Profiling & Optimising Contur

Sean Bray

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Physics

University College London

August 25, 2021

I, Sean Bray, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

My research is about stuff.

It begins with a study of some stuff, and then some other stuff and things.

There is a 300-word limit on your abstract.

Contents

List of Figures

Chapter 1

Introduction

Good software development does not just entail writing functional code. Factors that don't impact the final output of the code, like clear well written syntax, inclusion of code testing, and making efficient use of CPU resources for the task at hand can significantly increase the usability and reliableness of software in a research setting. This thesis focuses on making efficient use of CPU resources, or more intuitively writing code that does the required task as fast as is possible with the tools at hand. The project focuses in particular on the **Contur** package, which is open source software used in particle physics to test the consistency of theoretical predictions from newly proposed physics models against realised experimental data. The project entailed profiling the **Contur** code base and from the profile results identifying parts of the code where potential inefficiencies exist which can be improved upon. A brief summary of the contents of each chapter of this thesis follows below.

To start the thesis in Chapter 2 we attempt to provide the necessary **Contur** background. This background contains a brief sketch of the current situation within particle physics research that creates a gap for a tool like **Contur** to be useful, followed by an overview of the **Contur** package itself in terms of what it does and how it works. The outline of the workings of **Contur** will be necessarily brief but should be sufficient to give the reader the required background to understand later changes made to better optimise the code.

From Chapter 3 on in this thesis we start to discuss work actually carried out by this author as part of the research project. Chapter 3 specifically begins with a

discussion of the benefits to **Contur** users arising from the improved runtime which will result from code efficiencies. The chapter then moves onto outlining the tools used to perform and visualise the profiling results. The chapter then concludes by presenting the profiling results for the last version of **Contur** that exists before we started this project. This initial profile is taken as the bench mark to judge the effectiveness of later attempts to optimise **Contur**.

In Chapter 4 we digress slightly from the main theme of the project to discuss introducing automatic tests into the **Contur** package. When making code changes in **Contur** for optimisation purposes, an important consideration is that we don't unintentional break or introduce errors into **Contur** . Especially errors that will have an impact on the final output of **Contur**. One means we can reduce the chances of introducing errors is via robust testing infrastructure. Chapter 4 outlines the step taken as part of this research project to improve **Contur's** testing infrastructure.

Chapter 5 then finally outlines the changes made to **Contur** as part of this research project. Each of the changes outlined has been made because they can carry out the exact same process as the existing **Contur** code in a more efficient way. For each change made we give the background necessary to understand the purpose of the section of code we are changing, we explain the change we made and we give updated profiling results showing the impact of the change in terms of run time.

We will then conclude the thesis in Chapter 6 by summarising the accomplishments of the project. Additionally the final profile results of **Contur** after all the optimisation changes have been implemented will be presented. From these final results we can understand the most computational intensive remaining parts of a **Contur** run thus providing the target areas for future optimisation attempts with **Contur**.

Chapter 2

Contur Overview

2.1 Contur and the Standard Model

The standard model (SM) of particle physics is the name for the collective of quantum field theories that successfully describe three of the four fundamental forces of nature[1] and the observable matter content of the universe. The success of the SM is evidenced by the generally strong agreement between SM predictions and experimental data. Despite this success there is a widespread view that the SM is not the complete picture. This viewpoint arises from the inability of the SM to accommodate gravity or provide any insight into postulated quantities like dark matter[2] and dark energy[3]. As a result the task of the developing new fundamental theories in physics that extend the SM, which we will term Beyond Standard Model theories, is an active area of research.

Contur(Constraints On New Theories Using Rivet) has been developed to aid the search for new BSM theories. The approach aims to leverage the large amount of experimental data produced at the Large Hadron Collider to set constraints on the type of BSM theories which are possible. This is done by considering experimental results which have already been shown to agree with SM expectations. The **Contur** procedure then asks the question for a given completely specified BSM theory, where

---

[1] The forces successfully described are the electromagnetic force, the weak force and the strong force, while a quantum theory of gravity still eludes us

[2] Dark matter is hypothetical form of matter postulated to explain discrepancies between observed astrophysical motion and what we would expect based off the gravitational equation of motion in general relativity

[3] Dark energy is a hypothetical force postulated to explain the observed expansion of the universe

by completely specified we mean that the values of all free parameters have been set, "at what significance do existing measurements which agree with the SM already exclude this BSM".

The approach thus checks the consistency of predictions of the BSM theory against experimental results that have already been shown to align well with SM predictions. The idea being if the BSM theory is inconsistent with the experimental data and fails to predict processes accurately that are already well understood within the SM, then the BSM with parameter values we specified is not viable. Through running **Contur** for the BSM theory for multiple different free parameter values we can rule out parameter values which are inconsistent with realised data or potentially even rule out the whole BSM theory if there are no values for the free parameters of the BSM theory that produce consistent predictions.

2.2   Input Data

Contur's consistency checks compare simulated data from a new BSM theory against realised experimental data. We will now outline in greater detail how these two main sources of data are sourced.

2.2.1   Simulated BSM Data

At first sight the need to produce a large amount of simulated results for a BSM theory for a **Contur** run would seemingly jeopardise the desire that **Contur** to be a relatively quick and easy way to check the consistency of the BSM theory with realised data. If for every BSM theory run on **Contur** scripts of bespoke ancillary code was necessary to produce simulations of the BSM theory something like the **Contur** procedure would likely not be practicable.

Fortunately there exists mature packages in the particle physics community specifically devoted to the tasks of simulating particle collision events. These packages are sufficiently flexible that they can be used to simulate both SM events and a wide array of BSM events. A large part of this flexibility comes from the Universal FeynRules Output (UFO)[1], which allows the encoding of the Feymann diagram information of a process in a standard form which can then be passed to

an event generator which can use Monte Carlo methods to simulate the event in question.

Contur is event generator agnostic in the sense that any generator that can produce simulations with data in the required final format could be used. Yet, despite this, the main event generator currently used by **Contur** is **Herwig**[2] and the default assumption within the current set up is that the user generates BSM data with **Herwig**. For a fully specified BSM theory **Herwig** will generate events for a range of scenarios, with the output of these simulations being a collection of histograms[4], where any given bucket in these plots is the number of signal events which were counted in that bucket for the BSM theory. All of this output is stored in a single **YODA** file[3], this **YODA** file contains all the simulation data required to run **Contur** for a fully specified BSM theory.
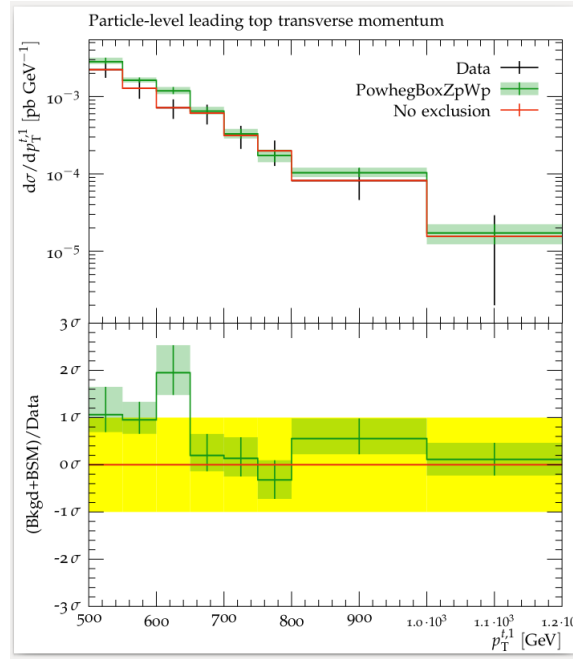
Figure 2.1: Example of histogram used in Contur

Recall that we previously defined a fully specified BSM theory to be the BSM with all free parameters given a single value. So running **Contur** on one fully specified BSM will only check the consistency of the BSM for the parameter values we have specified, different parameter values for the BSM will possible give different

---

[4]See figure 2.1 below for an example of a histogram used

signal events changing the output. Thus for any given BSM it is likely to be the case that we want to produce multiple **YODA** files containing simulated signal events for the BSM using different parameter values across **YODA** files.

Contur has tools to create a collection of such **YODA** files which is termed a grid. These tools sit within **Contur's** batch process capabilities[5]. It is easiest to understand this process if we think of a simple example if we have a BSM theory with just two free parameters with values that can range from 1 to 10^6, from this **Contur** batch can create a grid a 10×10 grid of 100 **YODA** files[7]. We can then pass this grid to **Contur** and run a consistency check for each point on the grid, this is an example of a **Contur** grid run. The grid run is the main area of **Contur** we will later focus on in our optimisation efforts.

2.2.2 Experimental Data

Contur sources experimental data via a combination of the HepData[10] repository and Rivet[11]. The HepData repository contains a digitized record of detector level results for run experiments. The Rivet library contains a collection of routines to account for the specifics of the detector and convert the detector level results to particle level result independent of detector effects. These particle level measurements can then be directly compared with the particle level signal effects simulated by the BSM. Carrying out the comparison at particle level ensures that the results being compared do not contain theory dependent extrapolations within them, making the Contur constraints arrived independent of theoretical assumptions.

The interface with Rivet is built into Contur, so the user does not need to provide any Rivet input when initiating a Contur run. Instead from experiments simulated in the BSM data passed in the yoda file Contur can call pull the appropriate experimental data from Rivet and HepData. In addition to realised results for experiments, simulated SM expectations can also be sourced from Rivet. These simulated SM expectations are created in the same way as the BSM simulations

---

[5]See Chapter 4 of the **Contur** User Manuel[4] for further details

[6]For simplicity take the parameters to be dimensionless

[7]**YODA** file one will be when both parameters have values 1, **YODA** file two the first parameter will have value 1, the second value 2 etc....

discussed in the previous subsection and stored in yoda files that can be accessed via Rivet. In its default run Contur does not make use of SM expectations, however there exists on optional theory run in Contur which we will discuss in the next section which makes use of SM expectations.

