

SP  
1.0

Generated by Doxygen 1.5.9

Sun Jun 13 22:01:04 2010



# Contents

<b>1</b>	<b>SP Documentation</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Usage examples . . . . .	1
<b>2</b>	<b>License</b>	<b>3</b>
2.1	LICENSE . . . . .	4
<b>3</b>	<b>Build</b>	<b>5</b>
3.1	Build . . . . .	6
3.1.1	Windows . . . . .	6
<b>4</b>	<b>Class Index</b>	<b>7</b>
4.1	Class Hierarchy . . . . .	7
<b>5</b>	<b>Class Index</b>	<b>9</b>
5.1	Class List . . . . .	9
<b>6</b>	<b>File Index</b>	<b>11</b>
6.1	File List . . . . .	11
<b>7</b>	<b>Class Documentation</b>	<b>13</b>
7.1	SphereBaroclin Class Reference . . . . .	13
7.1.1	Detailed Description . . . . .	14
7.1.2	Member Function Documentation . . . . .	14
7.1.2.1	L_1_step . . . . .	14
7.1.2.2	L_1_step . . . . .	15
7.1.2.3	L_step . . . . .	15
7.1.2.4	L_step . . . . .	15
7.1.2.5	p2u . . . . .	16
7.1.2.6	S_step . . . . .	16

7.1.2.7	S_step	16
7.1.2.8	u2p	17
7.2	SphereBarvortex Class Reference	18
7.2.1	Detailed Description	18
7.2.2	Member Function Documentation	19
7.2.2.1	L_1_step	19
7.2.2.2	L_step	19
7.2.2.3	LT_step	19
7.2.2.4	p2u	19
7.2.2.5	S_step	20
7.2.2.6	u2p	20
7.3	SphereChafe Class Reference	21
7.3.1	Detailed Description	21
7.4	SphereChafeConf Struct Reference	22
7.4.1	Detailed Description	22
7.5	SphereLaplace Class Reference	23
7.5.1	Detailed Description	23
7.5.2	Member Function Documentation	23
7.5.2.1	calc	23
7.5.2.2	solve	24
7.6	SphereNorm Class Reference	25
7.6.1	Detailed Description	25
7.6.2	Constructor & Destructor Documentation	25
7.6.2.1	SphereNorm	25
7.6.3	Member Function Documentation	26
7.6.3.1	dist	26
7.6.3.2	norm	26
7.6.3.3	scalar	26
7.7	SphereOperator Class Reference	27
7.7.1	Detailed Description	27
7.7.2	Constructor & Destructor Documentation	27
7.7.2.1	SphereOperator	27
7.7.3	Member Function Documentation	28
7.7.3.1	func2coef	28
7.7.3.2	coef2func	28
<b>8</b>	<b>File Documentation</b>	<b>29</b>

---

8.1	operator.h File Reference . . . . .	29
8.1.1	Detailed Description . . . . .	29
<b>9</b>	<b>Example Documentation</b>	<b>31</b>
9.1	test_baroclin.cpp . . . . .	31
9.2	test_barvortex.cpp . . . . .	37
9.3	test_chafe.cpp . . . . .	43
9.4	test_lapl.cpp . . . . .	45



# Chapter 1

## SP Documentation

### 1.1 Introduction

This library is to solve partial differential equations on sphere. It extends SpherePack library and make it easy to use in C++. It also implements some of the common partial differential equations such as Laplace equation, Chafe-Infante equation, Barotropic vorticity equation and two-dimensional baroclinic atmosphere equations.

### 1.2 Usage examples

#### 1. Laplace equation on the sphere

$$\begin{aligned}\Delta\psi &= f(\varphi, \lambda) \\ \Delta\psi &= \frac{1}{\cos\varphi} \frac{\partial}{\partial\varphi} \cos(\varphi) \frac{\partial}{\partial\varphi} \psi + \frac{1}{\cos^2\varphi} \frac{\partial^2}{\partial\lambda^2} \psi\end{aligned}$$

#### 2. Chafe-Infante equation on the sphere

$$\begin{aligned}\frac{du}{dt} &= \mu\Delta u - \sigma u + f(u) \\ u(x, y, t)|_{t=0} &= u_0\end{aligned}$$

#### 3. The Barotropic vorticity equation on the sphere

$$\begin{aligned}\frac{\partial\Delta\psi}{\partial t} + J(\psi, \Delta\psi) + J(\psi, l + h) + \sigma\Delta\psi - \mu\Delta^2\psi &= f(\varphi, \lambda) \\ \psi|_{t=0} &= \psi_0\end{aligned}$$

#### 4. The two-dimensional baroclinic atmosphere equations on the sphere

$$\begin{aligned}\frac{\partial\Delta u_1}{\partial t} + J(u_1, \Delta u_1 + l + h) + J(u_2, \Delta u_2) + \frac{\sigma}{2}\Delta(u_1 - u_2) - \mu\Delta^2 u_1 &= f(\phi, \lambda) \\ \frac{\partial\Delta u_2}{\partial t} + J(u_1, \Delta u_2) + J(u_2, \Delta u_1 + l + h) + \frac{\sigma}{2}\Delta(u_1 + u_2) - \mu\Delta^2 u_2 &-\end{aligned}$$

$$-\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(u_1, u_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 + g(\phi, \lambda) \right) = 0,$$
$$u_1|_{t=0} = u_{10}$$
$$u_2|_{t=0} = u_{20}$$

## **Chapter 2**

## **License**

## 2.1 LICENSE

Copyright (c) 2010 Alexey Ozeritsky  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## **Chapter 3**

# **Build**

## 3.1 Build

SP uses Linal library method to solve linear equations. And Spherepack library (my C port!) for Spherical Harmonics method. So you must put those libraries into SP root directory in the following way:

```
sp_directory/  
sp_directory/linal  
sp_directory/spherepack_c  
*  
* @subsection Unix-like  
* @verbatim  
mkdir build-directory  
cd build-directory  
cmake -DCMAKE_BUILD_TYPE=Debug path-to-sources # for Debug build  
cmake -DCMAKE_BUILD_TYPE=Release path-to-sources # for Release build  
make
```

### 3.1.1 Windows

```
mkdir build-directory  
cd build-directory  
cmake -G "Visual Studio 2009" #place your version of Visual Studio here
```

# Chapter 4

## Class Index

### 4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

SphereChafe . . . . .	21
SphereChafeConf . . . . .	22
SphereNorm . . . . .	25
SphereBaroclin . . . . .	13
SphereBarvortex . . . . .	18
SphereOperator . . . . .	27
SphereLaplace . . . . .	23



# Chapter 5

## Class Index

### 5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">SphereBaroclin</a> (Solve the two-dimensional baroclinic atmosphere equations ) . . . . .	13
<a href="#">SphereBarvortex</a> (Solve the Barotropic vorticity equation ) . . . . .	18
<a href="#">SphereChafe</a> (Chafe-Infante equation on sphere ) . . . . .	21
<a href="#">SphereChafeConf</a> (Chafe-Infante equation on sphere ) . . . . .	22
<a href="#">SphereLaplace</a> (Solve Laplace equation and calculate Laplace operator ) . . . . .	23
<a href="#">SphereNorm</a> (Spherical norm implementation ) . . . . .	25
<a href="#">SphereOperator</a> (The base class for all of spherical operators ) . . . . .	27



# Chapter 6

## File Index

### 6.1 File List

Here is a list of all documented files with brief descriptions:

<b>baroclin.cpp</b>	??
<b>baroclin.h</b>	??
<b>barvortex.cpp</b>	??
<b>barvortex.h</b>	??
<b>chafe.cpp</b>	??
<b>chafe.h</b>	??
<b>div.cpp</b>	??
<b>div.h</b>	??
<b>grad.cpp</b>	??
<b>grad.h</b>	??
<b>helm.c</b>	??
<b>helm.cpp</b>	??
<b>jac.cpp</b>	??
<b>jac.h</b>	??
<b>lapl.cpp</b>	??
<b>lapl.h</b>	??
<b>norm.cpp</b>	??
<b>norm.h</b>	??
<b>operator.cpp</b>	??
<a href="#">operator.h</a>	29
<b>statistics.h</b>	??
<b>test_baroclin.cpp</b>	??
<b>test_barvortex.cpp</b>	??
<b>test_chafe.cpp</b>	??
<b>test_jac.cpp</b>	??
<b>test_lapl.cpp</b>	??
<b>vorticity.cpp</b>	??
<b>vorticity.h</b>	??



