1

Computing Performance

This section evaluates the computing performance of the code. Our goal is to show that the new algorithm of angular convolution is faster than the old naive one, and the huge amount of simulation has proven that to be the case. But a raw result, where the implementation goes for an indefinite number of iterations during minimization, cannot give a proper and systematic performance evaluation. This is the purpose of this section.

Only the $\mathcal{F}_{\text{exc}}$ term

As discussed in appendix **??**, two main factors influence the performance of a sequential code: the algorithm complexity, and the memory delay. To study the algorithm complexity, testing with respect to parameters is done on some simple but important components. The result can match the theoretical algorithm complexity, or be completely different due to the overhead of function calling or the inhomogeneity of memory access. Study of these small parts permits a further understanding of the entire code.

1.1 FFT

The **FFT!** play an important role in the implementation, which is used by the spatial convolution and the **FGSHT!** process.

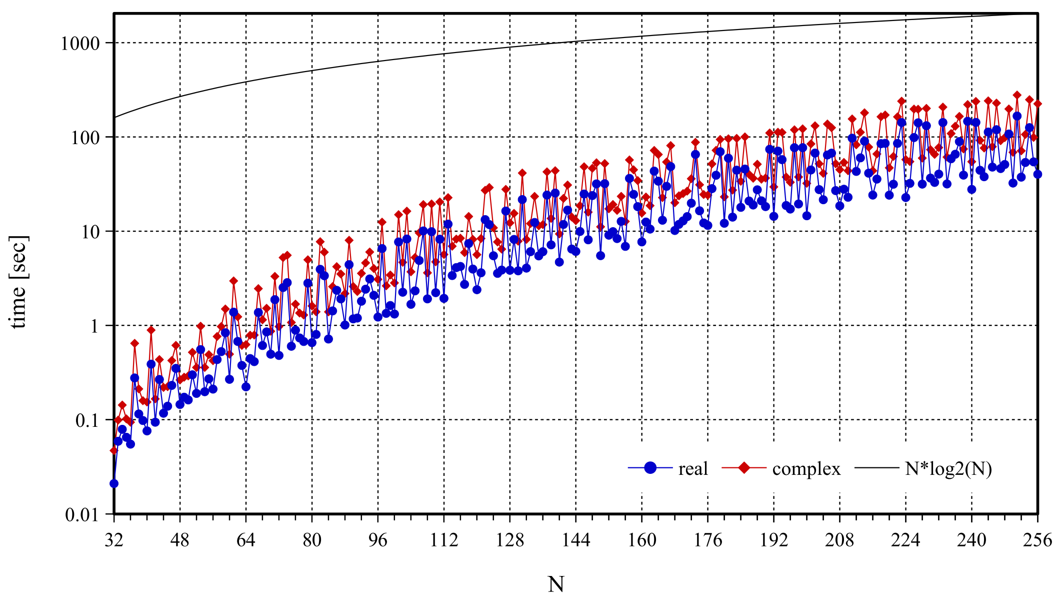

Figure 1.1: timing FFT of real numbers and of complex numbers

Referring to figure 1.1, the dependance on $O(N \log_2 N)$ [Numerical_Recipes_3ed] doesn't totally exist, but of the same form, depending on the algorithm of **FFT!** [ref dft]. It should be noted that a grid of prime number is always at the peaks in the figure, which means it can be 2 or more times longer than that of the composite number around. Therefore we should use an even number grid, where the k-border correction in §**??** is absolutely needed. Apart from this conclusion, to compare between the algorithms for angular part involved in this thesis, we are not really interested in computing performance with respect to the number of spatial grid. However, the ratio of real and complex **FFT!** timing is important, illustrated in figure 1.2, which is near the theoretical ratio 0.5. For example, we process n_{angle} real to complex **FFT!**, then $n_{\text{spatial}}/2$ complex to complex **FGSHT!**. Or we process n_{spatial} real to complex **FGSHT!**, then $n_{\text{proj}}/2$ complex to complex **FFT!**. This should not give a great difference if $n_{\text{angle}} \sim n_{\text{proj}}$ for small $n_{\max}$. If that is not the case, it will have an influence on the choice of algorithm.