2.3 Calculating Likelihoods

The final output of a **Contur** run on a single **YODA** file is a single CL_s exclusion limit, given in the form of the CL(s) technique[12]. This exclusion limit is an expression of our confidence that a fully specified BSM theory produces signal events inconsistent with understood SM processes. The calculation of these exclusion limits is the core of the **Contur** procedure, we will now give a high level overview of the steps involved in this calculation, let us start with the default **Contur** run on a single **YODA** file.

The CL_s exclusion limit is defined to be a ratio of p-values,

$$CL_s := \frac{CL_{s+b}}{CL_b} = \frac{p_{s_b}}{1 - p_b},$$

where in the above p_{s+b} is defined as the p-value for the signal plus the background event and p_b is the p-value just for the background event. For each histogram **Contur** computes one CL_s by either taking the maximum CL_s out of all the bins that compose the histogram or if correlation information between the bins exists using this to compute a single CL_s for the histogram. For any given bin in a histogram, for the default **Contur** run p_b will always be a half, because we are just comparing the background data with itself, so p_{s+b} will compute the quantity of interest, namely how well the signal count for the BSM approach aligns with the realised count. The below code snippet shows a simplified example of the flow of the code for the CL_s calculation, the calculation takes place within the `likelihood` class within **Contur**.

Listing 1: Compute CLs For Histogram

```python
def EvaluateHistogram(Histogram):

    if Histogram has correlation and (build correlation =True)

        return Likelihood(Histogram)

    else:

        return max(Likelihood(Histogram.bins))
```

After computing a CL_s for each histogram the next step is to bucket the histograms into statistically independent pools. In **Contur** a pool is combination of the final particle state, the experiment and the beam energy of the experiment, each of the three chosen so that their combination is statistical independent. Histograms that share these three properties are grouped into the same pool. **Contur** then takes the histogram with the maximum CL_s in each pool. A snippet of example code to used to carry out this process for a single pool is given below

Listing 2: Sort Likelihood blocks into pools

```python
def EvaluatePool(Pool):

    scores = [] #empty list for results

    if Histogram in pool:

        scores.append(EvaluateHistogram(Histogram))

    return Histogram with max(scores)
```

Finally **Contur** takes the histograms from each pool and combines into a single histogram to calculate a final CL_s for the **YODA** file. In combining histograms like this **Contur** assumes each of the pools to be statistical independent, which they are by construction. A simplified example of the code to perform this step is given by

Listing 3: Build Full Likelihood

```python
def BuildFullLikelihood():

    tests = []

    for Pool in ConturPools:

        tests.append(EvaliatePool(Pool))

    return Likelihood(tests)
```

The main alternative run option also available in **Contur** is a theory run. In the theory run when computing the background p-value p_b, instead of just trivially comparing the data with itself, the SM expectations are used. So the theory **Contur** run provides more of a relative measure of which of the SM expectations or BSM expectations are in better agreement with the realised data.

As already highlighted the most common way **Contur** is used in practice is on a grid of **YODA** files as opposed to just a single file. The **Contur** grid run is not fundamentally different from the single **YODA** run, for the grid run **Contur** runs iteratively through all the **YODA** files in the grid, at each iteration calculating a single final CL_s for the **YODA** file at that point. So the final output of the **Contur** grid run is a CL_s value for each point on the grid. This grid of CL_s values output can be presented as a heat map, an example of which can be seen in figure
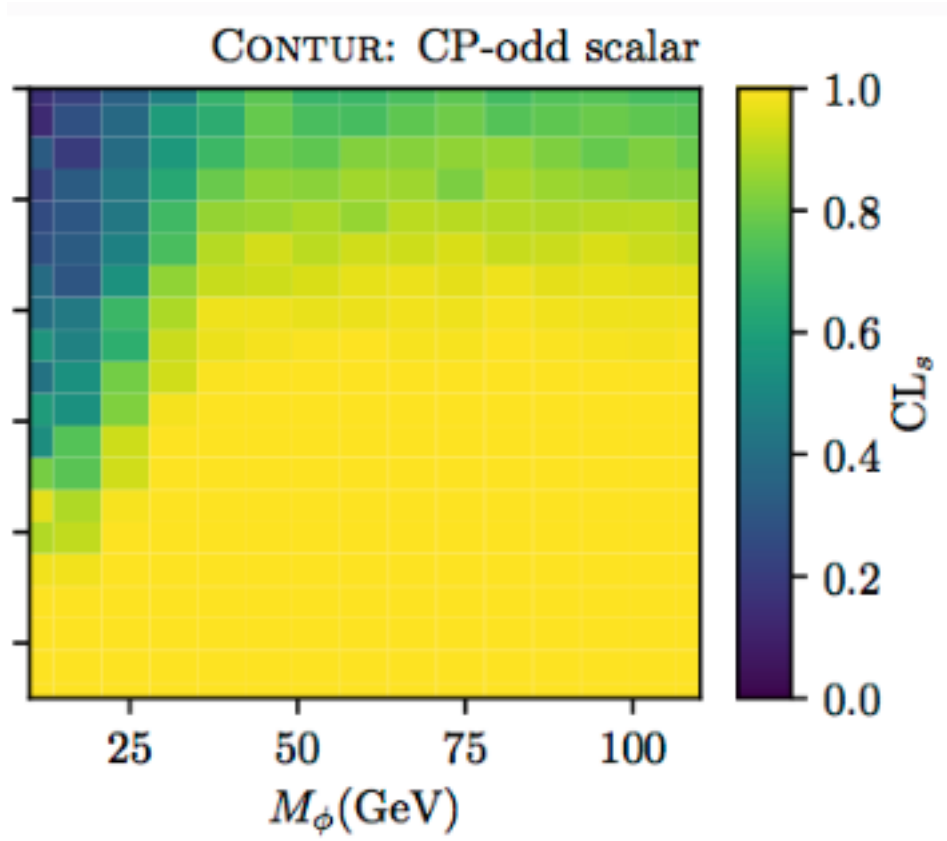
Figure 2.2: Example of heat map from grid run

Chapter 3

Profiling Contur

Before any attempt can be made to optimise code within **Contur**, it is first necessary to identify processes within the code where potential inefficiencies exist and we can make improvements. The best way to do this is to perform a profile of the code. A good profile should give us visibility not just on the total runtime of **Contur**, but how this run time breaks down between the different processes that compose a **Contur** run.

This section will outline the steps taken to produce a profile of **Contur** and how we used the results. We will start by introducing **cProfile**, which is the Python profiler which was used to carry out the profile. Then we will discuss **Snakeviz** and **gprof2dot**, these are the two tools which we used to visualise the profiling results produced by **cProfile**. Finally we will conclude the section by performing an initial profile of the **Contur** package before any code optimisation was attempted. This initial profile will serve as our benchmark to measure the effectiveness of our later attempts to improve the run time performance of **Contur**.

First however let us briefly consider why it is a worthwhile effort to try and improve the runtime of **Contur** code.

3.1   Why attempt to optimise Contur?

The obvious answer to the question "Why attempt to optimise **Contur**?" is simple that we will have code that runs faster. The immediate benefit of faster code for a researcher using **Contur** is that the wait time from run initiation to results is reduced,

all else equal this should increase productivity.

We can argue further though that faster code increases the range of analysis that a researcher can perform with **Contur**r. This argument follows from the observation that there likely exists a runtime above which **Contur** becomes impractical to use as a research tool[1]. If we combine with this the observation that will be discussed later in Chapter 4 that the runtime increases with the size of the grid used, then we can see that runtime puts an upper bound on the size of the grid that can be run with **Contur**. Improving run time will thus likely increase this upper bound which would allow researchers either to increase the span of a parameter space they evaluate, or look a the parameter space with a greater level of granularity.

A final obvious benefit of run time improvements follows from the fact that currently grids that are too large to run locally will be run on a HPC cluster. Increasing the range of grids that can be run locally will thus decrease the volume of runs going to the cluster, saving valuable CPU resources. Additionally for grids that still need to go to the cluster, making **Contur** code more efficient will reduce wasteful usage of the HPC CPU resources.

3.2 Profiling with cProfile

3.2.1 Why cProfile?

Let us consider some of the features we ideally require from our chosen profiler. At a minimum a profiler must obviously be able to time how long it takes our code to run. This basic requirement is essential to be able to determine if our attempted improvements to the code do in fact actually improve run performance. In addition to just providing the total run time of **Contur** we will also require our profiler to provide a split of the runtime among the functions/sub-functions which compose **Contur**. A split of the run time like this will highlight parts of the code that consume disproportionately large amounts of CPU or are repetitively called, suggesting optimisation improvements can be made.

cProfile[5] is a module within the Python standard library which meets these

---

[1]To take an extreme example, if the code takes over 24 hours to run, its utility to a researcher will be much less than code that takes under an hour to run.

requirements. Our main motivations for using **cProfile** are as follows:

- Provides a full profile of program with output include total run time, time taken at each individual step, and number of calls to individual functions;

- Easy to save the output of the profile in prof files which can then be read by tools built to visualize profiling results;

- Performing the profile with **cProfile** is quick and easy and requires minimal new code;

3.2.2 Using cProfile

We will demonstrate the usage of **cProfile** by profiling the last version of **Contur** which existed before any optimisation attempts were made[2]. All **Contur** code can be found in the main **Contur** repository[6], additionally all code contributions for this thesis can also be found as a commits in the main repository.

For the demonstration we will walk through the steps to profile a single **YODA** file. The steps required to perform the profile on a **Contur** grid run are the same, so the in the conclusion for this chapter we will just provide the results of the profile for the grid run without repeating the walk through.

