GFDセミナー2009分科会 2009年9月6日

SJPACKについて

石岡圭一

E-mail: ishioka@gfd-dennou.org

まずは球面のスペクトル法についてのおさらい

偏微分方程式中の従属変数を球面調和関数で展開.

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

ここに, λ: 経度, $\mu=\sin\theta$, θ: 緯度, t: 時刻, M: 切断波数.

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

$P_n^m(\mu)$は Legendre陪関数.

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

ルジャンドル陪関数による展開

球面スペクトル法には球面調和関数変換が必要．そのうち計算量の大部分を占めるのはルジャンドル陪関数に関する以下の変換である:

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

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

ここに，M: 切断波数，J: ガウス緯度の個数，$P_n^m(\mu)$: ルジャンドル陪関数，μ_j: ガウス緯度，w_j: ガウス重み．

必要な計算量

$m \geq 1$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．

ISPACKにおける球面調和関数変換のこれまでの実装

stpack(1995年) 最も素朴な実装

smpack(1998年) ベクトル化を追求

snpack(1999年) ベクトル化を追求しつつ省メモリ化

sjpack(2009年) スカラー計算機での高速化

変換のベクトル化(snpackでの話)

ベクトル計算機の能力を発揮させるためには，ベクトル長(最内側 DO-LOOP の長さ)を十分に長くとる必要がある.

最も素直にプログラミングすると(stpack)，逆変換・正変換の計算はともに最内側 DO-LOOPの DO変数は ガウス緯度の添字jにとるのが自然.

しかし，この方針だと効率面で2つ問題がある:

1. ベクトル長は $\frac{J}{2} = \frac{3}{4}M$ になるが，Mが十分大でなければベクトル長が短くなってVPの能力が十分発揮されない. ($M = 170$でもベクトル長 $= 128$)

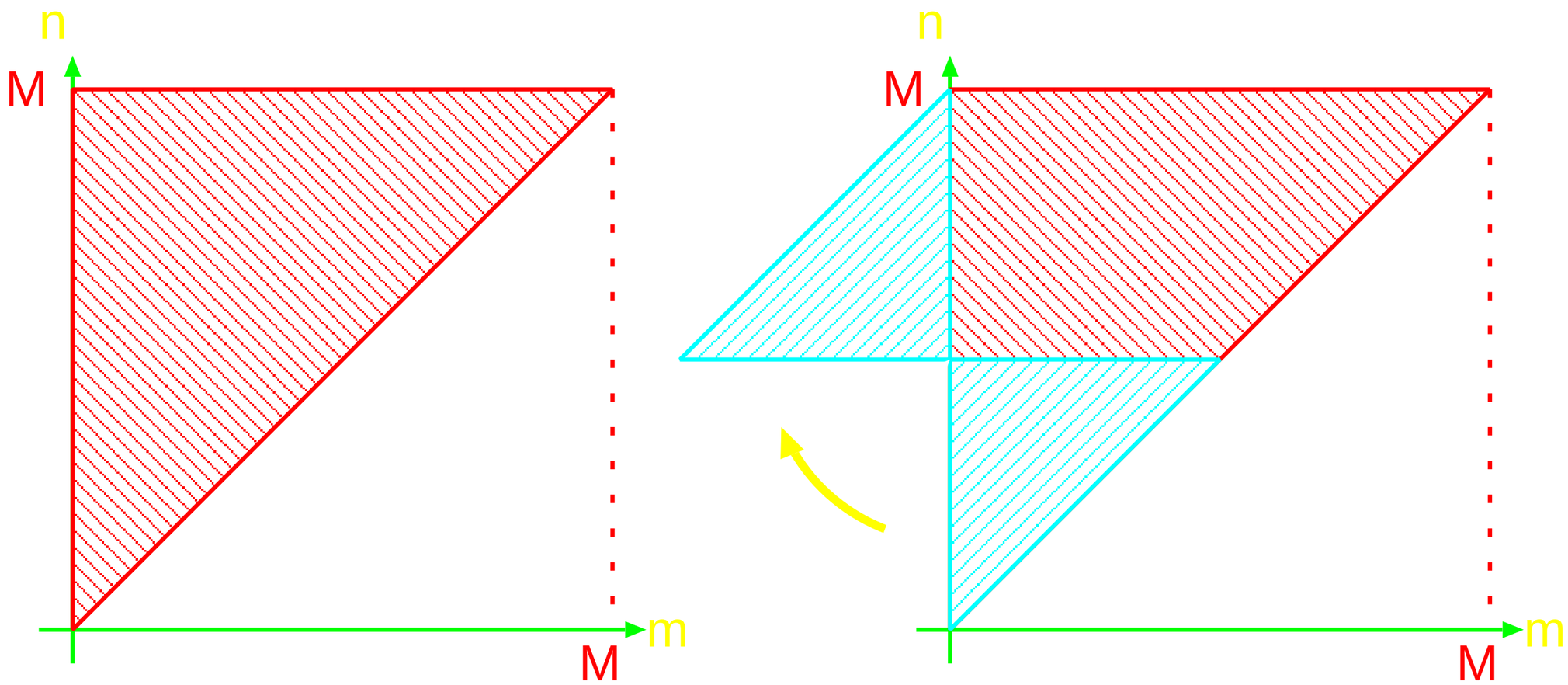
2. DO変数をjにとると，正変換の方の計算が内積型になり，逆変換の方の計算に較べて計算スピードが落ちる(約半分程度).

