

A Thesis Title

Author Name

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Something
University College London

August 23, 2021

I, Author Name, 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.

Impact Statement

UCL theses now have to include an impact statement. (*I think for REF reasons?*)

The following text is the description from the guide linked from the formatting and submission website of what that involves. (Link to the guide: <http://www.grad.ucl.ac.uk/essinfo/docs/Impact-Statement-Guidance-Notes-for-Research-Students-and-Supervisors.pdf>)

The statement should describe, in no more than 500 words, how the expertise, knowledge, analysis, discovery or insight presented in your thesis could be put to a beneficial use. Consider benefits both inside and outside academia and the ways in which these benefits could be brought about.

The benefits inside academia could be to the discipline and future scholarship, research methods or methodology, the curriculum; they might be within your research area and potentially within other research areas.

The benefits outside academia could occur to commercial activity, social enterprise, professional practice, clinical use, public health, public policy design, public service delivery, laws, public discourse, culture, the quality of the environment or quality of life.

The impact could occur locally, regionally, nationally or internationally, to individuals, communities or organisations and could be immediate or occur incrementally, in the context of a broader field of research, over many years, decades or longer.

Impact could be brought about through disseminating outputs (either in scholarly journals or elsewhere such as specialist or mainstream media),

education, public engagement, translational research, commercial and social enterprise activity, engaging with public policy makers and public service delivery practitioners, influencing ministers, collaborating with academics and non-academics etc.

Further information including a searchable list of hundreds of examples of UCL impact outside of academia please see <https://www.ucl.ac.uk/impact/>. For thousands more examples, please see <http://results.ref.ac.uk/Results/SelectUoa>.

Acknowledgements

Acknowledge all the things!

Contents

1	Introduction	11
2	Contur Overview	13
2.1	Contur and the Standard Model	13
2.2	Input Data	14
2.2.1	BSM Data	14
2.2.2	Experimental Data	16
2.3	Calculating Likelihoods	16
2.4	Contur Results	18
3	Profiling Contur	19
3.1	Why Profile and Optimise?	19
3.2	Profiling with cProfile	20
3.2.1	Why cProfile?	20
3.2.2	Using cProfile	21
3.3	Visualizing Profiling Results	24
3.3.1	Snakeviz	24
3.3.2	gprof2dot	25
3.4	Initial Profile Results	27
4	Testing Contur	29
4.1	Contur Existing Tests	29
4.2	Regression Testing	30
4.2.1	Contur Run Output Format	30

<i>Contents</i>	8
4.2.2 Implementing Regression Tests	31
4.2.3 Including Theory Runs	31
4.3 Unit Testing	31
4.3.1 Likelihood Class	31
4.3.2 YodaFactories Class	31
4.3.3 Functions	31
5 Optimising Contur	32
5.1 Single Yoda Run	32
5.2 Grid Run	32
5.2.1 Sort Blocks	33
5.2.2 Likelihood Calculation	36
6 General Conclusions	44
Appendices	45
A An Appendix About Stuff	45
B Another Appendix About Things	46
C Colophon	47

List of Figures

3.1	Output of cProfile run method	22
3.2	Contur single yoda run starting point - Example snakeviz icicle plot	25
3.3	Contur single yoda run starting point - Example gprof2dot	26
3.4	Contur grid run - icicle plot	27
3.5	Contur grid run - dot plot	28
4.1	Example output from single yoda file contur run - txt file	31
5.1	List comprehension in sort blocks - Run time info	35
5.2	Sort blocks grid run time - Before optimisation	36
5.3	Sort blocks grid run time - After optimisation	36
5.4	Likelihood object - Initial profile	37
5.5	Likelihood object - ts to pval details	38
5.6	Scipy Survival Function - Loop vs Array	40
5.7	Likelihood object and new functions - Profile after optimisation . .	43

List of Tables

Chapter 1

Introduction

This thesis's main focus is profiling and optimising the code which runs the Contur procedure. In section 2 we introduce Contur and provide the necessary background to understand what Contur is and what it does. We will also try to summarise the main parts of the flow of the Contur code. Understanding this flow of the Contur code will be helpful later when we outline the profiling done and the optimisations implemented on the back of this profiling.

