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 3, 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

List of Figures

List of Tables

Chapter 1

Introduction

Some stuff about things.[1] Some more things.

Inline citation: Anne Author. Example Journal Paper Title. *Journal of Classic Examples*, 1(1):e1001745+, January 1970

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 2

Contur Overview

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 3

Profiling Contur

This chapter will outline how we went about performing a profile of contur. We will start by introducing cProfile, 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 optimization 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.

3.1 Profiling with cProfile

3.1.1 Why cProfile?

Let us first begin by considering the features we ideally require from our profiler to make our task of improving the performance of contur easier. 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 our program we would also like our profiler to provide a split of this runtime between the component parts which compose the program. This requirement is especially important for a large code base like contur which is being profiled by someone not involved in the development of the code base.

cProfile is a module within the Python standard library which provides a profiler which meets all our requirements for a profiler, in addition it provides other useful features. Our main motivations for using cProfile are as follows:

1. 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;

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

3. Performing the profile with cProfile is quick and easy and requires minimal new code;

3.1.2 Using cProfile

cProfile is simple to use, this can be seen by considering the most straightforward profile of contur we can do using cProfile's run function. In the contur run script here we can just pass main to the cProfile run method as follows

```
import cProfile

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

When we run contur with the above update on a single yoda file we get the following terminal output with the profiling results

- 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 pstat 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;