→ breakthrough: DO変数をmにとる.

そうすると，まず上記2の問題はクリアされる．また，ベクトル長はMとより長くなり，さらに複数種類の変数を同時に変換する場合には，ベクトル長は M·(変数の種類) となり，一気にベクトル長が長くとれるようになる.

ただし，mをDO変数にとるには工夫が必要(そのままのコーディングでは，各nに対するベクトル長はnと可変になってしまう).

図のように三角切断の一部を折り返すことによってmに関するDO文のベクトル長を均一にMにする.



スカラー計算機での高速化に向けて(sjpack)

ここまではベクトル機がまだ元気だった頃の話.

周知のように, 京速計算機プロジェクトからNEC・日立連合が降りたことに象徴されるように, もはやベクトル機には将来性が無い.

ということで, スカラー計算機上での高速化を優先して設計していくべき(といっても, 最近の GPGPU計算においてはベクトル計算機向けに書かれたコードが移植しやすいという話もあるが).

スカラー計算機は, ベクトル計算機に比べてメモリアクセスが遅いので, いかにキャッシュメモリの中で計算を閉じさせるかということが高速化の肝.

といっても，素朴に実装しようとすると，球面調和関数変換はキャッシュが効きにくい．

なぜなら，変換に必要な球面調和関数のデータは倍精度なら

$$8\text{バイト} \times \frac{J}{2} \cdot (M+1)(M+2)/2 = 2J(M+1)(M+2)\text{バイト}.$$

例えば，$M = 170, J = 256$とすると，これは 15MB．最近のXeonのL3キャッシュにすら載らない．

ではどうするか？ $\rightarrow$ 変換する度にLegendre陪関数を毎回計算すればいい．そうすれば，$8\text{バイト} \times J/2 \times 7 = 28J\text{バイト}$の領域があればその中で計算を繰り返すことができる．$J = 256$のときはこれは 7168バイト．これなら Pentium4時代の Xeon でも L1Dキャッシュに収まる．

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．

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

ここに，$\epsilon_n^m = \sqrt{(n^2-m^2)/(4n^2-1)}$．この漸化式を1回計算するには，$\epsilon_n^m$および$1/\epsilon_n^m$を事前に計算しておくとすれば，乗算3回と減算1回．これ以上減らしようが無いようにも見えるが，

$$P_n^m(\mu) = \alpha_n^m p_n^m(\mu)$$

とα_n^m，p_n^mを新に導入し，α_n^mをうまく選ぶことによって，

$$p_{n+1}^m(\mu) = \beta_n^m \mu p_n^m(\mu) + p_{n-1}^m(\mu)$$

とすることができる．こうすれば乗算2回と減算1回となり，乗算を1回削減できる(このテクニックは snpack の段階で既に導入してあったが)．

さらに, Pentium4以降の x86系プロセッサで高速化を目指すにはSSE2 (Streaming SIMD Extension 2) 命令を最大限活用する必要がある. 最近ではコンパイラもそれなりに賢くなってきてはいるが, SSE2命令を活用しようとするならば,アセンブリ言語でのプログラミングは避けて通れない.

