CPSセミナー2021年7月22日

球面調和関数変換ライブラリの高速化について

石岡 圭一 (京大理・地惑)

E-mail: ishioka@gfd-dennou.org

はじめに

地球流体の研究のために球面のスペクトル法をずっと使い続けてきた関係で, それに必要な球面調和関数変換(SHT = Spherical Harmonic Transform) のための数値ライブラリ(ISPACK)の開発を細々と趣味的に続けてきた(ispack-0.0のリリースは1997/04/15).

このセミナーでは, 球面のスペクトル法の基礎的な部分から導入し, SHTライブラリの高速化のためにどのようなノウハウが必要なのかを解説する.

また, 最近, ISPACKのバージョン 3.1.0 をリリースしたので, そこに含めた A64FX向けチューニングの話も含める.

球面スペクトル法を用いた計算例:

回転球面上の２次元非圧縮流体の渦度方程式

$$\frac{\partial \zeta}{\partial t} + \frac{\partial \psi}{\partial \lambda}\frac{\partial \zeta}{\partial \mu} - \frac{\partial \psi}{\partial \mu}\frac{\partial \zeta}{\partial \lambda} + 2\Omega\frac{\partial \psi}{\partial \lambda} = D[\zeta]$$

$\zeta \equiv \nabla^2\psi$: 渦度, ψ: 流線関数,

Ω: 球の回転角速度,

λ: 経度, $\mu \equiv \sin\theta$, θ: 緯度, t: 時刻,

$D[\zeta]$: 高階粘性項.

球は単位時間あたり$\Omega/(2\pi)$回転する.

以下の数値計算の水平解像度は T682 (2048×1024 グリッド).

時間発展の例 1 (非回転の場合)

$$(\Omega = 0)$$

(クリックして下さい．アニメーションのURLに飛びます)

https://www.gfd-dennou.org/arch/ishioka/chikyu-no-butsuri/anim2000.mp4 (3.5MB)

渦度場 & 帯状平均帯状角運動量

赤: 正の渦度, 青: 負の渦度

時間発展の例 2 (高速回転の場合)

$$(\Omega = 2000)$$

(クリックして下さい．アニメーションのURLに飛びます)

https://www.gfd-dennou.org/arch/ishioka/chikyu-no-butsuri/anim2000.mp4 (5.7MB)

渦度場 & 帯状平均帯状角運動量

赤: 正の渦度, 青: 負の渦度

球面のスペクトルの応用例: 無限領域のスペクトル法

原点付近の運動が波動を励起し，それが遠方に伝播していくような問題は数多い．そのような問題を有限な領域で数値計算すると，境界からの反射波の影響で原点付近の解が汚染される．

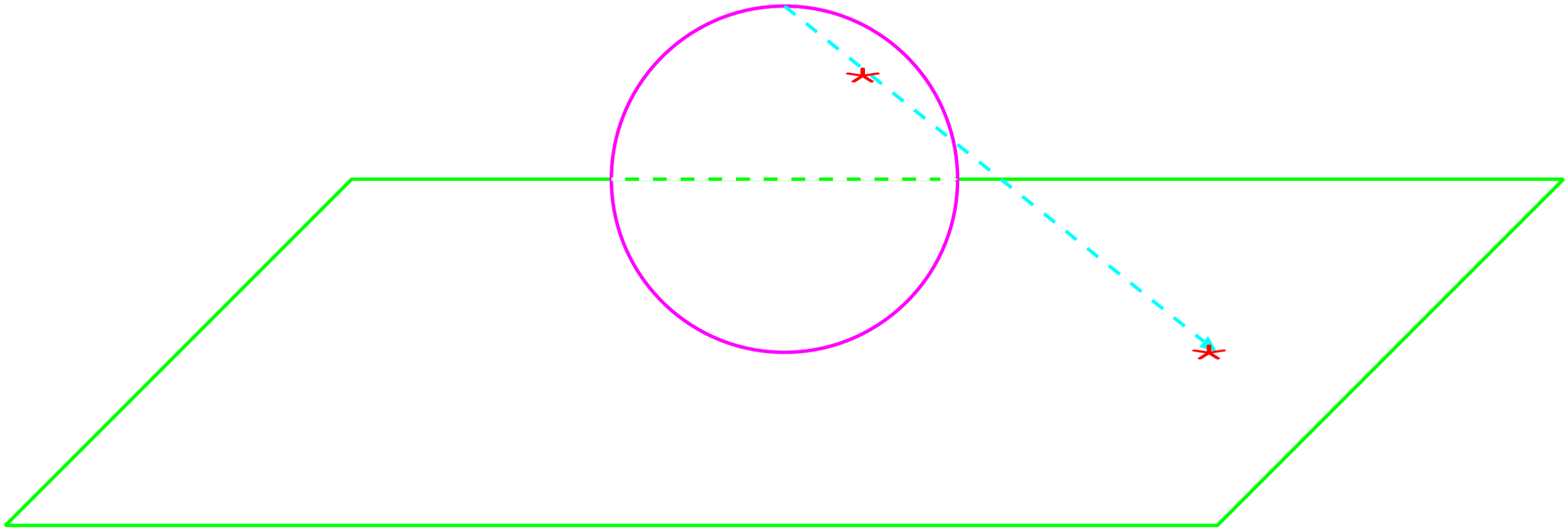
そこで，通常は境界条件の設定を工夫したり (NRBC = Non-Reflecting Boundary Condition)，波動を吸収するスポンジ層を導入したりして反射波を抑える．

分散性のある波動の場合，NRBCだけでは反射波を完全に抑えることはできず，またスポンジ層についてはその領域の広さや散逸の掛け方の設定が難しい．

そこで，最初から領域が無限であることを考慮した計算ができないか?

