Wave Field Library (WFL)
Polygon Source Library (PSL)
リファレンスマニュアル

関西大学
電気電子情報工学科
光情報システム研究室

松島 恭治

2021 年 11 月 2 日

WaveField ライブラリバージョン： Rel 3.8.0
PolygonSource ライブラリバージョン： Rel 2.2.0
マニュアルバージョン： Rel 2.22.2

Note

- このマニュアルはハイパーテキストになっています．このマニュアルを効率良く参照するためには Acrobat Reader のツールボタンの「前の画面」と「次の画面」を表示することを強く奨めます．
- これらのツールボタンは，デフォルトではツールバーに表示されないので，「ツール」⇒「ツールバーのカスタマイズ」から設定を行ってください．

目次

第 I 部

WaveField ライブラリ (WFL)

第 1 章

開発環境

1.1 WaveField ライブラリを用いる開発環境

1.1.1 インストール

セットアッププログラムだけでは完全なインストールはできない．WaveField および PolygonSource ライブラリを用いたプログラム開発には，別紙「追加インストール作業」に記載のインストール作業が必要である．また，次節に示す開発環境のいずれかが必要である．

1.1.2 開発環境とコンパイラ

WFL では以下の開発環境とコンパイラで用いることができる．

1.1.2.1 MS Visual Studio .NET (Visual Studio 2003)

WFL を使用できるはずであるが，テストはほとんど行われていない．推奨しない．

1.1.2.2 MS Visual Studio 2005/2008

WFL を使用できるはずであるが，テストはほとんど行われていないため，推奨しない．

このバージョンの Visual Studio では IntelliSense の機能が弱いため，メンバー関数の補完等がうまく行かないことが多い．なお，64 ビット CPU 用コードのコンパイルも可能 (64 ビットコンパイラを使用するためには Visual Studio のインストール時に指定が必要．デフォルトではインストールされない)．

1.1.2.3 MS Visual Studio 2010/2012

WFL を使用でき，64 ビット CPU 用コードのコンパイルも可能なはずである．ただし，十分なテストは行われていない．IntelliSense が正常に機能しない場合もある．

1.1.2.4 MS Visual Studio 2013 - 2017

WFL を使用でき，64 ビット CPU 用コードのコンパイルも可能なはずである．ただし，十分なテストは行われていない．IntelliSense は正常に機能する．

1.1.2.5 MS Visual Studio 2019

推奨開発環境．

1.1.2.6 Intel Parallel Studio XE (Intel C++ Compiler)

WFL を使用できる．現バージョンの WFL のコンパイルは Visual Studio 2019 と結合した Intel Parallel Studio XE 2020 によって行われている．

Note

- 実行時にコンパイラやライブラリのバージョン・開発環境等を取得するためには，実行環境・コンパイラ等の情報を取得・設定する関数を利用する．

1.1.3 ディレクトリ構造とファイル

注意
WFL3.0 Beta3 以降では，それ以前のバージョンとインストールポイントが変更されている．
WFL3.4.2 以降では 32 ビット版は提供されていない．

WFL のインストーラは次のディレクトリに必要なファイルをインストールする．なおデフォルトでは < インストールディレクトリ >=c:\WaveFieldTools である．

1.1.3.1 ヘッダ

ヘッダファイルは 64 ビット版と 32 ビット版で共通である．
< インストールディレクトリ >\include\
< インストールディレクトリ >\include\wfl\
ファイル：*.h

1.1.3.2 インポートライブラリ

WFL を呼び出すプログラムのビルド時には以下のフォルダにあるインポートライブラリが必要である．
< インストールディレクトリ >\lib\win32\
　32 ビット版のインポートライブラリ
< インストールディレクトリ >\lib\
　64 ビット版のインポートライブラリ
ファイル：wfl3.lib

1.1.3.3 ダイナミックリンクライブラリ

WFL を呼び出すプログラムの実行時には以下のフォルダにあるダイナミックリンクライブラリが必要である．これは PATH の通ったフォルダーか実行ファイル (.exe ファイル) と同じフォルダーに置かれている必要がある．
< インストールディレクトリ >\bin\win32\
32 ビット版のダイナミックリンクライブラリ
< インストールディレクトリ >\bin\
64 ビット版のダイナミックリンクライブラリ
ファイル：wfl3.dll

1.1.3.4 サンプルコード

チュートリアルのサンプリコードの一部は以下に存在する．
< インストールディレクトリ >\SampleCode\
ファイル:*.cpp

第 2 章

チュートリアル

2.1 WaveField ライブラリを用いたプログラミングの基本

2.1.1 コーディングの基本

WFL を用いたプログラムのソースコードの骨組みを次に示す.

Example 一般的なソースコードのスケルトン

```
#include    <wfl.h>
using namespace wfl;

void main(void)
{
    Start();

    /*----- 以下に必要なプログラムを記述する -----*/


}
```

2.1.1.1 ヘッダファイル

ヘッダファイルとして `wfl.h` をインクルードする．WFL のヘッダファイルは多数のファイルに分かれているが，`wfl.h` をインクルードすることにより関連するヘッダファイルがすべて読み込まれる．

2.1.1.2 名前空間

WFL の関数やクラスは全て `WaveFieldLib` 名前空間内で定義されている．`wfl` はその別名である．従って，名前空間 `wfl` の利用を宣言するのが一般的であるが，`using namespace` を使わず，例えば，

```
wfl::Start();
```

の様に記述することもできる．

2.1.1.3 初期化関数

WFL で定義された関数やクラスを利用する場合は，原則としてまず初めに `Start()` 関数を実行する．

Note

- ビルド時にはライブラリファイル `wfl`x`.lib` をリンクするが，これを明示的に設定する必要はない．「追加インストール作業」が正しく行われていれば，`wfl.h` の読み込みによって自動的に `wfl`x`.lib` がリンクされる．
- 実行時には，`PATH` の通ったディレクトリか，実行ファイル (`*.EXE`) と同じディレクトリに `wfl`x`.dll` が必要である．

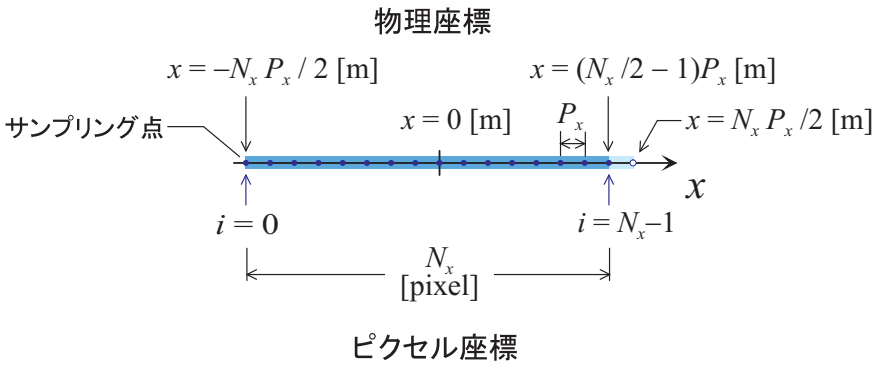


図 2.1　1 次元でのサンプリングのモデル

2.2　WaveField クラスを 1 次元複素数配列として利用する

WaveField クラスはリファレンス編の図 3.2 に示すとおり 2 次元の等間隔サンプリング格子の複素振幅分布をカプセル化したクラスとして定義されているが，1 次元でサンプリングした複素数配列と考えて用いることもできる．この場合のモデルを図 2.1 に示す．

2.2.1　簡単な例

WaveField クラスを 1 次元複素数分布 (配列) として扱ったソースリストの例を下記に示す．

Example　(ソース: ExSimple1D-a.cpp)

```
1  #include    "wfl.h"
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      WaveField a(256, 1, 0.01), b(1024, 1, 0.02);
8
9      double f = 0.1;                      // 空間周波数 0.1 [1/m]
10     double omega = 2 * Pi * f;           // 角周波数
11
12     int i;
13     a.Clear();                           // aをゼロクリア
14     for (i = 0; i < a.GetNx(); i++)      // i < 256 のループ
15     {
16         double x = a.X(i);               // サンプル点iの物理座標x
17         a.SetReal(i, 0, cos(omega*x));   // 点iのサンプル値の実部としてcos(x)を設定
18     }
19
20     b.Clear();
21     for (i = 0; i < b.GetNx(); i++)      // i < 1024 のループ
22     {
23         double x = b.X(i);               // サンプル点iの物理座標x
24         b.SetImag(i, 0, sin(omega*x));   // 点iのサンプル値の虚部としてsin(x)を設定
25     }
26
27     a.SaveAsCsv("cos関数.csv");          // CSVファイルとして保存
28     b.SaveAsCsv("sin関数.csv");
29 }
```

ソースリストの 7 行目は WaveField クラスのコンストラクタである．1 次元分布として用いるので，縦方向のサンプリング数を $N_y = 1$ としてコンストラクタを呼び出している．ここではオブジェクト a は横方向サンプリング数 $N_x = 256$，サンプリング間隔 $P_x = 0.01$[m] であり，オブジェクト b は横方向サンプリング数 $N_x = 1024$ でサンプリ

