**Program usage:**

Included in this repository are three different versions of the code: Python, Fortran and an original reference implementation in Matlab. The Python code has been tested in Anaconda 3.6.1 and Enthought 3.5.2, requiring only standard libraries such as NumPy, SciPy and Matplotlib. Use of the included F2PY code requires a correctly configured Python environment and suitable compilers, see documentation for more information. The Fortran code has been developed using Cygwin on Windows 10, however has also been tested on a Ubuntu installation, and suitable Linux Makefiles and Windows batch files are included for use. Required Fortran libraries include the standard LAPACK and BLAS, in addition to the matrix exponential library Expokit. Currently the Expokit library is linked a run-time as a static library via included object files, compiled for use on Windows 10 and Ubuntu, however ideally this library should be built from source and linked dynamically.

To change the simulation parameters, modify those included in ‘\*\_main.f90’, as due to time constrains a common parameters file has not been created however this would be a useful inclusion in the future. If required the OMP declarations can be modified to change the simulation from series to parallel performance, note that if the master loop in ‘\*\_main.f90’ is to be run in parallel then the ‘\*\_dynamics.f90’ file must be set to run in series manually as OMP declarations are present in both files.

**Program structure:**

Included in the Fortran directory are a number of files including: .f90 Fortran source files, .py Python source files, .cmd Windows batch files, Linux Makefiles and an ‘out’ directory where the simulation output is stored. There are two common Fortran files: ‘functions.f90’ where common routine Fortran functions and subroutines are stored, and ‘interactions.f90’ where subroutines for calculating particle interactions are stored. There are a further two simulation specific Fortran files, for solid effect: ‘solid\_effect\_dynamics.f90’ where the dynamics code is stored, and ‘solid\_effect\_main.f90’ where the parameters, I/O operations and master loop are stored. Python plotting code such as ‘solid\_effect\_plotting.py’ are also included. Similar structure is present for the included Python code.

**Program performance:**

For a single solid effect run Fortran performs around 22x faster than Matlab, and for a parallel run of 40 jobs performs 25x faster. However, for a single cross effect run this is reduced to only 2x faster. As can be seen in the included benchmarks and profiles the Matlab BLAS routines scale significantly better than the Fortran ones, this is likely because Matlab uses a highly optimised multithreaded MKL BLAS implementation. In comparison Fortran and Python are both using single threaded BLAS implementations, with Fortran using the default LAPACK/BLAS installation available from Cygwin (Windows 10) and Ubuntu. It is highly likely that if Fortran and Python are linked against more optimised BLAS implementations such as MKL or OpenBLAS their performance would be greatly improved, hopefully comparable to or even exceeding Matlab. This has not been performed due to time constraints and the difficulty of doing so.

**Relaxation:**

Initial benchmarks suggest that the Mance relaxion method is consistently faster than the original method, for all three tested languages and for both cross effect and solid effect. This is likely due to the high computational cost of Kronecker products used in the original method, despite the addition of many for loops in the Mance relaxation method which are typically slow in languages such as Matlab and Python. While the Mance relaxation method requires ordering of eigenvalues and eigenvectors (as performed automatically by ‘Eigenshuffle’ in Matlab), this adds a negligible cost of around 1ms for solid effect and 9ms for cross effect in Fortran.

However, when a custom Python Kronecker product function for use with identity matrices is used the performance of the two methods becomes comparable. Sadly, this is not the case in Fortran likely because the cost of reshaping the input array is comparable to simply multiplying the values by one.

Therefore, further work is needed to benchmark the two relaxation methods, with emphasis on the language of interest for performing the simulations.