→ 無限領域のスペクトル法

アイディア: 立体射影により，無限平面 ($\mathbb{R}^2$) を球面に射影し，球面のスペクトル法を用いる.



計算例:

β平面上の線形化された準地衡渦度方程式

$$\frac{\partial \xi}{\partial t} + \beta \frac{\partial \psi}{\partial x} = 0.$$

ψ: 流線関数, ξ: ポテンシャル渦度,

$$\xi(x, y, t) = (\triangle - \gamma^2)\psi(x, y, t); \quad \triangle = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}.$$

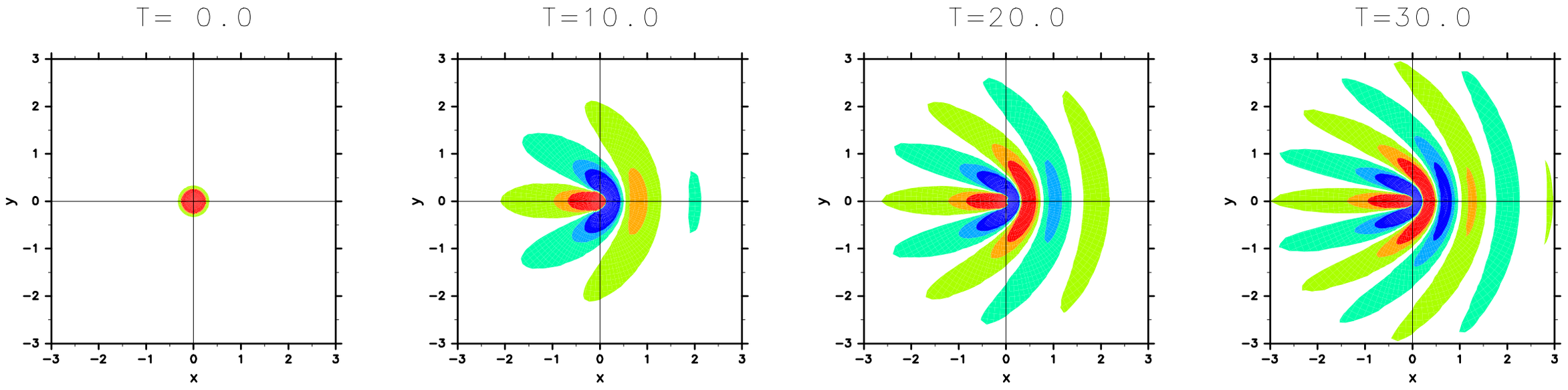
γ: ロスビー変形半径の逆数, x: 東向き座標, y: 北向き座標, t: 時刻.

初期条件:

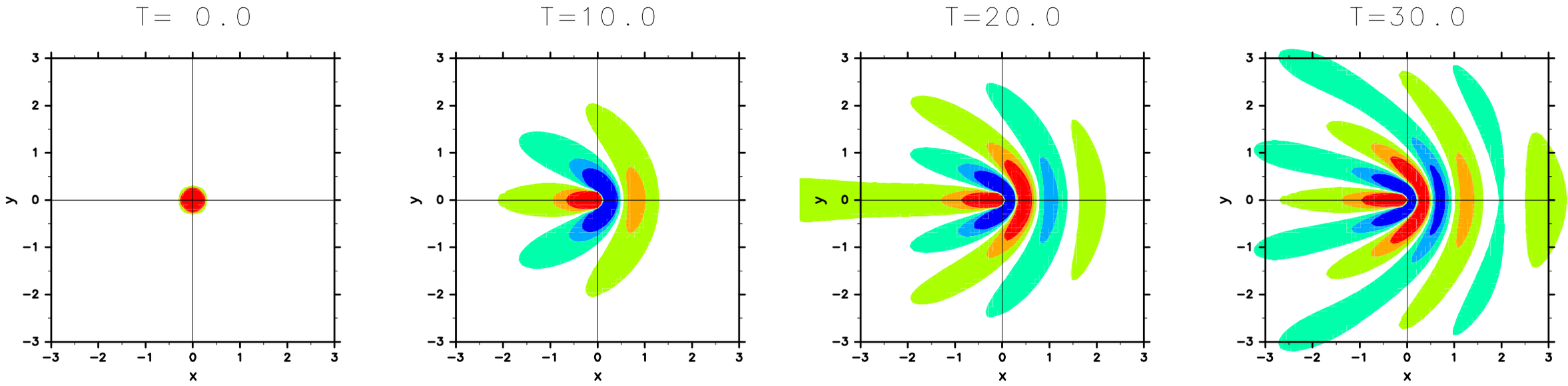
$$\xi = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right).$$

ポテンシャル渦度場の時間発展(Rossby波の伝播を表す)