# Chapter 7

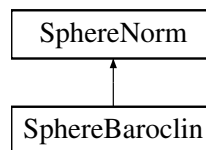
## Class Documentation

### 7.1 SphereBaroclin Class Reference

Solve the two-dimensional baroclinic atmosphere equations.

```
#include <baroclin.h>
```

Inheritance diagram for SphereBaroclin::



#### Public Member Functions

- void **S\_step** (double \*u1\_o, double \*u2\_o, const double \*u1, const double \*u2, double t)  
*Solve the two-dimensional baroclinic atmosphere equations.*
- void **S\_step** (double \*out, const double \*in, double t)  
*Provided for convince.*
- void **L\_step** (double \*u1\_o, double \*u2\_o, const double \*u1, const double \*u2, const double \*z1, const double \*z2)  
*Solve the linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2)*

$$\begin{aligned} \frac{\partial \Delta u_1}{\partial t} + J(u_1, \Delta z_1 + l + h) + J(z_1, \Delta u_1) + J(z_2, \Delta u_2) + J(u_2, \Delta z_2) + \frac{\sigma}{2} \Delta(u_1 - u_2) - \mu \Delta^2 u_1 &= 0 \\ \frac{\partial \Delta u_2}{\partial t} + J(u_1, \Delta z_2) + J(z_1, \Delta u_2) + J(u_2, \Delta z_1 + l + h) + J(z_2, \Delta u_1) + \frac{\sigma}{2} \Delta(u_1 + u_2) - \mu \Delta^2 u_2 &- \\ -\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(z_1, u_2) + J(u_1, z_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 \right) &= 0, \end{aligned}$$

- void **L\_step** (double \*out, const double \*in, const double \*z)  
*Solve the linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).*

- void `L_1_step` (double \*u1\_o, double \*u2\_o, const double \*u1, const double \*u2, const double \*z1, const double \*z2)  
*Solve the invert linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).*
- void `L_1_step` (double \*out, const double \*in, const double \*z)  
*Solve the invert linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).*
- void `p2u` (double \*u, const double \*p)  
*Add boundary conditions.*
- void `u2p` (double \*p, const double \*u)  
*Remove boundary conditions.*

### 7.1.1 Detailed Description

Solve the two-dimensional baroclinic atmosphere equations.

$$\begin{aligned} \frac{\partial \Delta u_1}{\partial t} + J(u_1, \Delta u_1 + l + h) + J(u_2, \Delta u_2) + \frac{\sigma}{2} \Delta(u_1 - u_2) - \mu \Delta^2 u_1 &= f(\phi, \lambda) \\ \frac{\partial \Delta u_2}{\partial t} + J(u_1, \Delta u_2) + J(u_2, \Delta u_1 + l + h) + \frac{\sigma}{2} \Delta(u_1 + u_2) - \mu \Delta^2 u_2 &- \\ -\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(u_1, u_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 + g(\phi, \lambda) \right) &= 0, \end{aligned}$$

where  $J(\cdot, \cdot)$  is spherical jacobian and  $\Delta$  is spherical Laplace operator.

**See also:**

[SphereJacobian](#), [SphereLaplace](#)

**Examples:**

[test\\_baroclin.cpp](#).

Definition at line 54 of file baroclin.h.

### 7.1.2 Member Function Documentation

#### 7.1.2.1 void SphereBaroclin::L\_1\_step (double \* out, const double \* in, const double \* z)

Solve the invert linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).

Provided for convince. u1 = &in[0] u2 = &in[nlat \* nlon]

**Parameters:**

*out* - result

*in* - input  
*t* - time

Definition at line 366 of file baroclin.cpp.

**7.1.2.2 void SphereBaroclin::L\_1\_step (double \* u1\_o, double \* u2\_o, const double \* u1, const double \* u2, const double \* z1, const double \* z2)**

Solve the invert linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).

**Parameters:**

*u1\_o* - output vector  
*u2\_o* - output vector  
*u1* - input vector (previous time step)  
*u2* - input vector (previous time step)  
*z1* - vector z1  
*z2* - vector z2

Definition at line 359 of file baroclin.cpp.

**7.1.2.3 void SphereBaroclin::L\_step (double \* out, const double \* in, const double \* z)**

Solve the linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2).

Provided for convince. u1 = &in[0] u2 = &in[nlat \* nlon]

**Parameters:**

*out* - result  
*in* - input  
*t* - time

Definition at line 354 of file baroclin.cpp.

**7.1.2.4 void SphereBaroclin::L\_step (double \* u1\_o, double \* u2\_o, const double \* u1, const double \* u2, const double \* z1, const double \* z2)**

Solve the linearized two-dimensional baroclinic atmosphere equations in a neighbourhood of point (z1, z2)

$$\begin{aligned} \frac{\partial \Delta u_1}{\partial t} + J(u_1, \Delta z_1 + l + h) + J(z_1, \Delta u_1) + J(z_2, \Delta u_2) + J(u_2, \Delta z_2) + \frac{\sigma}{2} \Delta(u_1 - u_2) - \mu \Delta^2 u_1 &= 0 \\ \frac{\partial \Delta u_2}{\partial t} + J(u_1, \Delta z_2) + J(z_1, \Delta u_2) + J(u_2, \Delta z_1 + l + h) + J(z_2, \Delta u_1) + \frac{\sigma}{2} \Delta(u_1 + u_2) - \mu \Delta^2 u_2 &- \\ -\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(z_1, u_2) + J(u_1, z_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 \right) &= 0, \end{aligned}$$

**Parameters:**

*u1\_o* - output vector

*u2\_o* - output vector  
*u1* - input vector (previous time step)  
*u2* - input vector (previous time step)  
*z1* - vector z1  
*z2* - vector z2  
*t* - time

Definition at line 347 of file baroclin.cpp.

#### 7.1.2.5 void SphereBaroclin::p2u (double \* u, const double \* p)

Add boundary conditions.

Just copy p to u. Provided for convince.

Definition at line 371 of file baroclin.cpp.

#### 7.1.2.6 void SphereBaroclin::S\_step (double \* out, const double \* in, double t)

Provided for convince.

`u1 = &in[0] u2 = &in[nlat * nlon]`

##### Parameters:

*out* - result  
*in* - input  
*t* - time

Definition at line 139 of file baroclin.cpp.

