1. If you choose to implement this example, uncomment either of the two functions called `tempOnDay`. One of the functions accepts a month and a day as arguments. The other one accepts a date (which is an integer in the range 1 to 365, or 366 if you design your code to understand leap years). Implement one or both. If you have one, the other should be simple!

- Create a histogram like the one shown in Figure 1.
- What is the mean temperature of the given day?
- What is the standard deviation of the temperature?
- Can you predict the probability of observing a particular temperature?

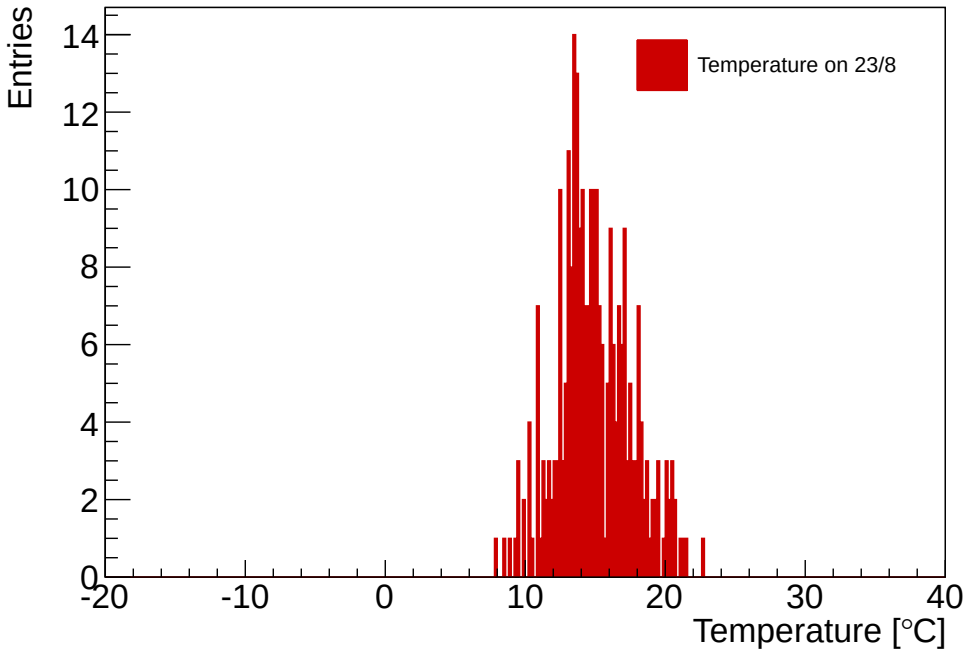


Figure 1: Histogram showing the temperature on August 23:rd every year since 1722.

Hints In ROOT, you often write functions that create one or more plots. If you want to look at those plots after the function completes, you have to put the histograms on the heap. If you put them on the stack, they will go out of scope and be deleted automatically once the function completes. Once a histogram is deleted it disappears from all plots. This can lead to a situation where you create histograms on the heap that you never delete. Technically, this fits the description of a memory leak. It's a design flaw of ROOT that it encourages memory leaks in this way. But since you will only be creating a handful of histograms for plotting, you shouldn't worry about it. A handful of histograms won't cause the program to run out of memory. The example code below shows how to create a histogram of integers with 365 bins between 1 and 366. The title of the axes can be specified directly in the constructor using the trick shown. We then set the fill color to a darker red, increment a bin and calculate some properties of the distribution. Finally, we create a new canvas and draw the histogram.

```
TH1I* hist = new TH1I("temperature", "Temperature;Temperature
    [#circC];Entries", 300, -20, 40);
hist->SetFillColor(kRed + 1);
hist->Fill(-3.2); //Increment the bin corresponding to -3.2 C
double mean = hist->GetMean(); //The mean of the distribution
double stdev = hist->GetRMS(); //The standard deviation
TCanvas* can = new TCanvas();
hist->Draw();
```

3.2 The temperature for every day of the year

If we can create a histogram showing the temperature of *one* day, why not do it for every day of the year? Uncomment the `tempPerDay` function and try it out! Figure 2 shows how the mean temperature varies throughout the year. Of course, just knowing the mean is not all that interesting. In order to make useful predictions, we also need to know the standard deviation of the temperature of each day.