The simplest way of performing a profile with **cPorfile** is via **cProfile's** run method. To profile **Contur** using the run method we just pass **Contur's** main function to the run method. We can make this adjustment to **Contur's** code by updating the main run script[3] as follows

```
import cProfile


if __name__ == "__main__":
        cls_args = get_args(sys.argv[1:],'analysis')
        cProfile.run("main(cl_args)", sort=cumtime)
```

After updating the run script as above we can now run **Contur** as normal to get the below terminal output from the profile

---

[2]The version can be found in commit 49a67e03

[3]Can be found here

```
Parameter values not known for this run.
INFO - Combined exclusion for these plots is 95.45 %

        17275900 function calls (17255906 primitive calls) in 20.838 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3/1    0.000    0.000   21.311   21.311 {built-in method builtins.exec}
        1    0.001    0.001   21.311   21.311 <string>:1(<module>)
        1    0.000    0.000   21.310   21.310 run_analysis.py:368(main)
        1    0.000    0.000   21.308   21.308 depot.py:101(add_point)
        1    0.000    0.000   20.656   20.656 yoda_factories.py:843(__init__)
        1    0.061    0.061   20.655   20.655 yoda_factories.py:856(__get_likelihood_blocks)
        1    0.153    0.153   16.199   16.199 yoda_factories.py:31(init_ref)
       33    8.617    0.261    9.028    0.274 {yoda.core.read}
     2963    0.692    0.000    4.513    0.002 yoda_factories.py:113(<listcomp>)
  1831930    1.120    0.000    3.893    0.000 aopaths.py:16(stripOptions)
  1832310    0.835    0.000    2.776    0.000 re.py:203(sub)
      380    0.027    0.000    1.948    0.005 yoda_factories.py:295(__init__)
      380    0.026    0.000    1.417    0.004 yoda_factories.py:653(__fillBucket)
  1835172    0.801    0.000    1.347    0.000 re.py:289(_compile)
     1014    0.488    0.000    1.298    0.001 yoda_factories.py:96(<listcomp>)
      381    0.011    0.000    1.131    0.003 plotinfo.py:317(mkStdPlotParser)
      382    1.113    0.003    1.113    0.003 {rivet.core.getAnalysisPlotPaths}
      380    0.006    0.000    1.080    0.003 likelihood.py:52(__init__)
       53    0.003    0.000    1.052    0.020 likelihood.py:110(__pval)
     4072    0.030    0.000    0.964    0.000 likelihood.py:174(__ts_to_pval)
     4072    0.238    0.000    0.934    0.000 _distn_infrastructure.py:1902(sf)
       53    0.002    0.000    0.932    0.018 likelihood.py:258(__chisq)
      965    0.021    0.000    0.925    0.001 likelihood.py:138(__ts_ts_to_cls)
      380    0.036    0.000    0.885    0.002 utils.py:96(writeHistoDat)
       53    0.002    0.000    0.851    0.016 likelihood.py:323(<listcomp>)
  1832513    0.762    0.000    0.762    0.000 {method 'sub' of 're.Pattern' objects}
  1259496    0.731    0.000    0.731    0.000 {method 'search' of 're.Pattern' objects}
        1    0.014    0.014    0.651    0.651 yoda_factories.py:904(sort_blocks)
6295/6293    0.042    0.000    0.631    0.000 <frozen importlib._bootstrap>:986(_find_and_load)
     4268    0.381    0.000    0.602    0.000 yoda_factories.py:958(<listcomp>)
      380    0.001    0.000    0.572    0.002 plotinfo.py:223(getHeaders)
      380    0.014    0.000    0.570    0.002 plotinfo.py:46(getSection)
     1520    0.150    0.000    0.528    0.000 plotinfo.py:128(_readHeadersFromFile)
6295/6293    0.032    0.000    0.463    0.000 <frozen importlib._bootstrap>:956(_find_and_load_unlocked)
48007/31560    0.077    0.000    0.450    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
  2512530    0.424    0.000    0.424    0.000 {method 'group' of 're.Match' objects}
  1910862    0.412    0.000    0.412    0.000 {built-in method builtins.isinstance}
     6295    0.071    0.000    0.406    0.000 <frozen importlib._bootstrap>:890(_find_spec)
     1210    0.088    0.000    0.393    0.000 build_covariance.py:51(buildCovFromErrorBar)
     6295    0.010    0.000    0.286    0.000 <frozen importlib._bootstrap_external>:1334(find_spec)
    56932    0.285    0.000    0.285    0.000 {method 'points' of 'yoda.core.Scatter2D' objects}
     6295    0.029    0.000    0.276    0.000 <frozen importlib._bootstrap_external>:1302(_get_spec)
        1    0.002    0.002    0.264    0.264 utils.py:50(getHistos)
     4072    0.015    0.000    0.236    0.000 _distn_infrastructure.py:513(argsreduce)
```

Figure 3.1: Output of cProfile run method

From figure 3.1 above we can summarise the main output from the single
YODA file **Contur** run:

- From line one of the profiling results we can see that the run had c.a. 17
  million function calls and took c.a. 20 seconds to run;

- The next line tells us that we are ordering the profiling results by cumulative
  time (`cumtime` column). The cumulative time for a function is the time spent
  to run a function and all other functions called within the function (so the
  `cumtime` for the main function will be the total run time of the program as all
  other functions are called within main);

- From line three on we have the profiling information for the functions and
  sub-functions which compose the **Contur** run. The main columns which stand
  out here are `ncalls` which gives the number of calls made to the function,

`tottime` which gives the total time spent in the function excluding calls to sub functions and finally `cumtime` which as already explained gives the run time for each function including all the calls to sub functions.;

The above profiling is already useful, it gives us things like the run time and the break down of the run time between the components of **Contur** . However the printed results in the current form are not very readable, a detailed knowledge of the functions that compose **Contur** would be needed to take any advantage of the run time broken down by components in its current form. Additionally we don't just want to print result to the terminal and work from there, we would preferable save the profiling results to some file format so our results are reusable across time.

To meet both these objectives for the profiling we from here on we will print the data from our profile into ".prof" files which can then be read by tools which help visualise the profiling results. We do this by introducing the Profile class of **cProfile** and using this to perform our profiles from here on in as opposed to using the run method, the updated code to perform the profiling with the **cProfile** class is given below.

```python
import cProfile, pstats, io


if __name__ == "__main__":
        cls_args = get_args(sys.argv[1:],'analysis')


        pr = cProfile.Profile()
        pr.enable()


        main(cl_args)


        pr.disable()
        pr.dump_stats('outfile.prof'')
```

3.3 Visualizing Profiling Results

To visualise our profiling results we will use two open source tools **Snakeviz** and **gprof2dot**. As what follows will show, we can use both of these tools in a complementary way, as opposed to a simple choose of one or the other, to help make best of use of the profiling data we produce with **cProfile**.

3.3.1 Snakeviz

Snakeviz[7] is a browser based graphical viewer for the output of Python's **cProfile** profiler module. **Snakeviz** can easily be pip installed with the following terminal command

```
$ pip install snakeviz
```

once installed we can invoke **Snakeviz** to visualise an arbitrary .prof file as follows

```
$ snakeviz profile_file.prof
```

After invoking **Snakeviz** as outlined above the web browser interface for the tool will open and the user can explore the profiling results. **Snakeviz** allows user interaction to adjust how results are rendered, the two main plotting options available in **Snakeviz** are icicle plots and sunburst plots[4].

From here on we will use **Snakeviz's** icicle plot to explore profiling results, additionally due to the constraints of the static form of this document is written in we will just examine static snapshots of the overall display in **Snakeviz's** viewer. These static snapshots of the **Snakeviz** viewer are sufficient to summarise profiling results. Using **Snakeviz's** viewers ability to adjust rendering though can be useful to get a feel and understanding for new profiling results, the interested reader is recommended to play around with **Snakeviz's** viewer functionality further.

Below in figure 3.2 we show a snapshot of an icicle plot from a profile of our initial starting **Contur** code on a single **YODA** file. From the figure we can seen that the icicle plot is showing the same information as figure 3.1 in just a more visually

---

[4]A nice overview of these plots in **Snakeviz** can be found here

appealing way, with the addition that in the icicle plot we can see the ordering of the calls to the components of code that compose a **Contur** run. This ordering is very useful additional information, for example from the ordering it jumps out at us that the call to `yoda.core` to read the **YODA** file passed to **Contur** takes a large proportion of the run time for a single **YODA** run. From this we can already understand that a lot of the run time for a single **YODA** run comes from just reading in data.
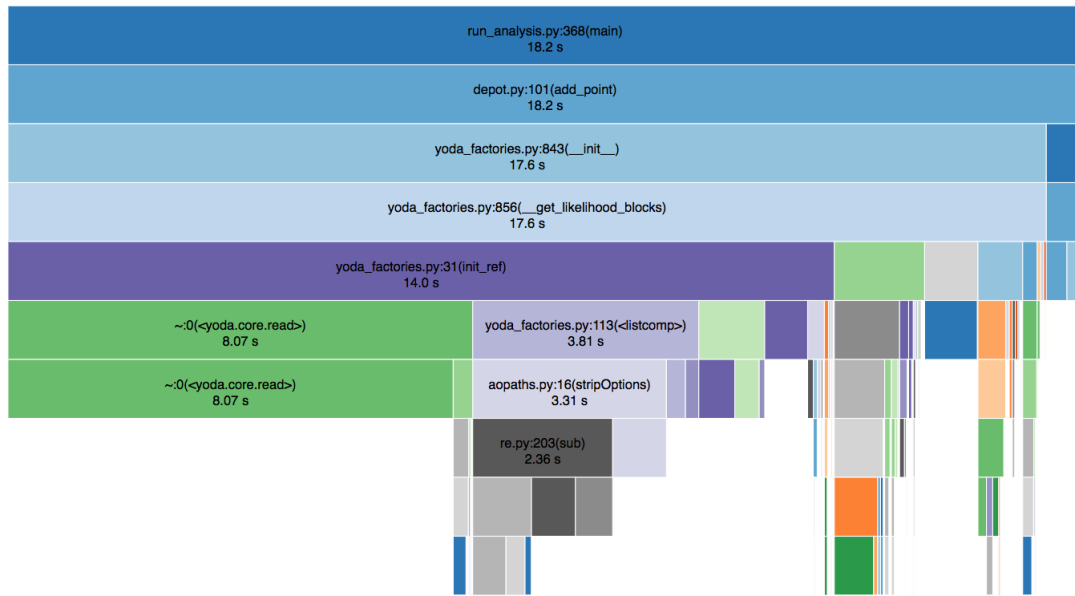
Figure 3.2: Contur single **YODA** run starting point - Example **Snakeviz** icicle plot