ング間隔 $P_x = 0.02$[m] としている．生成された WaveField オブジェクトのサンプリングデータは全く初期化されていないため，13 行目と 20 行目の様に原則として Clear() メンバー関数でゼロクリアする方が良い．

Note

- 図 2.1 に示すとおり，物理座標やサンプリング間隔の単位は [m] であり，その他の単位も基本的に SI 単位系を用いる．
- WaveField クラスではサンプリング数は必ず 2^n(n: 正の整数) でなければならない．2^n でないサンプリング数を指定した場合，指定したサンプリング数以上で最も小さな 2^n のサンプリング数でオブジェクトが生成される．

サンプル点は整数座標 i で指定され，$0 \leq i < N_x$ である．従って，ソースリスト 14 行目と 21 行目にあるように，ループはこの範囲で回す．この形の for ループが基本であり，ループの条件設定には必ず GetNx() メンバー関数や Nx() メンバー関数を用い，256 や 1024 等の定数をソースに埋め込まないのを原則とする．これは，意図的にサンプリング数を変更した場合にソースプログラムの変更を不要にするためと，自動的にサンプリング数が変更された場合でも適切に対応するためである．

この例では物理座標の原点は特に重要ではないが，$i = N_x/2$ が物理座標の原点 (実際にはローカル原点) である．従って物理座標の範囲は

$$-N_x P_x/2 \leq x \leq +(N_x/2 - 1)P_x$$

あるいは

$$-N_x P_x/2 \leq x < +N_x P_x/2$$

である ($x = +N_x P_x/2$ のサンプル点を含まない無いことに注意)．

16 行目と 23 行目にあるように，整数座標から物理座標への変換には X() メンバー関数を用いる．また，逆変換 (物理座標 → 整数座標の変換) が必要な場合は I() メンバー関数を用いる．これらの変換の厳密な定義は式 (3.1) と (3.2) を参照すること．

WaveField クラスの各サンプル値は Complex クラスのインスタンス (オブジェクト) であり，複素数値である．この例では，SetReal() メンバー関数 (17 行目) と SetImag() メンバー関数 (24 行目) を用いてオブジェクト a と b に実部あるいは虚部に正弦波と余弦波を設定している．10 行目で用いてる変数 (定数)Pi は円周率として WFL で定義されている．この例では，最後に SaveAsCsv() メンバー関数を用いて結果を CSV 形式ファイルとして保存している．

2.2.2 デフォルト値設定とオーバーロード演算子を用いた例

もう一つ例を下記に挙げる．

Example (ソース: ExSimple1D-b.cpp)

```
1  #include    "wfl.h"
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      WaveField::SetDefault(512, 1);  // デフォルト値の設定
8      WaveField::SetDefaultPx(0.01);  // デフォルト値の設定
9
10     WaveField a, b;                 // デフォルト値で生成するコンストラクタ
11
12     double f = 0.1;                 // 空間周波数 0.1 [1/m]
13     double omega = 2 * Pi * f;      // 角周波数
14     int i;
15     b.Clear();                      // ゼロクリア
16     for (i = 0; i < a.GetNx(); i++) // i < 512 のループ
17     {
18         Complex c = (0, omega*a.X(i));
19         a.SetValue(i, 0, exp(c));   // 点 i のサンプル値に複素数値 exp(i2πfx) を設定
```

```
20          b.SetReal(i, 0, b.X(i)*b.X(i)); // 点 i の実部に x の 2 乗を設定
21      }
22      a *= b;                             // a = a * b
23      a *= 3.0;                           // a = a * 3
24      a.SaveAsCsv("function.csv");        // CSV ファイルとして保存
25  }
```

この例では，7 行目と 8 行目で SetDefault() メンバー関数と SetDefaultPy() メンバー関数でサンプリング数やサンプリング間隔のデフォルト値を設定している．10 行目のコンストラクタでは引数を省略しているため，これらのデフォルト値が用いられる．従って，オブジェクト a も b もサンプリング数 512×1 でサンプリング間隔 $P_x = 0.01$[m] として生成される．いくつものオブジェクトを同じ設定で生成したい場合には，この様にデフォルト値設定関数を用いるのが便利である．

この例では a には exp() 関数を用いて関数 $\exp(i2\pi fx)$ を設定し (18, 19 行目)，b には x^2 を設定している (20 行目)．また，22 行目と 23 行目ではオーバーロード演算子を用いてサンプル点間の演算を行っている．これらの演算によりオブジェクト a は関数 $3x^2\exp(i2\pi fx)$ となる．

Note

- この例の場合，19 行目の exp() 関数の引数の実部はゼロであるため，計算が冗長であり余計な処理時間がかかる．このような場合には euler() 関数の方が効率が良い．euler() 関数を用いて 18 行目と 19 行目をまとめて書くと

  ```
  a.SetValue(i, 0, euler(omega*a.X(i)));
  ```

  となる．

2.2.3　FFT を用いた例

1 次元の離散フーリエ変換を用いた例を以下に示す．

Example　(ソース: Ex1DFft-a.cpp)

```
1   #include    <wfl.h>
2   using namespace wfl;
3
4   void main(void)
5   {
6       Start();
7       WaveField::SetDefault(1024, 1, 0.01);
8       WaveField a, b;
9       int i;
10
11      // 矩形関数を a に設定
12      a.Clear();
13      for (i = a.I(-0.5); i < a.I(+0.5); i++)
14          a.SetReal(i, 0, 1.0);
15      a.SaveAsCsv("Rect.csv");
16      a.Fft(-1);                          // 高速フーリエ変換
17      a.SaveAsCsv("FFT-Rect.csv");
18
19      // Λ(ラムダ)関数を b に設定
20      b.Clear();
21      for (i = b.I(-1.0); i < b.I(+1.0); i++)
22          b.SetReal(i, 0, 1.0 - fabs(b.X(i)));
23      b.SaveAsCsv("lambda.csv");
24      b.Fft(-1);                          // 高速フーリエ変換
25      b.SaveAsCsv("FFT-lambda.csv");
26  }
```

矩形関数は $-1/2 \le x \le +1/2$，Λ 関数は $-1 \le x \le +1$ の範囲で値を有するので，ここでは 13 行や 21 行のように，I() メンバー関数を用いて，その範囲の i 座標値でループを回している．離散 (高速) フーリエ変換は Fft() メンバー関数を用いて行う．フーリエ変換の定義は様々であるが，本ライブラリでは Fft() メンバー関数の引数が -1 の場合をフーリエ変換，$+1$ の場合を逆フーリエ変換と定義している．離散フーリエ変換の定義については式 (3.3) を参照す

ること．

Note

- 矩形関数の設定には SetRect() メンバー関数を用いることもできる．この場合，12~14 行を

  ```
  a.SetRect(1.0, 1.0);
  ```

  で置き換える．
- 離散 (高速) フーリエ変換の性質上，関数値の大きさはフーリエ積分による結果と一致しない．関数値の大きさをフーリエ積分による結果と一致させるためには，

  ```
  a *= a.GetPx();
  b *= b.GetPx();
  ```

  のようにして，サンプリング間隔を乗算する必要がある．

2.3 WaveField クラスを 2 次元複素振幅分布として利用する

WaveField クラスの本来の使用法は，平面上の 2 次元複素振幅分布を表すことである．以下にその最も簡単な例を示す．

2.3.1 球面波の複素振幅分布

これは球面波の複素振幅分布を位相と振幅に分けて BMP ファイルとして保存する例である．

Example (ソース: ExSphericalWave-a.cpp)

```
#include    <wfl.h>
using namespace wfl;

void main(void)
{
    Start();
    WaveField f(512, 512, 10e-6);   // 512x512の2次元分布
    int i, j;
    double z = 100e-3;  //原点から100mmの距離に点光源がある

    // 球面波をfに設定
    for (j = 0; j < f.GetNy(); j++)
        for (i = 0; i < f.GetNx(); i++)
        {
            double x = f.X(i), y = f.Y(j);
            double r = sqrt(x*x + y*y + z*z);
            f.SetValue(i, j, polar(1/r, f.GetWavenumber()*r));
        }

    f.Normalize();      //この例ではあまり意味はない
    f.SaveAsGrayBmp("SphericalWave-amp.bmp", AMPLITUDE);
    f.SaveAsGrayBmp("SphericalWave-phs.bmp", PHASE);
}
```

位置 (x, y, z) に位置する点光源からの球面波は，$(x, y, 0)$ 平面上では

$$f(x,y) = r^{-1}\exp[ikr]$$
$$r = \sqrt{x^2+y^2+z^2} \tag{2.1}$$

として与えられる．このプログラムでは，polar() 関数を用いて $f(x, y)$ をインスタンス f に設定している．

$f(x, y)$ を 2 次元分布をグレイスケールの BMP ファイルとして保存するためには SaveAsGrayBmp() メンバー関数を用いる．WaveField オブジェクトは複素振幅であるのに対して BMP ファイルは実数分布であるので，セーブする際にどのような分布をセーブするか指定する必要がある．ここでは，21 行では振幅分布を，22 行では位相分布をグレイスケールでセーブしている．セーブする分布を指定するためには Mode 列挙型を使用する．位相分布の画像を図 2.2