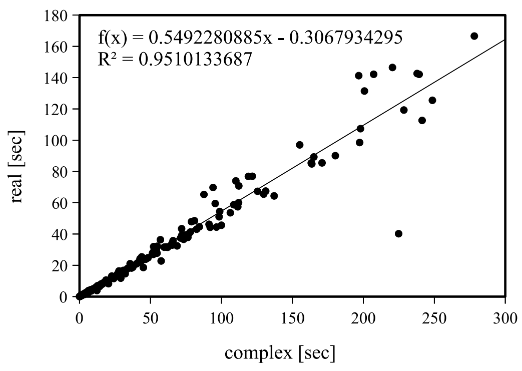

Figure 1.2: FFT real to complex (legendre pas claire)

1.2 FGSHT

The computing times of **GSHT!** and **FGSHT!** are shown in figure 1.3. There is no reason to see in detail how much **FFT!** has accelerated the **GSHT!** process, but clearly **FGSHT!** can be 100 times faster than **GSHT!**, and **GSHT!** for the symmetry of Ψ, $s = 1$ is on average 5 times longer than $s = 2$. As accuracy test shows that **GSHT!** and **FGSHT!** give exactly the same result, therefore it is possible to utilize **FGSHT!** in all the cases to have a faster performance.

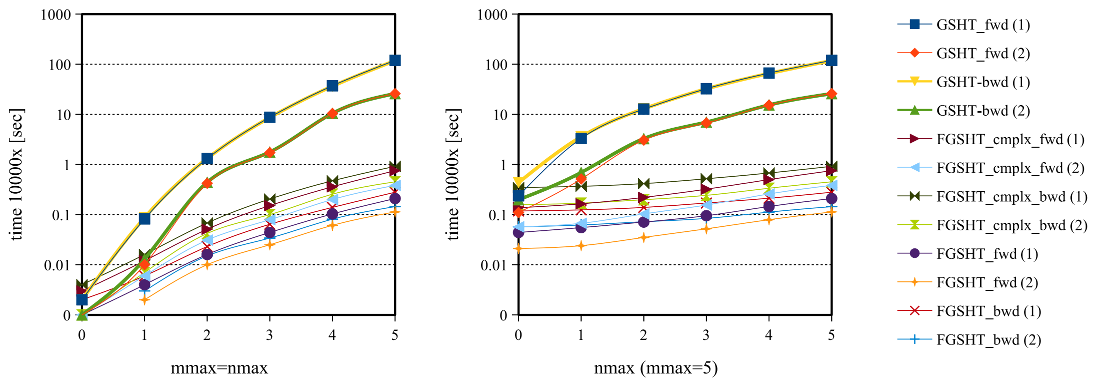

Figure 1.3: Computing time of **GSHT!** and **FGSHT!** (per 10000 times), between parentheses is the order of symmetry axes s

However, it is important to know the ratio between real and complex **FGSHT!** processes for the same reason as **FFT!**. It is demonstrated that this number is 0.3 in all cases, and it does not depend on $n_{\max}$. The difference between these two is that the real one performs real-to-complex **FFT!** for the Φ, Ψ grid and calculates only slightly more than half of projections ($\mu \geq 0$) than the complex one. Normally, the ratio should be greater than 0.5. This could mean there may be an extra process in the complex one, or it is controlled by the memory. Ultimately, the final result 0.3 means, that doing n_{spatial} real to complex **FGSHT!** takes only 0.6 the time of doing $n_{\text{spatial}}/2$ complex to complex **FGSHT!**, which means in `convolution_standard` we use less time to compute **FGSHT!** than in `convolution_pure_angular`.