3.3.2 gprof2dot

gprof2dot[8] is a python script that converts the output of the **cProfile** to dot plots. These dot plots can be used to complement the information we get from the icicle plots. The icicle plots and the user interface offered by **Snakeviz** offer a means to see the absolute run time of our code and how this absolute run time breaks down among the components of the program. The dot plot complements this information by providing a rendering which makes the flow of the code (i.e. the progression of the code from the call to main through the components that compose the program) more easily visible and additionally showing the relative weight run time wise of the components of the code. This visualisation can be useful to both quickly spot

bottlenecks in the code and also just to get a better understand of how a large code base works.

Figure 3.3: Contur single **YODA** run starting point - Example **gprof2dot**

We can see example of the dot plots produced by **gprof2dot** in figure 3.3 above. This plot is visualising the same single **YODA** run as in figure 3.2, so is a good way of demonstrating the complementary nature of the icicle plot and the dot plots for visualising our profiling results. Following the colouring scheme in the dot plot (red to yellow to green) the observation we previously made using the icicle plot about the weight of data reading in the runtime can be seen in the dot plot where we can

see c.a. 42% of run time is spent reading **YODA** files.

3.4   Initial Profile Results

In the previous section while introducing the visualisation tools we gave the initial profiling results resulting from running **Contur** on a single **YODA** file (see figure 3.2 and 3.3) before any optimisation of the code was attempted.

As previously discussed, in practical settings **Contur** is generally run on a grid of **YODA** files as opposed to a single **YODA** file, so along with our initial single **YODA** run profile we will also perform an initial profile of **Contur** on a test grid. The grid we use to perform this profile is a 10×10 grid, so composed of 100 **YODA** files in total, we will use this reference grid throughout to profile **Contur's** grid run.

In figure 3.4 below we see the icicle plot for the grid run, from this we can see that for the grid of 100 **YODA** files we have a run time of around 1100 seconds or close to 20 minutes.
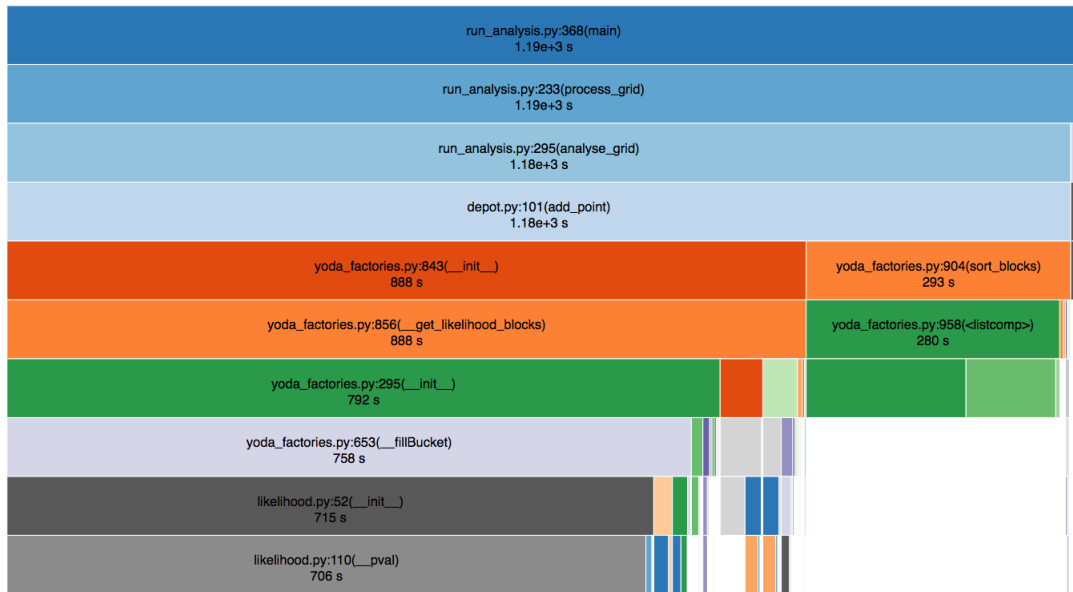
Figure 3.4: Contur grid run - icicle plot

We can also see from the plot that the main contribution to the run time seems to be coming from two blocks of the code. This is best seen in the dot plot figure 3.5 below where we can see that the `sort_blocks()` method contributes c.a. 25% of the run and the `ts_to_pval()` method which contributes c.a. 49%, so both of these

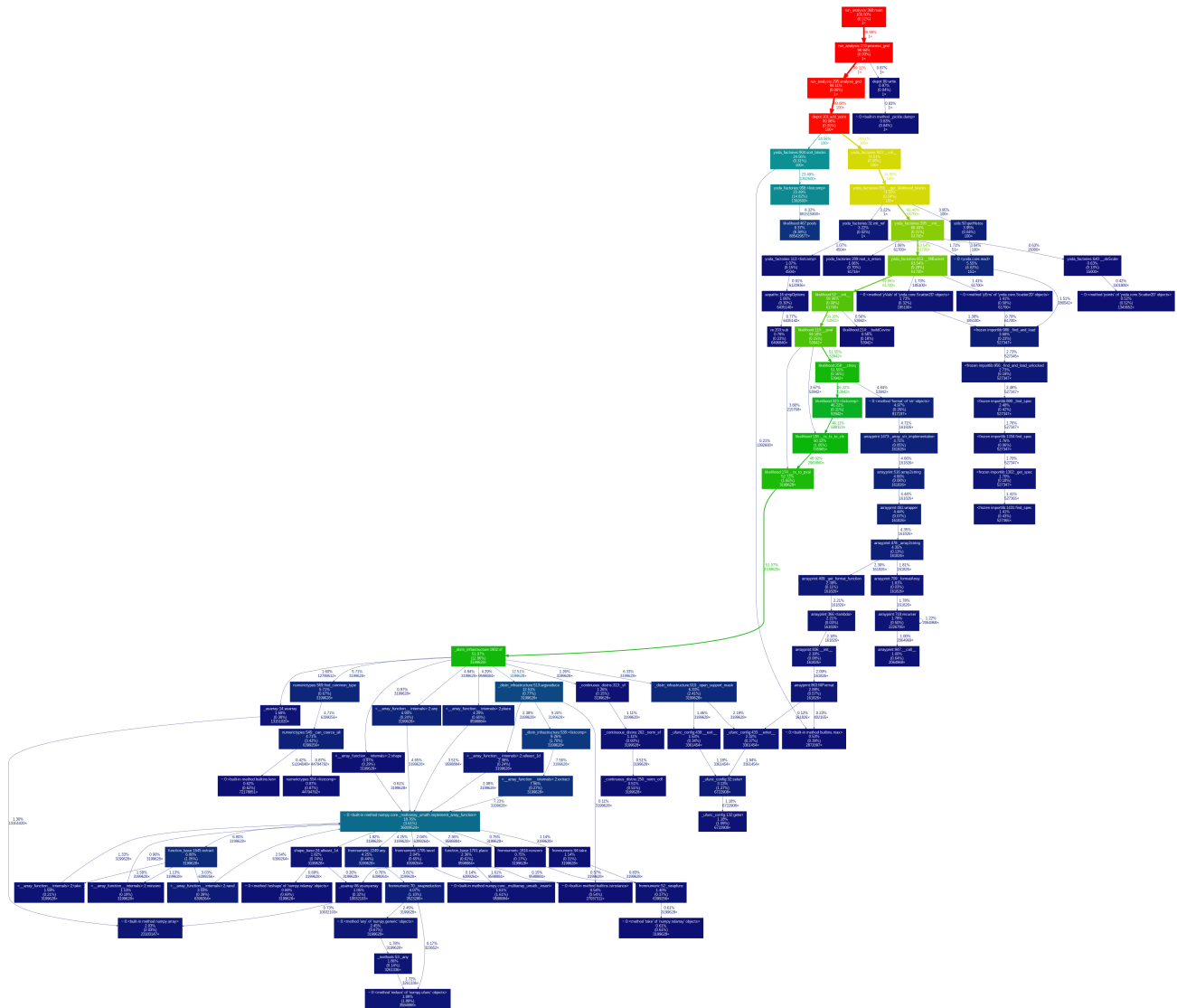methods in combination are close to three quarters of the run time for the **Contur** grid run.

Figure 3.5: Contur grid run - dot plot

Chapter 4

Testing Contur

As already highlighted in the introduction, an important concern when changing live code for optimisation purposes is that we don't unintentionally break or change the final output from **Contur**. We identified a robust testing infrastructure as a means to reduce the risk of this happening. In this chapter we slightly digress from profiling and optimisation to discuss the additions we made to **Contur's** testing infrastructure as part of this project.