に示す．グラデーションをカラースケールにするためには SaveAsGrayBmp() メンバー関数の代わりに SaveAsBmp() メンバー関数を用い，Gradation 列挙型として COLOR を指定する．詳細は SaveAsBmp() メンバー関数を参照．

SaveAsGrayBmp() メンバー関数は振幅値 A をグレイスケールで保存する際，$A = 0 \rightarrow$ 黒，$A = 1 \rightarrow$ 白に変換している．そのため，振幅値のレベルが $0 \leq A \leq 1.0$ から外れる場合は真っ白にハレーションしたり，あるいは真っ黒になったりする場合がある．それを防ぐためには，一般に Normalize() メンバー関数を用いる．ただし，この例では元々 z がある程度大きい場合には球面波では振幅値が変化が生じないことから，Normalize() メンバー関数があってもなくても振幅分布は真っ白な画像になる．

Note

- 球面波は z 値によっては容易にエイリアシング誤差を発生するため，本来は最大回折角の範囲で窓関数を用いなければならない．この例では z 値が大きいためその必要がなく，簡単なソースとなっている．
- WFL には AddSphericalWaveSqr() メンバー関数等，球面波を高速に計算するための関数が複数用意されている．

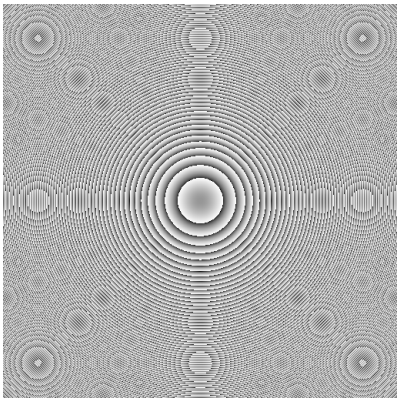


図 2.2　球面波の位相分布画像 (SphericalWave-phs.bmp)

2.3.2　2 次元矩形関数のフーリエ変換

以下は，幅が縦横とも 1 であるような矩形関数のフーリエ変換を求める例である．

Example　(ソース: Ex2DFftRect.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main(void)
5  {
6      Start();
7      WaveField a(512, 512, 0.1);      // 512x512の2次元分布
8      a.SetRect(1.0, 1.0);             // 縦横1の矩形関数をaに設定
9      a.Fft(-1);                       // フーリエ変換
10     a.Normalize();
11     a.SaveAsBmp("RectFFT-amp.bmp", AMPLITUDE, COLOR);
12 }
```

この例では 11 行目の SaveAsBmp() メンバー関数でグラデーションとして Gradation 列挙型 COLOR を指定しているため，カラースケールで保存されている．実際の画像を図 2.3 に示す．グラデーションとしてカラースケールを指定した場合には，デフォルトで画像の右端にカラーチャートが埋め込まれる．カラーチャートの最上部の色が振幅値の 1 に対応し，最下部の色が 0 に対応する．カラーチャートを埋め込みたくない場合は，次の様に，SaveAsBmp() メン

バー関数でチャート幅として 0 を指定する．

```
a.SaveAsBmp("RectFFT-amp.bmp", AMPLITUDE, COLOR, 0);
```

Note

- 2 次元 FFT はもっとも計算時間を要する演算の一つである．FFT については，現在，種々の演算パッケージが入手できる．WFL では，利用できる複数の FFT パッケージを組み込んでおり，また GPGPU 等ハードウェア支援を用いた FFT パッケージも利用できる (実装中)．
- Fft() メンバー関数で用いる FFT パッケージを切り替えるためには wfl::SetFftLib() 関数を用いる．

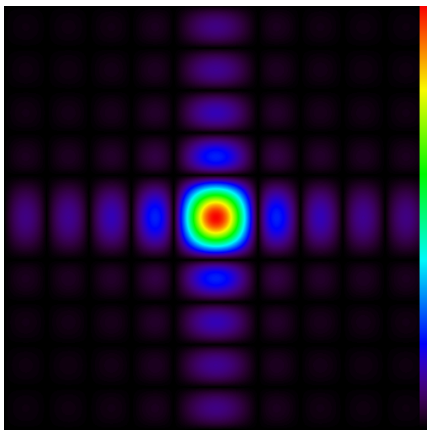


図 2.3 矩形関数のフーリエ変換分布画像 (RectFFT-amp.bmp)

2.3.3 FFT による画像の sinc 補間

ディジタル信号処理の一つの手法として FFT を用いた sinc 補間がある．以下は，そのプログラム例である．

Example (ソース: ExSincInterpol.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main(void)
5  {
6      Start();
7      WaveField a;
8      a.LoadBmp("shion-128gs.bmp", INTENSITY, 0, 2.2);    // 入力画像の読み込み．ガンマを2.2に設定
9      a.ModRandomPhase();                                  // 位相のランダム化
10     a.SaveAsGrayBmp("input-int.bmp", INTENSITY);         // 振幅画像の出力
11     a.SaveAsGrayBmp("input-phs.bmp", PHASE);             // 位相画像の出力
12     a.Fft(-1);                                           // フーリエ変換
13     a.Embed();                                           // サンプリング数の拡張
14     a.Normalize();
15     a.SaveAsGrayBmp("EmbededSpectrum.bmp", INTENSITY);   // 拡張したスペクトル
16     a.Fft(1);                                            // 逆フーリエ変換
17     a.Normalize();
18     a.SaveAsGrayBmp("OutputImage.bmp", INTENSITY);       // 出力画像
19 }
```

この例では 8 行目の LoadBmp() メンバー関数で入力画像を WaveField オブジェクト a の強度分布として読み込んでいる．この例のように，写真などの画像を読み込む際には LoadBmp() メンバー関数のガンマ値の引数として 2.2 を設定するのがふつうである．画像は位相値を持たないので，9 行目で ModRandomPhase() メンバー関数を用いて位相を乱数化している．この入力画像の強度像と位相像を図 2.4 に示す．

この様な画像を sinc 補間するとどうなるか調べるため，12 行目で離散フーリエ変換し，13 行目で Embed() メンバー


input-int.bmp

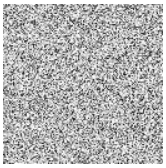
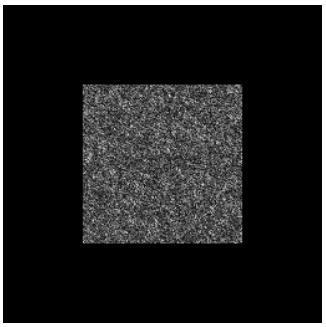

input-phs.bmp

図 2.4　入力複素画像

関数を用いてサンプリング数を縦横 2 倍で合計 4 倍に拡張している．これを逆フーリエ変換したのが図 2.5 の右の画像である．

以上の計算は，乱数位相を与えた複素振幅分布の遠視野像のシミュレーションを与えており，単純な位相の乱数化では像が劣化することを示している．これは，乱数位相には多数の “phase dislocation”(位相欠陥と訳される) が含まれるためである．ここで，phase dislocation(単に「スペックル」と呼ばれることもある) とは，位相 unwrapping (位相接続) が不可能であるような 2 次元位相分布上の不連続点のことである．


EmbededSpectrum.bmp


OutputImage.bmp

図 2.5　拡張したスペクトル像と sinc 補間された出力画像

2.4　平行平面間の回折伝搬計算

回折伝搬計算は波動光学シミュレーションでも最も重要な演算の一つであり，最も計算時間を必要とする演算である．現在，種々の回折伝搬計算アルゴリズムが提案されており，本ライブラリでも複数のアルゴリズムを実装している．利用に際しては，そのアルゴリズムの特性を把握することが重要である．

2.4.1　帯域制限角スペクトル法による円形開口からの回折像計算

次のソースコード例は，直径 1mm の円形開口からの回折像を回折距離 20mm で求めるものである．

2.4.1.1　バイナリの円形開口

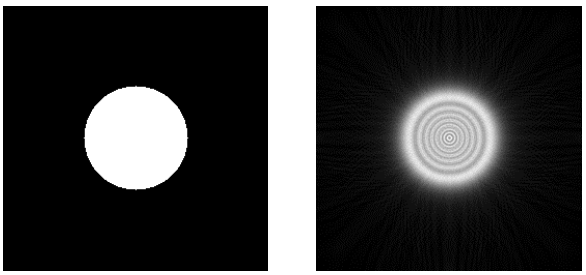
まず，開口パターンを 2 値で与えた円形開口からの計算例を示す．

Example　円形開口からの回折像計算 (1) (ソース: `ExCircAperture-a.cpp`)

```
1 #include      <wfl.h>
2 using namespace wfl;
3
4 void main()
```

```
5  {
6      Start();
7      WaveField a(256, 256, 2e-6);
8
9      //===== 円形開口をaに生成する
10     int i, j;
11     for (i = 0; i < a.Nx(); i++)
12     {
13         for (j = 0; j < a.Ny(); j++)
14         {
15             double x = a.X(i);
16             double y = a.Y(j);
17             double r = sqrt(x*x + y*y);
18             if (r < 0.1e-3)                          //半径0.1mmの円形開口
19                 a.SetValue(i, j, Complex(1.0, 0));   //開口内は1.0
20             else
21                 a.SetValue(i, j, Complex(0, 0));     //開口外は0.0
22
23         }
24     }
25     a.SaveAsGrayBmp("Aperture.bmp", AMPLITUDE);      //開口形状の保存
26
27     //===== 帯域制限角スペクトル法による回折伝搬計算
28     a.AsmProp(1e-3);                                 //1mmの伝搬回折計算を実行
29     a.Normalize();
30     a.SaveAsGrayBmp("Diffraction.bmp", INTENSITY);   //回折像(強度像)の保存
31 }
```