#### 7.1.2.7 void SphereBaroclin::S\_step (double \* u1\_o, double \* u2\_o, const double \* u1, const double \* u2, double t)

Solve the two-dimensional baroclinic atmosphere equations.

$$\begin{aligned}
 \frac{\partial \Delta u_1}{\partial t} + J(u_1, \Delta u_1 + l + h) + J(u_2, \Delta u_2) + \frac{\sigma}{2} \Delta(u_1 - u_2) - \mu \Delta^2 u_1 &= f(\phi, \lambda) \\
 \frac{\partial \Delta u_2}{\partial t} + J(u_1, \Delta u_2) + J(u_2, \Delta u_1 + l + h) + \frac{\sigma}{2} \Delta(u_1 + u_2) - \mu \Delta^2 u_2 &- \\
 -\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(u_1, u_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 + g(\phi, \lambda) \right) &= 0,
 \end{aligned}$$

##### Parameters:

*u1\_o* - output vector  
*u2\_o* - output vector  
*u1* - input vector (previous time step)  
*u2* - input vector (previous time step)

$t$  - time

**Examples:**

[test\\_baroclin.cpp](#).

Definition at line 150 of file baroclin.cpp.

**7.1.2.8 void SphereBaroclin::u2p (double \*  $p$ , const double \*  $u$ )**

Remove boundary conditions.

Just copy  $p$  to  $u$ . Provided for convince.

Definition at line 376 of file baroclin.cpp.

The documentation for this class was generated from the following files:

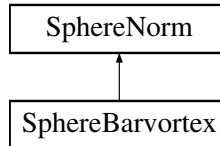
- baroclin.h
- baroclin.cpp

## 7.2 SphereBarvortex Class Reference

Solve the Barotropic vorticity equation.

```
#include <barvortex.h>
```

Inheritance diagram for SphereBarvortex::



### Public Member Functions

- void [S\\_step](#) (double \*out, const double \*in, double t)  
*Solve the Barotropic vorticity equation.*
- void [L\\_step](#) (double \*out, const double \*in, const double \*z)  
*Solve the linearized Barotropic vorticity equation in a neighbourhood of point (z).*
- void [LT\\_step](#) (double \*out, const double \*in, const double \*z)  
*Adjoint operator to L\_step.*
- void [L\\_1\\_step](#) (double \*out, const double \*in, const double \*z)  
*Solve the invert linearized Barotropic vorticity equation in a neighbourhood of point (z).*
- void [p2u](#) (double \*u, const double \*p)  
*Add boundary conditions.*
- void [u2p](#) (double \*p, const double \*u)  
*Remove boundary conditions.*

### 7.2.1 Detailed Description

Solve the Barotropic vorticity equation.

$$\frac{\partial \Delta \psi}{\partial t} + k_1 J(\psi, \Delta \psi) + k_2 J(\psi, l + h) + \sigma \Delta \psi - \mu \Delta^2 \psi = f(\varphi, \lambda)$$

where  $J(\cdot, \cdot)$  is spherical jacobian operator and  $\Delta$  is spherical Laplace operator.

**See also:**

[SphereJacobian](#), [SphereLaplace](#)

**Examples:**

[test\\_barvortex.cpp](#).

Definition at line 46 of file barvortex.h.

## 7.2.2 Member Function Documentation

### 7.2.2.1 void SphereBarvortex::L\_1\_step (double \* out, const double \* in, const double \* z)

Solve the invert linearized Barotropic vorticity equation in a neighbourhood of point (z).

#### Parameters:

- out* - output vector
- in* - input vector (previous time step)
- z* - vector z

Definition at line 293 of file barvortex.cpp.

### 7.2.2.2 void SphereBarvortex::L\_step (double \* out, const double \* in, const double \* z)

Solve the linearized Barotropic vorticity equation in a neighbourhood of point (z).

$$\frac{\partial \Delta \psi}{\partial t} + k_1 J(\psi, \Delta z) + k_1 J(z, \Delta \psi) + k_2 J(\psi, l + h) + \sigma \Delta \psi - \mu \Delta^2 \psi = 0$$

#### Parameters:

- out* - output vector
- in* - input vector (previous time step)
- z* - vector z

Definition at line 161 of file barvortex.cpp.

### 7.2.2.3 void SphereBarvortex::LT\_step (double \* out, const double \* in, const double \* z)

Adjoint operator to L\_step.

$$(L u, v) = (u, LT v)$$

#### Parameters:

- out* - output vector
- in* - input vector (previous time step)
- z* - vector z

Definition at line 247 of file barvortex.cpp.

### 7.2.2.4 void SphereBarvortex::p2u (double \* u, const double \* p)

Add boundary conditions.

Just copy p to u. Provided for convince.

Definition at line 332 of file barvortex.cpp.

**7.2.2.5 void SphereBarvortex::S\_step (double \* out, const double \* in, double t)**

Solve the Barotropic vorticity equation.

$$\frac{\partial \Delta \psi}{\partial t} + k_1 J(\psi, \Delta \psi) + k_2 J(\psi, l + h) + \sigma \Delta \psi - \mu \Delta^2 \psi = f(\varphi, \lambda)$$

**Parameters:**

*out* - output value

*in* - input vector (previous time step)

*t* - time

Definition at line 65 of file barvortex.cpp.

**7.2.2.6 void SphereBarvortex::u2p (double \* p, const double \* u)**

Remove boundary conditions.

Just copy p to u. Provided for convince.

Definition at line 337 of file barvortex.cpp.

The documentation for this class was generated from the following files:

- barvortex.h
- barvortex.cpp

## 7.3 SphereChafe Class Reference

Chafe-Infante equation on sphere.

```
#include <chafe.h>
```

### 7.3.1 Detailed Description

Chafe-Infante equation on sphere.

$$\frac{du}{dt} = \mu\delta u - \sigma u + f(u)$$

**Examples:**

[test\\_chafe.cpp](#).

Definition at line 52 of file chafe.h.

The documentation for this class was generated from the following files:

- chafe.h
- chafe.cpp

## 7.4 SphereChafeConf Struct Reference

Chafe-Infante equation on sphere.

```
#include <chafe.h>
```

### Public Attributes

- long [nlat](#)  
*latitude*
- long [nlon](#)  
*longitude*
- double [tau](#)  
*time step*
- double [mu](#)  
 $\mu$
- double [sigma](#)  
 $\sigma$
- double [theta](#)  
*parameter 0-1. 0.5 for Crank-Nicolson, 1.0 for backward Euler*
- [rp\\_t rp](#)  
*right part*

### 7.4.1 Detailed Description

Chafe-Infante equation on sphere.

Configuration.

#### Examples:

[test\\_chafe.cpp](#).

Definition at line 37 of file [chafe.h](#).

The documentation for this struct was generated from the following file:

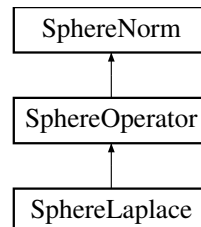
- [chafe.h](#)

## 7.5 SphereLaplace Class Reference

Solve Laplace equation and calculate Laplace operator.

```
#include <lapl.h>
```

Inheritance diagram for SphereLaplace::



### Public Member Functions

- void [solve](#) (double \*out, const double \*in, double m=1.0, double d=0.0)  
*Solve the equation.*
- void [calc](#) (double \*out, const double \*in)  
*Calculate Laplace operator.*
- void [make\\_psi](#) (double \*psi, const double \*u, const double \*v)  
*TODO: describe me!*

### 7.5.1 Detailed Description

Solve Laplace equation and calculate Laplace operator.