In section 3 we will begin to start discussing the profiling performed. We will start with a quick outline of the benefits of improving Contur runtime and then move onto outlining the tool used to perform the profile and then the tools used to visualise the results. The section will then conclude with a presentation of the initial profiling results performed on Contur before any optimisation was attempted, these initial profiling results will be taken as our benchmark to judge the impact of later optimisation attempts.

In section 4 we will digress slightly to discuss introducing automatic tests into the Contur package. An important consideration is that when making changes to Contur for optimisation purposes, is that we don't unintentional break code or introduce errors. One means we can reduce the chances of introducing errors into Contur is via a robust testing infrastructure. The addition of these tests will form the content of section 4.

In section 5 we will outline the changes made the Contur code for optimisation. The section will outline the motivation for each of the changes, attempting to explain

how it improves Contur run wise. Additionally the section will provide updated profiles after each change so we have a high level visibility on each changes impact.

We will then conclude the thesis in section 6 by briefly 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.

Some stuff about things.^[?] Some more things.

Inline citation:

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 nature¹ 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 and dark energy. 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.

Constraints On New Theories Using Rivet (Contur) 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 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

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

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 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 with contur. Yet, despite this, the main event generator currently used by Contur is Herwig^[2] and the default assumption within the current Contur 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 and scatters plots, 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.

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 signal events changing the Contur 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 capabilities². 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 10^3 , from this contur batch can create a grid a 10×10 grid of 100 yoda files⁴. We can then pass this grid to contur and contur can run a consistency check for each on the grid, this is an example of a contur grid run, and is the main area of contur we will later focus on in our optimisation efforts.

²See Chapter 4 of the Contur User Manuel^[4] for further details

³For simplicity take the parameters to be dimensionless

⁴Yoda 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....

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 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 an 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,

$$CL_s := \frac{CL_{sb}}{CL_b} = \frac{p_{sb}}{1 - p_b},$$

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 buckets that compose the histogram or if correlation information between the buckets exists using this to compute a single CL_s for the histogram. For any given bucket 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.

```
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 pools. In Contur a pool is combination of the final particle state, the experiment and the beam energy of the experiment, 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

```
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

```
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 practise 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.

2.4 Contur Results

Chapter 3

Profiling Contur

The first task in the process of optimising Contur is to perform a profile of the code. The purpose of this profile is get a clear idea of the starting run time and how this run time is split between the functions and sub-functions which compose the whole Contur program.

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 visualize 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 Profile and Optimise?

The obvious reason why having faster code is of value to a researcher using Contur is because all else equal, the faster Contur can run a scan, the faster the researcher can get results and perform their analysis, thus improving productivity and convenience.

We can argue further though that faster code increases the range of analysis that a researcher can perform with Contur. This argument follows from the observation

that there likely exists a runtime above which Contur becomes impractical to use as a research tool¹. 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 at 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 requirements. Our main motivations for using cProfile are as follows:

- Provides a full profile of program with output include total run time, time

¹To 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.

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 made². 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 cProfile 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 script³ as follows