ここで，10 ～ 24 行はオブジェクト a 内に円形開口を設定している．ここでは，単純にオブジェクト a の全サンプル点をスキャンして，物理座標の原点から半径 0.1mm 以内ではサンプル値を 1，それ以外では 0 に設定している．作成した円形開口を図 2.6(a) に示す．



(a) Aperture.bmp　(b) Diffraction.bmp

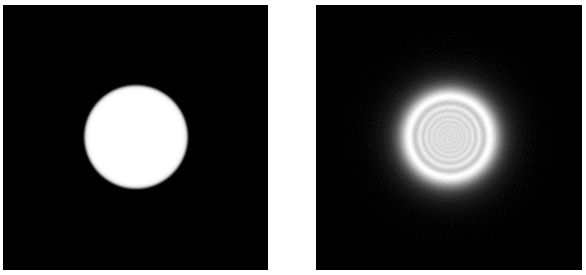
図 2.6　バイナリの円形開口とその回折像

回折計算は，28 行で WaveField クラスの AsmProp() メンバー関数を呼び出して行っている．オブジェクト a の内容は AsmProp() メンバー関数の実行により，回折計算の結果に置き換わる．つまり，WaveField オブジェクトを一種のスクリーンと考え，スクリーンが指定の距離だけ平行移動していると考えれば良い．求めた回折像 (強度像) を図 2.6(b) に示す．

ここでは，回折計算の手法として角スペクトル法 [1] を発展させた帯域制限角スペクトル法 [2] 用いてる．日本の光学の参考書にはあまり記述がないが，角スペクトル法は平面波展開法とも呼ばれ，フレネル近似条件を用いずに波動方程式の解を求める手法の一つである．フレネル近似を用いないため，伝播距離などの制限がなく，少なくとも解析的にはヘルムホルツ方程式の完全な解となっている．実際ここでも，わずか 1mm と短距離の伝搬計算を行っている．

2.4.1.2　ジャギーの影響を軽減した円形開口

よく見ると，図 2.6(b) の回折像には本来光がないはずの周辺部にも放射状の光が見られる．これは，円形開口が透過度 0 と 1 の 2 値パターンであり，開口の縁で，ビットマップ画像におけるジャギーと同様の効果が発生しているためであると考えられる．ジャギーを軽減するために，高次ガウス関数を用いアンチエイリアシングと同様の効果を持たせたのが次のコード例である．



(a) Aperture.bmp　　(b) Diffraction.bmp

図 2.7　アンチエイリアス効果を持たせた円形開口とその回折像

Example　円形開口からの回折像計算 (2) (ソース: ExCircAperture-b.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      WaveField a(256, 256, 2e-6);
8
9      //===== 円形開口をaに生成する
10     a.SetGaussian(0.1e-3, 50);                    //半径0.1mmの円形開口
11     a.SaveAsGrayBmp("Aperture.bmp", INTENSITY);   //開口形状の保存
12
13     //===== 帯域制限角スペクトル法による伝搬計算
14     a.AsmProp(1e-3);                              //1mmの伝搬回折計算
15     a.Normalize();
16     a.SaveAsGrayBmp("Diffraction.bmp", INTENSITY); //回折像(強度像)の保存
17 }
```

この例では，10 行目で WaveField クラスの SetGaussian() メンバー関数を用いて，円形開口を作成している．ガウス関数は本来指数部が 2 次の関数であるが，SetGaussian() メンバー関数は第 2 引数で 2 次を超える高次 (ここでは 50 次) を指定[*1]できる．この様にすると，非常に急峻な立ち上がりを有するがバイナリではない円形開口ができる．

開口パターンとその回折像を図 2.7 に示す．この回折像では，図 2.6(b) に見られたような放射状のノイズが消えていることがわかる．

2.4.1.3　伝搬距離が長い場合

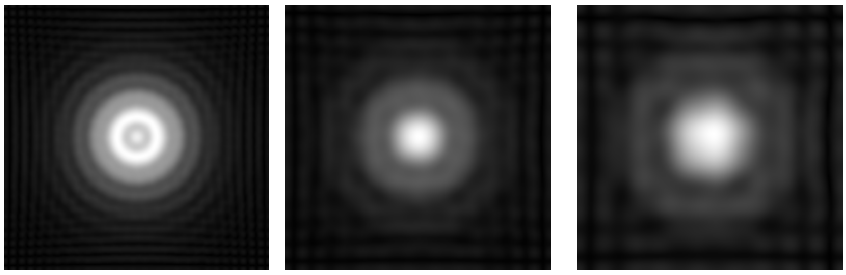
帯域制限角スペクトル法による回折伝搬計算は，従来の角スペクトル法に比べて伝搬距離が長い場合にも誤差が生じにくくなっている．これらの数値計算法では FFT を用いた離散高速畳み込み演算を行う．しかし，WaveField クラスの AsmProp() メンバー関数は単純な円状畳み込みを行うため，伝搬距離が伸び，光がサンプリング領域の境界近くまで広がってくると誤差が生じる．AsmProp() メンバー関数を用い，伝播距離を変えて図 2.6(a) の開口からの伝搬計算を行った結果を図 2.8 に示す．用いたソースコードは以下のとおりである．

Example　伝搬距離を変えた回折像計算 (ソース: ExCircAperture-c.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      WaveField a(256, 256, 2e-6);
8
9      a.SetGaussian(0.1e-3, 50);
10     a.SaveAsGrayBmp("Aperture.bmp", INTENSITY);   //開口形状の保存
11
```

[*1] このような関数はしばしばスーパーガウス関数と呼ばれる．

```
12      //===== 帯域制限角スペクトル法による伝搬計算
13      for (int i = 1; i <= 3; i++)
14      {
15          WaveField b = a;                                //オブジェクトbに開口aをコピーする
16          double d = 10e-3 * i;                           //dは伝搬距離
17          b.AsmProp(d);                                   //bを距離dだけ伝搬する
18          char fname[80];                                 //伝搬距離を入れたファイル名を作成
19          sprintf(fname, "Diffraction-%d.bmp", (int)(d/1e-3));
20          b.Normalize();
21          b.SaveAsGrayBmp(fname, INTENSITY);              //回折像(強度像)の保存
22      }
23  }
```



(a) Diffraction-10.bmp　(b) Diffraction-20.bmp　(c) Diffraction-30.bmp

図 2.8　AsmProp() で伝搬距離を増加した場合の回折像．伝搬距離を，(a)10mm，(b)20mm，(c)30mm とした結果．

15 行目では二つ目の WaveField オブジェクトとして b を定義し，開口データの入っている a をコピーしているが，これは AsmProp() メンバー関数の実行によりそのオブジェクトの内容が回折計算結果に置き換わり，元の開口データが破壊されるためである．

図 2.8(c) では，回折パターンが円形でなくなってきていることがわかる．円形開口からの回折像としてはこれは明らかにおかしい．これは，円状畳み込み演算のいわゆる “端効果” により，サンプリング領域の端で畳み込みが正しく行われていないためである．

この様な端効果を防ぐためには，回折計算前にサンプリング数を 4 倍 (縦横 2 倍) に拡張して周辺を 0 で埋め，回折計算後に中心の 4 分の 1 を切り取って元に戻せばよい．その処理には WaveField クラスの Embed() メンバー関数と Extract() メンバー関数を用いる．すなわち，次のソースのとおり，これらで AsmProp() メンバー関数を挟めばよい．

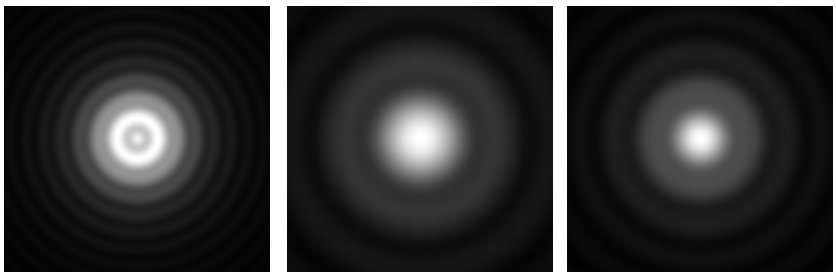
Example　より正確な回折像計算 (ソースの一部のみ記載) (ソース: ExCircAperture-d.cpp)

```
1       //===== 帯域制限角スペクトル法による伝搬計算(伝搬距離が長い場合)
2       for (int i = 1; i <= 3; i++)
3       {
4           WaveField b = a;
5           double d = 10e-3 * i;
6           b.Embed(1);                 //サンプリング領域を4倍拡張
7           b.AsmProp(d);               //伝搬する
8           b.Extract(1);               //サンプリング領域を4分の1に縮小
9           char fname[80];
10          sprintf(fname, "Diffraction-%d.bmp", (int)(d/1e-3));
11          b.Normalize();
12          b.SaveAsGrayBmp(fname, INTENSITY);
13      }
```