**Examples:**

[test\\_barvortex.cpp](#), and [test\\_lapl.cpp](#).

Definition at line 36 of file `lapl.h`.

### 7.5.2 Member Function Documentation

#### 7.5.2.1 void SphereLaplace::calc (double \* out, const double \* in)

Calculate Laplace operator.

$$out = \Delta in$$

**Parameters:**

*out* – result function

*in* – input function

Definition at line 79 of file lapl.cpp.

**7.5.2.2** `void SphereLaplace::solve (double * out, const double * in, double m = 1.0, double d = 0.0)`

Solve the equation.

$$m\Delta u + du = f$$

out[i][j] i - latitude (from the south pole to the north pole) j - longitude

**Parameters:**

*out* – result function

*in* – right part

*m* – Laplace multiplier

*d* – coefficient

Definition at line 42 of file lapl.cpp.

The documentation for this class was generated from the following files:

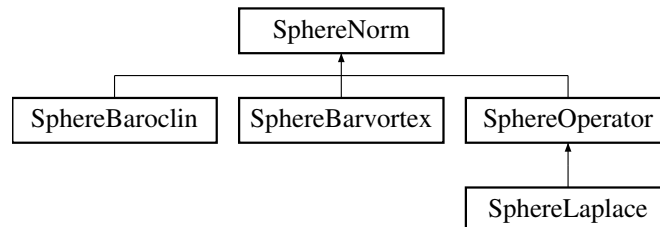
- lapl.h
- lapl.cpp

## 7.6 SphereNorm Class Reference

Spherical norm implementation.

```
#include <norm.h>
```

Inheritance diagram for SphereNorm::



### Public Member Functions

- [SphereNorm](#) (long *nlat*, long *nlon*)  
*Constructor.*
- double [scalar](#) (const double \**u*, const double \**v*)  
*Scalar product of two functions.*
- double [dist](#) (const double \**u*, const double \**v*)  
*the Distance between two functions.*
- double [norm](#) (const double \**u*)  
*the norm of functions.*

### 7.6.1 Detailed Description

Spherical norm implementation.

Definition at line 34 of file norm.h.

### 7.6.2 Constructor & Destructor Documentation

#### 7.6.2.1 SphereNorm::SphereNorm (long *nlat*, long *nlon*)

Constructor.

#### Parameters:

- nlat* - latitude
- nlon* - longitude

Definition at line 33 of file norm.cpp.

### 7.6.3 Member Function Documentation

#### 7.6.3.1 `double SphereNorm::dist (const double * $u$ , const double * $v$ )`

the Distance between two functions.

**Parameters:**

$u$   
 $v$

**Examples:**

[test\\_baroclin.cpp](#).

Definition at line 61 of file norm.cpp.

#### 7.6.3.2 `double SphereNorm::norm (const double * $u$ )`

the norm of functions.

**Parameters:**

$u$

Definition at line 56 of file norm.cpp.

#### 7.6.3.3 `double SphereNorm::scalar (const double * $u$ , const double * $v$ )`

Scalar product of two functions.

**Parameters:**

$u$   
 $v$

Definition at line 42 of file norm.cpp.

The documentation for this class was generated from the following files:

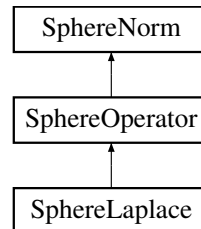
- norm.h
- norm.cpp

## 7.7 SphereOperator Class Reference

The base class for all of spherical operators.

```
#include <operator.h>
```

Inheritance diagram for SphereOperator::



### Public Member Functions

- [SphereOperator](#) (long nlat, long nlon, long isym)  
*Constructor.*
- void [func2koef](#) (double \*k, const double \*f)  
*Converts function to coefficients.*
- void [koef2func](#) (double \*f, const double \*k)  
*Converts coefficients to function.*

#### 7.7.1 Detailed Description

The base class for all of spherical operators.

It allocates memory for Spherepack and implements some usefull methods.

#### Examples:

[test\\_barvortex.cpp](#), and [test\\_lapl.cpp](#).

Definition at line 150 of file operator.h.

#### 7.7.2 Constructor & Destructor Documentation

##### 7.7.2.1 SphereOperator::SphereOperator (long nlat, long nlon, long isym)

Constructor.

#### Parameters:

- nlat* - latitude
- nlon* - longitude
- isym* - unused yet

Definition at line 75 of file operator.cpp.

### 7.7.3 Member Function Documentation

#### 7.7.3.1 void SphereOperator::func2koef (double \* *k*, const double \* *f*)

Converts function to coefficients.

**Parameters:**

*k* – array 2\*nlat\*nlat

*f* – array nlat\*nlon

Definition at line 130 of file operator.cpp.

#### 7.7.3.2 void SphereOperator::koef2func (double \* *f*, const double \* *k*)

Converts coefficients to function.

**Parameters:**

*k* – array 2\*nlat\*nlat

*f* – array nlat\*nlon

Definition at line 151 of file operator.cpp.

The documentation for this class was generated from the following files:

- [operator.h](#)
- operator.cpp

# Chapter 8

## File Documentation

### 8.1 operator.h File Reference

```
#include <vector>
#include "norm.h"
```

#### Classes

- class [SphereOperator](#)  
*The base class for all of spherical operators.*

#### 8.1.1 Detailed Description

**Author:**

Alexey Ozeritsky <[aozeritsky@gmail.com](mailto:aozeritsky@gmail.com)>

Definition in file [operator.h](#).