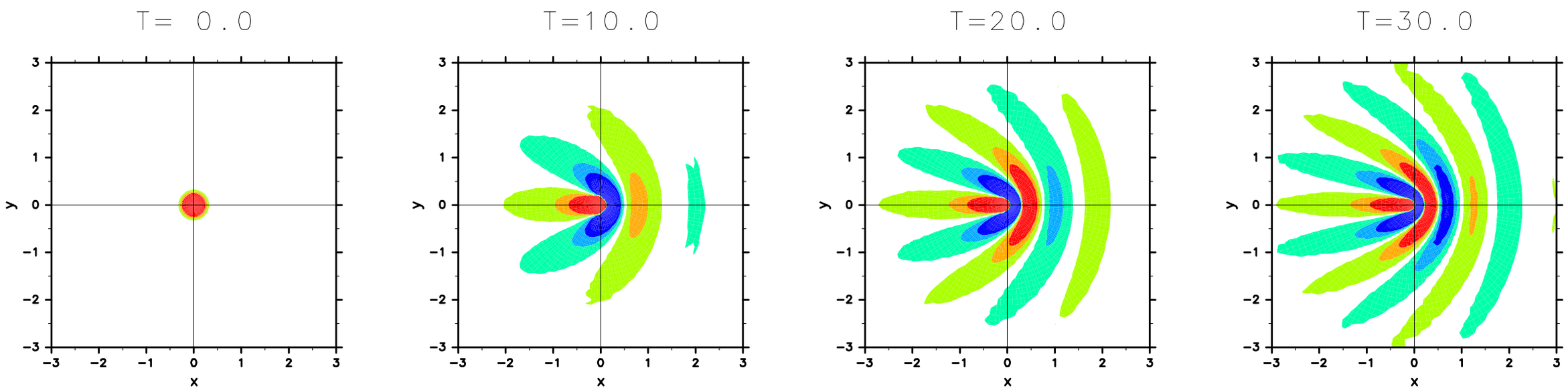
厳密解($\beta = 1, \gamma = 0, \sigma = 0.2$):



周期境界条件のフーリエスペクトル法による数値解(領域: $[4\pi \times 4\pi]$):



無限領域のスペクトル法による数値解(射影に使う球半径＝２):



球面のスペクトル法のざっくりとしたレビュー

従属変数を球面調和関数で展開.

$$f(\lambda,\mu,t)=\sum_{n=0}^{M}\sum_{m=-n}^{n}a_n^m(t)Y_n^m(\lambda,\mu).$$

λ: 経度, $\mu=\sin\theta$, θ: 緯度, t: 時刻,

M: 切断波数.

$$Y_n^m(\lambda,\mu)=P_n^{|m|}(\mu)e^{im\lambda}.$$

$P_n^m(\mu)$: Legendre 陪関数.

$$P_n^m(\mu)\equiv\sqrt{(2n+1)\frac{(n-m)!}{(n+m)!}}\frac{1}{2^n n!}$$
$$\times(1-\mu^2)^{m/2}\frac{d^{n+m}}{d\mu^{n+m}}(\mu^2-1)^n\quad(0\le m\le n).$$

Legendre陪関数による展開

球面調和関数展開では，以下のLegendre陪関数による展開に最も計算コストがかかる．

逆変換:

$$g_j^m = \sum_{n=m}^{M} s_n^m P_n^m(\mu_j) \quad (m = 0, \cdots, M;\ j = 1, \cdots, J).$$

正変換:

$$s_n^m = \sum_{j=1}^{J} w_j g_j^m P_n^m(\mu_j) \quad (m = 0, \cdots, M;\ n = m, \cdots, M).$$

J: ガウス緯度の個数, μ_j: ガウス緯度, w_j: ガウス重み.

必要な計算量

Legendre陪関数の $m \geq 1$ の成分に対しては実部・虚部を扱わねばならないことと，ルジャンドル陪関数の対称性から添字 j に関する計算は半分の $J/2$ でよいことを考慮すると，逆変換・正変換に必要な計算量は N はルジャンドル陪関数をすべて事前に計算しておくとしても (乗算と加算が必要なことを考慮して)

$$N = \frac{J}{2} \cdot (M+1)^2 \cdot 2 = J(M+1)^2.$$

さらに，正変換時の数値計算で誤差が出ないことを保証するためには $J > \frac{3}{2}M$ としておく必要があるので，

$$N \sim \frac{3}{2}M^3.$$

M が大のときは，この変換計算をいかに効率化するかが全体の計算スピードを決める．

x64 な CPU 上 (というか, ベクトル機でない普通の CPU) での最適化

今日, x64 な CPU が広く普及しているので, コードをそれに合わせてチューンしておくことは有意義.

x64 な CPU はメモリバンド幅が狭いので, キャッシュの活用が重要.

しかし, Legendre陪関数変換を素朴に実装すると, 行列×ベクトル型の演算になるので, キャッシュに載りにくい.

→ 解決策: Legendre陪関数を変換と同時に計算する.

ただし，Legendre陪関数を計算する計算量は変換そのものの計算量と同じオーダー．なので，Legendre陪関数の計算コストを下げられる工夫が可能ならやるべき．

Legendre陪関数は，普通，以下の漸化式で計算される．

$$P^m_{n+1}(\mu) = (\mu P^m_n(\mu) - \epsilon^m_n P^m_{n-1}(\mu))/\epsilon^m_{n+1}.$$

$$\epsilon^m_n = \sqrt{(n^2 - m^2)/(4n^2 - 1)}.$$

この漸化式を１回計算するには，ϵ^m_nおよび$1/\epsilon^m_n$を事前に計算しておくとすれば，乗算３回と減算１回．

変換の式および漸化式を工夫することによって，漸化式部分のコストを削減した．積和算命令にも馴染むものにした(１段あたり実質積和算１回)．

元々の漸化式

$$\mu P_n^m(\mu) = \epsilon_{n+1}^m P_{n+1}^m(\mu) + \epsilon_n^m P_{n-1}^m(\mu)$$

を2段階組み合わせると,

$$\mu^2 P_n^m(\mu) = \epsilon_{n+1}^m \epsilon_{n+2}^m P_{n+2}^m(\mu) + \{(\epsilon_{n+1}^m)^2 + (\epsilon_n^m)^2\} P_n^m(\mu) + \epsilon_n^m \epsilon_{n-1}^m P_{n-2}^m$$