```
import cProfile

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

²The version can be found in commit 49a67e03, link provided here

³Can 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_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 contour 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 contour 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 Profile class is given below.

```
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 plots⁴.

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 contour code on a single yoda file. From the figure we can see that the icicle plot is showing the same information as figure 3.1 in just a more visually

⁴A 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 Contur run. From this we can already understand that a lot of the run time for a single Contur run comes from just reading in data.

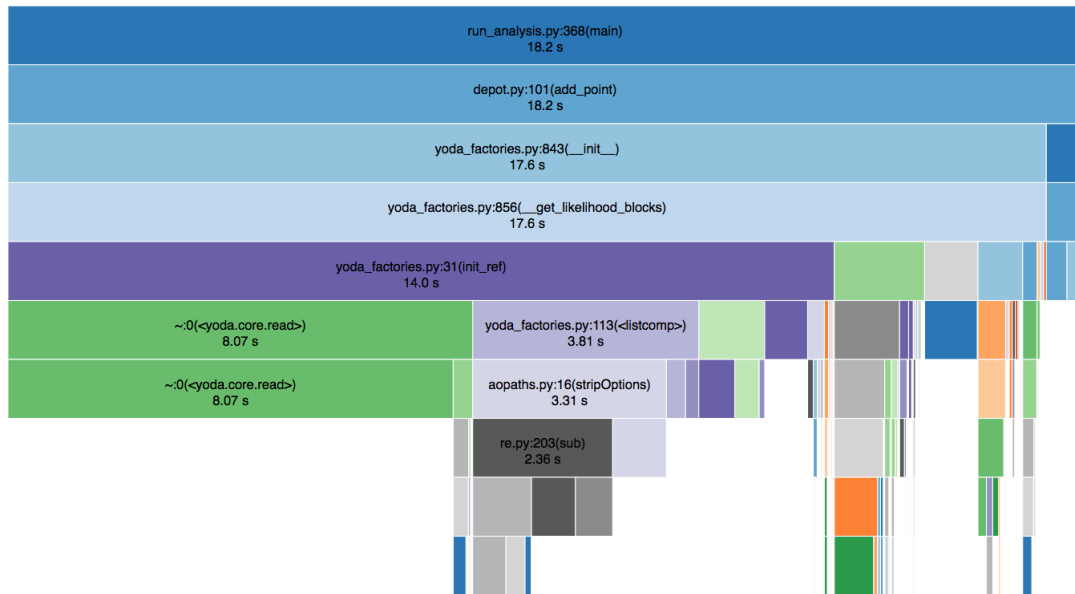


Figure 3.2: Contur single yoda run starting point - Example snakeviz icicle plot

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

[illegible]

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 contur 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 coloring 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 run time 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.

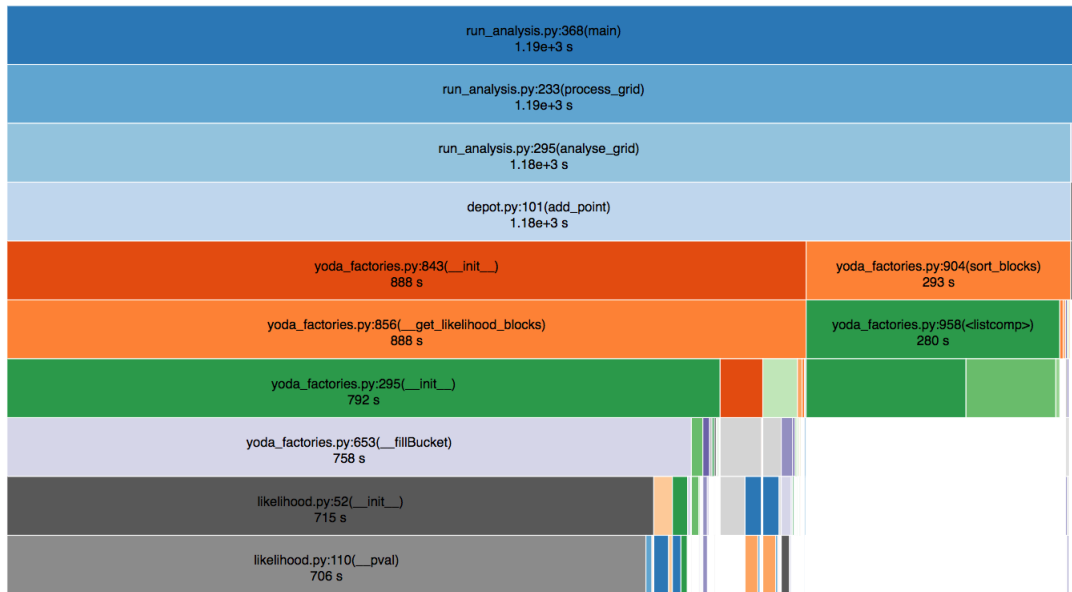


Figure 3.4: Contur grid run - icicle plot

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.

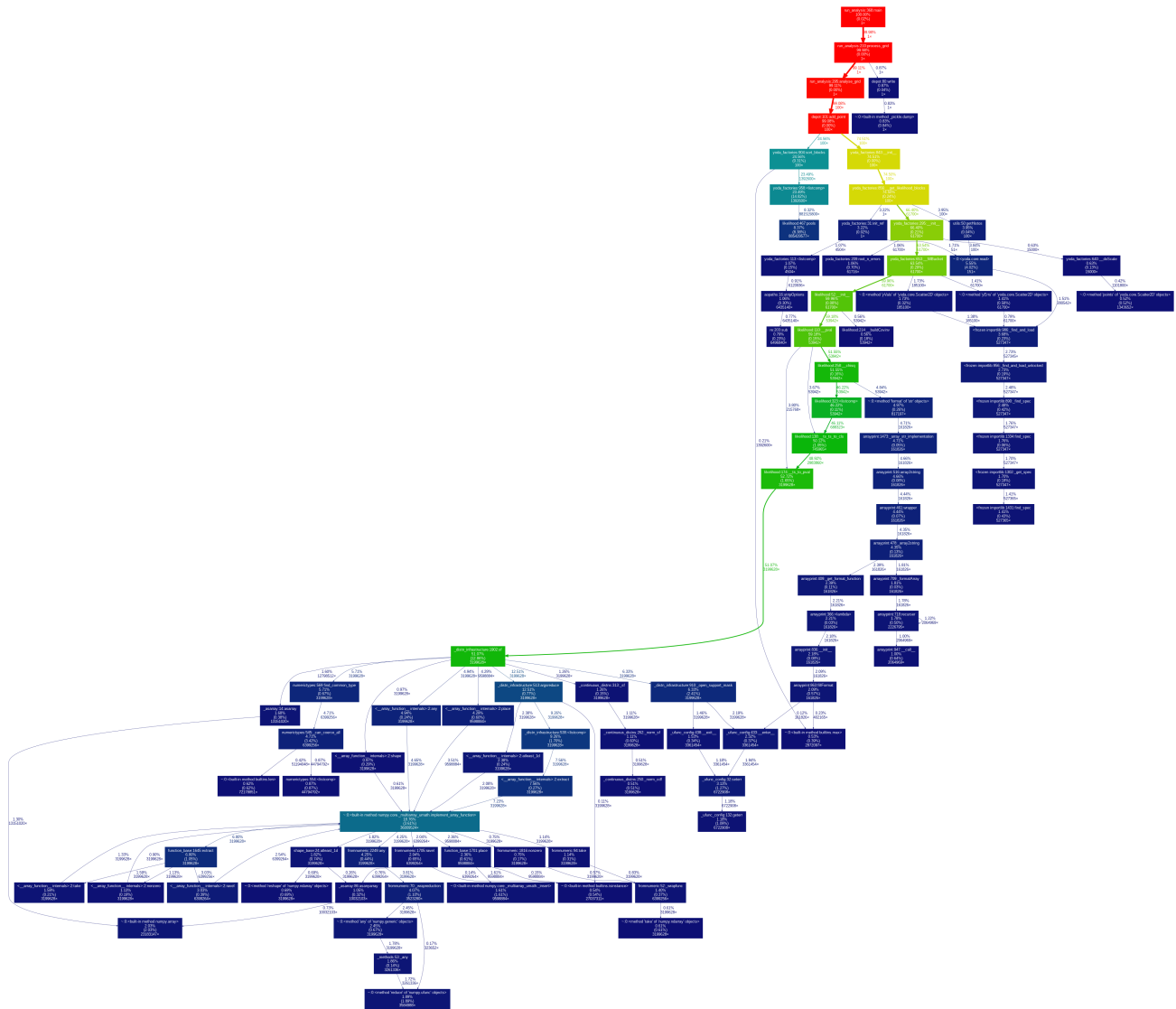


Figure 3.5: Contur grid run - dot plot

Chapter 4

Testing Contur

In this chapter will make a slight digression from profiling/optimisation to discuss testing within Contur. One risk with updating Contur code for optimisation purposes is that we unintentional break existing functionality or introduce errors into the code that impact the final results. To reduce this risk as much as is possible we devoted time during this project to improve Contur's automatic testing infrastructure.

4.1 Contur Existing Tests

Prior to work carried out in this thesis Contur had a limited set of tests¹ implemented within python's pytest framework^[9]. In the contur repository these tests can be found in the test folder. Within the tests folder there are two separate scripts to run tests, `test_batch_submit.py` and `test_executables.py`. These tests effectively test that functionality within contur runs without error, however the tests don't have any visibility on the output of the Contur run (except if the run throws an error before completion) or perform any form of unit testing.

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 Contur run². 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.

¹See initial tests here