ということで, Legendre陪関数変換のコアの部分はアセンブリ言語で書いて最適化しておくべき.

実際コアとなる部分はどんなものか?
例: jlswg-fort.f

```
      SUBROUTINE LJLSWG(JH,S,R,Y,QA,QB,W)

      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION W(JH,2),Y(JH),S(2)
      DIMENSION QA(JH),QB(JH)

      DO J=1,JH
        W(J,1)=W(J,1)+S(1)*QA(J)
        W(J,2)=W(J,2)+S(2)*QA(J)
        QB(J)=QB(J)+R*Y(J)*QA(J)
      END DO

      END
```

と短いサブルーチンなので, これを(SSE2命令等を使って)ガシガシに最適化すればいい(というか, このコアの部分をサブルーチンとして独立させてあるところが設計の肝の一つ).

SSE2命令を使って最適化した例(64ビット版)
例: jlswg-sse64.f

```
.globl ljlswg_
ljlswg_:
        movl    (%rdi), %edi
        movhpd (%rsi), %xmm0
        movlpd (%rsi), %xmm0
        movhpd 8(%rsi), %xmm6
        movlpd 8(%rsi), %xmm6
        movhpd (%rdx), %xmm1
        movlpd (%rdx), %xmm1
        movq  8(%rsp), %r10
        shlq $3,%rdi
        movq %rcx,%rsi
        addq %rdi,%rsi
        movq %r10,%r11
        addq %rdi, %r11
.align 16
.L0:
        movaps (%rcx), %xmm4
        movaps (%r8), %xmm2
        movaps (%r9), %xmm3
        movaps (%r10), %xmm5
```

```
movaps (%r11), %xmm8
movaps %xmm0, %xmm7
mulpd %xmm1,%xmm4
mulpd %xmm2,%xmm4
addpd %xmm4,%xmm3
mulpd %xmm2,%xmm7
addpd %xmm7,%xmm5
mulpd %xmm6,%xmm2
addpd %xmm2,%xmm8
movaps %xmm3,(%r9)
movaps %xmm5,(%r10)
movaps %xmm8,(%r11)
addq $16,%rcx
addq $16,%r8
addq $16,%r9
addq $16,%r10
addq $16,%r11
cmpq %rcx,%rsi
jne .L0
ret
```

ベンチマーク結果

とりあえず T170の普通の設定($J = 256, I = 512$)での snpack との速度比較(なお, sjpack では FFT として FFTJ を使っている).

なお, Flops は実際の計算量(Legendre陪関数の毎回計算を含む)ではなく, Legendre陪関数をもし事前に計算してメモリに格納していたとする場合の変換そのものに絶対必要な計算量で算出している).

Intel Core2 Duo Xeon 3.0GHz (Woodcrest) 64bit mode
(L1: 32KB, L2: 4MB)において
SJPACK: 3.5 GFlops
SNPACK: 1.1 GFlops

Intel Pentium4 Xeon 3.06GHz (Prestonia)
(L1: 8KB, L2: 512KB)において
SJPACK: 1.3 GFlops
SNPACK: 0.17 GFlops

FLTSSとの比較

東大理・情報理工の須田礼仁氏の FLTSS
http://www.na.cse.nagoya-u.ac.jp/~reiji/fltss/
との比較． ただし，FLTSSは FFTまで面倒を見ないので，Legendre
陪関数変換の部分のみで比較(ifort -O3 でコンパイル).

T170の普通の設定($J = 256$)での比較

Intel Core2 Duo Xeon 3.0GHz (Woodcrest) 64bit mode
(L1: 32KB, L2: 4MB)において
LJPACK: 3.1 GFlops
FLTSS: 1.3 GFlops

Intel Pentium4 Xeon 3.06GHz (Prestonia)
(L1: 8KB, L2: 512KB)において
LJPACK: 1.4 GFlops
FLTSS: 0.73 GFlops