となる．赤道反対称のLegendre陪関数について考えて,

$n = m + 2l + 1$ $(l = 0, 1, \ldots)$ とすると，上の漸化式は,

$$\begin{aligned}\mu^2 P_{m+2l+1}^m(\mu) =& \epsilon_{m+2l+2}^m \epsilon_{m+2l+3}^m P_{m+2l+3}^m(\mu) \\ &+ \{(\epsilon_{m+2l+2}^m)^2 + (\epsilon_{m+2l+1}^m)^2\} P_{m+2l+1}^m(\mu) \\ &+ \epsilon_{m+2l+1}^m \epsilon_{m+2l}^m P_{m+2l-1}^m(\mu),\end{aligned}$$

と表せる．ここで,

$$P_{m+2l+1}^m(\mu) = \mu \alpha_l^m p_l^m(\mu),$$

とp_l^mを導入すれば,

$$\begin{aligned}\mu^2\alpha_l^m p_l^m(\mu) =& \epsilon_{m+2l+2}^m\epsilon_{m+2l+3}^m\alpha_{l+1}^m p_{l+1}^m(\mu)\\ &+\{(\epsilon_{m+2l+2}^m)^2+(\epsilon_{m+2l+1}^m)^2\}\alpha_l^m p_l^m(\mu)\\ &+\epsilon_{m+2l+1}^m\epsilon_{m+2l}^m\alpha_{l-1}^m p_{l-1}^m(\mu),\end{aligned}$$

となる．この漸化式を簡単にするために，α_l^mに対して

$$\epsilon_{m+2l+2}^m\epsilon_{m+2l+3}^m\alpha_{l+1}^m = -\epsilon_{m+2l+1}^m\epsilon_{m+2l}^m\alpha_{l-1}^m,$$

を満すことを課す．この式の両辺にα_l^mを掛けて変形すると,

$$\alpha_{l+1}^m\alpha_l^m = -\frac{\epsilon_{m+2l+1}^m\epsilon_{m+2l}^m}{\epsilon_{m+2l+3}^m\epsilon_{m+2l+2}^m}\alpha_l^m\alpha_{l-1}^m,$$

を得る．この式を繰り返し適用すると,

$$\alpha_{l+1}^m\alpha_l^m = (-1)^l\frac{\epsilon_{m+3}^m\epsilon_{m+2}^m}{\epsilon_{m+2l+3}^m\epsilon_{m+2l+2}^m}\alpha_1^m\alpha_0^m,$$

を得る． 従って，あとはα_0^mとα_1^mを定めさえすれば，順次，α_l^m $(l = 2,3,\ldots)$を定めることができる．

α_0^m と α_1^m の定めかたには任意性があるが，後々の便利のため，以下のように定めることにする．

$$\alpha_0^m = \frac{1}{\epsilon_{m+1}^m},\ \alpha_1^m = \frac{\epsilon_{m+1}^m}{\epsilon_{m+2}^m \epsilon_{m+3}^m}.$$

こう定めると，結局，

$$\alpha_{l+1}^m \alpha_l^m = \frac{(-1)^l}{\epsilon_{m+2l+3}^m \epsilon_{m+2l+2}^m}.$$

となる．先の $p_l^m(\mu)$ に対する漸化式の両辺に $(-1)^l \alpha_l^m$ を掛けて，上の式を用いると，

$$\begin{aligned} & p_{l+1}^m(\mu) \\ & = \left[(-1)^l (\alpha_l^m)^2 \mu^2 - (-1)^l (\alpha_l^m)^2 \left\{ (\epsilon_{m+2l+2}^m)^2 + (\epsilon_{m+2l+1}^m)^2 \right\} \right] p_l^m(\mu) \\ & + p_{l-1}^m(\mu). \end{aligned}$$

となる．

さらに, a_l^m と b_l^m を

$$a_l^m = (-1)^l(\alpha_l^m)^2, \quad b_l^m = -a_l^m\left\{(\epsilon_{m+2l+2}^m)^2 + (\epsilon_{m+2l+1}^m)^2\right\},$$

と導入すれば, 結局, $p_l^m(\mu)$ に対する漸化式

$$p_{l+1}^m(\mu) = (a_l^m\mu^2 + b_l^m)p_l^m(\mu) + p_{l-1}^m(\mu).$$

が得られる. この漸化式においては, a_l^m と b_l^m および μ^2 を前もって計算しておけば, 漸化式一段あたり, 2回の FMA(乗算2回, 加算2回) で計算できる. この漸化式は元の P_n^m で考えれば, n を2つ増す漸化式に相当するので, n の一段あたり, 1回の FMA で済んでいて, 元々の 3回の FMA の必要性に比べて, 計算量が 1/3 になることになる.

ただし，この2段とばしの漸化式を $P_n^m(\mu)$ の偶奇モード両方について計算していては全く意味が無い(むしろ以前のアルゴリズムの方が速いことになる)ので，どちらか一方についてのみ計算することになる．実際には，$\mu = 0$付近の精度を考えて，奇モードのみについて計算する．

隅モードについては，

$$\mu P_n^m(\mu) = \epsilon_{n+1}^m P_{n+1}^m(\mu) + \epsilon_n^m P_{n-1}^m(\mu)$$

を用いて，2つの奇モードの重ね合わせ(の $1/\mu$)として計算できる．

このアルゴリズムを利用することの副産物として，乗算の削減だけでなく，Legendre陪関数に関するロードとストアが半分で済むので，その分のコスト削減にもつながる．実際には，例えば，逆変換は以下のように計算される．

$$g_j^m = \sum_{l=0}^{\left[\frac{M-m}{2}\right]} (t_l^m \alpha_l^m) p_l^m(\mu_j) + \mu_j \sum_{l=0}^{\left[\frac{M-m-1}{2}\right]} (s_{m+2l+1}^m \alpha_l^m) p_l^m(\mu_j).$$