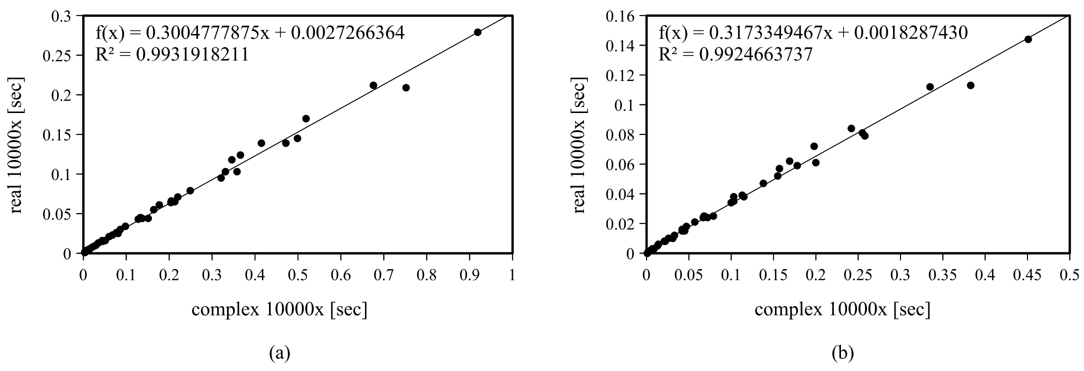

Figure 1.4: FGSHT real to complex (legendre pas claire)

1.3 k-KERNEL

As discussed in the previous section, the final result of energy and structure is independent of the choice of path inside a k-kernel. That means we are free in terms of precision cost to choose the fast path. As path (1) and (2) in figure **??** introduce the transform from $\Delta\hat{\rho}^{m}_{\mu'\mu}(\mathbf{k})$ to $\Delta\hat{\rho}(\mathbf{k}, \boldsymbol{\Omega})$ which has no interest in timing, and the entire set of branches will be compared in later implementations, here we only compare the paths (3) and (4), which correspond to eq. (**??**) and (**??**).

The theoretical predictions of the computing time of **OZ!** equation with respect to $n_{\max}$ are listed in table **??**. If the **OZ!** equation is the most time-consuming part, the result should have the same proposal. Figure (xx) shows the experimental timing of the whole path (3) and (4).

It is shown that... (There is a problem of code that gives backtrace but obviously (4) is 100 times faster than (3).)

1.4 ENTIRE ITERATION OF $\mathcal{F}_{\text{exc}}$ EVALUATION

Apart from all the `naive` methods that will be discussed in §1.4.1, figure 1.5 shows all the comparable `convolution` timing data. We can see `convolution_standard` is the fastest algorithm, and **OZ!** equation is not the longest part in the iteration. All the tests are performed for a $L = 24$, nfft $= 72$ grid with 4 series: the three `convolution` methods with$m_{\max} = n_{\max}$, and `convolution_standard` with $m_{\max} = 5$, varying $n_{\max}$.

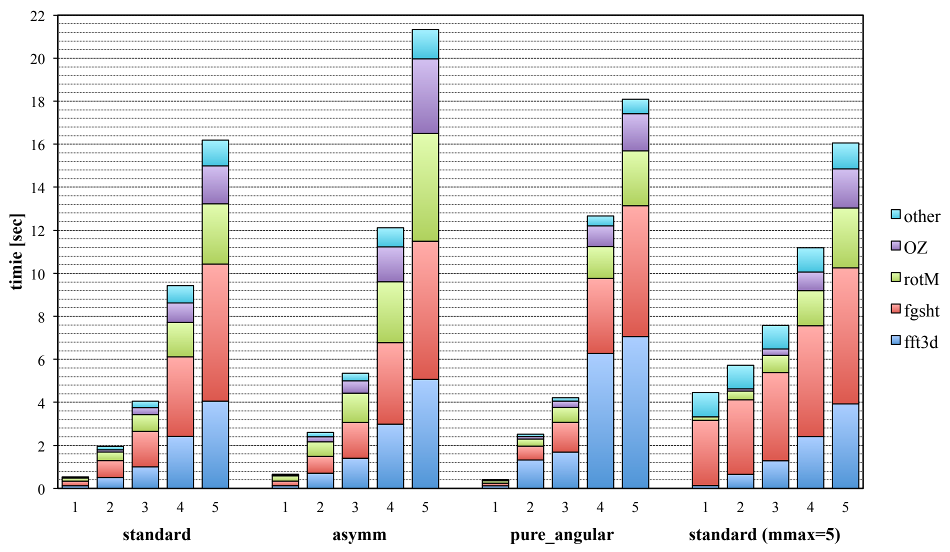

Figure 1.5: Entire iteration of $\mathcal{F}_{\text{exc}}$ evaluation: timing overall/ decomposition of timing for 1 itertation evaluation fexc