T341の普通の設定($J = 512$)での比較

Intel Core2 Duo Xeon 3.0GHz (Woodcrest) 64bit mode

(L1: 32KB, L2: 4MB)において
LJPACK: 3.1 GFlops
FLTSS: 1.6 GFlops

Intel Pentium4 Xeon 3.06GHz (Prestonia)
(L1: 8KB, L2: 512KB)において
LJPACK: 1.2 GFlops
FLTSS: 0.92 GFlops

課題

・OpenMPによる並列化はしてあり，Xeon ではそれなりに並列化効率が出ているが，Opteronでは今一つ．さらに SPARC64VII では悲惨なよう(Thanks to 竹広さん)．→改善の余地あり？

そもそも，SPARC64VII ではアセンブリレベルの最適化の余地もある筈．ただ，私はアカウント持ってないので手が出ないが．

・GPGPUやLarrabeeの動向は？ SSE2な最適化だけで「あと10年は戦える」か？ Intel は将来的には Larrabee もCPUに内蔵して命令セットも AVX というのに整理していくらしい．

ネット上の情報源

- IA32インテルアーキテクチャソフトウェアデベロッパーズマニュアル
(ftp://download.intel.co.jp/jp/developer/jpdoc/)

- Agner Fog氏のページ(x86アセンブリでの高速化についての情報源)
(http://www.agner.org/optimize/)

- IA32 SIMDの扉(x86のSIMD命令についてよくまとまっている)
(http://www.icnet.ne.jp/~nsystem/simd_tobira/)

「ネットは広大だわ……」(by 草薙素子, 1989年)

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．

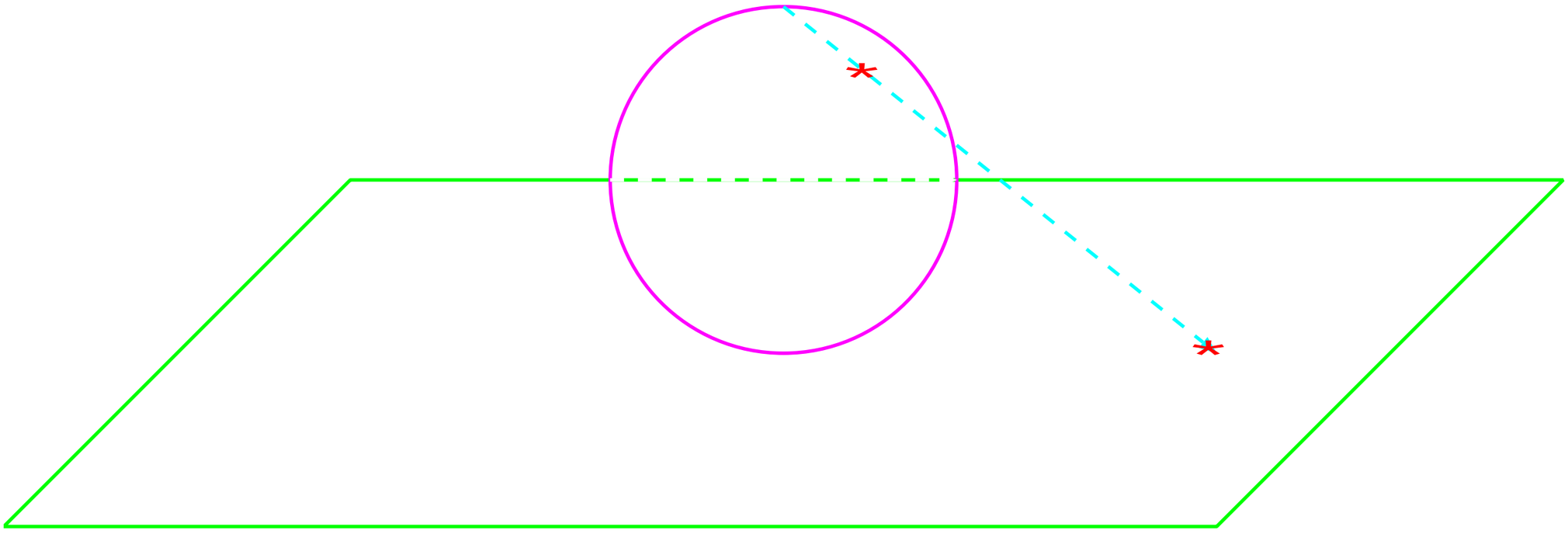
そこで，通常は境界条件の設定を工夫したり (NRBC = Non-Reflecting Boundary Condition)，波動を吸収するスポンジ層を導入したりして反射波を抑える．