# Chapter 9

## Example Documentation

### 9.1 test\_baroclin.cpp

The two-dimensional baroclinic atmosphere equations

$$\begin{aligned} \frac{\partial \Delta u_1}{\partial t} + J(u_1, \Delta u_1 + l + h) + J(u_2, \Delta u_2) + \frac{\sigma}{2} \Delta(u_1 - u_2) - \mu \Delta^2 u_1 &= f(\phi, \lambda) \\ \frac{\partial \Delta u_2}{\partial t} + J(u_1, \Delta u_2) + J(u_2, \Delta u_1 + l + h) + \frac{\sigma}{2} \Delta(u_1 + u_2) - \mu \Delta^2 u_2 &- \\ -\alpha^2 \left( \frac{\partial u_2}{\partial t} + J(u_1, u_2) - \mu_1 \Delta u_2 + \sigma_1 u_2 + g(\phi, \lambda) \right) &= 0, \end{aligned}$$

```
#include <string>
#include <vector>
#include <math.h>

#include "baroclin.h"
#include "linal.h"

using namespace std;
using namespace linal;

double u1_t (double x, double y, double t)
{
    return x*sin(y+t)*ipow(cos(x),4);
}

double u2_t (double x, double y, double t)
{
    return x*cos(y+t)*ipow(cos(x),4);
}

double rp_f1(double x, double y, double t, const SphereBaroclin::Conf * conf)
{
    double sigma = conf->sigma;
    double mu     = conf->mu;
    double sigma1= conf->sigma;
    double mu1    = conf->mu;
    double alpha  = conf->alpha;

    return -45*mu*sin(y+t)*x-(9./2.)*sigma*
        ipow(cos(x),3)*sin(y+t)*sin(x)+(9./2.)*sigma*
        ipow(cos(x),3)*sin(x)*cos(y+t)-10*sigma*
        ipow(cos(x),4)*x*sin(y+t)+10*sigma*
```

```

        ipow(cos(x),4)*x*cos(y+t)+(15./2.)*sigma*
        ipow(cos(x),2)*x*sin(y+t)-(15./2.)*sigma*
        ipow(cos(x),2)*x*cos(y+t)-360*mu*sin(y+t)*sin(x)*
        ipow(cos(x),3)+147*mu*sin(y+t)*sin(x)*cos(x)-400*mu*sin(y+t)*x*
        ipow(cos(x),4)+390*mu*sin(y+t)*x*
        ipow(cos(x),2)-20*x*cos(y+t)*
        ipow(cos(x),4)-9*cos(y+t)*
        ipow(cos(x),3)*sin(x)+15*x*cos(y+t)*
        ipow(cos(x),2);
    }

double rp_g1(double x, double y, double t, const SphereBaroclin::Conf * conf)
{
    double sigma = conf->sigma;
    double mu    = conf->mu;
    double sigma1= conf->sigma;
    double mu1   = conf->mu;
    double alpha = conf->alpha;

    double alpha2 = alpha * alpha;
    double r = 20*x*sin(y+t)*
        ipow(cos(x),4)+9*sin(y+t)*
        ipow(cos(x),3)*sin(x)-15*x*sin(y+t)*
        ipow(cos(x),2)+390*mu*cos(y+t)*x*
        ipow(cos(x),2)-10*sigma*
        ipow(cos(x),4)*x*cos(y+t)-(9./2.)*sigma*
        ipow(cos(x),3)*sin(y+t)*sin(x)-alpha2*
        ipow(cos(x),7)*x-45*mu*cos(y+t)*x+18*
        ipow(cos(x),6)*
        ipow(cos(y+t),2)*sin(x)-9*
        ipow(cos(x),6)*sin(x)+9*x*
        ipow(cos(x),5)-(9./2.)*sigma*
        ipow(cos(x),3)*sin(x)*cos(y+t)-30*x*x*
        ipow(cos(x),4)*sin(x)-18*x*
        ipow(cos(x),5)*
        ipow(cos(y+t),2)+alpha2*
        ipow(cos(x),4)*x*sin(y+t)+(15./2.)*sigma*
        ipow(cos(x),2)*x*sin(y+t)+(15./2.)*sigma*
        ipow(cos(x),2)*x*cos(y+t)-400*mu*cos(y+t)*x*
        ipow(cos(x),4)+147*mu*cos(y+t)*sin(x)*cos(x)+60*x*x*
        ipow(cos(x),4)*
        ipow(cos(y+t),2)*sin(x)-360*mu*cos(y+t)*sin(x)*
        ipow(cos(x),3)-10*sigma*
        ipow(cos(x),4)*x*sin(y+t)+4*alpha2*
        ipow(cos(x),6)*x*x*sin(x)-9*alpha2*
        ipow(cos(x),3)*mul*cos(y+t)*sin(x)-20*alpha2*
        ipow(cos(x),4)*mul*cos(y+t)*x+15*alpha2*
        ipow(cos(x),2)*mul*cos(y+t)*x-alpha2*
        ipow(cos(x),4)*sigma1*x*cos(y+t);
    r /= alpha2;
    return r;
}

double rp_f2(double x, double y, double t, const SphereBaroclin::Conf * conf)
{
    double sigma = conf->sigma;
    double mu    = conf->mu;
    double sigma1= conf->sigma;
    double mu1   = conf->mu;
    double alpha = conf->alpha;

    return -9*cos(y+t)*
        ipow(cos(x),3)*sin(x)-20*x*cos(y+t)*
        ipow(cos(x),4)+15*x*cos(y+t)*
        ipow(cos(x),2)-(9./2.)*sigma*
        ipow(cos(x),3)*sin(y+t)*sin(x)+(9./2.)*sigma*
        ipow(cos(x),3)*sin(x)*cos(y+t)-10*sigma*

```

```

        ipow(cos(x),4)*x*sin(y+t)+10*sigma*
        ipow(cos(x),4)*x*cos(y+t)+(15./2.)*sigma*
        ipow(cos(x),2)*x*sin(y+t)-(15./2.)*sigma*
        ipow(cos(x),2)*x*cos(y+t)-360*mu*sin(y+t)*sin(x)*
        ipow(cos(x),3)+147*mu*sin(y+t)*sin(x)*cos(x)-400*mu*sin(y+t)*x*
        ipow(cos(x),4)+390*mu*sin(y+t)*x*
        ipow(cos(x),2)-45*mu*sin(y+t)*x;
    }

double rp_g2(double x, double y, double t, SphereBaroclin::Conf * conf)
{
    double sigma = conf->sigma;
    double mu     = conf->mu;
    double sigma1= conf->sigma;
    double mu1   = conf->mu;
    double alpha = conf->alpha;

    return -15*x*sin(y+t)*
        ipow(cos(x),2)+20*x*sin(y+t)*
        ipow(cos(x),4)+9*sin(y+t)*
        ipow(cos(x),3)*sin(x)-(9./2.)*sigma*
        ipow(cos(x),3)*sin(x)*cos(y+t)-10*sigma*
        ipow(cos(x),4)*x*sin(y+t)-10*sigma*
        ipow(cos(x),4)*x*cos(y+t)-(9./2.)*sigma*
        ipow(cos(x),3)*sin(y+t)*sin(x)+(15./2.)*sigma*
        ipow(cos(x),2)*x*sin(y+t)+(15./2.)*sigma*
        ipow(cos(x),2)*x*cos(y+t)-360*mu*cos(y+t)*sin(x)*
        ipow(cos(x),3)+147*mu*cos(y+t)*sin(x)*cos(x)-400*mu*cos(y+t)*x*
        ipow(cos(x),4)+390*mu*cos(y+t)*x*
        ipow(cos(x),2)-45*mu*cos(y+t)*x;
}

double rp1(double phi, double lambda, SphereBaroclin::Conf * conf)
{
    double omg = 2.*M_PI/24./60./60.; // ?
    double T0  = 1./omg;
    double R   = 6.371e+6;
    double c   = T0*T0/R/R;
    double x   = phi;

    double pt1 = -0.5 * (sin(x)*M_PI*x-2*sin(x)*x*x);
    if (fabs(pt1) > 1e-14) {
        pt1 /= cos(x);
    }

    double pt2 = -0.5*(-M_PI+4*x);
    return -T0/R * 16.0 / M_PI / M_PI * 30.0 * (pt1 + pt2);
}

double rp2(double phi, double lambda, SphereBaroclin::Conf * conf)
{
    double omg = 2.*M_PI/24./60./60.; // ?
    double T0  = 1./omg;
    double R   = 6.371e+6;
    double c   = T0*T0/R/R;
    double x   = phi;

    double pt1 = -0.5 * (sin(x)*M_PI*x-2*sin(x)*x*x);
    if (fabs(pt1) > 1e-14) {
        pt1 /= cos(x);
    }

    double pt2 = -0.5*(-M_PI+4*x);
    return -T0/R * 16.0 / M_PI / M_PI * 30.0 * (pt1 + pt2);
}

double cor(double phi, double lambda, double, const SphereBaroclin::Conf * conf)