ここに,

$$t_l^m = \begin{cases} s_{m+2l}^m \epsilon_{m+2l+1}^m + s_{m+2l+2}^m \epsilon_{m+2l+2}^m & (l = 0, 1, \ldots, [\frac{M-m}{2}] - 1) \\ s_{m+2l}^m \epsilon_{m+2l+1}^m & (l = [\frac{M-m}{2}]). \end{cases}$$

SIMD命令の活用

x64なCPUでは，演算性能向上のためには SIMD命令(AVX，AVX2, AVX512) の活用が必須(A64FXの話はまた後で)

コンパイラが賢くなってはきているが，少なくとも計算のホットスポットはアセンブリでコーディングするのはかなり有効.

Intel の Haswell 以降では 1サイクルあたり，256ビット長(倍精度 4つ) の積和算が 2組実行できる.
Intel の Skylake-SP 以降では 1サイクルあたり，512ビット長(倍精度 8つ) の積和算が 2組実行できる.

ということで，逆変換/正変換において，jによるループを SIMD長ごとに分割してSIMD命令を有効に使えるようにすればよい.

逆変換のホットスポットのコーディングの例(SIMD長が8の場合)

```
DO I=1,JB
   DO J=1,8
      Q1=Q(J,1,I)
      Q2=Q(J,2,I)
      Q0=Q(J,0,I)
      Q3=Q(J,3,I)+S(1,2)*Q1+S(1,4)*Q2
      Q4=Q(J,4,I)+S(1,1)*Q1+S(1,3)*Q2
      Q5=Q(J,5,I)+S(2,2)*Q1+S(2,4)*Q2
      Q6=Q(J,6,I)+S(2,1)*Q1+S(2,3)*Q2
      Q1=Q1+(A(0)*Q0+A(1))*Q2
      Q2=Q2+(A(2)*Q0+A(3))*Q1
      Q(J,3,I)=Q3+S(1,6)*Q1+S(1,8)*Q2
      Q(J,4,I)=Q4+S(1,5)*Q1+S(1,7)*Q2
      Q(J,5,I)=Q5+S(2,6)*Q1+S(2,8)*Q2
      Q(J,6,I)=Q6+S(2,5)*Q1+S(2,7)*Q2
      Q1=Q1+(A(4)*Q0+A(5))*Q2
      Q(J,1,I)=Q1
      Q(J,2,I)=Q2+(A(6)*Q0+A(7))*Q1
   END DO
END DO
```

逆変換のホットスポットのコーディングの例(AVX512アセンブリ版)

```
L00:
  vmovapd   (%rax),%zmm14
  vmovapd 64(%rax),%zmm12
  vmovapd 128(%rax),%zmm13
  vmovapd %zmm9,%zmm0
  vfmadd231pd %zmm14,%zmm8,%zmm0
  vfmadd213pd %zmm12,%zmm13,%zmm0
  vmovapd %zmm11,%zmm1
  vfmadd231pd %zmm14,%zmm10,%zmm1
  vfmadd213pd %zmm13,%zmm0,%zmm1
  vmovapd %zmm5,%zmm15
  vfmadd231pd %zmm14,%zmm4,%zmm15
  vfmadd213pd %zmm0,%zmm1,%zmm15
  vfmadd213pd %zmm7,%zmm6,%zmm14
  vfmadd213pd %zmm1,%zmm15,%zmm14
  vmovapd %zmm15,64(%rax)
  vmovapd %zmm14,128(%rax)
  vmovapd 192(%rax),%zmm14
  vfmadd231pd %zmm18,%zmm12,%zmm14
  vfmadd231pd %zmm22,%zmm13,%zmm14
  vfmadd231pd %zmm26,%zmm0,%zmm14
  vfmadd231pd %zmm30,%zmm1,%zmm14
  vmovapd %zmm14,192(%rax)
  vmovapd 256(%rax),%zmm14
  vfmadd231pd %zmm16,%zmm12,%zmm14
  vfmadd231pd %zmm20,%zmm13,%zmm14
  vfmadd231pd %zmm24,%zmm0,%zmm14
  vfmadd231pd %zmm28,%zmm1,%zmm14
  vmovapd %zmm14,256(%rax)
  vmovapd 320(%rax),%zmm14
  vfmadd231pd %zmm19,%zmm12,%zmm14
  vfmadd231pd %zmm23,%zmm13,%zmm14
  vfmadd231pd %zmm27,%zmm0,%zmm14
  vfmadd231pd %zmm31,%zmm1,%zmm14
  vmovapd %zmm14,320(%rax)
  vmovapd 384(%rax),%zmm14
  vfmadd231pd %zmm17,%zmm12,%zmm14
  vfmadd231pd %zmm21,%zmm13,%zmm14
  vfmadd231pd %zmm25,%zmm0,%zmm14
  vfmadd231pd %zmm29,%zmm1,%zmm14
  vmovapd %zmm14,384(%rax)
  addq $448,%rax
  cmpq %rax,%r10
  jne L00
```

Skylake-SPでは，1サイクルあたり，積和算2回，ロード2回，ストア1回ができる設計になっている． 上記のループは，反復1回あたり，積和算24回，ロード7回，ストア6回となっているので，ロードとストアがボトルネックになることなく(理論上は) 12サイクルで実行できることになる．

論文:

Ishioka, K. (2018). A New Recurrence Formula for Efficient Computation of Spherical Harmonic Transform. *JMSJ*, **96**, 241–249.