- Plot the mean temperature of each day of the year.
- Use error bars in order to visualize the standard deviation.

Hints The example code shows how to loop over every bin in a histogram and explicitly set the contents and error of each one. In ROOT, bin 0 is the underflow bin. This bin is not shown on plots. Bins 1 to `nBins` are what you see when drawing the histogram. Finally, bin `nBins` + 1 is the overflow bin.

```
for(int bin = 1; bin <= hist->GetNbinsX(); ++bin) {
  hist->SetBinContent(bin, 5);
  hist->SetBinError(bin, sqrt(5.));
}
```

3.3 The warmest and coldest day of each year

In Sweden, summers and cold and winters are colder. But which day is actually the coldest? Implement the `hotCold` function and find out! Figure 3 shows when the warmest and coldest days have typically occurred during the last few hundred years. To make things more interesting, note that each dataset may contain readings from different places (for example, the Uppsala

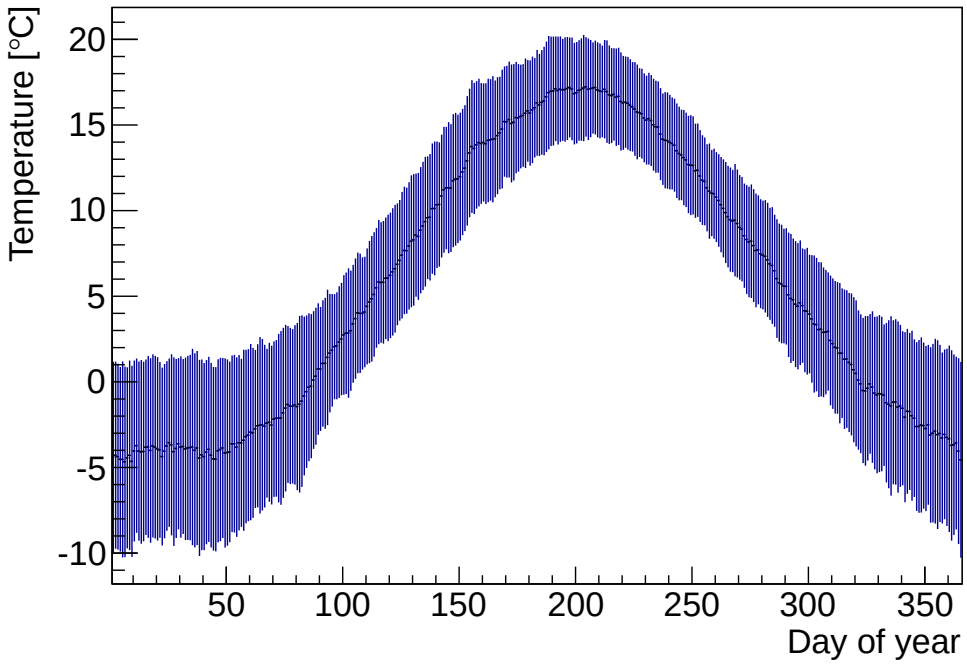


Figure 2: Histogram showing the mean temperature on each day of the year.

dataset has a few readings from Stockholm). If you use the Uppsala dataset, write your function such that all readings from a region other than Uppsala are ignored.

- Create histograms of the warmest and coldest day each year.
- Predict when the warmest and coldest day is most likely to occur.
- Can you show both histograms in the same plot?
- Can you make a fit function that "wraps around" as in Figure 3?

Hints The mean of a histogram could be obtained with the `GetMean` member function. A more fancy, and sometimes more useful, way of extracting information from a distribution is to fit a mathematical model to it. The example code shows how to define a custom function (a Gaussian in this case) and then fit that function to a distribution contained in a histogram. The function is defined in the range $[1, 366]$ and takes three parameters. They are (as defined in the `Gaussian` function) an amplitude, a mean and a standard