```

```

{
    return 2.*sin(phi) + // l
           0.5 * cos(2*lambda)*sin(2*phi)*sin(2*phi); //h
}

double zero_cor(double phi, double lambda, double, const SphereBaroclin::Conf * c
onf)
{
    return 0;
}

double u0(double phi, double lambda)
{
    double omg = 2.*M_PI/24./60./60.; // ?
    double T0 = 1./omg;
    double R = 6.371e+6;

    return -T0/R * 16.0 / M_PI / M_PI * 30.0 *
           (M_PI/4 * phi * phi - phi * phi * phi / 3);
}

#define pOff(i, j) ( i ) * conf.nlon + ( j )

template < typename T >
void proj(double * u, SphereBaroclin & bv, SphereBaroclin::Conf & conf, T rp, dou
ble t)
{
    double dlat = M_PI / (conf.nlat - 1);
    double dlon = 2. * M_PI / conf.nlon;

    for (int i = 0; i < conf.nlat; ++i) {
        double phi = -0.5 * M_PI + i * dlat;
        for (int j = 0; j < conf.nlon; ++j) {
            double lambda = j * dlon;
            u[pOff(i, j)] = rp(phi, lambda, t);
        }
    }
}

void test_barvortex()
{
    SphereBaroclin::Conf conf;
    double R = 6.371e+6;
    double H = 5000;
    double omg = 2.*M_PI/24./60./60.; // ?
    double T0 = 1./omg;
    conf.k1 = 1.0;
    conf.k2 = 1.0;
    conf.tau = 0.0001;
    conf.sigma = 1.14e-2;
    conf.mu = 6.77e-5;

    conf.sigmal= conf.sigma;
    conf.mu1 = conf.mu;

    conf.nlat = 19;
    conf.nlon = 36;
    conf.theta = 0.5;

    conf.alpha = 1.0;
    conf.rp1 = rp_f1;
    conf.rp2 = rp_g1;

    // conf.alpha = 0.0;
    // conf.rp1 = rp_f2;
    // conf.rp2 = rp_g2;
}

```

```

conf.cor      = zero_cor;

int n = conf.nlat * conf.nlon;

double t = 0;
double T = 30 * 2.0 * M_PI;
double nrl;
double nr2;
int i = 0;

SphereBaroclin bv(conf);
vector < double > u1(n);
vector < double > u2(n);
vector < double > u1l(n);
vector < double > u2l(n);
vector < double > u1r(n);
vector < double > u2r(n);

fprintf(stderr, "#mesh_w:%d\n", conf.nlon);
fprintf(stderr, "#mesh_h:%d\n", conf.nlat);
fprintf(stderr, "#tau:%.16lf\n", conf.tau);
fprintf(stderr, "#sigma:%.16lf\n", conf.sigma);
fprintf(stderr, "#mu:%.16lf\n", conf.mu);
fprintf(stderr, "#sigma1:%.16lf\n", conf.sigma1);
fprintf(stderr, "#mul:%.16lf\n", conf.mul);
fprintf(stderr, "#alpha:%.16lf\n", conf.alpha);
fprintf(stderr, "#k1:%.16lf\n", conf.k1);
fprintf(stderr, "#k2:%.16lf\n", conf.k2);
fprintf(stderr, "#theta:%.16lf\n", conf.theta);
fprintf(stderr, "#rp:kornev1\n");
fprintf(stderr, "#coriolis:kornev1\n");
fprintf(stderr, "#initial:kornev1\n");
fprintf(stderr, "#build:$Id: test_baroclin.cpp,v d1c6e0fc40b1 2010/06/11
12:47:16 aozeritsky $");

proj(&u1[0], bv, conf, u1_t, 0);
proj(&u2[0], bv, conf, u2_t, 0);

while (t < T) {
    bv.S_step(&u1l[0], &u2l[0], &u1[0], &u2[0], t);
    t += conf.tau;

    proj(&u1r[0], bv, conf, u1_t, t);
    proj(&u2r[0], bv, conf, u2_t, t);

    nrl = bv.dist(&u1l[0], &u1r[0]);
    nr2 = bv.dist(&u2l[0], &u2r[0]);
    fprintf(stderr, "t=%le; nr=%le; nr=%le; min=%le; max=%le;\n",
            t, nrl, nr2,
            vec_find_min(&u1l[0], n),
            vec_find_max(&u1l[0], n));

    if (isnan(nrl) || isnan(nr2)) {
        return;
    }

    u1l.swap(u1);
    u2l.swap(u2);
    i++;
}

int main(int argc, char ** argv)
{
    fprintf(stderr, "#cmd:");
    for (int i = 0; i < argc; ++i) {
        fprintf(stderr, "%s ", argv[i]);
    }
}

```

```
    }  
    fprintf(stderr, "\n");  
    //set_fpe_except();  
    test_barvortex();  
}
```

## 9.2 test\_barvortex.cpp

the Barotropic vorticity equation

$$\frac{\partial \Delta \psi}{\partial t} + J(\psi, \Delta \psi) + J(\psi, l + h) + \sigma \Delta \psi - \mu \Delta^2 \psi = f(\varphi, \lambda)$$

```

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include <vector>

#include "barvortex.h"
#include "grad.h"
#include "vorticity.h"
#include "statistics.h"
#include "linal.h"

#ifdef max
#undef max
#endif

#ifdef WIN32
#define snprintf _snprintf
#endif

using namespace std;
using namespace linal;

static inline double max (double a, double b)
{
    return (a > b) ? a : b;
}

double ans (double x, double y, double t)
{
    return x*sin (y + t) *ipow (cos (x), 4);
}

double zero_coriolis (double phi, double lambda, double t, const SphereBarvortex:
:Conf * conf)
{
    return 0.0;
}

double f (double x, double y, double t, const SphereBarvortex::Conf * conf)
{
    double mu    = conf->mu;
    double sigma = conf->sigma;
    return 390*mu*sin (y+t) *x*ipow (cos (x), 2)
        +147*mu*sin (y+t) *sin (x) *cos (x) -
        400*mu*sin (y+t) *x*
        ipow (cos (x), 4) -360*mu*sin (y+t) *
        sin (x) *ipow (cos (x), 3) -20*sigma*sin (y+t) *
        ipow (cos (x), 4) *x+15*sigma*sin (y+t) *
        ipow (cos (x), 2) *x-9*sigma*sin (y+t) *
        ipow (cos (x), 3) *sin (x) +30*cos (y+t) *
        ipow (cos (x), 4) *sin (y+t) *x*x*sin (x) +9*cos (y+t) *
        ipow (cos (x), 6) *sin (y+t) *sin (x) -45*mu*sin (y+t) *x-
        9*cos (y+t) *ipow (cos (x), 5) *sin (y+t) *x-20*x*cos (y+t) *
        ipow (cos (x), 4) -9*cos (y+t) *
        ipow (cos (x), 3) *sin (x) +15*x*cos (y+t) *
        ipow (cos (x), 2);
}

```

```

void solve()
{
    long nlat = 19;
    long nlon = 36;

    SphereBarvortex::Conf conf;
    conf.nlat = nlat;
    conf.nlon = nlon;
    conf.mu = 8e-5;
    conf.sigma = 1.6e-2;
    conf.tau = 0.001;
    conf.theta = 0.5;
    conf.k1 = 1.0;
    conf.k2 = 1.0;
    conf.rp = f;
    conf.cor = zero_coriolis;

    double dlat = M_PI / (nlat - 1);
    double dlon = 2. * M_PI / nlon;
    double t = 0;

    int i, j, it = 0;

    vector < double > u (nlat * nlon);
    vector < double > v (nlat * nlon);
    vector < double > r (nlat * nlon);

    SphereBarvortex bv (conf);

    double nev1 = 0;

    for (i = 0; i < nlat; ++i)
    {
        for (j = 0; j < nlon; ++j)
        {
            double phi = -0.5 * M_PI + i * dlat;
            double lambda = j * dlon;

            r[i * nlon + j] = ans (phi, lambda, t);
        }
    }

    while (true)
    {
        bv.S_step (&u[0], &r[0], t);
        t += conf.tau;

        if (it % 100 == 0)
        {
            nev1 = 0.0;
            for (i = 0; i < nlat; ++i)
            {
                for (j = 0; j < nlon; ++j)
                {
                    double phi = -0.5 * M_PI + i * dlat;
                    double lambda = j * dlon;

                    v[i * nlon + j] = ans (phi, lambda, t);
                    //fprintf(stderr, "%.16lf %.16lf \n", u[i
* nlon + j], v[i * nlon + j]);
                }
            }

            nev1 = bv.dist(&u[0], &v[0]);

            fprintf (stderr, "time=%lf nev1=%.16le \n", t, nev1);
        }
    }
}