※ Legendre陪関数の漸化式にかかる計算コストを従来のものの 1/3にする技法．この技法は，他の開発者の球面調和関数変換ライブラリにも既に利用されている．cf. libsharp by Martin Reinecke
(https://gitlab.mpcdf.mpg.de/mtr/libsharp/).
彼には，libsharpの中に “This update features significant speedups thanks to important algorithmic discoveries by Keiichi Ishioka (https://www.jstage.jst.go.jp/article/jmsj/96/2/96_2018-019/ and personal communication)”．と謝辞を書いてもらっている．

他の実装との速度比較

比較対象: SHTns

作者: Nathanaël Schaeffer (ISTerre, Université de Grenoble)

URL: http://users.isterre.fr/nschaeff/SHTns/

論文: Schaeffer(2013): Efficient spherical harmonic transforms aimed at pseudospectral numerical simulations, *Geochemistry, Geophysics, Geosystems*, vol. 14, pp.751–758.

開発履歴: Changelog を見ると, 2010年に v1.0 がリリースされていたようだ. 現在は v3.4.6 (2021年5月15日リリース)

※実装の方向性は私のものに非常に近い(Legendre陪関数を on-the-fly で計算するとか, SIMD命令の活用とか).

SHTns も, v3.2-r660 以降では, “New recurrence formula of Ishioka (2018) leading to faster transforms, especially for large sizes. see https://doi.org/10.2151/jmsj.2018-019” として私の手法が取り入れられているので, それとの比較も入れる. 以前は, configure 時に –enable-ishioka というスィッチをオプションを付けてから make する必要があったが, 今はそちらがデフォルトになっている. 従って, あえて –disable-ishioka というスィッチを有効にした場合とも比較しておく.

ベンチマーク結果(AVX2)

プラットホーム:

CPU: Broadwell Xeon E5-2699v4 (22コア)×2

コンパイラ: ifort 19.0.0.117 -xHost -qopenmp -align array64byte

(SHTns は, gcc 6.3.0, option -O2 -march=native -ffast-math)

44スレッドの場合のみを示す.

各スペクトルに$(-1,1)$の一様乱数を与え，逆変換 + 正変換を行った誤差評価(SHTns' は, SHTnsを –disable-ishioka で makeした場合).

		$M=1023$	$M=2047$	$M=4095$
$\epsilon_{\max}$	SHTns'	1.2×10^{-10}	5.9×10^{-11}	2.0×10^{-9}
	SHTns	1.2×10^{-12}	4.4×10^{-12}	2.0×10^{-11}
	ISPACK3	5.1×10^{-13}	1.2×10^{-12}	5.8×10^{-12}
ϵ_{rms}	SHTns'	7.7×10^{-13}	2.6×10^{-13}	6.7×10^{-12}
	SHTns	7.4×10^{-14}	1.7×10^{-13}	3.9×10^{-13}
	ISPACK3	4.3×10^{-14}	8.9×10^{-14}	1.9×10^{-13}

		$M=8191$	$M=16383$
$\epsilon_{\max}$	SHTns'	4.9×10^{-10}	2.5×10^{-8}
	SNTns	4.3×10^{-11}	5.9×10^{-11}
	ISPACK3	1.8×10^{-11}	4.1×10^{-11}
ϵ_{rms}	SHTns'	1.5×10^{-12}	4.7×10^{-11}
	SHTns	7.2×10^{-13}	1.2×10^{-12}
	ISPACK3	4.3×10^{-13}	7.9×10^{-13}

スペクトル⇔格子点値　1回あたりの変換にかかる時間(sec)

	$M = 1023$		$M = 2047$	
	bwd	fwd	bwd	fwd
SHTns'	0.0021	0.0022	0.014	0.015
SHTns	0.0015	0.0015	0.0097	0.0098
ISPACK3	0.0017	0.0014	0.011	0.0094

	$M = 4095$		$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd	bwd	fwd
SHTns'	0.11	0.11	0.86	0.86	7.4	7.3
SHTns	0.070	0.071	0.53	0.52	4.5	4.3
ISPACK3	0.075	0.064	0.53	0.45	4.0	3.4

スペクトル⇔格子点値 の変換における実効 GFlops値(このシステムにおける理論ピーク性能は, AVX使用時の CPU周波数を 1.8GHz とすると, $1.8 \times 16 \times 44 = 1267$GFlops. Turboが効いて, 2.6GHzまで上がるとするなら, $2.6 \times 16 \times 44 = 1830$GFlops. MKLのDGEMMで$4000 \times 4000$と$4000 \times 2000$の行列同士の行列積を計算させて計測すると, 1230GFlops).

	$M = 1023$		$M = 2047$	
	bwd	fwd	bwd	fwd
ISPACK3	651	823	813	940

	$M = 4095$		$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd	bwd	fwd
ISPACK3	933	1090	1052	1245	1111	1311

ベンチマーク結果その2(AVX512)

プラットホーム:

CPU: Skylake-W Xeon W-2135 (3.7GHz, 6コア)

コンパイラ: ifort 19.1.3.304 -xHost -qopenmp -align array64byte

(SHTns は, gcc 8.3.0, option -O2 -march=native -ffast-math)

6スレッドの場合のみを示す.