**Program benchmarks (laptop Intel i7 3.2MHz)**

**Solid effect (original relaxation, 1 run):**

Matlab (s): 9.58

Python + NumPy (s): 14.19 (default Kronecker product), 12.38 (custom Kronecker product with identity matrix)

Fortran (s): 1.87

Fortran + OpenMP (s): 0.82

Fortran + OpenMP, Ubuntu (s): 0.44, **22x faster**

**Solid effect (Mance relaxation, 1 run):**

Matlab (s): 8.42

Python + NumPy (s): 12.48

Fortran (s): 1.82

Fortran + OpenMP (s): 0.84

**Solid effect (original relaxation, 40 runs):**

Matlab (s): 286.4

Python + NumPy (s): 551.4

Fortran (s): 75.5

Fortran + OpenMP in series (s): 35.4

Fortran + OpenMP in parallel (s): 20.6

Fortran + OpenMP in parallel, Ubuntu (s): 11.6, **25x faster**

**Cross effect (original relaxation, 1 run):**

Matlab (s): 33.5

Python + NumPy (s): 60.3

Fortran (s): 51.2

Fortran, Ubuntu (s): 46.3

Fortran + OpenMP (s): 18.0

Fortran + OpenMP, Ubuntu (s): 17.5, **2x faster**

**General benchmarks (laptop Intel i7 3.2MHz)**

**Calculating 1E4 matrix exponentials 4x4:** *(Solid effect propagator)*

Matlab (s): 2.78

Python + SciPy (s): 5.58

Fortran + Expokit (s): 0.76

**Calculating 1E4 matrix exponentials 8x8:** *(Cross effect propagator)*

Matlab (s): 14.1

Python + SciPy (s): 43.1

Fortran + Expokit (s): 41.5

Fortran + Expokit + OpenMP (s): 10.4

**Calculating 1E4 matrix exponentials 4, 8, 30, 60, 80:** *(Simplified)*

Matlab (s): 0.91, 0.97, 2.36, 5.45, 8.69

Python + SciPy (s): 2.59, 2.89, 9.70, 35.4, 72.7

Fortran + Expokit (s): 0.037, 0.12, 4.14, 30.6, 74.9

Fortran + Expokit + OpenMP (s): 0.0039, 0.016, 0.52, 3.87, 9.53

**Calculating 1E5 Kronecker products 8x8:** *(Cross effect Liouville space eigenvectors)*

Matlab (s): 1.87

Python + NumPy (s): 3.95

Fortran (s): 0.65