4.1 Contur Existing Tests

Prior to work carried out in this thesis, **Contur** had a limited set of tests[1] implemented within python's pytest framework[9]. In the **Contur** repository these tests can be found in the tests folder. Within the tests folder there are two separate scripts to run tests, `test_batch_submit.py` and `test_executables.py`. These test scripts effectively check that the functionality within **Contur** runs without error, however the tests don't have any visibility on the output of a run (except if the run throws an error before completion).

The main one of these scripts of relevance for the additional tests we will add in subsequent sections is `test_executables.py` which checks that **Contur** runs on a single **YODA** file and on a grid without errors. To carry out these tests pytest does a single **YODA** and grid run[2]. These runs are of relevance to us because we can use their outputs to create regression tests as we will outline in the next section.

---

[1] See initial tests here

[2] The tests folder contains a single yoda file and a 4×4 grid for the grid run

4.2 Regression Testing

The simplest test we can put in place to try and mitigate the risks from changes to code is to try and ensure that the change does not alter the final output of a **Contur** run. This can be achieved by introducing regression testing into **Contur's** suite of tests. Regression tests will consist of comparing the output of the run we make with the updated code (labelled the target) against the output of a run made before the update to the code (labelled the base). The regression test is passed if our target output is equal to base output[3].

Implementing these regression tests within the pytest framework will allow us to carry out the comparison of new results against old results automatically just by running pytest. Thus the regression tests we implemented as part of this project will be of use to other **Contur** developers to help ensure their changes don't unintentional alter output. Before outlining in greater detail how went about implementing the regression tests it is useful to first give greater clarity on the file format of the results output by **Contur**.

4.2.1 Contur Run Output Format

Single **YODA** and grid runs output their results in different file formats. A single **YODA** run outputs a text file with the results printed in a text file. A example of such an output is shown in figure 4.1 below. For regression testing purposes we can simple compare that base and target text files are the same excluding the first three lines of the text file which give the location where **Contur** is running to produce the text file[4]

---

[3]The target output in this case can be said to regress to base, hence the name regression testing.

[4]This can be seen 4.1, if we included these first three lines in our comparison then the single **YODA** file regression test would always fail whenever **Contur** is run from a different location which is not something we want to happen.

```
Run Information
Contur is running in /Users/jonbutterworth/gitstuff/contur-dev/tests
on analysis objects in ['sources/testPoint.yoda']
Using search analyses
Excluding Higgs to WW measurements
Excluding secret b-veto measurements
Excluding ATLAS WZ SM measurement
Building all available data correlations, combining bins where possible
Building default background model from data, ignoring (optional) SM theory predictions

Sampled at:
CZdL1x1: 1.062202380952381
CZdL3x3: -1.062202380952381
CZuL1x1: 1.062202380952381
CZuL3x3: -1.062202380952381
CZuR1x1: 1.062202380952381
CZuR3x3: -1.062202380952381
cotH: 4.5
mZp: 3578.9473684210525
Combined exclusion for these plots is 100.00 %

pools
ATLAS_13_METJET
0.39844652
/ATLAS_2016_I1458270/d05-x01-y01
ATLAS_13_EEJET
0.03740851
/ATLAS_2019_I1718132/d59-x01-y01
ATLAS_13_MMJET
0.08701654
```

Figure 4.1: Example output from single yoda run - txt file

The grid run returns a pickled[5] ".map" file. The ".map" file contains the **Contur** `Depot` object from the run. The `Depot` object contains all information about a run in its attributes, using the it's `make_df` attribute we can create a **Pandas**[6] `DataFrame` like what we see in figure 4.2 below. For the regression test for the grid we can compare a base and target `DataFrame`.

---

[5]For further information about pickling objects in Python see here

[6]**Pandas** is an open source data analysis Python package, further information can be found here

	CZdL1x1	mZp	CL
0	0.35406746031746034	1000.0	1.000000
1	0.35406746031746034	1444.4444444444443	1.000000
2	0.35406746031746034	1888.888888888889	1.000000
3	0.35406746031746034	2333.3333333333335	1.000000
4	0.35406746031746034	2777.777777777778	1.000000
...	...	...	...
95	1.062202380952381	3222.222222222222	1.000000
96	1.062202380952381	3666.666666666667	0.999996
97	1.062202380952381	4111.111111111111	0.999404
98	1.062202380952381	4555.555555555556	0.984761
99	1.062202380952381	5000.0	0.861149

100 rows × 3 columns

Figure 4.2: Example output from grid run - dataframe

4.2.2 Implementing Regression Tests

Adding the regression tests to **Contur's** testing infrastructure is straight forward enough. For the single **YODA** regression we added the function given in listing 4 to the test_executables.py script. This function first reads in the base text file and then reads in the target text file created from the run. Then after removing the first three lines from each text file the function checks that the text files are the same.

Listing 4: Single YODA Regression Test

```python
def test_regression_single_yoda_run():
    args_path = os.path.join(test_dir, 'base.txt')
    with open(args_path) as sf:
        base = sf.read().splitlines(True)


    args_path = os.path.join(test_dir, 'target.txt')
    with open(args_path) as sf:
        target = sf.read().splitlines(True)
    assert base[3:] == target[3:]
```

The regression test for the grid run is similar to the single **Contur** regression. We can see the function for the grid regression in listing 5. Instead of text files this function pickles two map files and then calls the `Depot._build_frame()` attribute to create a base and target `DataFrame`. We then use **Pandas's** `assert_frame_equal()`[7] function to check if the two `DataFrame` are equal. Using the `assert_frame_equal()` default settings, this test will pass if the numerical differences between any two corresponding cells in the two `DataFrame` is less than 10^{-8}.

Listing 5: Grid Regression

```python
def test_regression_grid_run():
    args_path = os.path.join(test_dir, 'base.map')
    with open(args_path, 'rb') as file:
        base = pkl.load(file)._build_frame()
    args_path = os.path.join(test_dir, 'target.map')
    with open(args_path, 'rb') as file:
        target = pkl.load(file)._build_frame()
    assert_frame_equal(base, target)
```

4.2.3 Including Theory Runs

In addition to including regression test for the default **Contur** runs we also added regression tests for the theory runs too. Recall when the theory option is specified **Contur** includes SM expectations so the theory option will potentially give different output than the default run. The theory regression tests were added using similar functions to listing 4 and 5.

---

[7]See here for **Pandas** documentation for this function

Chapter 5

Optimising Contur

In this section we will outline the optimisation changes made to **Contur** as part of this research project. Our focus will be on optimising the grid run, mainly because for research purposes users will in general be spending most of their time running **Contur** on a grid as opposed to single **YODA** files. So focusing our optimisation efforts on the grid run is likely to produce more practical benefits for users. In addition we have two other motivations for focusing our efforts on optimising the grid run:

- There is more scope for achieving meaningful improvements in runtime with the grid run. This viewpoint comes from observing from figure 3.2 that the single **YODA** runtime only takes around 20 seconds, while from figure 3.4 we can see that the runtime for a smallish grid[1] takes up to 20 minutes. Thus decreasing runtime for the single **YODA** by 50% will only save us 10 seconds in absolute terms, while the equivalent decrease for the grid run would save us 10 minutes[2];

- There is more scope for the grid runtime to increase with changing research needs. The grid runtime is dependent on the size of the grid, as the grid grows in size the runtime will increase. There is a practical limit to how big

---

[1]The grid we are profiling is 10×10, so contains 100 **YODA** files, for research purposes it is common to run such a 10×10 grid across three different energy buckets (7,10 and 13 TeV), with each bucket having 100 **YODA** files for a total of 300. So the 20 minutes we profiled for a single 10×10 grid would likely be close to an hour if run across the three energy buckets

[2]Or 30 minutes for the case where the grid has three energy buckets.

a grid can be resulting from this increasing runtime, in effect once a grid is so large it is too slow to run on. Optimisations to the grid run that not only improve runtime on the current standard size grids but also reduce the speed that runtime increases with increasing grid size could have very practical benefits like making runs feasible on large grids where previously the run time was too slow;

For the above reasons the main focus of the optimisation from here on in will be on the grid run. From the dot plot in figure 3.5 arising from the data produced from our initial grid profile we can see that grid runtime arises from two branches in the code flow, the first arising from the `sort_blocks()` method[3], which takes c.a. 25% of runtime and the second being the calculation of the CL_s exclusions for each histogram[4] which takes most of the remainder runtime. Our initial efforts will focus on making optimisation improvements for these two parts for the program.

5.1 Sort Blocks Method

5.1.1 Background

In section 2.3 we gave a brief overview into how **Contur** calculates the CL_s exclusion for a single passed **YODA** file. Additionally we also highlighted the similarities between the single **YODA** and the grid runs. We will now go into a bit greater detail to give the necessary background to understand the optimisation change covered in this section.

Contur contains three main classes to coordinate a run, the `depot` class, `yoda_factories` and the `likelihood` class. The `depot` class deals with the overall control of a run and is where all results are stored, while the `likelihood` class calculates the CL_s exclusion for each histogram. The `yoda_factories` class sits between these two classes in the flow of a run. Each **YODA** file that the `depot` object finds in a grid gets passed to `yoda_factories`. Upon instantiation of a `yoda_factories` object with a **YODA** file, `yoda_factories` gathers all necessary

---

[3]See dark green box in dot plot

[4]sequence of boxes in the dot plot starting as yellow and morphing to green

data[5], and then for each histogram in the passed **YODA** file, yoda_factories instantiates a likelihood object to calculate the CL_s exclusion for the histogram. The end result of instantiating yoda_factories is that its likelihood_blocks attribute will contain a list of the likelihood objects instantiated for each histogram. Each of these likelihood objects will contain the CL_s for a histogram as an attribute.