²The 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 risk that changes to code don't break Contur in some way is to try and ensure that these changes don't alter the final output of the 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 our Contur run with the updated code (labeled the target) against the output of Contur before we made the change (labeled the base). The regression test is passed if our target output is equal to base output³.

Implementing these regression tests within the pytest framework will allow us to carry out these comparison of new results against old results automatically just by running pytest. Thus the regression tests we implement in Contur will be of wider use to other Contur developers to help ensure updates to Contur code do not unintentionally alter the output of Contur. 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 file contur runs and grid runs output their results in their formats. A single yoda file contur run outputs a text file with the results printed on the 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 file⁴

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

⁴This 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 file contur run - txt file

The grid run returns a .map file which contains a pickled

4.2.2 Implementing Regression Tests

4.2.3 Including Theory Runs

4.3 Unit Testing

4.3.1 Likelihood Class

4.3.2 YodaFactories Class

4.3.3 Functions

Chapter 5

Optimising Contur

5.1 Single Yoda Run

5.2 Grid Run

For research purposes, contur 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 run time with the grid run. This viewpoint comes from observing from figure 3.2 that the single yoda run time only takes around 20 seconds, while from figure 3.4 we can see that the run time for smallish grid¹ 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 minutes²;
- There is more scope for the grid run runtime to increase with changing research needs. The runtime for the grid run is highly dependent on the size of the grid

¹The grid we are profiling contur on 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), which each bucket having 100 yoda files for a total of 300 yoda files. So the 20 minutes we profiled for a single 10×10 would likely be close to an hour if run across the three energy buckets

²Or 30 minutes for the case where the grid has three energy buckets and 300 yoda files.

used, as the grid grows in size so will the runtime. There is a practical limit how big a grid can be for `contur` resulting from this increasing runtime, in effect once a grid is so large it is too slow to run `contur` on it. Optimisation 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 runtime could have very practical benefits like making `contur` 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` method³, which takes c.a. 25% of runtime and the second being the likelihood calculation⁴ which takes most of the remainder of the runtime. Our initial efforts will focus on making optimisation improvements for these two parts for the program.

5.2.1 Sort Blocks

5.2.1.1 Background