```

```

        r.swap(u);

        it += 1;
    }
}

inline double sign(double k)
{
    if (k > 0) {
        return 1.0;
    } else {
        return -1.0;
    }
}

double test_coriolis (double phi, double lambda)
{
    return 2 * sin(phi) + 0.5 * cos(2 * lambda) * sign(2 * phi) * ipow(sin(2
* phi), 2);
}

void output_psi(const char * prefix, const char * suffix,
               const double * psi, long nlat, long nlon,
               double U0, double PSI0,
               SphereGrad & grad)
{
    vector < double > u (nlat * nlon);
    vector < double > v (nlat * nlon);
    vector < double > Psi (nlat * nlon);

    grad.calc(&u[0], &v[0], &psi[0]);

    vec_mult_scalar(&u[0], &u[0], -1.0, nlat * nlon);

    char ubuf[1024]; char vbuf[1024]; char psibuf[1024];
    char Ubuf[1024]; char Vbuf[1024]; char Psibuf[1024];

    snprintf(ubuf, 1024, "out/%snorm_u%s.txt", prefix, suffix);
    snprintf(vbuf, 1024, "out/%snorm_v%s.txt", prefix, suffix);
    snprintf(psibuf, 1024, "out/%snorm_psi%s.txt", prefix, suffix);

    snprintf(Ubuf, 1024, "out/%sorig_u%s.txt", prefix, suffix);
    snprintf(Vbuf, 1024, "out/%sorig_v%s.txt", prefix, suffix);
    snprintf(Psibuf, 1024, "out/%sorig_psi%s.txt", prefix, suffix);

    mat_print(ubuf, &u[0], nlat, nlon, "%23.16lf ");
    mat_print(vbuf, &v[0], nlat, nlon, "%23.16lf ");
    mat_print(psibuf, &psi[0], nlat, nlon, "%23.16lf ");

    vec_mult_scalar(&u[0], &u[0], U0, nlon * nlat);
    vec_mult_scalar(&v[0], &v[0], U0, nlon * nlat);
    vec_mult_scalar(&Psi[0], &psi[0], PSI0, nlon * nlat);

    mat_print(Ubuf, &u[0], nlat, nlon, "%23.16le ");
    mat_print(Vbuf, &v[0], nlat, nlon, "%23.16le ");
    mat_print(Psibuf, &Psi[0], nlat, nlon, "%23.16le ");
}

void run_test(const char * srtm)
{
    long nlat = 5 * 19;
    long nlon = 5 * 36;

    SphereBarvortex::Conf conf;
    conf.nlat = nlat;
    conf.nlon = nlon;
    conf.mu = 1.5e-5;
}

```

```

conf.sigma      = 1.14e-2;
int part_of_the_day = 48;
conf.tau        = 2 * M_PI / (double) part_of_the_day;
conf.theta      = 0.5;
conf.k1         = 1.0;
conf.k2         = 1.0;
conf.rp         = 0;
conf.cor        = 0;//test_coriolis;

double dlat = M_PI / (nlat - 1);
double dlon = 2. * M_PI / nlon;
double t = 0;
double T = 2 * M_PI * 1000.0;

int i, j, it = 0;

vector < double > u (nlat * nlon);
vector < double > v (nlat * nlon);

vector < double > r (nlat * nlon);
vector < double > f (nlat * nlon);
vector < double > cor(nlat * nlon);
vector < double > rel(nlat * nlon);

double nr = 0;
double omg = 2.*M_PI/24./60./60.; // ?
double TE  = 1./omg;
double RE  = 6.371e+6;
double PSI0 = RE * RE / TE;
double U0  = 6.371e+6/TE;
const char * fn = srtm ? srtm : "";

if (fn) {
    FILE * f = fopen(fn, "rb");
    if (f) {
        size_t size = nlat * nlon * sizeof(double);
        if (fread(&rel[0], 1, size, f) != size) {
            fprintf(stderr, "bad relief file format ! \n");
        }
    } else {
        fprintf(stderr, "relief file not found ! \n");
    }
    fclose(f);
}

double rel_max = 0.0;
for (i = 0; i < nlat * nlon; ++i) {
    if (rel_max < rel[i]) rel_max = rel[i];
    //if (rel_max < fabs(rel[i])) rel_max = fabs(rel[i]);
}

for (i = 0; i < nlat; ++i)
{
    for (j = 0; j < nlon; ++j)
    {
        double phi    = -0.5 * M_PI + i * dlat;
        double lambda = j * dlon;

        //double ff = -(M_PI / 4 * ipow(phi, 2) - fabs(ipow(phi,
3)) / 3.0) * 16.0 / M_PI / M_PI * 3.0 / U0;
        //r[i * nlon + j] = (phi > 0) ? ff : -ff;
        if (phi > 0) {
            u[i * nlon + j] = (phi * (M_PI / 2. - phi) * 16 /
M_PI / M_PI * 30.0 / U0);
        } else {
            u[i * nlon + j] = (-phi * (M_PI / 2. + phi) * 16
/ M_PI / M_PI * 30.0 / U0);

```

```

    }
    v[i * nlon + j] = 0;
    //cor[i * nlon + j] = conf.coriolis(phi, lambda);
    //cor[i * nlon + j] = 1000 * rel[i * nlon + j] / rel_max
+ 2 * sin(phi);
    //
    // rel[i * nlon + j] = 0.5 * sign(phi) * cos(2 * lambda) * i
pow(sin(2 * phi), 2);

    if (rel[i * nlon + j] > 0) {
        rel[i * nlon + j] = 1.0 * rel[i * nlon + j] / rel
_max;
    } else {
        rel[i * nlon + j] = 0.0;
    }
    cor[i * nlon + j] = rel[i * nlon + j] + 2 * sin(phi);
}

}

SphereOperator op(nlat, nlon, 0);
SphereLaplace lapl(op);
SphereGrad grad(op);
SphereVorticity vor(op);

vor.calc(&f[0], &u[0], &v[0]);
vor.test();
vec_mult_scalar(&f[0], &f[0], -1.0, nlat * nlon);
#if 0
for (i = 0; i < nlat; ++i)
{
    for (j = 0; j < nlon; ++j)
    {
        double phi = -0.5 * M_PI + i * dlat;
        if (phi < 0) {
            f[i * nlon + j] *= -1.0;
        }
    }
}
#endif
lapl.solve(&r[0], &f[0]);
vec_mult_scalar(&f[0], &f[0], conf.sigma, nlat * nlon);

conf.rp2 = &f[0];
conf.cor2 = &cor[0];

SphereBarvortex bv (conf);

Variance < double > var(u.size());

mat_print("out/cor.txt", &cor[0], nlat, nlon, "%23.16lf ");
mat_print("out/rel.txt", &rel[0], nlat, nlon, "%23.16lf ");
mat_print("out/rp.txt", &f[0], nlat, nlon, "%23.16lf ");
mat_print("out/u0.txt", &u[0], nlat, nlon, "%23.16lf ");
mat_print("out/v0.txt", &v[0], nlat, nlon, "%23.16lf ");

//exit(1);

while (t < T)
{
    if (it % part_of_the_day == 0) {
        char buf[1024];
        nr = bv.norm(&r[0]);
        if (isnan(nr)) break;
        fprintf(stderr, "nr=%.16lf, t=%.16lf of %.16lf\n", nr, t,
T);