スペクトル⇔格子点値　1回あたりの変換にかかる時間(sec)

	$M = 1023$		$M = 2047$	
	bwd	fwd	bwd	fwd
SHTns	0.0044	0.0044	0.033	0.033
ISPACK3	0.0048	0.0045	0.029	0.027

	$M = 4095$		$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd	bwd	fwd
SHTns	0.20	0.19	1.6	1.5	11.0	10.9
ISPACK3	0.18	0.18	1.3	1.3	9.9	9.6

スペクトル⇔格子点値 の変換における実効 GFlops値(このシステムにおける理論ピーク性能は, AVX512使用時の CPU周波数を 3.9GHz と仮定すると, $3.9 \times 32 \times 6 = 749$GFlops. MKLの DGEMMで$8000 \times 8000$の行列同士の行列積を計算させて計測すると, 660GFlops 程度).

	$M = 1023$		$M = 2047$	
	bwd	fwd	bwd	fwd
ISPACK3	235	253	310	326

	$M = 4095$		$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd	bwd	fwd
ISPACK3	383	383	426	429	445	458

ベンチマーク結果その3(AVX512, Thanks to 竹広さん)

プラットホーム(阪大 SQUID):
CPU: Icelake Xeon Platinum 8368 (2.4GHz, 38コア)×2 76スレッドの場合のみを示す.

スペクトル⇔格子点値 の変換における実効 GFlops値(このシステムにおける理論ピーク性能は, AVX512使用時の CPU周波数が仮に 2.4GHz のままととするなら, $2.4 \times 32 \times 76 = 5837$GFlops)

	$M = 4095$		$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd	bwd	fwd
ISPACK3	2590	2493	3186	3181	3785	3569

A64FXでのベンチマーク結果その1(Thanks to 齋藤泉君)

ispack-3.0.1の状態で, 単純な Fortran90コードの状態でA64FX(名大不老)上でとりあえず1ノード(48コア) でベンチマークをとってもらったとき.

	$M = 8191$	
	bwd	fwd
ISPACK-3.0.1	1206	305

不老ではA64FXが 2.2GHzで運用されているので, 理論ピーク性能は, $2.2 \times 32 \times 48 = 3379$GFlops)である筈なので, 逆変換の方で 36%程度は出ているが, 正変換の方は 9%程度と悲惨. ということで少しは何とかしたかった.

A64FXのためのチューニングに向けて

ということで，x86上と同様に，まずA64FX上のアセンブリから勉強を始めて手でチューンすることを考えた．以下，やっていったことを時系列てきに散漫に書く．

A64FX上で使えるアセンブリ(AArch64 + SVE)はとっても書き易い．屋上屋を重ねた Intel x64用のアセンブリ + avx512 とかに比べると，はるかに分かりやすい．また，SVEの movprtfx で実質的に 4オペランド命令が使えるのはとっても楽(Intelにも欲しい..)．例えば，

```
movprfx z4.d, p0/z, z0.d
fmla z4.d, p0/m, z1.d, z2.d
```

のように書けば，実質的に

```
z4 = z0 + z1 * z2
```

のような計算が1命令で実行できる．

ということで, 最初は A64FX上でアセンブリを書き始めたのだが, A64FX はやはり命令体系は ARMアーキテクチャになったとはいえ, ハード的には富士通の石なので, 京(SPARC64VIIIfx)の頃の残念な性質をかなり受け継いでしまっていて, よほど条件が整わないとピーク性能が出ない. 理由は:

・命令のレイテンシが大きすぎる: 積和算のレイテンシが 9サイクル, L1 からの vector load のレイテンシが 11サイクルもあるっていったい何? ちなみに Intel の Skylakeでは, それぞれ, 4サイクルと3サイクル.

・物理レジスタが少ない: A64FXでは, 物理レジスタが 32+96で 128 しかない. ちなみに Intel のSkylakeでは 168

ということで, よほど条件が整わない限り (例えば, DGEMMのように, dependency chainが深くない計算を同時並列的にできるような場合以外), レイテンシの隠蔽がちゃんとできない.

ということで，A64FX上でアセンブリを書く場合には，命令のレイテンシや物理レジスタの枯渇まで気にして命令の順番を考える必要があり，これは A64FXの癖を知りつくしていないとできないので，結構，人間業を超えてしまう．

ところが，A64FX上の Fortranコンパイラ，frtpx「は」かなり賢く，元になるFortranのプログラムが SIMD化しやすく書いてあれば，かなり最適なアセンブリコードを吐いてくれる．それを眺めてみると，「あー，そこで loadを先読みするのね」とか，「あー，そこでこの積和算やっとく方がレイテンシの関係で得なのね」とまあ解釈はできるが，なかなか人間業では思いつかないような順番で命令を発行している．

ということで，自分でA64FX用のアセンブリを書くのは放棄して，正変換のコア部分を frtpxが SIMD化しやすいように，frtpxの気持になって書き換えてみた．

チューニングのために書き換えたところの例(正変換の元の形)

```
DO I=1,JB
  DO J=1,8
    Q0=Q(J,0,I)
    Q1=Q(J,1,I)
    Q2=Q(J,2,I)
    Q3=Q(J,3,I)
    Q4=Q(J,4,I)
    S(1)=S(1)+Q4*Q1
    S(2)=S(2)+Q3*Q1
    S(3)=S(3)+Q4*Q2
    S(4)=S(4)+Q3*Q2
    Q1=Q1+(AC(0)*Q0+AC(1))*Q2
    Q2=Q2+(AC(2)*Q0+AC(3))*Q1
    S(5)=S(5)+Q4*Q1
    S(6)=S(6)+Q3*Q1
    S(7)=S(7)+Q4*Q2
    S(8)=S(8)+Q3*Q2
    Q1=Q1+(AC(4)*Q0+AC(5))*Q2
    Q(J,1,I)=Q1
    Q(J,2,I)=Q2+(AC(6)*Q0+AC(7))*Q1
  END DO
END DO
```