Before discussing the `sort blocks` method in detail it is first necessary to give some background into the flow of a `contur` grid run and where the `sort blocks` method sits within that flow. A grid run in `contur` is effectively just a group of single `yoda` runs performed within a loop with some slight differences⁵⁶. So we can focus in on the operations `contur` performs on each `yoda` file it is passed in the grid.

The `yoda factories` class in `contur` coordinates the run on each `yoda` file. When passed a `yoda` file `yoda factories` collects the data in the `yoda` file⁷ and the background

³See dark green box in dot plot

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

⁵The most notable difference being in a single `yoda` `contur` run a large part of run time is spent reading in `yoda` files from `Rivet` with the background data. In the grid run this data reading is done for the first `yoda` file in the grid, but it is not necessary to repeat it for each new `yoda` file in the grid because the data read in for the first `yoda` file can be reused.

⁶The `Depot` class handles coordinating the `contur` run across the grid, the `add point` method in the `Depot` class used to run `contur` on individual `yoda` files in the grid

⁷The `yoda` file passed to `yoda factories` contains the signal events for the beyond standard model we are using `contur` to test

experimental data needed to test against is also collected from Rivet⁸. The yoda file is composed of analysis objects, each analysis object is a histogram (is this correct? namely what I see as analysis object in the code is just another name for histogram) for which a confidence level can be calculated. The yoda factories class loops through these analysis objects, for each analysis object an instance of *contur*'s likelihood class is created, the likelihood class compute the confidence level for the analysis object and stores it as its CLS attribute. The yoda factory performs this whole loop upon instantiation⁹ and stores a list of likelihood object¹⁰ in its likelihood block attribute.

Yoda factories's sort blocks method is called after instantiation of a yoda factories object, so after all of the above calculations are carried out. Each likelihood object in yoda factories has collected has a pools attribute which gives the pool to which the analysis object used to create the likelihood object belongs. The sort blocks method simply buckets the likelihood objects by pool¹¹, and then for each pool it takes the likelihood object with the largest confidence level and discards the rest (Jon - please shout if this understanding is not correct).

5.2.1.2 Changes made

The profiling results in the dot plot in figure 3.2 are precise enough that we can pinpoint the major slow point in the sort blocks method to be the list comprehension in line 958 of yoda factories¹². This list comprehension sits within a loop and the whole blocks of code looks as follows