```
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 function 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, an in dept 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 do 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'')
```

3.2 Visualizing Profiling Results

To visualise our profiling results we will use two open source tools Snakeviz and gprof2dot. As the following will attempt to show both of these tools can be used in a complementary ways to aid in best using the profiling data output from cProfile.

3.2.1 Snakeviz

Snakeviz is a browser based graphical viewer for the output of Python's cProfile profiler module. Snakeviz can easily be piped 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. From here on we will use Snakeviz's icicle plot to explore profiling results, additionally due to the constraints of the static form 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 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 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.

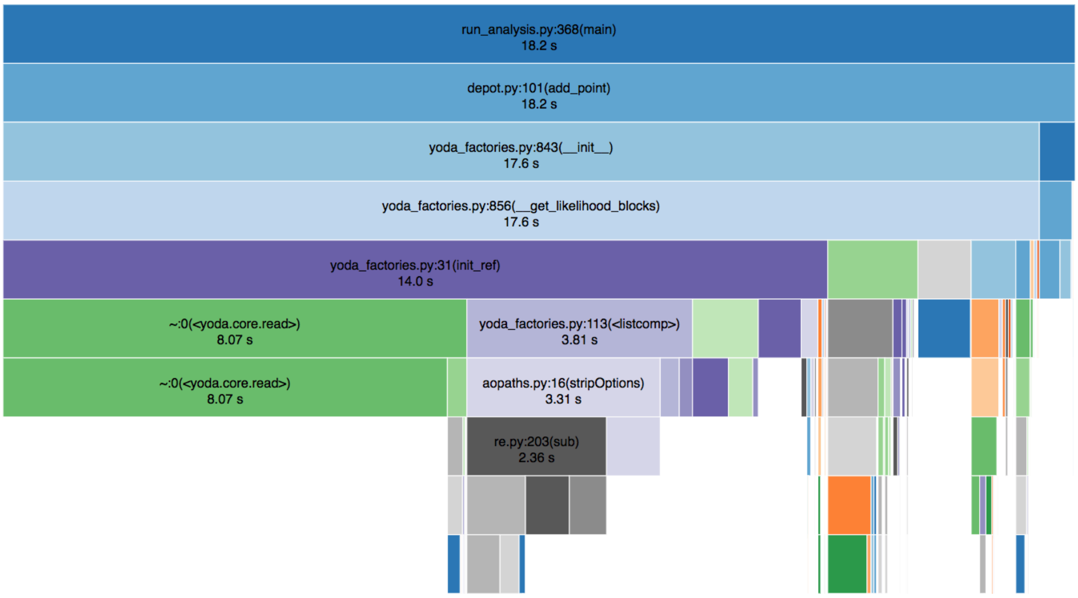

Figure 3.2: Contur single yoda run starting point - Example snakeviz icicle plot

3.2.2 gprof2dot

gprof2dot 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 of our code and how this absolute run time breaks down among the components of the program. The dot plot complement 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.

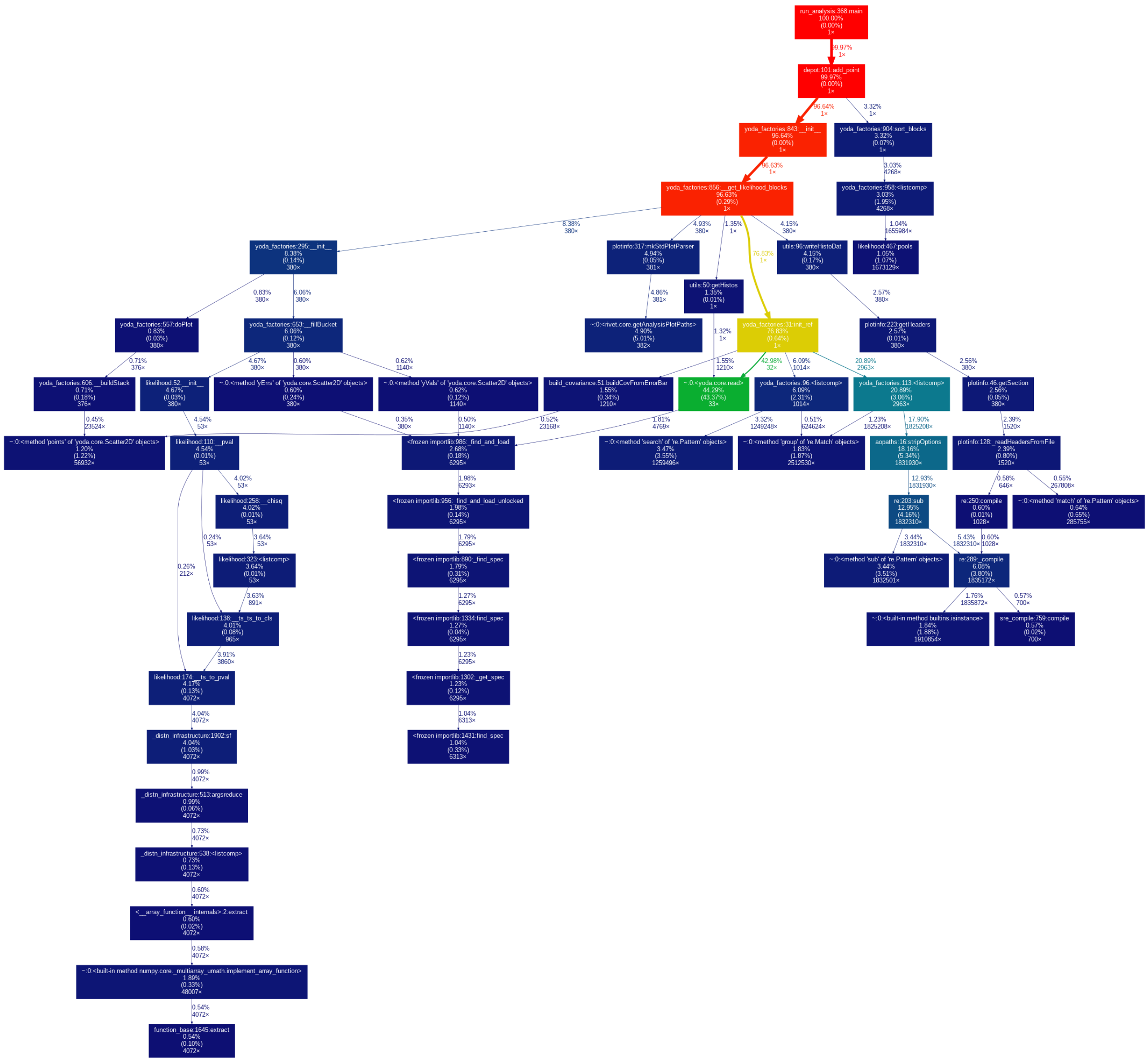

Figure 3.3: Contur single yoda run starting point - Example gprof2dot

We can see example of the dot plots produced by gprof2dot in figure 3.3 above. This plot is visualising the same single yoda 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.

3.3 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 through out 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.