1.4.1 *"naive" methods and "convolution_pure_angular"*

The **`naive_standard`**, **`naive_interpolation`**, and **`convolution_pure_angular`** methods share the same processes out of the k-kernel. Table 1.1 shows the timing of loop k of these three methods. It indicates that **`convolution_pure_angular`** takes far less time than the other two methods, of which the loop k takes time in the same order of magnitude as the rest of iteration. And once $m_{\text{max}} \geq 2$, **`naive_interpolation`** is faster than **`naive_standard`**. Note that order 2 of **`naive_interpolation`** can give good results for a **DCF!** of $n_{\text{max}} = 5$. So in every case of **`naive`** methods, **`naive_interpolation`** should be used. This verifies the conclusion of k-kernel test in that the path (4) in figure **??** is the fastest.

m_{max}	**`naive_standard`**	**`naive_interpolation`**	**`convo_pure_angular`**	OTHER
1	2.34	4.42	0.26	0.15
2	365.95	209.12	1.09	1.43
3	3295.00	752.70	2.37	1.85
4	too long	too long		
5	too long	too long		

Table 1.1: Timing [sec] of loop k of "naive_standard", "naive_interpolation" and "convolution_pure_angular", and the rest of iteration

1.4.2 *"convolution_standard" and "convolution_pure_angular"*

The comparison of **`convolution_standard`** and **`convolution_pure_angular`** appears in figure 1.6. Their difference lies in the inversion of **FFT!** and **FGSHT!**. We can see the other parts are almost identical, but the implementation of **FFT!** is different in terms of time. Because in **`convolution_standard`** the number of **FE!** we need for **FFT!** is the number of projections, and in **`convolution_pure_angular`** it is the number of angular grid nodes. As there are fewer projections than angular nodes, **`convolution_standard`** reasonably takes less time.

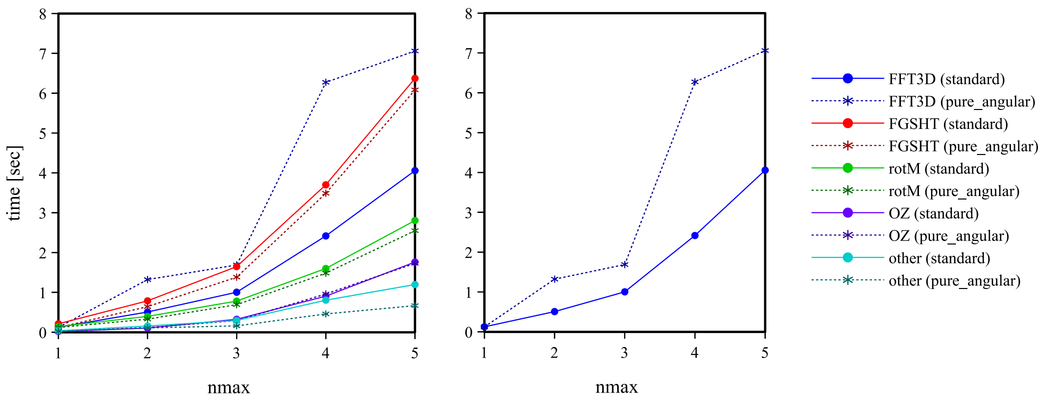

Figure 1.6: comparison of **`convolution_standard`** and **`convolution_pure_angular`**

1.4.3 *"convolution_standard" and "convolution_asymm"*

We compare **`convolution_standard`** and **`convolution_asymm`** in figure 1.7. The difference is that **`standard`** calculates a half k in the $k$loop and **`asymm`** calculates all k in the $k$loop. They share the same process of **FGSHT!**; for the processes in a k loop (rotM, OZ) **`asymm`** always takes longer time. As in **`asymm`** we calculate the **FFT!** for all the projections and in **`standard`** we calculate only a half projections with $\mu \geq 0$, the time consumed by **FFT!** is also different.