**Calculating 1E6 Kronecker products 8x8 (identity):** *(Cross effect Hamiltonian frame transformation)*

Matlab full (s): 17.9

Python full (s): 28.5

Python identity (s): 10.2

Fortran full (s): 3.26

Fortran identity (s): 3.38

**Calculating 1E8 matrix products (2x2 real):** *(Spin matrices xz)*

Matlab (s): 49.27

Python + NumPy (s): 95.2

Fortran (s): 0.13

Fortran + BLAS (s): 5.86

Fortran + BLAS direct (s): 2.12

**Calculating 1E6 matrix products (8x8 real):** *(Cross effect Sz frame transformation)*

Matlab (s): 1.09

Python + NumPy (s): 2.56

Fortran (s): 2.23

Fortran + BLAS (s): 1.39

Fortran + BLAS direct (s): 1.21

**Calculating 1E4 matrix products (64x64 real):** *(Cross effect propagation real component)*

Matlab (s): 0.18

Python + NumPy (s): 0.27

Fortran (s): 0.70

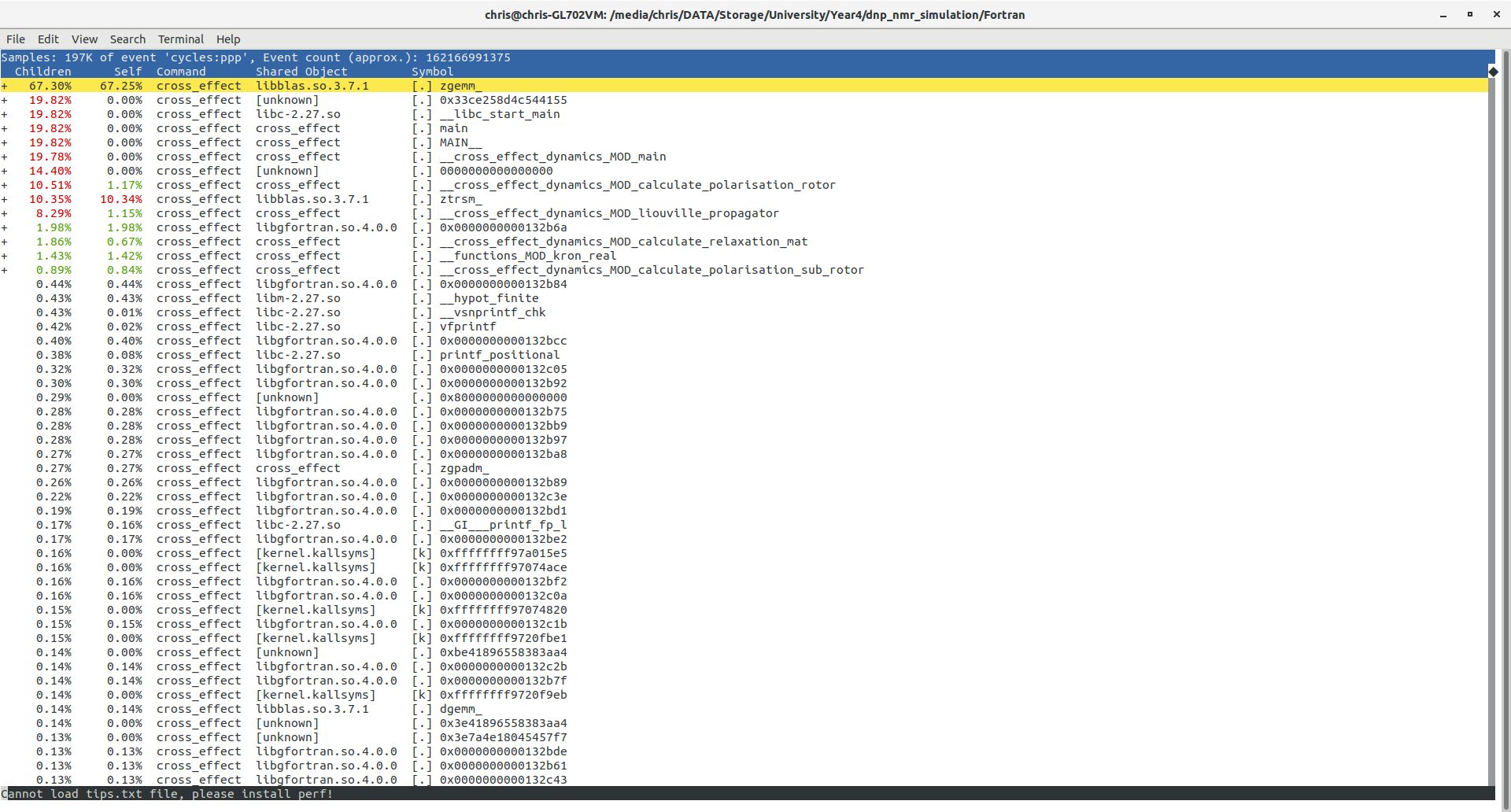
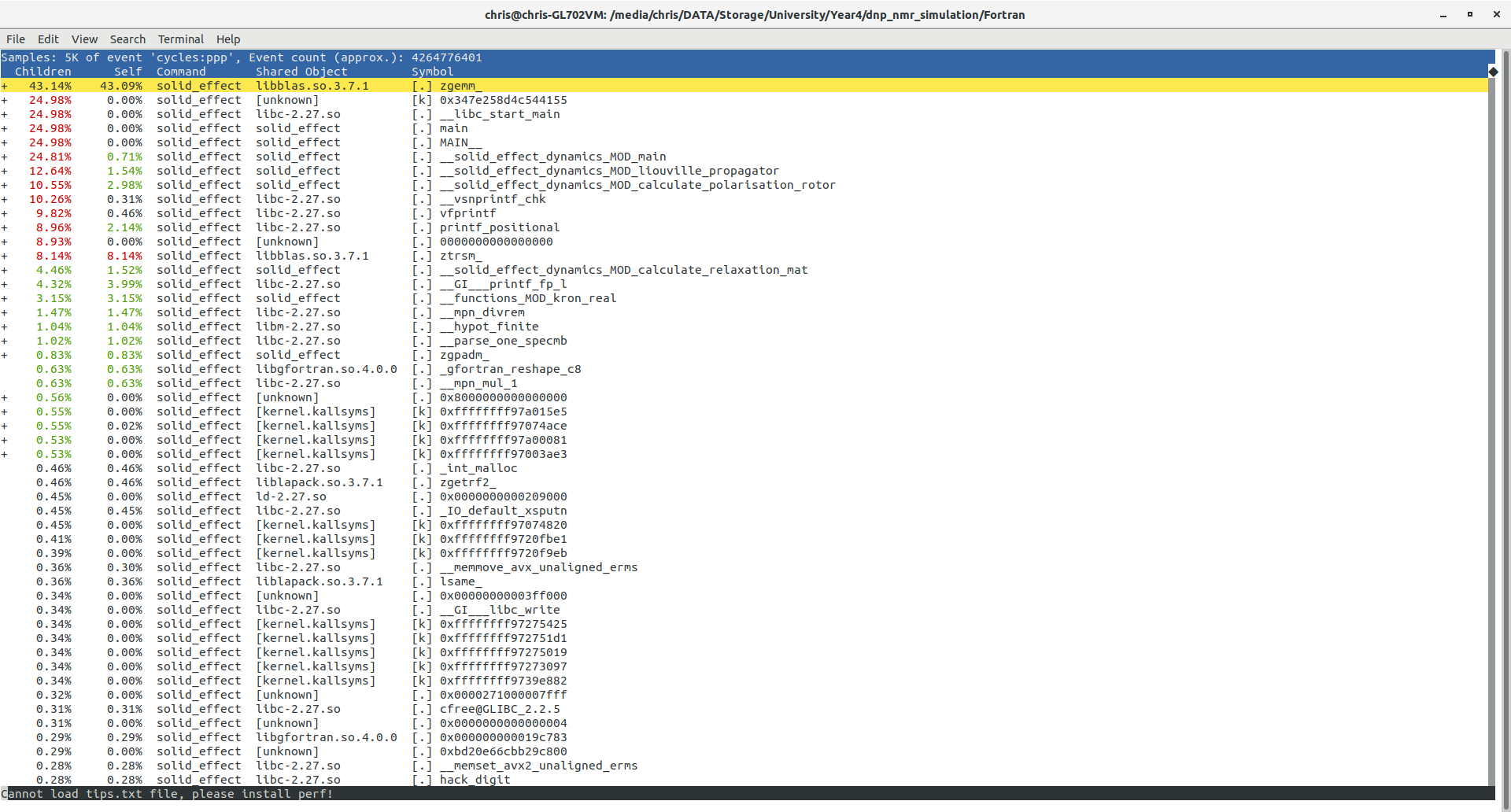
Fortran + OpenMP (s): 0.17