The sort_blocks() method of yoda_factories is called after an object has been instantiated and the likelihood_blocks attribute has been populated. The method takes the list of Likelihood objects in the likelihood_blocks list and buckets the Likelihood objects into pools. After this bucketing for each pool the method then finds the Likelihood object in each pool with the largest CL_s, the selected Likelihood objects are used to create a new list of Likelihood objects which are stored in yoda_factories sorted_blocks attribute.

5.1.2 Changes made

The profiling results in the dot plot in figure 3.2 are precise enough that we can pin point the major slow point in the sort_blocks() method to be the list comprehension in line 958 of yodafactories[6]. This list comprehension sits within a loop and the whole block of code can be seen in listing 6 below.

Listing 6: Sort Blocks List Comprehension

```
for p in pools:
  if not p == omitted_pools:
    for item in l_blocks:
      if item.CLs ==
        max([x.CLs for x in l_blocks if x.pools == p])
```

In the above block of code the list comprehension in the last line is used to find the Likelihood object in each pool with the largest CL_s. The list comprehension on its own is not particularly slow, it is however placed within a nested for loop, which loops over pools and then within each pool iterate we have another loop

---

[5]This consists of both the simulated data contained in the passed **YODA** file and experimental data from **Rivet**

[6]This can also be easily picked up in the icicle plot when using Snakeviz's graphical viewer

over the `likelihood_blocks` list. So in each **YODA** file the number of times listing 6 is called is given by the number of pools multiplied by the length of the `likelihood_blocks` list. While over a whole grid of 100 **YODA** files we will effectively be multiplying this number again by 100. Figure 5.1 below shows that for our 10×10 grid we are calling listing 6 over a million times. So although each comprehension on its own takes less than 0.002 seconds this multiplied by a million still gives us a run time of over 200 seconds.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1392600	176.5	0.0001268	280.1	0.0002012	yoda_factories.py:958(<listcomp>)

Figure 5.1: List comprehension in sort blocks - Run time info

The optimisation change adopted here was to move the maximum CL_s calculation for each pool performed by listing 6 outside of the nested for loop. This change dramatically reduces the number of times the calculation is performed. In its place we now only perform the calculation once per **YODA** file[7] and store the results in a dictionary whose key is the pool and value is the maximum CL_s for the pool. This dictionary is then used in place of the list comprehension in listing 6. So within the nested for loop we have replaced a call to a list comprehension with a run time per call of 10^{-4} with a dictionary with a runtime per call[8] of 10^{-7}, so the 10^6 calls made in the loop will take 10^2 seconds with our old implementation but only 10^{-1} seconds with the new implementation.

5.1.3 Impact of changes

Below we see the impact of the optimisation on the runtime for the `sort_blocks()` method. In figure 5.2 we can see that prior to optimisation the runtime of the list comprehension is in line with our expectations from the previous section with a runtime of 280 seconds, order 10^2 as expected and total runtime[9] for `sort_blocks()` is 293 seconds. While in figure 5.3 we see that we have a post optimisation runtime

---

[7]So in our 10×10 grid we go from perform the calculation over 1 million times to exactly 100 times, once for each **YODA** file in the grid
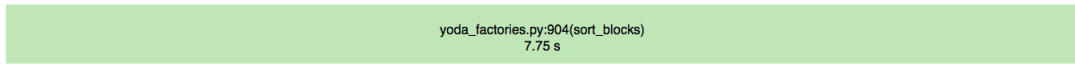
[8]See here for profile information for Python's dictionary object

[9]Remember the list comprehension is only part of the sort blocks method, not the whole of it so we will have additional runtime from other parts of the method

of 7 seconds for the whole of the `sort_blocks()` method. This runtime supports the hypothesis that the runtime for calling the dictionary in place of the list comprehension is of order 10^{-1} seconds and additionally would suggest that the optimisation change made the calculation of the maximum CL_s for each pool slightly faster too, as prior to optimisation `sort_blocks()` had 13 seconds of runtime outside of the list comprehension, now after optimisation total run time is just 7 seconds.

yoda_factories.py:904(sort_blocks)
293 s

yoda_factories.py:958(<distcomp>)
280 s
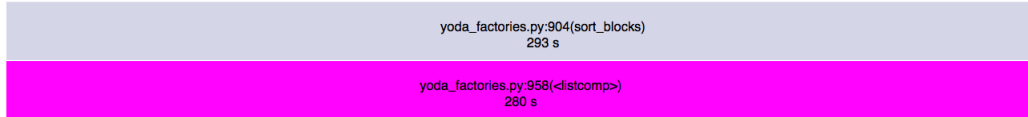
Figure 5.2: Sort blocks grid run time - Before optimisation

yoda_factories.py:904(sort_blocks)
7.75 s

Figure 5.3: Sort blocks grid run time - After optimisation

5.2 Likelihood Calculation

5.2.1 Background

The outlines of the workings of **Contur** to this point[10] have avoided delving too deeply into how the `Likelihood` object computes the CL_s, instead it has just been sufficient to highlight that for each histogram the calculation of the CL_s takes place within the `Likelihood` object. In this section we will need to delve a bit more deeply into the likelihood calculation. The reason for this comes from examining the icicle plot in figure 3.4. From this plot we can that a material proportion of **Contur's** runtime is spent within the `Likelihood` object[11].

Drilling down into the initial grid run profile results for the `Likelihood` object we can see in figure 5.4 that the `ts_to_pval()` method is where most of the run time is coming from in the `Likelihood` object. This method takes the test statistics which are computed by the `Likelihood` object's `chisqr()` method and converts them into

---

[10]See chapter 2 Contur overview and the outline for the `sort_blocks` method in section5.1

[11]From the icicle plot we can see that out of a run time of around 1100 seconds we spend 715 seconds in the likelihood class calculations

p-values. The test statistics are taken to have the distribution of a standard normal, so the p-value is computed just by computing the survival probability for the value of the passed test statistic. This is done using the the `scipy.stats.norm` object in the Scipy library and using the `sf()` method of this object to compute the survival probability for the passed test statistic. The only function that the `ts_to_pval()` method in the `Likelihood` object performs is to call the `sf()` method, so from this we can see that the majority of our run time in the `Likelihood` object is coming from calling this one method which from here on in we will refer to as the survival function.
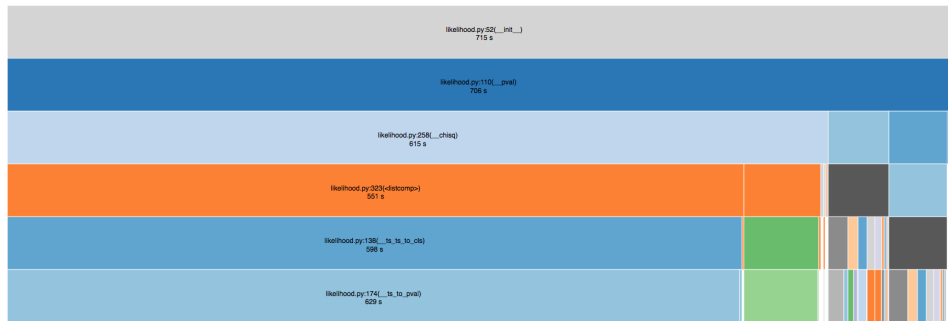
Figure 5.4: Likelihood object - Initial profile

From figure 5.5 below it becomes apparent that the large run time we observe from calling the survival function results from the large number of calls we make to the function. Each call to the survival function takes c.a. 0.0002 seconds, but we make over 3 million calls resulting in a total run of over 600 seconds.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
3199628	19.69	6.153e-06	628.7	0.0001965	likelihood.py:174(__ts_to_pval)

Figure 5.5: Likelihood object - ts to pval details

5.2.2 Scipy Normal Distribution Survival Method

From the previous section we can conclude that greater efficiencies in the likelihood calculation can be achieved by reducing the number of times we call the survival function. It is helpful to first understand better where the calls to the survival function arise when a `Likelihood` object is instantiated. Within a `Likelihood` object we get calls to the survival function via the following routes:

1. Every time the `ts_to_pval()` method is called we have one call to survival function. Upon instantiation this method is called twice explicitly, giving us two calls to survival function ;

2. The `ts_ts_to_cls()` method calls `ts_to_pval()` twice internally, so every time this method is called we have two calls to the survival function. The method is called once upon instantiation giving us two more calls to the survival function.;

3. The `chisq()` method which computes the test statistics will only call the survival function when an inverse for the covariance matrix of the analysis object cannot be computed[12]. When the method calls the survival function the number of times it makes the calls is twice the number of buckets in the analysis object. So this is a minimum of 2 calls but potentially much more than 2 ;

From the above we see that each histogram will have at least 4 calls to the survival function[13], so across a whole **YODA** file the number of calls to the survival function will be at least 4 multiplied by the number of histograms in the **YODA** file[14]. Finally we have 100 **YODA** files in our grid, which need to be summed across to give the total number of calls we make to the survival function in our grid run.

To reduce the number of calls to the survival function we will adopt two approaches. The first is simple to check if we are making any unnecessary calls to the survival function anywhere that can easily be removed, this approach is simple but will not likely give high returns. The second approach is to take greater advantage of numpy's array functionality to see if via collecting test statistics into numpy arrays and passing these arrays of test statistics to the survival function, reducing the overall number of calls[15].

---

[12]In the code the survival function will only be called when the condition "self.covBuilt and sb nuisance is not None" is false

[13]In practise it will be a lot more than this because of the `chisq()` calls

[14]For perspective here, the 10×10 grid we profiled on had c.a. 60,000 likelihood objects created across 100 **YODA** files suggesting on average each **YODA** file had 600 valid histograms

[15]So for example if we had four test statistics we wanted to pass to the survival function we would collect the test statistics into a single array and pass the array once to the survival function as opposed to four individual calls to the survival function

For the second approach to be effective we require the runtime for the survival function if passed a numpy array of length *n* to be significantly less than the runtime if we just made *n* separate calls to the survival function. We would expect this to be the case as the survival function like all methods in Scipy is built to enable fast array based computation when passed numpy arrays.