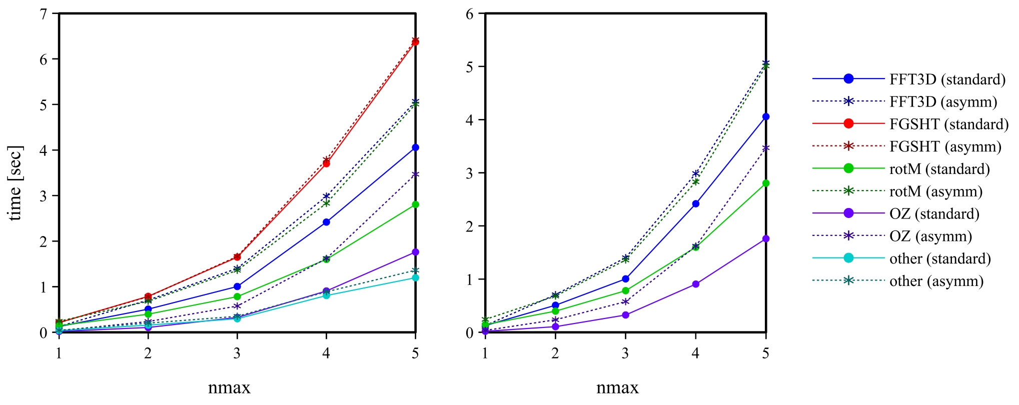

Figure 1.7: comparison of **`convolution_standard`** and **`convolution_asymm`**

1.4.4 $m_{\max}$ *and* $n_{\max}$

The comparison of $m_{\max} = n_{\max}$ and $m_{\max} = 5$ for **`convolution_standard`** is shown in figure 1.8. We see that the choice of quadrature order $m_{\max}$ only affects the **FGSHT!** process and the lecture/storage of density variable (other).

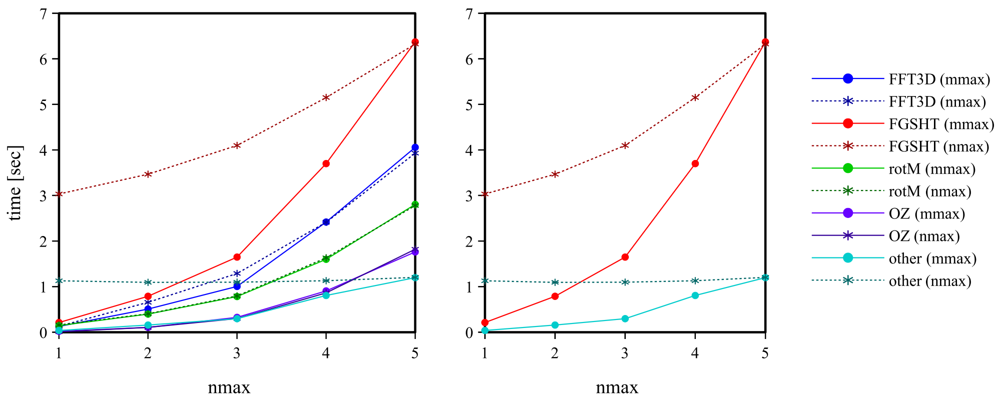

Figure 1.8: comparison of **`convolution_standard`** for $m_{\max} = n_{\max}$ and $m_{\max} = 5$

1.5 GLOBAL VIEW OF THE SEQUENTIAL CODE PERFORMANCE

We can see that **`convolution_standard`** is the fastest. The **`convolution`** methods are orders of magnitude faster than **`naive`** methods.

implementation of a method very fast of fexc evaluation by take advetatge of properties of convolution spatial and angular

we can now use nmax5 insdead of nmax1 in previous method in a resonable timing