```
for p in pools:
    if not p == omitted_pools:
        for item in like_blocks:
            if item.CLS == max([x.CLS for x in like_blocks if x.pools == p]) \
```

⁸When running with theory option standard model simulated results will also be collected from Rivet

⁹This statement is true for our starting version of *contur*, as will be shown in 5.2.2, updates we will make for the likelihood optimisation will alter this flow

¹⁰Each likelihood object in this list will have the confidence level for the analysis object it refers to stored in its CLS attribute

¹¹So collects all the likelihood objects with the same pool into a single bucket

¹²This can also be easily picked up in the icicle plot when using Snakeviz's graphical viewer

In the above block of code the list comprehension in the last line is used to find the likelihood block in the pool with the largest confidence. The list comprehension on its own is not what makes the code slow, the list comprehension is placed within a nested for loop, which loops over pools and then within each pool iterate we have another loop over likelihood blocks. So in each yoda file the number of times we perform this list comprehension is the number of pools multiplied by the number of likelihood objects, and this is only for a single yoda file, across the whole grid we are doing this 100 times. Figure 5.1 below shows that for our 10×10 grid we are calling this list comprehension over a million times. So although each comprehension on its own takes less than .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 calculation of the max confidence level for each pool performed by the list comprehension outside of the nested for loop, thus reducing dramatically the number of times the calculation is performed. In its place we now only perform the calculation once per yoda file¹³ and store the results in a dictionary whose key is the pool and value is the max confidence level for the pool. This dictionary is then used in place of the list comprehension in the above for loop. 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 call¹⁴ 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.

5.2.1.3 Impact of changes

Below we see the impact of the optimisation on the run time for the sort blocks method. In figure 5.2 we can see that prior to optimisation the run time of the list

¹³So 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

¹⁴See <https://towardsdatascience.com/faster-lookups-in-python-1d7503e9cd38>

comprehension is in line with our expectations from the previous section with a run time of 280 seconds, order 10^2 as expected and total run time¹⁵ for sort blocks is 293 seconds. While in figure ?? we see that we have a post optimisation run time of 7 seconds for the whole of the sort blocks method. This run time would support the hypothesis that the run time for calling the dictionary in place of the list comprehension is of order 10^{-1} seconds and additional would suggest that the optimisation change made the calculation of the maximum confidence level for each pool slightly faster too, as prior to optimisation sort blocks had 13 seconds of run time outside of the list comprehension, now after optimisation total run time is just 7 seconds.



Figure 5.2: Sort blocks grid run time - Before optimisation

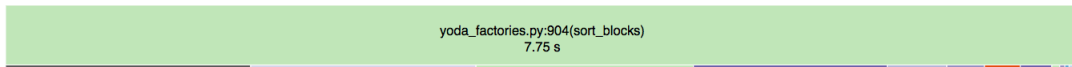


Figure 5.3: Sort blocks grid run time - After optimisation

5.2.2 Likelihood Calculation

5.2.2.1 Background

In the outline of the background for the sort blocks method in section 5.2.1 we touched on when the likelihood class comes into the flow of a contour run, namely how the yoda factory loops through all the analysis objects in the passed yoda file, instantiating a likelihood object for each analysis object which computes a confidence level. We however ignored the details of how these confidence levels are computed. We can see from the icicle plot in figure 3.4 that a material proportion of contour's run time is spent within the likelihood class¹⁶.

¹⁵Remember the list comprehension is only part of the sort blocks method, not the whole of it so we will have additional run time from other parts of the method