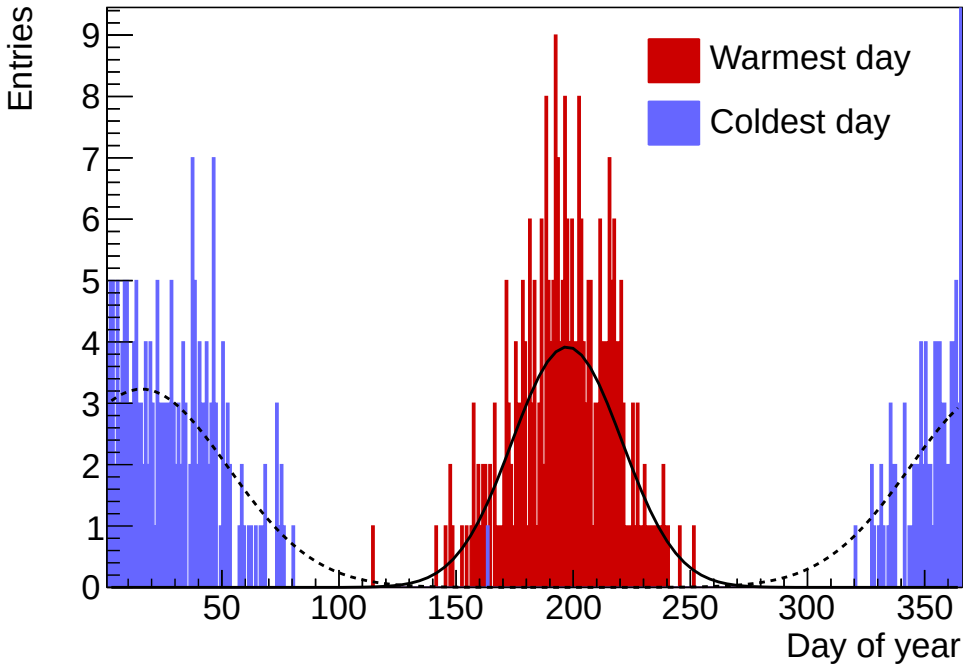


Figure 3: The plot shows how often a given day of the year was the warmest or coldest for every year since 1722. Lines show Gaussian fits of the distributions.

deviation. The meaning of the parameters passed to the `Fit` function is to fit quietly (Q), to not automatically plot the function when the histogram is plotted (0) and to fit only within the range in which the function is defined (R). After fitting, we print the mean of the Gaussian as well as the uncertainty of that mean. Finally, the example shows how to create a legend containing two histograms and then draw those histograms in the same plot.

```
double Gaussian(double* x, double* par) { //A custom function
  return par[0]*exp(-0.5*(x[0]*x[0] - 2*x[0]*par[1] +
    par[1]*par[1])/(par[2]*par[2]));
}

TF1* func = new TF1("Gaussian", Gaussian, 1, 366, 3);
func->SetParameters(5, 200, 50); //Starting values for fitting
hist->Fit(func, "QOR");
cout << "The mean is " << func->GetParameter(1) << endl;
cout << "Its uncertainty is " << func->GetParError(1) << endl;
```

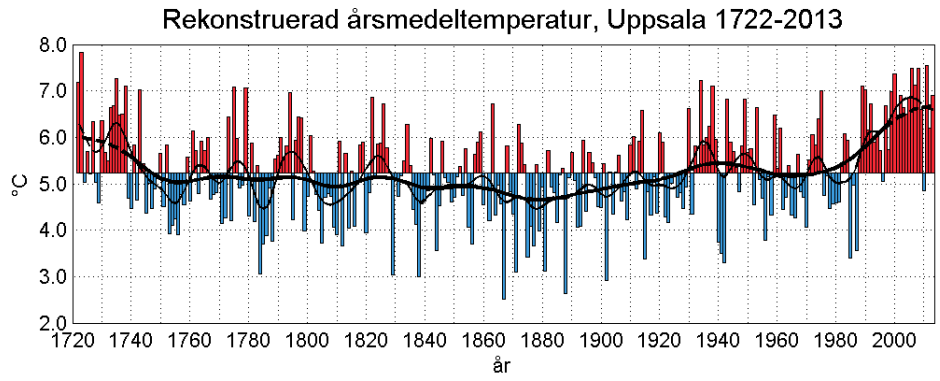
```
TLegend *leg = new TLegend(0.65, 0.75, 0.92, 0.92, "", "NDC");
leg->SetFillStyle(0); //Hollow fill (transparent)
leg->SetBorderSize(0); //Get rid of the border
leg->AddEntry(hist, "", "F"); //Use object title, draw fill
leg->AddEntry(anotherHist, "A title", "F"); //Use custom title

hist->Draw();
anotherHist->Draw("SAME"); //Draw on top of the existing plot
leg->Draw(); //Legends are automatically drawn with "SAME"
```