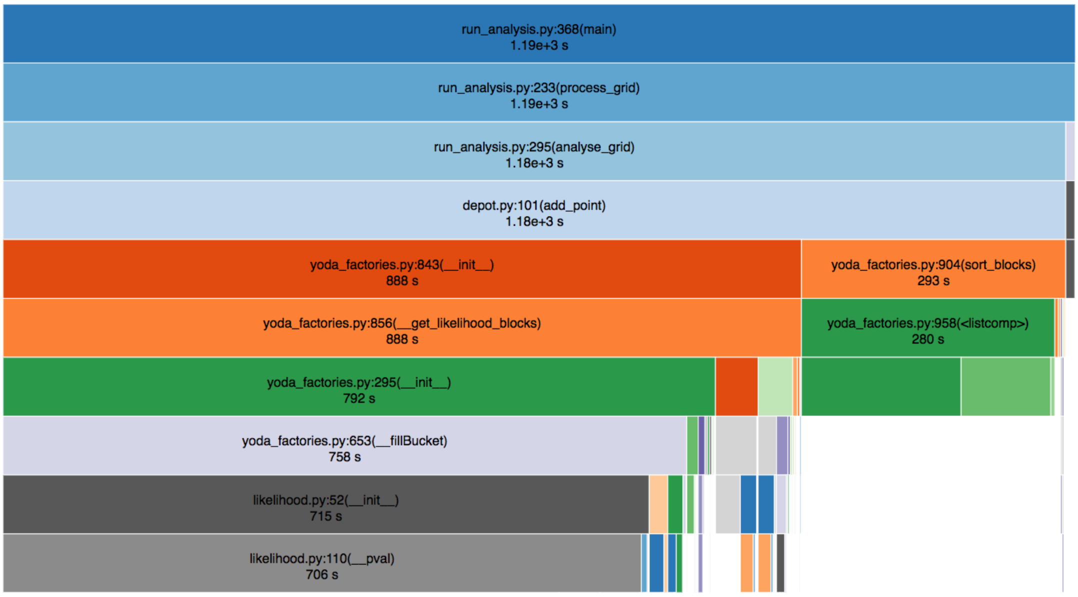

Figure 3.4: Contur grid run - icicle plot

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.

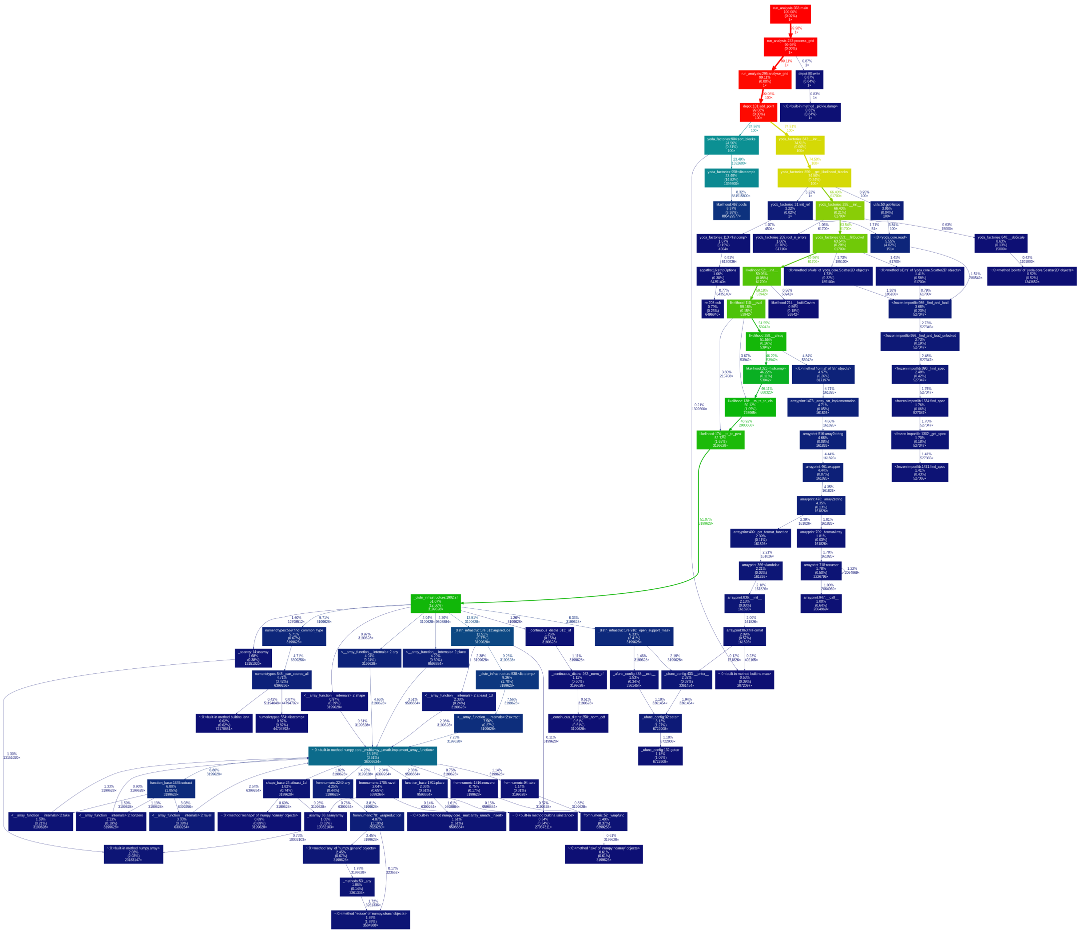

Figure 3.5: Contur grid run - dot plot

Chapter 4

Testing Contur

Before attempting to update contur code to improve it's run time performance we need to give some consideration into how we can ensure that we don't break some of contur's existing functionality via our changes. Our changes for optimisation should complete the same tasks as the code we are updating and return the same outputs. In this chapter we will outline how we attempt to ensure the optimisation changes don't break existing contur functionality via improving existing testing within contur.

4.1 Contur Existing Tests

Prior to work carried out in this thesis contur had a limited set of tests implemented within python's pytest framework. 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 grid runs without errors. To carry out these tests pytest does a single yoda and grid contur run[1]. 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]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 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 output[2].

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 file[3]

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

[3]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 file contur run - txt file

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 Sort Blocks

5.2 Likelihood Calculation

Chapter 6

General Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

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 LaTeX and BibTeX, composed with a text editor.

Bibliography

[1] Anne Author. Example Journal Paper Title. *Journal of Classic Examples*, 1(1):e1001745+, January 1970.