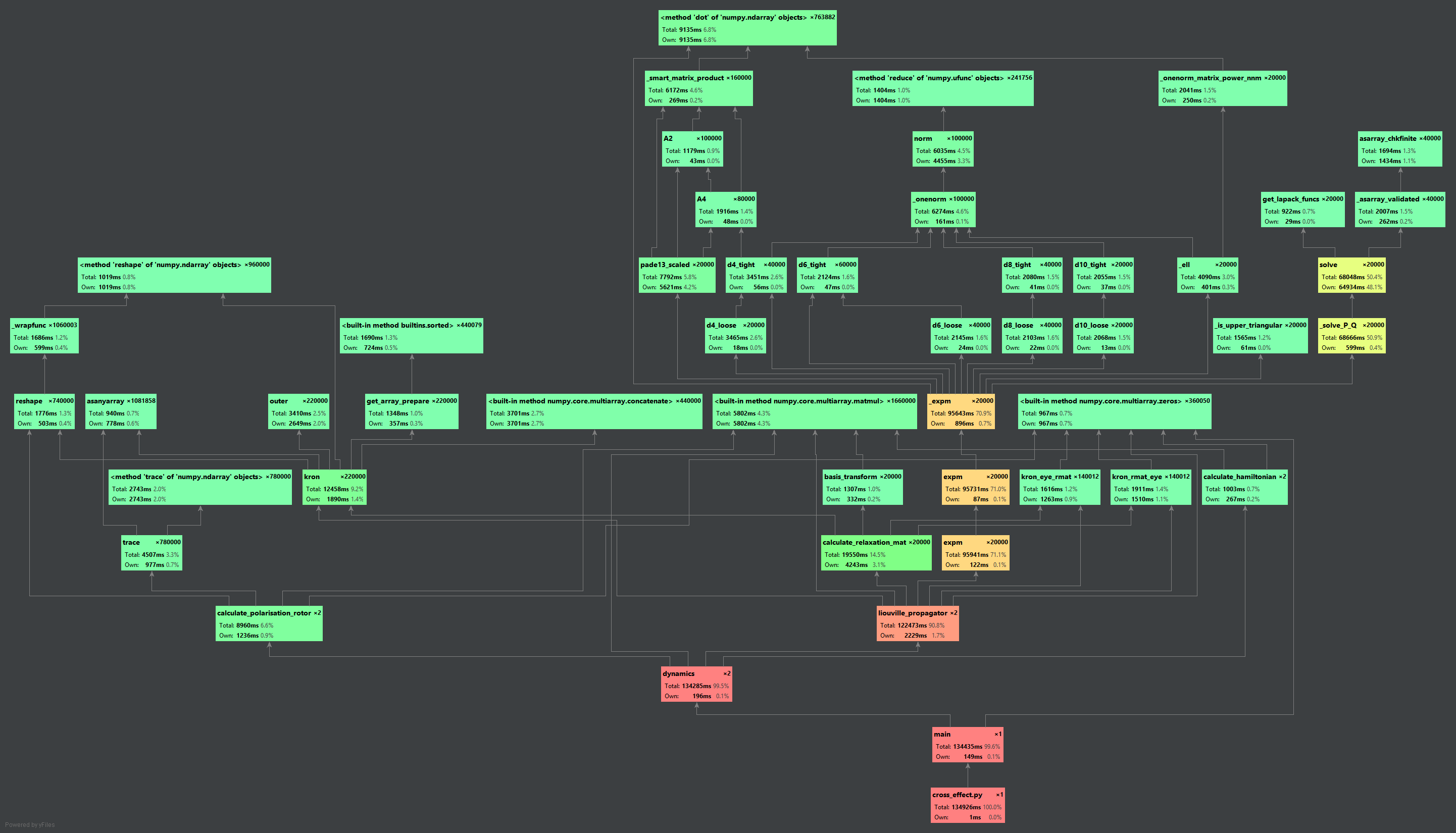
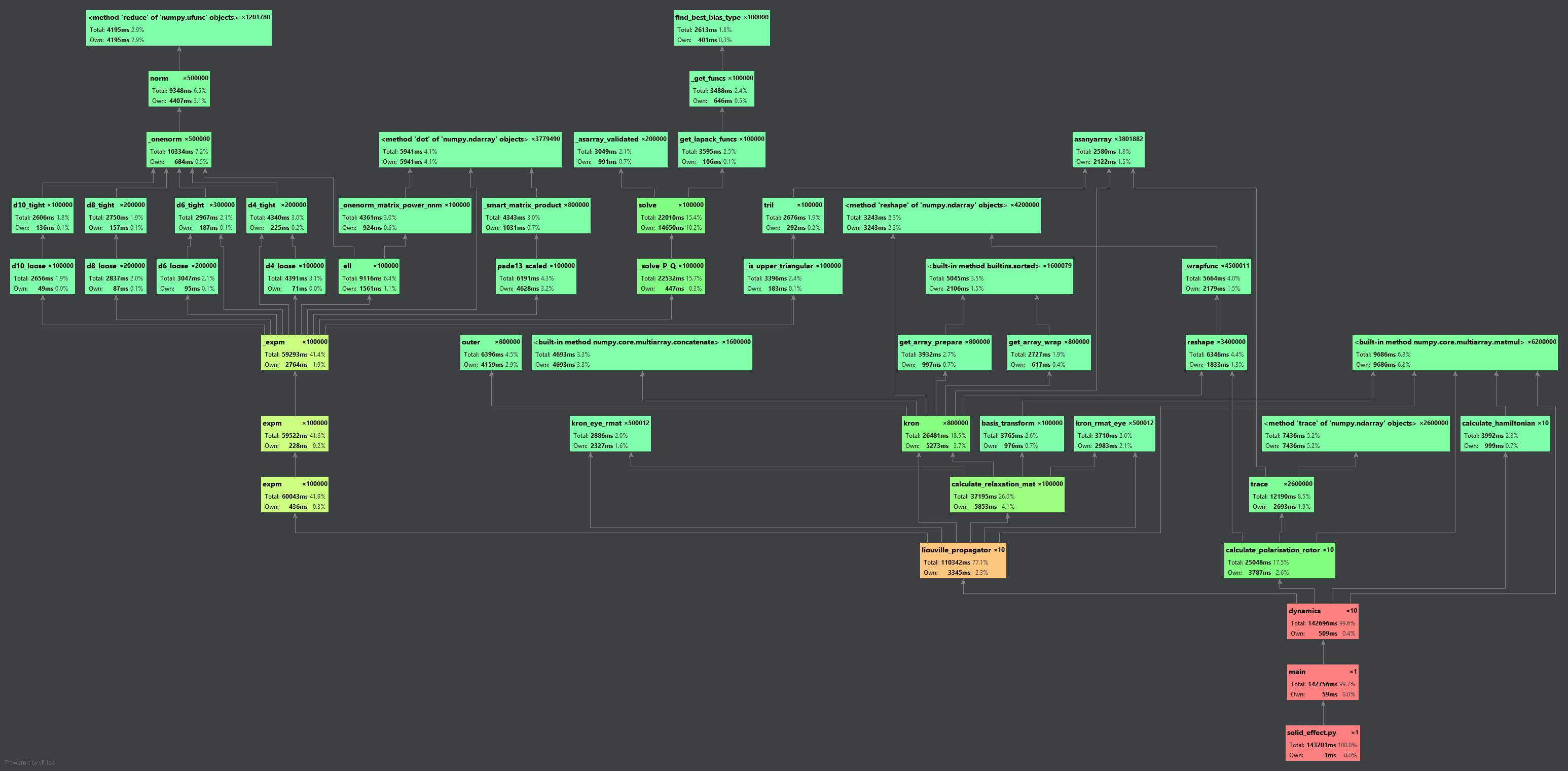
Fortran + BLAS (s): 3.20

Fortran + BLAS + OpenMP (s): 0.75

Fortran + BLAS direct (s): 3.15

Fortran profiles for cross effect and solid effect simulations showing the program bottleneck is in performing the BLAS function zgemm(), complex matrix multiplication as used by the matrix exponential. Therefore, optimising this function call by using a more optimised BLAS implementation is very important.



Python profiles for cross effect and solid effect simulations showing the program bottleneck is in performing the complex matrix exponential, as is the case in the Fortran code also.