# profiling

## Line by line profiling and code coverage for GAP

2.5.1

10 October 2022

**Christopher Jefferson**

**Christopher Jefferson**
Email: caj21@st-andrews.ac.uk
Homepage: https://caj.host.cs.st-andrews.ac.uk/
Address: St Andrews
Scotland
UK

# Contents

# Chapter 1

# Tutorial

## 1.1 Line-by-line profiling

The purpose of this section is to show how to use GAP's line-by-line profiling / code coverage. For this, you need GAP 4.10 or newer.

Do you just care which lines of code are executed? Then you should switch to the coverage guide (these two guides are very similar!)

We will start with a quick guide to profiling, with some brief comments. We will explain later how to do these things in greater depth!

Let's start with some code we want to profile. Here I am going to profile the function `f` given below, and use a group from the AtlasRep package.

```
LoadPackage("atlasrep");
a := AtlasGroup("U6(2)", NrMovedPoints, 12474);
b := a^(1,2,3);
f := function() Intersection(a,b); end;
```

Firstly, we will record a profile of the function `f`:

```
# Code between ProfileLineByLine and UnprofileLineByLine is recorded
# to a file output.gz
ProfileLineByLine("output.gz"); f(); UnprofileLineByLine();
```

You should write this all on a single line in GAP, as profiling records the real time spent executing code, so time spent typing commands will be counted.

This creates a file called `output.gz`, which stores the result of running `f`. Now we want to turn that into a nice output. This requires loading the profiling package, like this:

```
LoadPackage("profiling");
OutputAnnotatedCodeCoverageFiles("output.gz", "outdir");
```

If loading the profiling package produces errors, make sure you have compiled both the profiling and IO packages.

`OutputAnnotatedCodeCoverageFiles` (2.3.1) reads the previously created `output.gz` and produces HTML output into the directory `outdir`.

You must view the result of your profiling in a web-browser outside of GAP. Open `index.html` from the `outdir` directory in the web browser of your choice to see what happened.

At the very top is a link to a flame graph. These give a quick overview of which functions took the most time. Functions are stacked, so lower functions call higher functions.

From this graph we can see that `f` called `Intersection` (**Reference: Intersection**), which called the function `Intersection2 perm groups` near line 2950 in `stbcbckt.gi`. This function spent most of its time in `PartitionBacktrack`, and a little time in `Stabilizer`.

Whenever you generate a profile which contains timing information, a flame graph link will be show on the first page of your generated profile!

## 1.2 FAQ / Problems

- `ProfileLineByLine` (**Reference: ProfileLineByLine**) records the wall time (also known as clock time) that occurs between `ProfileLineByLine` (**Reference: ProfileLineByLine**) and the next `UnprofileLineByLine` (**Reference: UnprofileLineByLine**). This is why we start profiling, run our code, and then stop profiling all on a single line.

- If you want to profile how long everything in GAP takes, including the startup, then you can do this by giving the command line option `-prof filename` when starting GAP. This is equivalent to GAP calling `ProfileLineByLine("filename");` before loading any of the standard library (obviously, give your own filename).

- Giving your output file the gz extension makes GAP automatically compress the file. This is a great idea, because the files can get very big otherwise! Even then, the files can grow quite large very quickly, keep an eye on them.

- `ProfileLineByLine` (**Reference: ProfileLineByLine**) takes an optional second argument which is a record, which can set some configuration options. Here are some of the options:

- `wallTime`: Boolean (defaults to `true`). Sets if time should be measured using wall-clock time (true) or CPU time (false). Measuring CPU-time has a higher overhead, so avoid it if at all possible!

- `resolution`: Integer (defaults to `0`). By default GAP will record a trace of all executed code. When non-zero, GAP instead samples which piece of code is being executed every `resolution` nanoseconds. Setting this to a non-zero value improves performance and produces smaller traces, at the cost of accuracy. GAP will still accurately record which statements are executed at least once. This is mainly useful when you wish to consider very long-running code.

## 1.3 Function-based profiling

Sometimes you will have code that just runs too long to easily profile line-by-line. You can profile this in GAP's older function-based profiler. You can read more about this profiler in GAP's documentation ( (**Reference: Profiling**)), but here is a quick example to get you going!

```
ProfileGlobalFunctions(true);
ProfileOperationsAndMethods(true);
```

```
f();
ProfileGlobalFunctions(false);
ProfileOperationsAndMethods(false);
DisplayProfile();
```

# Chapter 2

# Functionality provided by the profiling package

## 2.1 Reading line-by-line profiles

### 2.1.1 ReadLineByLineProfile

▷ `ReadLineByLineProfile(`*`filename`*`)` (function)

Read *`filename`*, a line-by-line profile which was previously generated by GAP, using the `ProfileLineByLine` (**Reference: ProfileLineByLine**) or `CoverageLineByLine` (**Reference: CoverageLineByLine**) functions from core GAP. A parsed profile can be transformed into a human-readable form using either `OutputAnnotatedCodeCoverageFiles` (2.3.1) or `OutputFlameGraph` (2.2.1)

### 2.1.2 MergeLineByLineProfiles

▷ `MergeLineByLineProfiles(`*`filenames`*`)` (function)

Read *`filenames`*, a list of line-by-line profiles which were previously generated by GAP, using the `ProfileLineByLine` (**Reference: ProfileLineByLine**) or `CoverageLineByLine` (**Reference: CoverageLineByLine**) functions from core GAP. The elements of *`filenames`* can be either filenames, or files previously parsed by `ReadLineByLineProfile` (2.1.1).