この場合の結果を図 2.9 に示す．正しく円形の回折パターンになっていることがわかる．

Note

- 上に示したとおり，4 倍拡張・縮小を行うとより正確な計算が行えるが，その代償として一時的にオブジェクトの 4 倍のメモリを消費する．また，計算時間も 4 倍以上遅くなることに注意しなければならない．
- 4 倍拡張と縮小を一つの関数で行うのが，ExactAsmProp() メンバー関数である．



(a) Diffraction-10.bmp　(b) Diffraction-20.bmp　(c) Diffraction-30.bmp

図 2.9　4 倍拡張して伝搬距離を増加した場合の回折像．伝搬距離を，(a)10mm，(b)20mm，(c)30mm とした結果．

- 畳み込み演算における FFT を分離した関数として AsmPropFs() メンバー関数がある．フーリエ空間で畳み込み以外の処理が必要な場合はこちらが便利である．

2.4.2　フレネル回折計算による円形開口からの回折像

フレネル回折を数値計算する方法は，大きくわけると FFT を 1 回だけ用いる方法と，FFT を 2 回用いて高速畳み込みを行う方法の二つがある．このうち，高速畳み込みを行う方法は角スペクトル法とほぼ同じ方法であり，フレネル近似を用いる分だけ制約が強く，計算時間も角スペクトル法とほぼ同等なため[*2]，高速畳み込みを用いてフレネル回折計算を行うメリットはほぼ全くない．そのため，以下では省略する．

2.4.2.1　FFT を 1 回用いるフレネル回折計算

WaveField ライブラリでこの計算を行うには，WaveField クラスの FresnelProp() メンバー関数を用いる．以下のソースでは，フレネル回折条件にできるだけ適合するように，サンプリング間隔，開口サイズ，伝搬距離を調整してある．

Example　1 回の FFT によるフレネル回折計算 (ソース: ExSftFresnel.cpp)

```
#include      <wfl.h>
using namespace wfl;

void main()
{
    Start();
    WaveField a(256, 256, 10e-6);
    a.SetGaussian(0.5e-3, 50);

    //===== フレネル回折計算
    for (int i = 1; i <= 3; i++)
    {
        WaveField b = a;                          //オブジェクトbに開口aをコピーする
        double d = 20e-3 * i;                     //dは伝搬距離
        b.FresnelProp(d);                         //フレネル回折でbを距離dだけ伝搬する
        b.Normalize();
        char fname[80];                           //伝搬距離を入れたファイル名を作成
        sprintf(fname, "Diffraction-%d.bmp", (int)(d/1e-3));
        b.SaveAsGrayBmp(fname, INTENSITY);        //回折像(強度像)の保存
        printf("回折距離:%2.0lf[mm]  サンプリング領域の大きさ:%4.2lf×%4.2lf [mm^2]\n",
                    d/1e-3, b.GetWidth()/1e-3, b.GetHeight()/1e-3);
    }
}
```

[*2] 角スペクトル法がその周波数応答関数にルート演算を含むのに対して，フレネル回折ではルートを用いないという点では，一見フレネル回折が計算時間の点で有利に見える．しかし，現代の CPU ではルート演算は除算とほぼ同じ速度で行えるため，このことは数値計算上のメリットにならない．

15 行目で FresnelProp() メンバー関数を実行して，フレネル回折計算を行っている．また，21 行目では，WaveField クラスの GetWidth() メンバー関数と GetHeight() メンバー関数を用いてサンプリング領域の大きさを取得している．このプログラムの実行結果を下記に，また回折像 (振幅像) を図 2.10 に示す．

図 2.10 では，一見，回折距離が伸びるに従って回折像が縮んでいるように見え，おかしな結果となっている．しかし，これは実行結果を見るとわかるように，サンプリング領域の物理的サイズが変化しているためであることがわかる．FFT を用いた計算では一般にサンプリング数は変化しないが，1 回の FFT によるフレネル回折計算では，その性質上サンプリング間隔が回折距離に比例して拡大するため，サンプリング領域の物理的サイズが拡大し，画像のサイズに比べて回折像が縮んでいるように見える．この性質のため，1 回の FFT によるフレネル回折計算法の用途は限定されたものとなっている．

Example　フレネル回折計算の実行結果

```
*** WFL <Intel C++/X64/SSE1> Rel.3.05.00 [MATSU2016] No. of Cores: 8
回折距離:20[mm]  サンプリング領域の大きさ:1.27×1.27 [mm^2]
回折距離:40[mm]  サンプリング領域の大きさ:2.53×2.53 [mm^2]
回折距離:60[mm]  サンプリング領域の大きさ:3.80×3.80 [mm^2]
続行するには何かキーを押してください . . .
```

(a) Diffraction-20.bmp　(b) Diffraction-40.bmp　(b) Diffraction-60.bmp

図 2.10　1 回の FFT によるフレネル回折像．伝搬距離を，(a)20mm，(b)40mm，(c)60mm とした結果．

2.4.3　回折伝搬計算の応用

2.4.3.1　スリットを通過したレーザービームの回折像

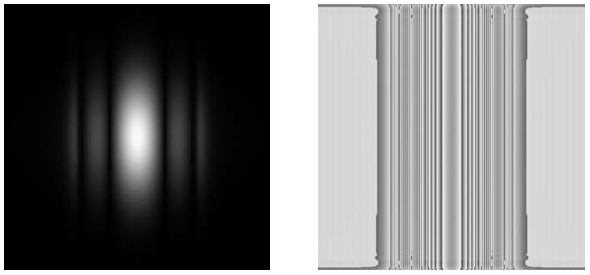
単純なシミュレーションであるが，次のソースコードは，幅 50μm のスリットを通過する波長 532nm のレーザービームの回折像を求めるプログラムである．

Example　レーザービームの回折像 (ソース: ExSlitBeam-a.cpp)

```
#include    <wfl.h>
using namespace wfl;

void main()
{
    Start();
    WaveField::SetDefault(512, 512, 10e-6, 10e-6, 532e-9);
    WaveField a, b;
    a.SetGaussian(1e-3);                          // 1/eビーム径2mmのガウスビーム
    b.SetRect(50e-6, 5e-3);                       // 幅0.05mmのスリット
    b *= a;
    b.ExactAsmProp(50e-3);                        // 距離50mm伝搬
    b.Normalize();
    b.SaveAsGrayBmp("intensity.bmp", INTENSITY);  //回折像(強度像)
    b.SaveAsGrayBmp("phase.bmp", PHASE);          //位相像
}
```

ここでは，オブジェクト a がレーザービーム，b がスリットとスリット背後の光波分布である．レーザービームは

$1/e$ 径 2mm のガウスビームで，スリットの位置にビームウェストがあるとしている．そのため，9 行目で WaveField クラスの SetGaussian() メンバー関数を用いてその振幅分布を設定している．この関数では “次数” の引数を省略するとデフォルト値として 2 次の通常のガウス関数になる．一方，スリットは WaveField クラスの SetRect() メンバー関数を用いて 10 行目で設定している．後は，この両者を乗算して (11 行)，ExactAsmProp() メンバー関数で回折伝搬演算している．得られた振幅分布と位相分布を図 2.11 に示す．



(a) 強度像　　(b) 位相像

図 2.11　スリットを通過したレーザービームの回折像．

2.4.3.2　レンズによる結像のシミュレーション

次のソースコードはレンズによる結像の簡易的な波動光学シミュレーションである．

Example　レンズによる結像 (ソース: ExImageFormationByLens.cpp)

```
#include    <wfl.h>
using namespace wfl;

void main()
{
    Start();
    WaveField::SetDefault(128, 128, 2e-6);
    double  R = 1e-3;           // 瞳の半径

    //===== aに物体データ読み込む
    WaveField a;
    a.LoadBmp("Shion-128gs.bmp", INTENSITY, 0, 2.2);
    a.Embed(3);                 // 8x8倍に拡張

    //===== bにレンズの透過関数を設定する
    WaveField b;
    b.Embed(3);                 // aと同じになるように拡張しておく
    b.SetGaussian(R, 100);      // 瞳関数設定．瞳は半径Rの円形瞳
    b.SetQuadraticPhase(10e-3); // レンズの位相関数設定．焦点距離は10mm
    b.SaveAsWf("lens.wf");      // 確認のためwf形式で保存

    //===== aをレンズ位置まで伝搬する
    a.ExactAsmProp(20e-3);      // レンズ位置は物体位置から20mm

    //===== aとbを乗算する
    b *= a;

    //===== bをスクリーン位置まで伝搬して縮小する
    b.ExactAsmProp(20e-3);      // スクリーン位置はレンズ位置から20mm
    b.Extract(3);               // 1/(8*8)に縮小
    b.Normalize();
    b.SaveAsGrayBmp("image.bmp", INTENSITY);// 保存
}
```