¹⁶From 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

Before outlining the steps taken to reduce the likelihood run time it is worth briefly outlining in greater detail the steps within the likelihood class to compute the confidence level for each analysis object. Upon instantiation the likelihood class computes two chi-square test statistics, a background and target test statistic (greater detail needs to be added here). These test statistics are then converted into p-values by assuming the test statistics are normally distributed and computing their survival function value¹⁷. We use the sf method of the norm class found in `scipy.stats` to compute the p-values. Once a background and target p-values are computed they can be combined to calculate the confidence level for the analysis object.

Drilling down into the initial grid run profile results for the likelihood class 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 computes p-values to test statistic and really just calls `scipy.stats.norm.sf` under the hood, so from here on we can focus our efforts on the `scipy.stats.norm.sf` method.

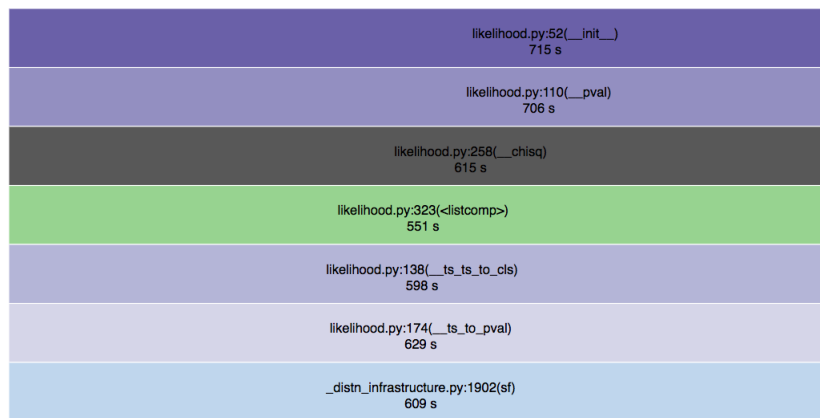


Figure 5.4: Likelihood object - Initial profile

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.

¹⁷Defined by 1-cdf, so the area of the distribution to the right of the test statistic

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

5.2.2.2 Scipy Survival Method

From the previous section it should be clear that the survival function and reducing the number of times we call it is key to reducing the run time in the likelihood object. It is thus helpful to first understand better where the calls to the survival function arise when a likelihood class is instantiated. Within a likelihood object we get calls to the scipy 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 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 chi square test statistics will only call the survival function when an inverse for the covariance matrix of the analysis object cannot be computed¹⁸. 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 analysis object will have at least 4 calls to the survival function¹⁹, so across a whole yoda file the number of calls to the survival function will be at least 4 multiplied by the number of analysis objects in the yoda

¹⁸In the code the survival function will only be called when the condition "self.covBuilt and sb nuisance is not None" is false

¹⁹In practise it will be a lot more than this because of the chisq calls

file²⁰. 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 got rid of, 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 calls²¹.

For the second approach to be effective we would require the run time for the survival function if passed a numpy array of length n to be significantly less than the run if we just made n separate calls to the survival function. We would expect this to be the case as the Scipy functions are built to enable fast array based computation on 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 run time 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 run time between 10^2 - 10^3 seconds which is in line with the c.a. 600 seconds we actually observe.

²⁰For 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 analysis objects

²¹So 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

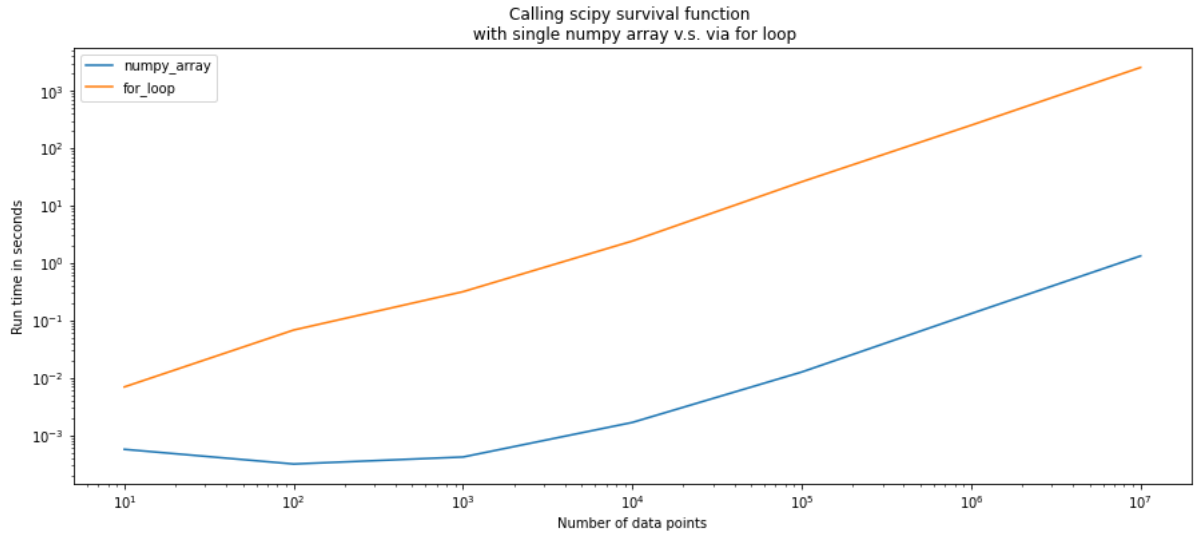


Figure 5.6: Scipy Survival Function - Loop vs Array

5.2.2.3 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. Making use of numpy arrays to reduce the number of calls to the survival function across likelihood objects;

Of the above changes the first two are least disruptive in terms of their impact in overall flow of a *contur* run as they only make changes within the likelihood class. While the third change is more substantial as it alters the flow of a *contur* run between yoda factories and the likelihood class.