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．

なので，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,

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

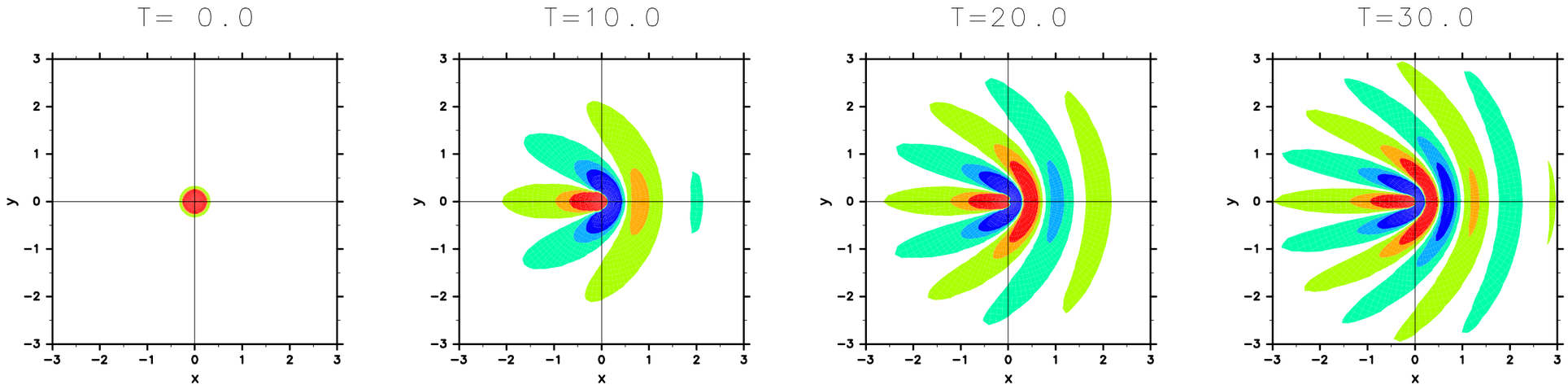
γ: ロスビー変形半径の逆数, x: 東向き座標, y: 北向き座標, t: 時刻.

初期条件:

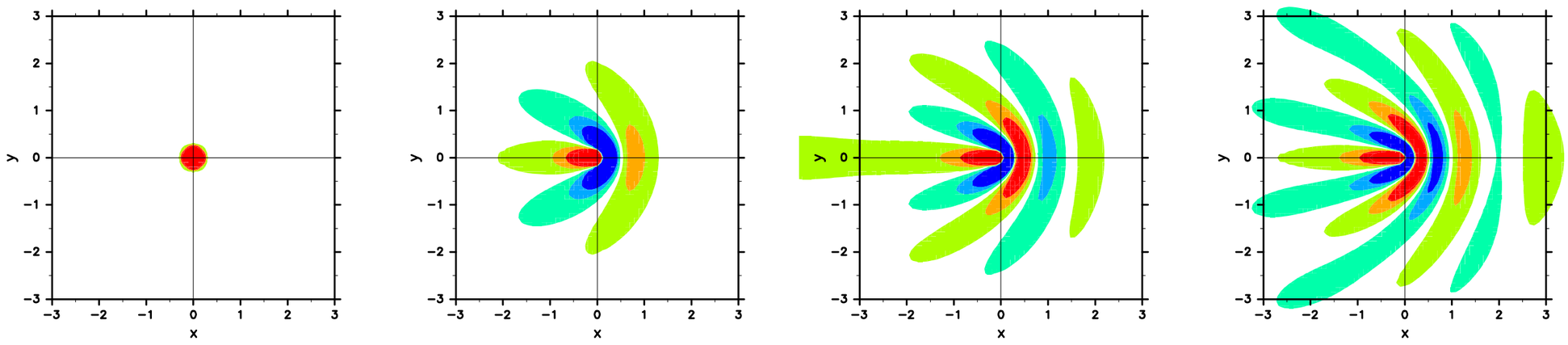
$$\xi = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right).$$

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):