Figure 5.6 below shows the result of the profile we performed to test the performance of calling the survival function *n* times within a loop or passing an array of length *n* once. The *x* axis gives the value of *n* on a log scale while the *y* axis gives the runtime for the loop (orange line) and the array (blue line) on a log scale. So from the plot we can see that our starting profile with *n* between 10^6-10^7 should give a runtime between 10^2-10^3 seconds which is in line with the c.a. 600 seconds we actually observe.
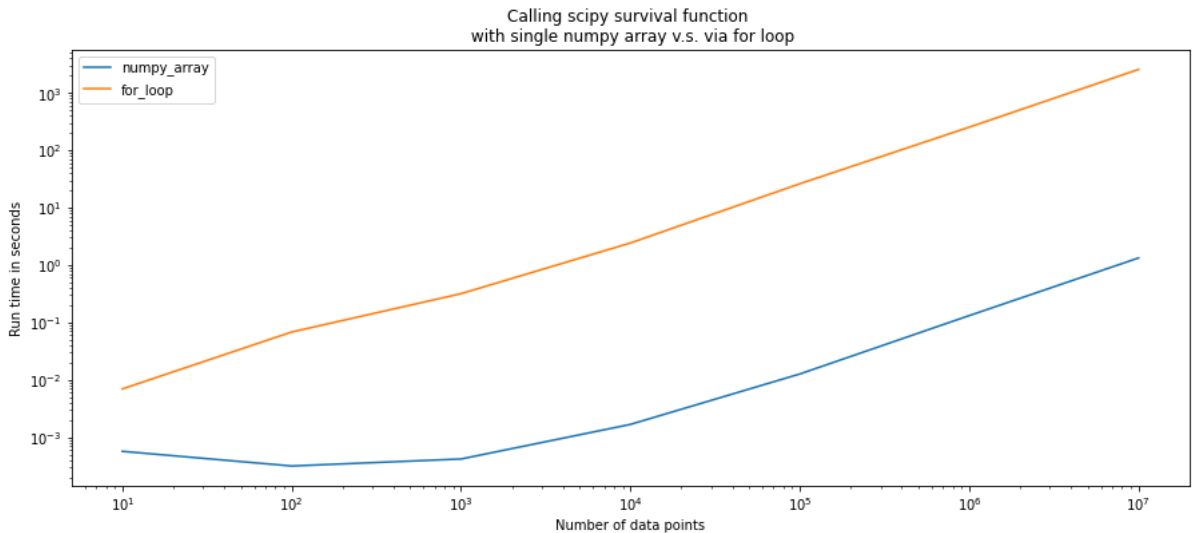
Figure 5.6: Scipy Survival Function - Loop vs Array

5.2.3 Changes made

The optimisation changes to the likelihood calculation were made via multiple commits to the **Contur** repository over the space of a couple weeks, they can be grouped into three groups of changes:

1. Within `Likelihood` objects making use of numpy arrays to reduce the number of calls to the survival function;

2. Within `Likelihood` objects reducing unnecessary calls to the survival function;

3. Across `Likelihood` objects, use numpy arrays to reduce the number of calls. So try to gather test statistics for multiple `Likelihood` objects across the **YODA** file together and make a single call to the survival function.

Of the above changes the first two are least disruptive in terms of their impact in the overall flow of a run as they only make changes within the `Likelihood` class. While the third change is more substantial structural alternation to flow of a run between `yoda_factories` and the `Likelihood` class.

The first change[16] involves passing a list of tuples of the background and signal test statistics to the `ts_to_pval()` method and to the `ts_ts_to_cls()` method, as opposed to passing the test statistics as separate calls. This reduces the calls to the survival function for these values from 4 to 2. Listing 7 below gives an example of this change, we can see when we pass the list containing the tuple we only need to call `ts_to_pval()` once as opposed to twice, which means we see a similar reduction in the number of times we call the survival function.

Listing 7: Sort Blocks List Comprehension

```
## Passing individual test statistics (before)
p_val_background = ts_to_pval(ts_background)
p_val_signal = ts_to_pval(ts_signal)


##Passing list of tuples (after)
list_ts = [(ts_background,ts_signal)]
p_val_background , p_val_signal = ts_to_pval(list_ts))
```

This change also has impact within the `chisq()` method, where we also now pass a list of tuples containing the test statistics as opposed to making separate calls to the survival function. This ensures that whenever the `chisq()` method makes a call to the survival function it only makes 1 call, so after all these changes, for a

---

[16]The change can be found in commit 0b807895

histogram we get at least 2 calls to the survival function and at most 3.

The second change[17] made is motivated from the observation that the `ts_ts_to_cls()` method has a call to `ts_to_pval()` within it, so it computes the p-value within the call. This is unnecessary as we have already computed the p-value. The change introduces the `pval_to_cls()` method which directly takes a p-value and computes the CL_s from it. Negating the need to make another call to the survival function, so after this change we now have a minimum of 1 call to the survival function and a maximum of 2 in each histogram.

From here on in let us simplify and assume that we have exactly 2 calls to the survival function for each histogram and for each **YODA** file we have m histograms. So in a grid with n **YODA** files after the above changes we have $2 \times n \times m$ calls to the survival function. So for our 100 **YODA** file grid assuming $m = 600$ this will still give us 120,000 calls to the survival function. Additionally we can see with an expanding grid size this number of calls will grow, for example with 1,000 **YODA** files the number of calls to the survival function will be above 1 million again. So the current configuration is not robust against future increases in grid size. The final set of optimisation changes we will make resolves this issue by getting rid of the runtime dependence on the number of histograms m per **YODA** file.

To achieve this, let us first remind ourselves how the current set up works. Upon instantiation of the `yoda_factories` class we loop through all histograms instantiating a `likelihood` object for each and doing the full likelihood calculation (i.e. computing the CL_s) upon instantiation of the `likelihood` class. So at the end of this process we have a `yoda_factories` object with a `likelihood_blocks` attribute that contains all the `likelihood` objects. This process can be split into steps that allows us to eliminate dependence on the number of histograms. This can be done by instead of using the survival function within the `likelihood` object, we can instead just collect test statistics in the `Likelihood` object which can then be collected into a numpy array in `yoda_factories` composed of test statistics from all the `likelihood` objects, this array can then be passed once to the survival function.

---

[17]The change can be found in commit 15ef923d

Implementing this change in practise is spread across two commits[18]. For the first commit we introduce the `likelihood_blocks_ts_to_cls()` function, which takes a list of `likelihood` objects and computes a CL_s for each `likelihood` object in the list. In terms of the flow of the run, with this change we alter the `likelihood` object so it no longer calculates the CL_s upon instantiation, so when we instantiate `yoda_factories` we now have a list of `likelihood` objects in the `llikelihood_blocks` attribute where each `likelihood` object just contains a test statistic, we pass this list to the new function which does the confidence level calculation. After this change we will have $n + (n \times m)$ calls to the survival function.

For the second commit we introduce the `likelihood_blocks_find_dominant_ts_function`. This function finds the test statistic in a histogram that gives the largest CL_s when the covariance matrix does not have a valid inverse. Thus it moves the calls to the survival function that take place within the `chisq()` method into a single call in `yoda_factories`. After this change the number of calls to the survival function will be $2n$ so we have removed the dependence on m of the number of calls. The impact of this third change in terms of the overall structure of **Contur** can be seen in listing 8 below.

Listing 8: Impact Of New Functions

---

[18]The first commit 299b03a8 introduces the `likelihood_blocks_ts_to_cls()` function, while the second commit 2769e1c2 introduces the `likelihood_blocks_find_dominant_ts_function()`

```
## Before: Just instantiate yoda_factories object
## and straight away call sort blocks
yFact = yoda_factories(yoda_file)
yFact.sort_blocks()


##After: Now after instantiating the
## yoda object we need to call the new
## functions to compute CLs
yFact = yoda_factories(yoda_file)
likelihood_blocks_find_dominant_ts(yFact.likelihood_blocks)
likelihood_blocks_ts_to_cls(yFact.likelihood_blocks)
yFact.sort_blocks()
```

5.2.4 Impact of changes

The impact of the optimisation on the run time for the `likelihood` class and the associated new functions can be seen in figure 5.7 below. From the figure we can see that impact of the optimisation on the run time for the CL_s calculation has been substantial, from a starting run time of c.a. 600 seconds the optimisation has reduced the total run time to just below 61 seconds. In the next section we will see that much of the remainder of this runtime for the likelihood calculation can be further reduced.
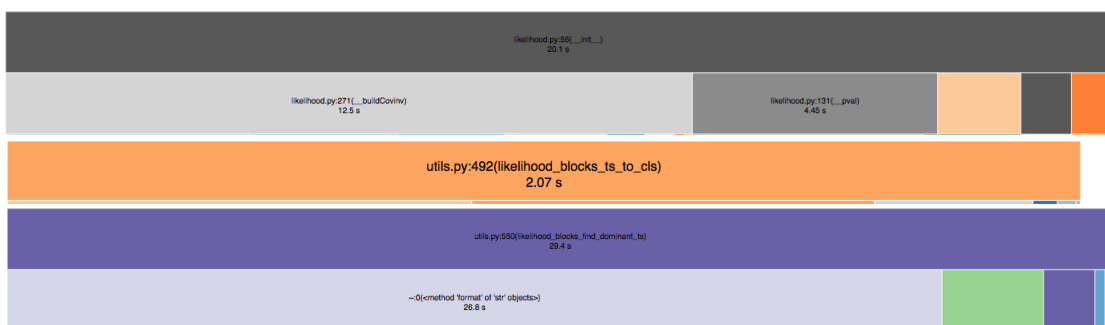
Figure 5.7: Likelihood object and new functions - Profile after optimisation

5.3 Other Changes

The `sort_blocks()` and `likelihood` class optimisations are the two main changes carried out as part of this thesis and the most effective in terms of their reduction of runtime. This section covers two more minor optimisation changes made.