```

```
        snprintf(buf, 1024, "_%06d", it);
        output_psi("", buf, &r[0], nlat, nlon, U0, PSI0, grad);

        vector < double > m = var.m_current();
        vector < double > d = var.current();

        output_psi("m_", "", &m[0], nlat, nlon, U0, PSI0, grad);
        output_psi("d_", "", &d[0], nlat, nlon, U0, PSI0, grad);
    }

    bv.S_step (&u[0], &r[0], t);
    t += conf.tau;

    var.accumulate(u);

    r.swap(u);

    it += 1;
}

if (!isnan(nr)) {
    vector < double > m = var.m_current();
    vector < double > d = var.current();

    output_psi("m_", "", &m[0], nlat, nlon, U0, PSI0, grad);
    output_psi("d_", "", &d[0], nlat, nlon, U0, PSI0, grad);
}
}

int main (int argc, char * argv[])
{
    //solve();

    // exe [relief in binary format!]
    run_test(argv[1]);
}
```

## 9.3 test\_chafe.cpp

Chafe-Infante equation on a sphere

$$\frac{du}{dt} = \mu \Delta u - \sigma u + f(u)$$

$$u(x, y, t)|_{t=0} = u_0$$

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include <vector>

#include "chafe.h"
#include "linal.h"

#ifdef max
#undef max
#endif

using namespace std;
using namespace linal;

static inline double max (double a, double b)
{
    return (a > b) ? a : b;
}

double ans (double x, double y, double t)
{
    return x*sin (y + t) *ipow (cos (x), 4);
}

double f (double x, double y, double t,
          double mu, double sigma)
{
    return ipow (cos (x), 2) *
           (x*cos (y + t) *ipow (cos (x), 2)
            + sigma*sin (y + t) *ipow (cos (x), 2) *x + 9*mu*sin (y + t) *sin
(x) *cos (x)
            - 15*mu*sin (y + t) *x + 20*mu*sin (y + t) *x*ipow (cos (x), 2) )
    ;
}

void solve()
{
    long nlat = 19;
    long nlon = 36;

    SphereChafeConf conf;
    conf.nlat = nlat;
    conf.nlon = nlon;
    conf.mu = 1.0;
    conf.sigma = +70.0;
    conf.tau = 0.01;
    conf.theta = 0.5;
    conf.rp = f;

    double dlat = M_PI / (nlat - 1);
    double dlon = 2. * M_PI / nlon;
    double t = 0;

    int i, j, it = 0;
```

```

vector < double > u (nlat * nlon);
vector < double > v (nlat * nlon);
vector < double > r (nlat * nlon);

SphereChafe chafe (conf);

double nev1 = 0;

for (i = 0; i < nlat; ++i)
{
    for (j = 0; j < nlon; ++j)
    {
        double phi    = -0.5 * M_PI + i * dlat;
        double lambda = j * dlon;

        r[i * nlon + j] = ans (phi, lambda, t);
    }
}

while (true)
{
    chafe.calc (&u[0], &r[0], t);
    t += conf.tau;

    if (it % 100 == 0)
    {
        nev1 = 0.0;
        for (i = 0; i < nlat; ++i)
        {
            for (j = 0; j < nlon; ++j)
            {
                double phi    = -0.5 * M_PI + i * dlat;
                double lambda = j * dlon;

                v[i * nlon + j] = ans (phi, lambda, t);
                nev1 = max (nev1, fabs (u[i * nlon + j] -
v[i * nlon + j] ) );
//
                fprintf(stderr, "%.16lf %.16lf \n", u[i *
nlon + j], v[i * nlon + j]);
            }
        }

        fprintf (stderr, "nev1=%.16le \n", nev1);
    }
    r.swap(u);

    it += 1;
}

int main (int argc, char * argv[])
{
    solve();
}

```

## 9.4 test\_lapl.cpp

Laplace equation on a sphere

$$\Delta\psi = f(\varphi, \lambda)$$

$$\Delta\psi = \frac{1}{\cos\varphi} \frac{\partial}{\partial\varphi} \cos(\varphi) \frac{\partial}{\partial\varphi} \psi + \frac{1}{\cos^2\varphi} \frac{\partial^2}{\partial\lambda^2} \psi$$

```

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include <vector>

#include "lapl.h"
#include "linal.h"

#ifdef max
#undef max
#endif

using namespace std;
using namespace linal;

static inline double max(double a, double b)
{
    return (a > b) ? a : b;
}

double rp(double x, double y) {
    return -6.0 * sin(y) * sin(2.0 * x);
}

double ans(double x, double y) {
    return sin(y) * sin(2.0 * x);
}

double ans2(double x, double y, double t)
{
    return x*sin(y+t)*ipow(cos(x),4);
}

double rp2(double x, double y, double t, double mu, double sigma)
{
    return sin(y+t)*ipow(cos(x),2)*(9*mu*sin(x)*cos(x)+20*mu*x*ipow(cos(x),2)
+sigma*x*ipow(cos(x),2)-15*mu*x);
}

void solve()
{
    long nlat = 3*19, nlon = 3*36;
    double dlat = M_PI / (nlat-1);
    double dlon = 2. * M_PI /nlon;
    int i, j;

    double mu = -0.5;
    double sigma = 1000;

    vector < double > u(nlat * nlon);
    vector < double > v(nlat * nlon);
    vector < double > r(nlat * nlon);

    SphereOperator op(nlat, nlon, 0);

```

```

SphereLaplace lapl(op);

double nev1 = 0;

for (i = 0; i < nlat; ++i) {
    for (j = 0; j < nlon; ++j) {
        double phi    = -0.5 * M_PI + i * dlat;
        double lambda = j * dlon;

        r[i * nlon + j] = rp2(phi, lambda, 0, mu, sigma);
    }
}

lapl.solve(&u[0], &r[0], -mu, sigma);

for (i = 0; i < nlat; ++i) {
    for (j = 0; j < nlon; ++j) {
        double phi    = -0.5 * M_PI + i * dlat;
        double lambda = j * dlon;

        nev1 = max(nev1, fabs(u[i * nlon + j] - ans2(phi, lambda,
0)));
    }
}

fprintf(stderr, "nev1=%.16le \n", nev1);

lapl.calc(&v[0], &u[0]);
nev1 = 0.0;

for (i = 0; i < nlat; ++i) {
    for (j = 0; j < nlon; ++j) {
        double phi    = -0.5 * M_PI + i * dlat;
        double lambda = j * dlon;

        nev1 = max(nev1, fabs(5 * v[i * nlon + j] - r[i * nlon +
j]));
    }
}

fprintf(stderr, "nev1=%.16le \n", nev1);
}

int main(int argc, char * argv[])
{
    solve();
}

```