## 2.2 Generating flame graphs

A 'flame graph' is a method of visualising where time is spent by a program.

### 2.2.1 OutputFlameGraph

▷ `OutputFlameGraph(`*`profile[, filename][, options]`*`)` (function)

Generate an 'svg' file which represents a 'flame graph', a method of visualising where time is spent by a program.

*profile* should be either a profile previously read by ReadLineByLineProfile (2.1.1), or a string giving the filename of a profile.

The flame graph will be written to *filename* (or returned as a string if *filename* is not present).

The final (optional) argument is a record of options. Currently, the allowed options are 'squash' (which is a boolean). If 'squash' is true then recursive functions calls will be squashed, so the graph will not show recursive functions calling themselves. The other allowed option is 'type', which can be "default" (a standard flamegraph), "reverse" (reverse the graph, showing the leaf functions first), and "chart" (where the graph shows the flow of time from left-to-right, rather than merging all calls from one function to another into one block).

### 2.2.2 OutputFlameGraphInput

▷ OutputFlameGraphInput(*profile[, filename]*) (function)

Generate the input required to draw a 'flame graph', a method of visualising where time is spent by a program. One program for drawing flame graphs using this output can be found at https://github.com/brendangregg/FlameGraph.

*profile* should be either a profile previously read by ReadLineByLineProfile (2.1.1), or a string giving the filename of a profile.

The flame graph input will be written to *filename* (or returned as a string if *filename* is not present).

## 2.3 Generating coverage reports

### 2.3.1 OutputAnnotatedCodeCoverageFiles

▷ OutputAnnotatedCodeCoverageFiles(*coverage[, indir], outdir[, options]*) (function)

Takes a previously generated profile and outputs HTML which shows the lines of code that were executed, and (if this was originally recorded) how long was spent executing these lines of code.

*coverage* should be either a profile previously read by ReadLineByLineProfile (2.1.1), or a string giving the filename of a profile which will first be read with ReadLineByLineProfile.

Files will be written to the directory *outdir*.

The optional second argument gives a filter, only information about filenames starting with *indir* will be outputted.

The final optional argument is a record of configuration options. The only currently allowed option is 'title', which will set the title of created pages.

### 2.3.2 OutputJsonCoverage

▷ OutputJsonCoverage(*coverage, outfile*) (function)

Takes a previously generated profile and outputs a json coverage file which is amongst other things accepted by codecov.io.

*coverage* should be either a profile previously read by ReadLineByLineProfile (2.1.1), or a string giving the filename of a profile which will first be read with ReadLineByLineProfile.

The output will be written to the file with name *outfile* (a string).

### 2.3.3 OutputLcovCoverage

▷ `OutputLcovCoverage(`*coverage, outfile*`)` (function)

Takes a previously generated profile and outputs an lcov coverage file.

*coverage* should be either a profile previously read by `ReadLineByLineProfile` (2.1.1), or the filename of a profile which will first be read with `ReadLineByLineProfile`.

The output will be written to the file with name *outfile* (a string).

### 2.3.4 OutputCoverallsJsonCoverage

▷ `OutputCoverallsJsonCoverage(`*coverage, outfile, pathtoremove[, opt]*`)` (function)

Takes a previously generated profile and outputs a json coverage file which is accepted by `coveralls.io`.

*coverage* should be either a profile previously read by `ReadLineByLineProfile` (2.1.1), or a string giving the filename of a profile which will first be read with `ReadLineByLineProfile`. *pathtoremove* is the path to the tested repository; this path prefix will be removed from all filenames in *coverage*. Finally, *opt* is a record. Its key/value pairs are directly inserted into the produced JSON, in the form of a JSON dictionary. This can be used to set the `service_name`, `service_job_id`, and more. If this record is not given, we try to guess the correct values based on the environment (currently only supported for Travis and AppVeyor).

The output will be written to the file with name *outfile* (a string).

## 2.4 Miscellaneous

### 2.4.1 LineByLineProfileFunction

▷ `LineByLineProfileFunction(`*function, arguments*`)` (function)

Calls *function* with the list of arguments *arguments*, and opens a time profile of the resulting call in the default web browser.

### 2.4.2 ProfileFile

▷ `ProfileFile(`*file[, opts]*`)` (function)

**Returns:** a string

Tests the file with name *file* in another GAP session, and produces a code coverage report of lines that were executed in the process. If *file* ends with `.tst` it will be called with `Test`; otherwise, it will be run directly.

The optional argument *opts* should be a record, and may contain any of the following components:

- `outdir`: a string denoting the directory into which the HTML files of the report will be placed (a temporary directory by default);

- `indir`: a string such that only file paths beginning with `indir` will be profiled (default `""`);
- `showOutput`: a boolean denoting whether to print test output to the screen (default `true`);
- `open`: a boolean denoting whether to open the report in a web browser on completion (default `false`).

This function returns the location of an HTML file containing the report.

### 2.4.3 ProfilePackage

▷ `ProfilePackage(`*`pkg_name[, opts]`*`)` (function)

**Returns:** a string

If *`pkg_name`* is the name of an installed package, then this function runs that package's test suite and produces a report on the code coverage of files inside the package. The string returned denotes the location of an HTML file containing the report. The optional argument *`opts`* behaves the same as in `ProfileFile` (2.4.2).

# Index