このソースにおける基本的なオブジェクトの役割は，a がレンズ前方の物体像，b がレンズとレンズ後方の像である．まず，11 ～ 13 行で物体像を a に設定している．ここでは，物体は図 2.12(a) に示すような平面画像である．LoadBmp() メンバー関数を用いた画像の読み込みに際してはガンマ値を 2.2 にすることに注意する．印画紙に焼き付けた本来の写真であれば，外光の反射によりそれ自体が拡散性のある光を発している．そのような散乱性のある光をシミュレーションするのは簡単ではないので，ここでは，この画像はピクセルピッチが 2μm の極めて小さな画像として

いる (画像サイズ $25.6 \times 25.6 \mu m^2$ のマイクロ画像!)．これは，画像が小さいとそれそのものによる回折が強く生じるため，その光の発散を利用してレンズ面での光の面積を広げるためである．これが，このシミュレーションを「簡易的」と表現する理由である．

この画像からの光はレンズに到達するまでにかなり拡散するので，13 行で Embed() メンバー関数により 8×8 倍にサンプリングエリアを拡大しておく．

次に 16 ～ 19 行目でオブジェクト b にレンズを作成している．レンズは円形としてその瞳を SetGaussian() メンバー関数を用いて 100 次のスーパーガウス関数で設定する．また SetQuadraticPhase() メンバー関数を用いて焦点距離 10mm のレンズとして働く 2 次位相を与えている．

25 ～ 32 行では，レンズへの入射光 a とレンズ透過度 b を乗算した結果を再び b に格納し，ExactAsmProp() メンバー関数で b をレンズから 20mm 離れたスクリーン位置まで伝搬した後，サンプリング領域を縮小してから保存している．この結果を図 2.12(b) に示している．ここでは，円形瞳のサイズを変えた結果を示してある．光学の教科書が教えるとおり，結像は倒立像となり，また瞳のサイズを縮小するとボケが強くなる．

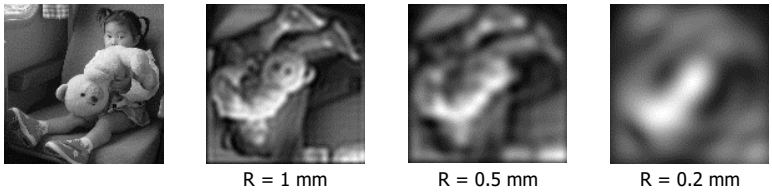


図 2.12　レンズによる結像シミュレーションの結果．(a) 現画像，(b) 様々な瞳半径における再生像．

Note

- 上記のソースプログラムでは，20 行目で WaveField クラスの SaveAsWf() メンバー関数を用いて，WF 形式で保存を行っている．WF 形式は光波の複素振幅分布の保存形式であり，波長等の補助データも含み，WaveField のオブジェクトを一時保管するのに都合が良いようにデザインされている．WF 形式で保存したデータは WaveField クラスの LoadWf() メンバー関数で再び WaveField オブジェクトに読み込むことができる．また，WaveFront アプリケーションを用いることにより，グラフィカルユーザーインターフェースを用いて WF 形式の光波の振幅像，強度像，位相像などを簡単に観察できるだけではなく，簡易的なシミュレーションを行うこともできる．

2.5　回転変換 (非平行平面間の回折伝搬計算)

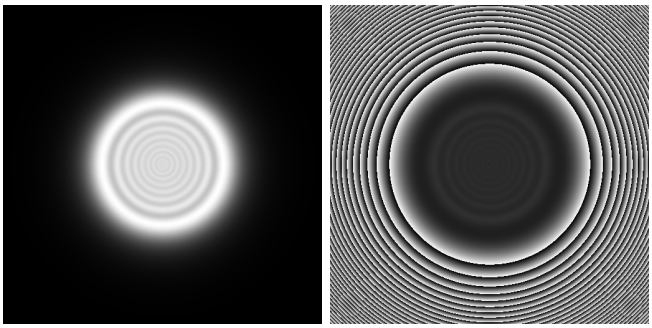
回転変換は，あるサンプリング平面で与えられる光波を傾いた別のサンプリング平面で観測した結果を与える．すなわち，回転変換は平行ではない平面間の回折伝搬計算と考えられる．

2.5.1　傾いた平面上で得られる円形開口からの回折像 (キャリア位相を無視)

次のソースコード例は，直径 0.5 mm の円形開口からの回折像を 45° 傾いた平面上で計算するプログラムである．ここで，回折像を得る平面を観測面と呼ぶことにする．観測面が伝搬元のサンプリング平面に対して大きく傾いている場合には，一般に観測面での光波のキャリア周波数が高くなるため，位相像では容易にエイリアシングが発生する．しかし，傾いた平面上での回折像 (振幅像または強度像) にのみ興味がある場合には，位相を無視して回折像のみを求めることができる．

Example　円形開口からのキャリア位相を無視した回折像計算 (ソース: ExRotationalTransform-a.cpp)

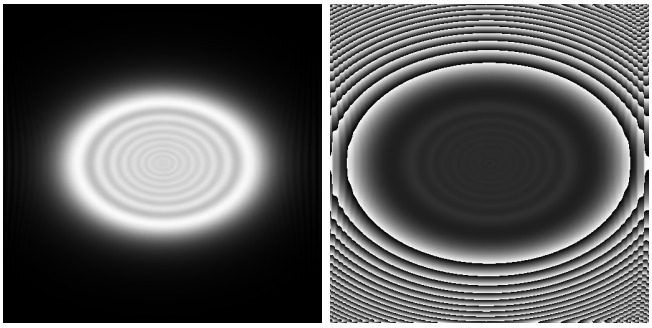
```
#include    <wfl.h>
using namespace wfl;

void main()
{
    Start();
    WaveField u0(2048, 2048, 0.5e-6);    //サンプリング数2K2K，間隔0.5μmのフィールド
    u0.SetGaussian(0.25e-3, 50);         //開口の生成
    u0.ExactAsmProp(6e-3);               //6mmの並進伝搬計算
    u0.SaveAsWf("Parallel.wf");          //伝搬元フィールドの保存
    RMatrix mat = CRMatrixY(45*DEG);     //y軸周りに45度座標回転する変換行列
    WaveField u1(2048, 2048, 0.5e-6);    //伝搬先フィールド
    u1.Rotate(u0, mat);                  //回転変換の実行(キャリア位相は無視する)
    u1.SaveAsWf("Rotation.wf");          //伝搬先フィールドの保存
}
```



振幅像　　位相像

図 2.13　開口から 6mm 伝搬した時の回折光波 (Parallel.wf)

ここで，9 行で円形開口からの光波を開口から 6mm 平面上で求め，次に 13 行で y 軸の周りに 45° 傾いた平面上での光波を求めている．回転角を設定するために，11 行目で RMatrix 型の回転行列 mat を設定している．ここでは y 軸周りの**座標回転**の変換行列を与える関数 wfl::CRMatrixY() 関数を用いてる．回転変換行列には，座標位置を回転する行列 RMatrixY() 関数と，座標系を回転する行列 CRMatrixY() 関数があることに注意しなければならない．回転変換で観測面を傾けるためには後者の座標系を回転する行列が必要である．なお，これらの関数の引数の角度の単位はラジアンである．11 行目で用いている DEG はマクロであり，$\pi/180$ の値に展開されるので，DEG を乗算することによりラジアン単位の角度が得られる．計算の結果を図 2.14 に示す．



振幅像　　位相像

図 2.14　45° 傾いた観測面での回折像 (Rotation.wf)

複数の軸の周りに回転する場合は，回転行列を乗算する．例えば，まず x 軸の周りに 15° 回転し，次に y 軸の周りに

45° 回転する場合には，11 行目は

```
RMatrix mat = CRMatrixY(45*DEG)*CRMatrixX(15*DEG);
```

となる．

Note

- 回転変換で用いる回転行列には，文字 C で始まる**座標回転**行列を用いる．
- 座標系として右手系を用いるので，座標軸周りの回転の向きは右ねじの向きであり，また角度の単位はラジアンである．

2.5.2　傾いた平面上で得られる円形開口からのキャリア位相を含む回折光波

次のソースコード例は，前節と同様，直径 0.5 mm の円形開口からの回折像を 45° 傾いた平面上で計算するプログラムである．しかし，ここでは回折光波をさらに伝搬するような場合を想定し，キャリア周波数を正しく反映した結果を得ている．

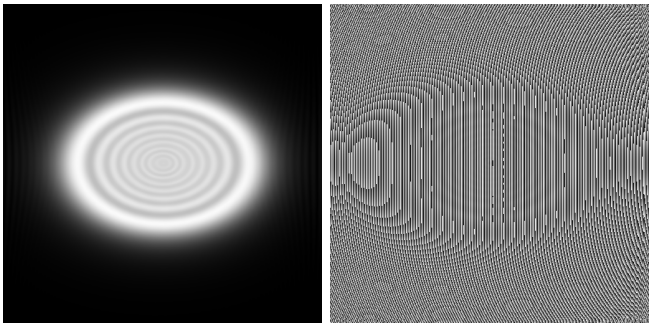


図 2.15　45° 傾いた観測面でのキャリア位相を含む回折光波 (Rotation-c.wf)