5.3.1 Printing Debug Statements

A large part of the remaining run time in the `Likelihood` object that we see in figure 5.7 comes from the following line of code

Listing 9: Logging Debug Statements

```
contur.config.contur_log.debug(
  "n data, {} signal, {} background {}, bin {} ".format(
    like_ob._n, like_ob._s,  like_ob._bg, like_ob._index))
```

In the above each of likelihood objects are numpy arrays and reformatting a large number of these to be printed as strings is time consuming. Further in general the user does not require Contur to print the debug statements in the logger, however in the current set up the reformatting happens regardless whether or not the user has specified to print the strings. The quick fix to this is to place listing 9 within a if statement that only runs if the user specifies to print the debug statements in the logger, this change has been made in listing 10 below.

Listing 10: Logging Debug Statements With If Statement

```
if contur.config.contur_log.isEnabledFor(logging.DEBUG):
    contur.config.contur_log.debug(
    "n data, {} signal, {} background {}, bin {} ".format(
    like_ob._n, like_ob._s, like_ob._bg, like_ob._index))
```

The impact of this change can be seen in figure 5.8 below. We can see that the change removes much of the remaining runtime in the `Likelihood` object and the `likelihood_blocks_find_dominant_ts_function()` function.
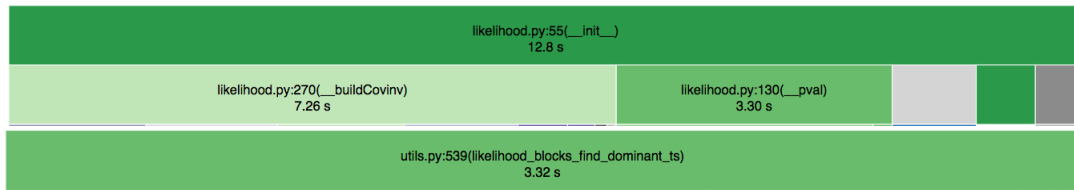
Figure 5.8: Likelihood object and new functions - Profile after removing debug print

5.3.2 Strip Options List Concatenation

In figure 5.9 below we can see a profile for the `init_ref()` function, this function deals with the data gathering phase of the Contur run. From the green box in the profile we can see that c.a. 15 seconds of runtime comes from a list comprehension.
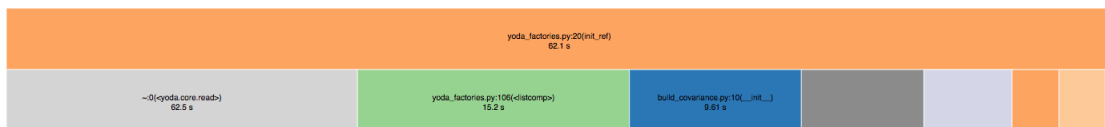
Figure 5.9: Likelihood object and new functions - Profile after optimisation

The specific list comprehension that contributes this 15 seconds is given in listing 11 below.

Listing 11: Strip Options List Concatenation

```
[match.group() in x for x in
[rivet.stripOptions(x) for x in aopath]]
```

Like other examples we have seen with list comprehensions the extra runtime we get here arises because the list comprehension is placed within another for loop, so the comprehension runs multiple times. In the above it is the inner comprehension that takes up runtime, a simple change we can make is to move this comprehension outside of the for loop and pass the list as a variable in the above line of code in the for loop.

In figure 5.10 below we see the results of this change, namely we get a close to 15 second reduction in run time as we would expect from removing the list comprehension.

Figure 5.10: Likelihood object and new functions - Profile after optimisation

Chapter 6

General Conclusions

In this chapter we will conclude this thesis by first summarising the impact of the optimisation changes made to **Contur** as part of this research project. We will then finish up by highlighting the remaining parts of **Contur** that could potentially be further optimised in the future.

6.1 Results Summary

In figure 6.1 below we can see the final profile of the grid run, comparing to figure 3.4 which shows the initial profile of the grid run before any optimisation, we can see the full impact of the changes made as part of this research project. From a starting runtime of close to 20 minutes before optimisation, the changes made as part of this project have taken the runtime below 4 minutes. The optimisation of the likelihood calculation outlined in section 5.2 is the largest contributor to this runtime reduction. The change to the `sort_blocks()` method outlined in section 5.1 was also a material contributor to the runtime reduction.
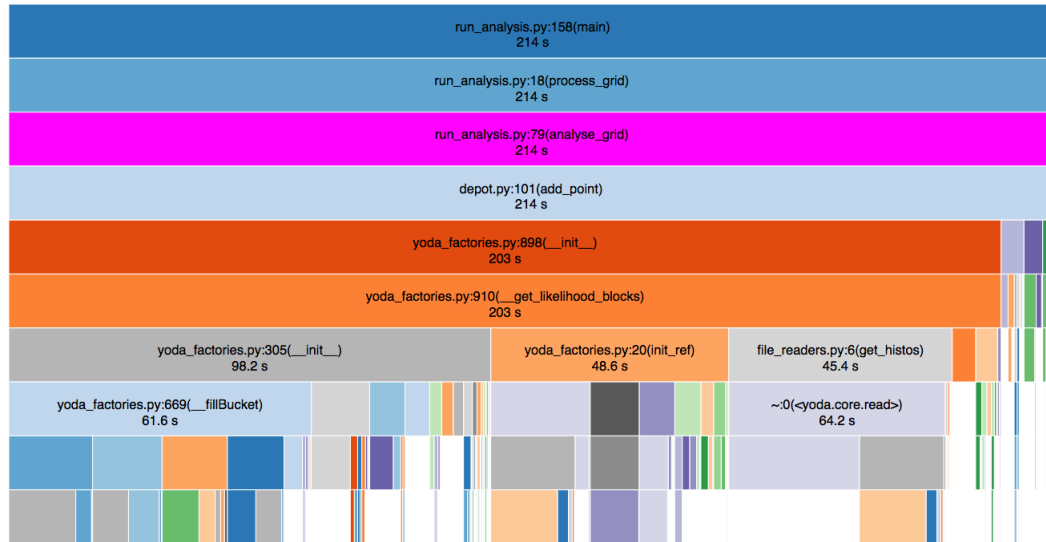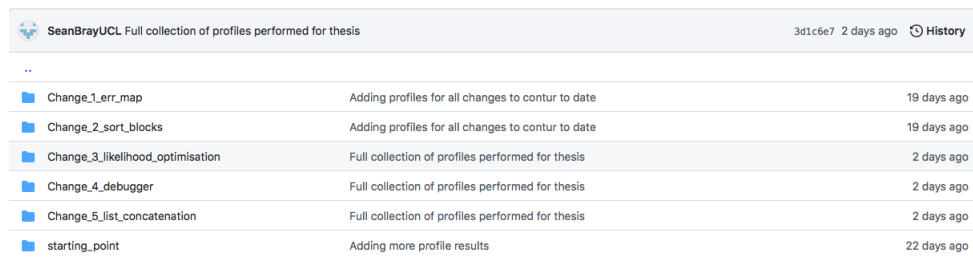
Figure 6.1: Final Grid Profile

6.2 Future Work

Appendix A

Viewing Profiling Results

All profiling results shown in this thesis are saved in the Git repository located here. The repository contain a Read.Me file which should make the navigation of the profiling results clear.

To briefly summarise, the `profiles` folder in the repository contains all the profiles, its contents are shown in figure A.1 below. The `starting_point` folder is the initial profile before optimisation, it contains a "prof" file which can be read by **Snakeviz** to create the browser interface discuss in section 3, additionally the folder also contain a png file which is the dot plot produced **gprof2dot** and discussed further in section 3.

Figure A.1: Profiles Folder

Each of the subsequent numbered folders are structured the same as the `starting_point` folder. They each contain an updated profile of **Contur** after an optimisation change has been made. The optimisation changes accumulate across the folders, so the profile in folder `Change_5_list_concatenation` contains all the optimisation changes, so can be taken as the complete post optimisation **Contur** profile.

Appendix B

Viewing Code Written For Project

All code written for this project, with one exception, can be found in the main **Contur** repository [6]. Links to specific commits made by this author can be found throughout this thesis, the approach taken has been to include a footnote with a link to the specific commits when discussing code changes in the thesis. A quick overview of code contributions to the main repository by this author can also be seen via the contributors page of the main **Contur** repository [6].

Bibliography

[1] UFO - The Universal FeynRules Output,
   `https://arxiv.org/abs/1108.2040`

[2] Herwig - Event Generator,
   `https://herwig.hepforge.org`

[3] Yoda,
   `https://yoda.hepforge.org`

[4] Contur User Manuel,
   `https://arxiv.org/abs/2102.04377`

[5] cProfile Documentation,
   `https://docs.python.org/3/library/profile.html`

[6] Main repository with Contur code,
   `https://gitlab.com/hepcedar/contur`

[7] Snakeviz Package,
   `https://jiffyclub.github.io/snakeviz/`

[8] gprof2dot Package,
   `https://github.com/jrfonseca/gprof2dot`

[9] Python pytest Documentation,
   `https://docs.pytest.org/en/6.2.x/`

[10] HEPData Repository,
   `https://arxiv.org/abs/1704.05473`

[11] Rivet Library,

https://rivet.hepforge.org

[12] CLS Technique,

https://doi.org/10.1088/0954-3899/28/10/313