3.4 The mean temperature of each year

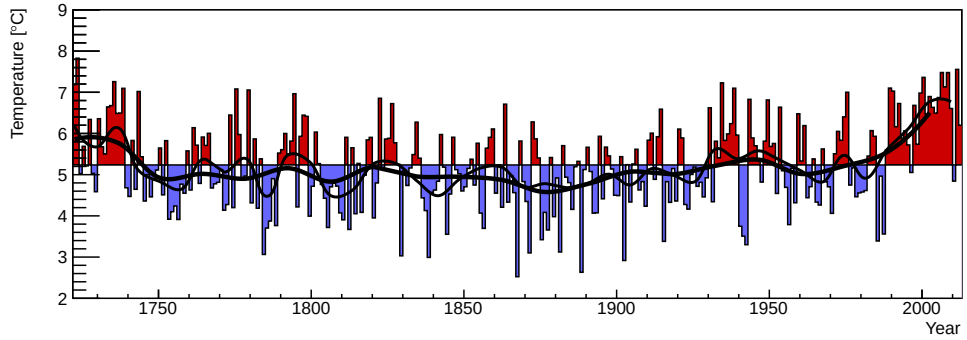
SMHI has produced some very nice plots that show the mean temperature of each year since the temperature measurement started. An example is shown in Figure 4 (a). Can we make something similar in ROOT? Of course we can. Figure 4 (b) shows one such attempt. Try to make your own plot with the `tempPerYear` function. The function accepts one argument. This is a future (or past) year for which to predict the temperature using extrapolation.

- Plot the mean temperature of each year in your dataset.
- Fit the data with a model of your choice. Use the model to predict the temperature of a given year in the future. Can you draw any conclusions when it comes to global warming?
- Can you plot the deviation from the average as in Figure 4?
- Can you plot the moving averages?

Hints The neat deviation-from-the-average effect is actually obtained by plotting several histograms in different colors on top of each other. Try it! A moving average can be done using, for example, a graph. Add one point to the graph for every few years in the histogram. The y-value of the point should be the average of the surrounding years. Then plot the graph with a smooth line between the points. The example shows how to create a new graph and fill it with one point for each bin in a histogram. The call to `Expand` increases the capacity of the graph by 100 if adding a new point would make the graph run out of space. Finally, the example shows how to draw the graph (on top of the current plot) with a smooth curve.



(a)



(b)

Figure 4: The plots show the mean temperature of each year since 1722. Positive and negative deviations from the mean are indicated by different colors. Lines show moving averages. Both a plot from SMHI (a) and one from ROOT (b) are shown.

```
TGraph* graph = new TGraph();
for(int bin = 1; bin < hist->GetNbinsX(); ++bin) {
  graph->Expand(graph->GetN() + 1, 100);
  graph->SetPoint(graph->GetN(), hist->GetBinCenter(bin),
    hist->GetBinContent(bin));
}
graph->Draw("SAME C");
```

4 Code skeleton

The code skeleton contains a file called `rootlogon.C`. If ROOT is started from a directory that contains a `rootlogon.C`, it will execute all the statements in the `rootlogon` function automatically. This is a convenient way to set up the environment, e.g. by executing some style scripts that determine how plots should look. The provided `rootlogon.C` makes some basic changes to the default plots. The code skeleton also comes with a class called `tempTrender` and a script called `project.cpp`. These files are compiled and loaded automatically by the `rootlogon` function. As you can see by looking at the files, they are almost empty. It is your job to implement the `tempTrender` class and any other classes or functions that you might need. The datasets from SMHI are in the directory called `datasets`. Each group should pick a dataset (or a couple) to work with. Good luck with the project!