Example　円形開口からのキャリア位相を含む回折光波計算 (ソース: ExRotationalTransform-b.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      WaveField u0(2048, 2048, 0.5e-6);    //サンプリング数2K2K，間隔0.5μmのフィールド
8      u0.SetGaussian(0.25e-3, 50);         //開口の生成
9      u0.ExactAsmProp(6e-3);               //6mmの並進伝搬計算
10     u0.SaveAsWf("Parallel.wf");          //伝搬元フィールドの保存
11     RMatrix mat = CRMatrixY(45*DEG);     //y軸周りに45度座標回転する変換行列
12     WaveField u1(2048, 2048, 0.5e-6);    //伝搬先フィールド
13
14     SFrequency cal;                      //キャリア周波数を入れる変数
15     u1.Rotate(u0, mat, &cal);            //回転変換の実行(キャリア周波数がcalに入る)
16     u1.MultiplyPlaneWave(cal);           //得られたキャリア周波数の平面波を乗算する
17
18     u1.SaveAsWf("Rotation-c.wf");        //伝搬先フィールドの保存
19 }
```

14 行目からが異なる部分である．Rotate() メンバー関数に第 3 の引数として SFrequency 型の変数のポインタを与えると，関数終了時には回転変換で除去したキャリア周波数がその変数に代入される．16 行目では MultiplyPlaneWave() メンバー関数を用いて，このキャリア周波数の平面波を乗算している．これにより，Rotate() メンバー関数内で除去されたキャリアを復旧することができる．計算の結果を図 2.15 に示す．図 2.14 と比べると，振幅像に違いはないが，位相像にはキャリア成分がのっていることがわかる．

2.6　その他のサンプルソース

2.6.1　マルチスレッディングの制御

次のソースコード例は，WFL で使用するスレッド数を変化して計算時間を計測するものである．

Example　マルチスレッディングによる回折計算 (ソース: ExMultiThreading-a.cpp)

```
1  #include    <wfl.h>
2  using namespace wfl;
3
4  void main()
5  {
6      Start();
7      StopWatch sw;
8      WaveField a(2048, 2048, 2e-6);
9      a.SetGaussian(2e-3, 50);
10
11     //===== シングルスレッドでの伝搬計算(デフォルト)
12     sw.Start();                  //ストップウォッチスタート
13     a.ExactAsmProp(10e-3);       //デフォルトではシングルスレッドで計算する
14     sw.Stop();                   //ストップウォッチストップ
15     printf("%dスレッドでの計算時間: %f [s]\n", wfl::GetNumThreads(), sw.Read());
16
17     for (int i = 2; i <= 4; i++)
18     {
19         //===== マルチスレッドを用いた伝搬計算
20         sw.Reset();
21         wfl::SetNumThreads(i);       //計算にi個のスレッドを用いるように設定
22         sw.Start();                  //ストップウォッチスタート
23         a.ExactAsmProp(10e-3);       //i個のスレッド数を用いて計算する
24         sw.Stop();                   //ストップウォッチストップ
25         printf("%dスレッドでの計算時間: %f [s]\n", wfl::GetNumThreads(), sw.Read());
26     }
27 }
```

ここで，計算時間の計測には 7 行目の StopWatch クラスを用いている．13 行目では ExactAsmProp() メンバー関数を用いて伝搬計算を行っている．WFL は，デフォルトではシングルスレッドで処理を行うため，ここでの計算はシングルスレッドで行われる．

マルチスレッドの処理に切り替えるためには，21 行目の wfl::SetNumThreads() 関数を用いる．この関数は，引数が無い場合には検出されるコア数をすべて用いて処理を行うように WFL を設定する．引数がある場合にはその引数で示されるスレッド数で処理を行うように設定する．このソースをコンパイルして実行した結果を次に示す．

Example 実行結果

```
*** WaveFieldLib <Intel C++/Win32/SSE1> Rel.3.00.00  CPU Core: 8
1スレッドを用いた伝搬計算時間: 1.185000 [s]
2スレッドを用いた伝搬計算時間: 0.718000 [s]
3スレッドを用いた伝搬計算時間: 0.562000 [s]
4スレッドを用いた伝搬計算時間: 0.546000 [s]
続行するには何かキーを押してください . . .
```

この結果からわかる通り，4 スレッドでの計算時間はシングルスレッドの場合の約半分になる．なお，Start() 関数のメッセージに示されている 8 という数字は，実行マシンで検出されたコア数であり，計算に利用するスレッド数ではないので注意しなければならない．

Note

- デフォルトでは，マルチスレッドで処理されない．wfl::SetNumThreads() 関数が実行されてマルチスレッド処理に切り替わる．
- 内部的には OpenMP を用いてマルチスレッディングが実装されているが，WFL の処理に用いるスレッド数は OpenMP の関数では設定できない．ただし，ネスト等の処理の設定には OpenMP 関数が利用できる場合もある．

第 3 章

リファレンス

3.1　名前空間内でグローバルな定義

3.1.1　初期化・ログ記録・実行時間計測関連する関数

WaveField の諸メソッド・関数を用いる前にまず実行しなければならない初期化関数，また長時間かかるシミュレーションにおいて，計算時間を計測したり，途中経過のメッセージをファイルに記録するための関数群．使用方法は下記の使用例を参照すること．

```
void Start(int MsgLevel = 1)
void Start(const char* fname)
void Start(int MsgLevel, const char* fname)
```

戻り値　なし

説　明　WaveField の初期化のために必ず初めに実行する関数．1 番目の形式では初期化時のメッセージのレベル MsgLevel のみを指定する．MsgLevel 引数の省略時は，メッセージレベルは 1 となり，WaveField のバージョン等だけを含む 1 行のメッセージが出力される．2 番目の形式では，ログファイルのファイル名を fname として指定する．メッセージレベルはデフォルト値の 1 となる．3 番目の形式では，メッセージレベル MsgLevel とログファイル名 fname の両方を指定する．

Note

- この関数より先に他の WaveFieldLib 関数を用いた場合の結果は保証されない．
- メッセージレベルは 0 ～ 3 であり，0 では初期化メッセージは出力されない．
- ログファイル名を指定しない場合，ログ記録関数では，コンソールにのみ出力を表示するが，ログファイル名を指定した場合，コンソールに表示した内容と同じ内容がログファイルに追記される．

```
void PrintLog(const char* format, 変数...)
void PrintL(const char* format, 変数...)
void Printf(const char* format, 変数...)
```

戻り値　なし．

説　明　format と変数で指定されるログをプリント&記録する．Start() 関数でログファイル名が指定されていないときは，コンソールにのみ表示する．

Note

- format と変数の使用方法は，標準ライブラリ関数の printf() と全く同じ．
- この関数の実行後は，改行されないため，format で改行を指定しない場合は，次行はその後に続く．
- PrintLog() 関数では，それを実行した時点までの CPU タイムとその時点の実際の時刻を表示する．
- PrintL() 関数では，それを実行した時点までの CPU タイムのみを表示する．
- Printf() 関数では，付加的な情報は表示されず，標準ライブラリ関数の printf() とほぼ同じ動作をするが，コンソールだけではなくログファイルにも同時に出力する．

Example

```
Start(3,"log.txt");
WaveField wf(256, 256);
float x = 1.0, y = 2.0;
PrintLog("これはPrintLog()関数の出力の例です. x = %f, y = %f\n", x, y);
PrintL("PrintL()関数はコンパクトに出力します. x = %f, y = %f\n", x, y);
Printf("Printf()関数は標準のprintf()関数と同じ出力をします. x = %f\n", x);
```

Example (出力結果)

```
*** WFL <Intel C++/X64/SSE1> Rel.3.04.00 [PCNAME] No. of Cores: 8
*** Build : Nov  9 2015 00:13:13 Intel C++ Ver.15.00, Copyright K. Matsushima
*** Logging Start from 2015/11/09 00:15:17, File : (null)
>>> CPU Time : 00000.000 Sec (2008/05/13 15:25:38) <<<
これはPrintLog()関数の出力の例です. x = 1.000000, y = 2.000000
00000.000s>PrintL()関数はコンパクトに出力します. x = 1.000000, y = 2.000000
Printf()関数は標準のprintf()関数と同じ出力をします. x = 1.000000
```

```
double GetTickSec()
```

戻り値　秒単位の時間

説　明　Start() 関数を実行してから経過した CPU 時間を取得する.

Note

- プログラム中のある場所から別の場所までの実行時間の計測には StopWatch クラスを利用した方が便利である.

```
FILE* GetLogFile(void)
```

戻り値　ログファイルのファイルポインタ

説　明　Start() 関数で指定したログファイルへのファイルポインタを取得する.

3.1.2 並列処理の制御

```
void SetNumThreads()
void SetNumThreads(int n)
```

戻り値　なし

説　明　WFL の関数が用いるスレッド数を設定する. 1 番目の形式では, スレッド数は CPU コアの数に設定される. 2 番目の形式では, スレッド数は n になる.

Note

- 実行しているマシンが有しているプロセッサーコア数は Start() 関数でメッセージレベルを 1 以上にした場合に表示される. また GetNumProcs() 関数や, OMP 関数 omp_get_num_procs() でも取得できる.
- デフォルトのスレッド数は (この関数を実行するまでは)1 である.