チューニングのために書き換えたところの例(正変換の新しい形)

```
SV=0
DO I=1,JB
   Q0V=Q(:,0,I)
   Q1V=Q(:,1,I)
   Q2V=Q(:,2,I)
   Q3V=Q(:,3,I)
   Q4V=Q(:,4,I)
   SV(:,1)=SV(:,1)+Q4V*Q1V
   SV(:,2)=SV(:,2)+Q3V*Q1V
   SV(:,3)=SV(:,3)+Q4V*Q2V
   SV(:,4)=SV(:,4)+Q3V*Q2V
   Q1V=Q1V+(A(0)*Q0V+A(1))*Q2V
   Q2V=Q2V+(A(2)*Q0V+A(3))*Q1V
   SV(:,5)=SV(:,5)+Q4V*Q1V
   SV(:,6)=SV(:,6)+Q3V*Q1V
   SV(:,7)=SV(:,7)+Q4V*Q2V
   SV(:,8)=SV(:,8)+Q3V*Q2V
   Q1V=Q1V+(A(4)*Q0V+A(5))*Q2V
   Q(:,1,I)=Q1V
   Q(:,2,I)=Q2V+(A(6)*Q0V+A(7))*Q1V
END DO
DO K=1,8
   S(K)=S(K)+SUM(SV(:,K))
END DO
```

A64FXでのベンチマーク結果その2

ispack-3.1.0でA64FX(名大不老)上でとりあえず1ノード(48コア) でベンチマークをとった.

	$M = 8191$		$M = 16383$	
	bwd	fwd	bwd	fwd
ISPACK-3.1.0	1205	1294	1316	1413

ということで, 今回のチューニングによって, 正変換のスピードが逆変換並みになった. 正変換の方が若干速いのは, x86上と同様. $M = 16383$ のときの正変換で, ピークの 42%程度の実効性能は出ていることになる.

ギネスに挑戦?

私の知る限り, これまでに行われた自由度最大の球面調和関数変換は,

Reinecke, M., & Seljebotn, D. S. (2013).
Libsharp–spherical harmonic transforms revisited..
Astronomy & Astrophysics, **554**, A112.

に記述されている TL262143 ($= 2^{18} - 1$)のケースだと思われる(ここでは, Intel E5-2670 (Sandy Bridge, 8コア, 2.6GHz)を4096コア分使って計算が行われている)

ギネスに挑戦ではないが, 富岳を使ってこの記録を破ることを試みた (Thanks to 竹広さん).

なお, ISPACK-3.1.0 では, 球面調和関数変換をMPI化したものも実装してある(sypack)のでそれを用いる.

MPI実装は素朴なもので, Legendre陪関数変換は経度方向波数mの方向に分割して並列化しており, 経度方向のFFTを実行する際は, 各緯度円毎に並列化する. 両者をつなぐ際, 転置的な処理が必要になるが, そこは単純にMPI_ALLTOALLを1回呼ぶだけの単純な実装(特に通信時間の隠蔽とかには凝っていない). また, MPI+OpenMP のハイブリッド並列化をしている.

A64FXでのベンチマーク結果その3(Thanks to 竹広さん)

ispack-3.1.0でA64FX(富岳)上で世界タイ記録およびその倍の切断波数でのベンチマークをとった(なお, 富岳では 2.0GHzのモードで計算)

TL262143の計算(256ノード使用, 48スレッド/ノード)
bwd: 64.3sec (実効 280TFlops)
fwd: 56.2sec (実効 321TFlops)
$\epsilon_{\max} = 4.71 \times 10^{-9}$, $\epsilon_{\mathrm{rms}} = 2.47 \times 10^{-11}$

TL524287の計算(1024ノード使用, 48スレッド/ノード)
bwd: 123sec (実効 1172TFlops)
fwd: 107sec (実効 1350TFlops)
$\epsilon_{\max} = 1.71 \times 10^{-8}$, $\epsilon_{\mathrm{rms}} = 7.70 \times 10^{-11}$

ということで，TL524287の方の正変換では，1ノードあたり，実効 1.32TFlops でていることになり，これは 2.0GHz駆動の A64FXのピーク性能の 43%程はでていることになる.

また，このような大きな切断波数の計算では，逆変換＋正変換での相対誤差が 10^{-8}のレベルになりさすがに大きくなるが，まあ，単精度よりはましな程度の精度は維持できていると見なせる.

A64FXでのベンチマーク結果その4(Thanks to 竹広さん)

TL262143の計算での不老と富岳の比較(どちらも 2.2GHz駆動モードでの計算, 256ノード)

不老:
bwd: 55.1sec (実効 327TFlops)
fwd: 44.4sec (実効 406TFlops)

富岳:
bwd: 55.9sec (実効 323TFlops)
fwd: 49.1sec (実効 367TFlops)

富岳の方がちょっと遅い．最初はMPI通信に要する時間の差とかがあるのかと思ったが, 不老でも富岳でも, 正変換の場合にMPI通信に要している時間はどちらも3秒程度なので，この差が説明できない． この点については同じバイナリを使ってみるなどいろいろチェックしたがちょっと謎のまま．

まとめとTODO

まとめ

- Legendre陪関数の漸化式の工夫など様々な技法をもりこんで，(少なくとも Intelの CPU上である程度以上のサイズの変換では) 世界最速のSHTライブラリを構築した.
- ここで導入した Legendre陪関数の漸化式の工夫については，他の開発者の実装にも既に取り入れられてきている.
- A64FXのようなCPUでもある程度の性能は出るようにチューニングを行った.

TODO

- 経度方向の回転対称性を仮定した変換の実装
- FFTについて，COS，SIN変換や，2次元，3次元のFFTの実装
- Pythonインターフェースの整備