**Program benchmarks (laptop)**

**Solid effect (1 run):**

Matlab (s): 9.93

Python + NumPy (s), Anaconda: 31.06

Python + NumPy (s), Enthought: 29.22

Python + NumPy (s), Intel: 29.64

Fortran (s): ~~4.18~~ ~~2.84~~ ~~2.67~~ ~~2.38~~ ~~2.34~~ ~~2.12~~ ~~1.97~~ 1.68

Fortran + OpenMP (s): 0.88 (11x faster than Matlab)

**Solid effect (40 runs):**

Matlab (s):

Python + NumPy (s), Enthought:

Fortran (s):

Fortran + OpenMP series (s):

Fortran + OpenMP parallel (s):

*\*Currently there is a bug(?) associated with access violation when running multithreaded Kronecker product function, which may be limiting performance. A fix is to apply OpenMP statements to the function, however this needs to be investigated further.*

*\* Currently there is only a partial F2PY implementation, limited to Hamiltonian and propagator calculation.*

The significantly higher performance of Matlab for all tested matrix operations is puzzling, as both Python and Matlab are linked to LAPACK Fortran library for matrix multiplication. The Fortran benchmark supports this to a degree however it appears Matlab is somehow significantly more optimised.

In conclusion Matlab is significantly faster than pure Python, however when combined with F2PY Python may perform up to an order of magnitude faster than Matlab. More work needs to be done to ensure Python speeds are representative, and to investigate other Fortran compilers and matrix multiplications.

An immediate improvement to Fortran speed would be to define all matrices as real, as without yz hyperfine solid effect matrices are all real. Also, worth considering whether it is possible to reduce precision of Fortran code. This would also improve Python speed significantly.

Fortran utilises double precision, to work with F2PY kind=8 is used instead of real64 however with gfortran double precision matches kind=8 so this is not an issue. However, with other compilers this may cause issues. All variables, constants and Numpy arrays are specified to double precision.

It seems Mance relaxation requires ordered eigenvalues/eigenvectors. This adds slightly to computational time.

Eigenvectors differ only in sign, which should be fine as eigenvectors are not unique

Mance relaxation has smaller t1 matrix, lower change in one rotor but greater over many

Matlab column major, Python row major (as in C), Fortran column major

(/ omega1, omega2, omega3, omega4 /) ! f90 notation  
[ omega1, omega2, omega3, omega4 ] ! f2003 alternative

**General benchmarks (laptop)**

**Calculating 1E7 matrix products (2x2 real):**

Matlab (s): 4.99

Python + NumPy (s): 9.69

Fortran (s): 6.97

Fortran + BLAS (s): 0.26

**Calculating 1E6 matrix products (8x8 real):**

Matlab (s): 0.73

Python + NumPy (s): 1.07

Fortran (s): 0.96

Fortran + BLAS (s): 0.35

**Calculating 1E6 matrix products (8x8 complex):**

Matlab (s): 1.33

Python + NumPy, Anaconda (s): 2.12

Python + NumPy, Enthought (s): 2.11

Python + Numpy, Intel (s): 2.15

Fortran (s): 1.90

As expected when inputs are both small real matrix operation is significantly faster, while when complex or larger are slower. The same pattern between functions and languages is present, showing that function dgemm() provided by BLAS constantly outperforms the intrinsic function matmul(). However, dgemm() only works on real matrices so its applications are limited. There appears to be no difference between the performance of three tested Python interpreters, as all are linked to Intel MKL implementation of LAPACK. The performance of other Fortran compilers such as ifort has not been examined, however gfortran is known to perform similarly despite being open source. Matlab is likely capable of the most significant pre-optimisation, hence it outperforms all other languages except for the dgemm() function. Further research is required to confirm these relationships and provide further justification. Check Fortran benchmarks on a Linux installation to ensure Cygwin/MingGW are not causing decreased performance.

**Calculating 1E5 Kronecker products:**

Matlab (s): 2.32

Python + NumPy, Anaconda (s): 4.45

Python + NumPy, Enthought (s): 4.37

Python + NumPy (s), Intel: 4.49

Fortran (s): 1.65

**Calculating 1E4 matrix exponentials:**

Matlab (s): 1.37

Python + SciPy, Anaconda (s): 4.35

Python + SciPy, Enthought (s): 4.32

Python + SciPy, Intel (s): 4.23

Fortran + Expokit (s): 0.21

**Links:**

[**https://software.intel.com/en-us/forums/intel-fortran-compiler-for-linux-and-mac-os-x/topic/269726**](https://software.intel.com/en-us/forums/intel-fortran-compiler-for-linux-and-mac-os-x/topic/269726)

[**https://stackoverflow.com/questions/31494176/will-fortrans-matmul-make-use-of-mkl-if-i-include-the-library**](https://stackoverflow.com/questions/31494176/will-fortrans-matmul-make-use-of-mkl-if-i-include-the-library)

[**https://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html**](https://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html)

[**https://modelingguru.nasa.gov/docs/DOC-1762**](https://modelingguru.nasa.gov/docs/DOC-1762)

[**https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Optimize-Options.html**](https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Optimize-Options.html)

[**https://groups.google.com/forum/#!topic/comp.lang.fortran/HLqObOUIAZc**](https://groups.google.com/forum/#!topic/comp.lang.fortran/HLqObOUIAZc)

[**https://www.ics.uci.edu/~paolo/FastMM/FMM-Reference/reference.html**](https://www.ics.uci.edu/~paolo/FastMM/FMM-Reference/reference.html)

[**https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2017-getting-started**](https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2017-getting-started)