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

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

Matlab (s): 9.58

Python + NumPy (s): ~~14.19~~ 12.38

Fortran (s): 1.87

Fortran + OpenMP (s): ~~0.91~~ 0.84

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

Matlab (s): 8.42

Python + NumPy (s): 12.48

Fortran (s): 1.82

Fortran + OpenMP (s): ~~0.86~~ 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

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

Matlab (s): 33.5

Python + NumPy (s): ~~68.4~~ 60.3

Fortran (s): ~~67.4~~ 51.2

Fortran + OpenMP (s): ~~22.2~~ 18.0

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

Python + NumPy (s): 67.8

Fortran (s): 63.9

Fortran + OpenMP (s): 22.3

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

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

Matlab (s): 1.37

Python + SciPy, Enthought (s): 4.32

Fortran + Expokit (s): 0.21

**Calculating 1E3 matrix exponentials 8x8:** *(Cross effect propagator)*

Matlab (s): 1.45

Python + SciPy (s): 4.38

Fortran + Expokit (6 terms) (s): 5.74

Fortran + Expokit (2 terms) (s): 4.17

**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

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

Matlab (s): 0.41

Python + NumPy (s): 0.75

Fortran (s): 3.03

Fortran + OpenMP (s): 0.76

Fortran + BLAS x4 (s): 13.69

**Program performance:**

For a single run Fortran performs around 11x faster than Matlab, and for a parallel run of 40 jobs performs 14x faster. It should be noted that parallel performance of Matlab and Python has not been tested, however as neither language support ‘true’ multithreading the gains are likely to be negligible. The greater performance of Fortran over Matlab and Python is expected as Fortran is a language optimised for High Performance Computing, however it is interesting that Matlab can defeat Fortran in certain areas such as matrix multiplication. This is likely due to the highly optimised LAPACK interface used by Matlab, able to pre-optimise matrix operations. It is likely that this is also responsible for the greater performance of Matlab over Python; as the bulk of the computational cost is in performing matrix operations, the LAPACK implementation is the deciding factor in the overall speed.

**Cross effect:**

Can’t parallelise propagation calculations as these are iterative, therefore the cost will increase. Double precision is required due to the parameters used, and the small but finite values in density matrix which alter the final polarisations. It may be possible to utilise mixed precision algorithms, however I do not have the time to investigate this further.

These benchmarks are also likely misleading due to loop unrolling, it is likely that all languages, in particular Matlab, are able to unroll the for loop and recognise the repeated operation. I have ensured that the loop is not being optimised out of existence altogether, however this may be an import caveat. It is not currently clear why cross effect performs so poorly, continue working on this.

For example, Fortran beats Matlab when real, however when complex is slower. This is very likely due to pre-optimisation, as Matlab can detect that only one matrix inputted is real and so number of operations that must be performed is smaller than if both were complex.

This is a little concerning, why is Fortran so bad here? However this 64x64 matrix multiplication only occurs 1E4 times in propagator function, never again so overall impact is only 2-3s, negligible especially when parallel processing advantage of Fortran is considered.

It seems my previous BLAS benchmarks were wrong (issue with real/complex casting), the BLAS matrix product is slower for small matrices as expected (according to literature is generally faster only consistently for very large matrices).

Attempting to create a custom BLAS function for complex inputs works, however as it requires performing four matrix products it is very slow in comparison to intrinsic function.

The BLAS wrapper adds a small overhead, only significant for small matrices and so for purposes of generalization the wrapper will be used.

Kronecker product. Sadly, while the custom Kronecker function makes a significant different to Python, it is slower for the Fortran code. This is likely because multiplication in Fortran is very fast, while creating and re-shaping matrices is slow. However, this at least shows that the Kronecker product is not a cause for concern in the Fortran program as Fortran is able to beat both Matlab and Python.

**Matrix exponential.** The matrix exponential does not perform significantly faster in Fortran, investigate whether I can perform single precision matrix exponential. Also work on trying to remove reshape function as this may consume a lot of time also. Look into this more after matrix multiplication. Expokit only supports double precision matrix exponentials, so using single precision is not an option.

**Matrix multiplication.**

**Relaxation:**

It appears 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 expense 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. In addition, the Mance relaxation method requires ordering of eigenvalues and eigenvectors (as performed automatically by ‘Eigenshuffle’ in Matlab), however this adds a negligible cost of around 1ms for solid effect and 9ms for cross effect. Therefore, the Mance relaxation method can be recommended for use without any reservations.

**Further work:**

* Look into stability of Fortran code (appears to be a few bugs remaining)
* Look into very low performance of Fortran cross effect code
* Consider BLAS matrix multiplication
* ~~Consider re-designing Kronecker product function, optimising for use with identity matrices~~
* Consider reducing precision of Fortran calculations
* Look into further Fortran optimisations
* Look into changing number of timesteps