The first change²² involves passing a tuple of background and signal test statistics to the *ts to pval* method and to the *ts ts to cls* methods, as opposed to passing the test statistics as separate calls. This reduces the calls to the survival function for

²²The change can be found in commit 0b807895

these values from 4 to 2. In addition within the `chisq` method we also now pass a list of tuples of test statistics as opposed to making separate calls to the survival function. This ensures that whenever the `chisq` method makes on a call to the survival function it only makes 1 call, so after all these changes, for analysis object we get at least 2 calls to the survival function and at most 3.

The second change²³ made is motivated from the observation that the `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 introduce the `pval` to `cls` method which directly takes a `p` value and computes a confidence level 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 analysis object.

From here on in let us simplify and assume that we have exactly 2 calls to the survival function for each analysis object and for each yoda file we have m analysis objects. 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 on 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 which may be used by `contur`. The final set of optimisation changes we will make will resolve this issue by getting rid of the run time dependence on m .

To achieve this, let us first remind ourselves how the the current set up works. Upon instantiation of the `yoda factories` class we loop through all analysis objects instantiating a likelihood object for each and doing the full likelihood calculation (i.e. computing the confidence level) upon instantiation of the class. So at the end of this process we have a yoda factory object with a `likelihood blocks` attribute that contains all the likelihood objects. This process can be split into steps that allow us eliminate dependence on the number of analysis objects. This can be done by

²³The change can be found in commit 15ef923d

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 the survival function.

Implementing this change in practise is spread across two commits²⁴.

For the first commit we introduce the likelihood blocks `ts to cls` function, which takes a list of likelihood blocks and computes a confidence level for each likelihood block in the list. In terms of the flow of `contur`, with this change we alter the likelihood object so it no longer calculates the confidence level upon instantiation, so when we instantiate yoda factories we now have a list of likelihood objects in the likelihood blocks attribute with just test statistics not confidence level, we pass this list to the new function which does the confidence level calculation. After this change we will $n + (n \times m)$ calls to the survival function.

For the second commit we introduce the likelihood blocks `find dominant ts` function. This function finds the chi square test statistic in an analysis object that gives the largest confidence when the covariance matrix does not have a valid inverse. Thus it moves the calls to the survival function that take place within the `chisqr` 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.

5.2.2.4 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 confidence level 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 20 seconds.

²⁴The first commit 299b03a8 introduces the likelihood blocks `ts to cls` function, while the second commit 2769e1c2 introduces the likelihood blocks `find dominant ts` function

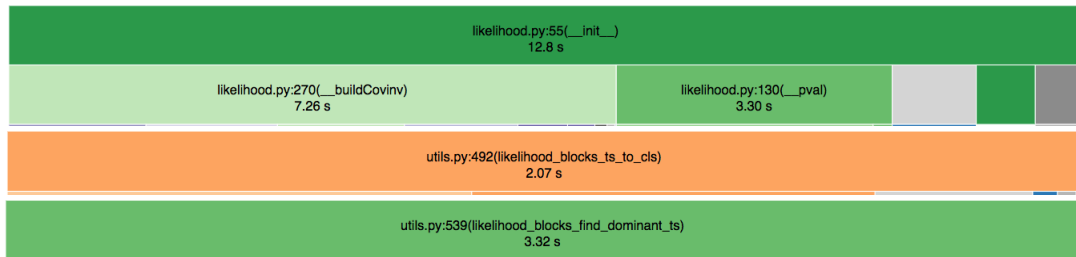


Figure 5.7: Likelihood object and new functions - Profile after optimisation

Chapter 6

General Conclusions

Appendix A

An Appendix About Stuff

(stuff)

Appendix B

Another Appendix About Things

(things)

Appendix C

Colophon

This is a description of the tools you used to make your thesis. It helps people make future documents, reminds you, and looks good.

(example) This document was set in the Times Roman typeface using L^AT_EX and BibT_EX, composed with a text editor.

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>