```
void PushNumThreads(int n)
```

戻り値　なし

説　明　WFL の関数が用いるスレッド数を一時的に n に変更する．この時，これ以前のスレッド数をスタックに保存し，回復できるようにする．

Note

- 以前のスレッド数を回復するには PopNumThreads() 関数を用いる．

```
int PopNumThreads(void)
```

戻り値　回復したスレッド数

説　明　PushNumThreads() 関数により一時的に変更していたスレッド数をそれ以前のスレッド数に戻す．

```
int GetNumThreads(void)
```

戻り値　スレッド数

説　明　この関数の実行時点で WFL が使用しているスレッド数を取得する．

Example

```
SetNumThreads();
Printf("実行しているスレッド数は%dである", GetNumThreads());
```

```
void SetDynamicThreads(bool onoff)
```

戻り値　なし

説　明　動的スレッド数制御をオン (onoff=true) またはオフ (onoff=false) する．

```
bool GetDynamicThreads(void)
```

戻り値　スレッド数

説　明　この関数の実行時点で設定されている動的スレッド数制御の有無を取得する．

```
int GetNumProcs(void)
```

戻り値　プロセッサコア数

説　明　WFL を実行中のマシンのプロセッサコア数を取得する．

Example

```
SetNumThreads();
Printf("このマシンのプロセッサコア数は%dである", GetNumProcs());
```

3.1.3 メモリの取得/解放と FFT パッケージの設定/取得する関数

```
void* Malloc(size_t size, const char* msg = NULL)
```

戻り値 取得したメモリブロックへのポインタ

説　明 メモリブロックを取得する．標準ランタイムライブラリの malloc() と同様の動作をするが，この関数では取得したメモリブロックが必ず 16 バイト境界で整列する．SSE 命令の利用にはこれが必要である．メモリブロックの取得に失敗した場合には，ErrorCode 列挙型 MEMORY のエラーを発生する．msg 文字列を指定した場合には，その際にメッセージを表示する．

```
void Free(void* p)
```

戻り値 なし

説　明 Malloc() 関数で取得したポインタ p から始まるメモリブロックを解放する．

```
void SetFftLib(FftLib fl)
```

戻り値 なし

説　明 FftLib 列挙型 fl で示される FFT パッケージを WFL で用いる FFT として設定する．

Note

- デフォルトでは FFT パッケージは MKL に設定されている．
- FFT の実行には，WaveField クラスの Fft() メンバー関数を用いる．

```
FftLib GetFftLib(void)
```

戻り値 FftLib 列挙型

説　明 現在設定されている FFT パッケージを取得する．

3.1.4 マルチパート/分割 WF 形式のファイルに関する関数

```
void CreateFileMpWf(const char* fname, unsigned short int size)
```

戻り値 なし

説　明 光波数 size のマルチパート WF 形式ファイルをファイル名 fname で作成する．作成できない場合はエ

ラーになる．

Note

- マルチパート形式ファイルの書き込み/読み込み等については，WaveField クラスの SaveAsMpWf() メンバー関数，LoadMpWf() メンバー関数，LoadParamMpWf() メンバー関数を参照．

```
int GetSizeMpWf(const char* fname)
```

戻り値　含まれる光波複素振幅データの数

説　明　ファイル名 fname のマルチパート WF 形式ファイルに含まれる光波の数を取得する．

```
void CreateFileSegWf(const char* fname, unsigned short int mx, unsigned short int my)
```

戻り値　なし

説　明　mx×my に分割した分割 WF 形式ファイルをファイル名 fname で作成する．作成できない場合はエラーになる．

Note

- 分割 WF 形式ファイルの書き込み/読み込み等については，WaveField クラスの SaveAsSegWf() メンバー関数，LoadSegWf() メンバー関数，LoadParamSegWf() メンバー関数を参照．

```
int GetSizeSegWf(const char* fname, int& mx, int& my)
```

戻り値　含まれる光波複素振幅データの総数

説　明　ファイル名 fname のマルチパート WF 形式ファイルに含まれる光波の数 mx×my を取得する．

3.1.5　エラー処理関数

```
void SetErrorHandling(ErrorHandling eh)
```

戻り値　なし

説　明　WFL では，エラー時の処理方法としてメッセージをコンソールとログファイルに出力してプログラムを終了する方法 (デフォルト設定) と，例外を発生する方法の 2 種類が用意されている．この関数を実行するとそれ以後，再設定されるまでエラー発生時の処理方法は ErrorHandling 列挙型 eh で指定される方法となる．

Note

- エラーコードとエラーメッセージについては ErrorCode 列挙型と GetErrorString() 関数を参照．
- 例外については WaveFieldException クラスを参照．
- CONSOLE が指定されている場合には，exit(1) でプログラムを終了する．

```
ErrorHandling GetErrorHandling(void)
```

戻り値 ErrorHandling 列挙型

説　明 この関数の実行時点で設定されているエラー処理方法を取得する．

```
void Error(ErrorCode n, const char* funcname, const char* msg)
```

戻り値 なし

説　明 ErrorCode 列挙型のエラーコード n のエラーを発生する．funcname はエラー発生箇所を示す関数名であり，msg は追加メッセージである．通常はこの関数を使用するのではなく，マクロ WFL_ERROR() 関数を用いる．

Note

- エラーコードとエラーメッセージについては ErrorCode 列挙型と GetErrorString() 関数を参照．

```
WFL_ERROR1(n)
WFL_ERROR(n, msg)
```

戻り値 なし

説　明 マクロ．ErrorCode 列挙型のエラーコード n のメッセージを表示してプログラムを終了する．2 番目の形式では msg を追加メッセージとして出力する．

Note

- このマクロではソースコード上でこのマクロが置かれた位置の関数名を取得している．
- エラーコードとエラーメッセージについては ErrorCode 列挙型と GetErrorString() 関数を参照．
- このマクロの定義は次のとおり．

```
#define WFL_ERROR1(n) Error(n, __FUNCTION__)
#define WFL_ERROR(n) Error(n, __FUNCTION__, msg)
```

```
const char* GetErrorString(ErrorCode n)
```

戻り値 エラーメッセージ文字列

説　明 ErrorCode 列挙型のエラーコード n のエラーメッセージを取得する．

Note

- エラーコードについては ErrorCode 列挙型を参照．

3.1.6 実行環境・コンパイラ等の情報を取得・設定する関数

```
int GetMajorVersion(void)
int GetMinorVersion(void)
int GetPatchNumber(void)
```

戻り値　バージョン番号

説　明　WFL のバージョンを取得する．バージョンは “xx.yy.zz” で表わされ，xx がメジャーバージョン，yy がマイナーバージョン，zz がパッチ番号である．

```
const char* GetVersionString(void)
```

戻り値　“xx.yy.zz” 形式でバージョンを表わす文字列

説　明　WFL のバージョンを取得する．

```
Platform GetPlatform(void)
```

戻り値　Platform 列挙型

説　明　WFL が動作するプラットフォームを取得する．

```
const char* GetPlatformString(void)
```

戻り値　プラットフォームを表わす文字列

説　明　WFL が動作するプラットフォームを取得する．

```
DevelopEnv GetCreateEnv(void))
```

戻り値　DevelopEnv 列挙型

説　明　WFL をコンパイルしたコンパイラを取得する．

```
const char* GetCreateEnvString(void)
```

戻り値　コンパイラを表わす文字列

説　明　WFL をコンパイルしたコンパイラを取得する．

```
DevelopEnv GetDevelopEnv(void)
```

戻り値　DevelopEnv 列挙型

説　明　現在実行しているコンパイラを取得する．

```
const char* GetDevelopEnvString(void)
```

戻り値 コンパイラを表わす文字列
説　明 現在実行しているコンパイラを取得する.

```
DevelopEnv GetTargetEnv(void)
```

戻り値 DevelopEnv 列挙型
説　明 WFL が想定しているコンパイラを取得する

```
const char* GetTargetEnvString(void)
```

戻り値 コンパイラを表わす文字列
説　明 WFL が想定しているコンパイラを取得する

```
CpuType GetCpuType(void)
```

戻り値 CpuType 列挙型
説　明 WFL が対応しているを CPU のタイプを取得する

```
const char* GetCpuTypeString(void)
```

戻り値 CPU タイプを表わす文字列
説　明 WFL が対応しているを CPU のタイプを取得する

```
const char* GetCreateEnvVersionString(void)
```

戻り値 コンパイラのバージョンを表わす文字列
説　明 WFL をコンパイルしたコンパイラのバージョンを取得する

```
const char* GetCreateTimeStamp(void)
```

戻り値 タイプスタンプを表わす文字列
説　明 WFL がコンパイルされた時点のタイプスタンプを取得する

3.1.7　物理空間の配置や幾何的問題に関わる関数

```
void SetPositionTolerance(double t)
```

戻り値　なし
説　明　位置許容誤差を設定する．

```
const double& GetPositionTolerance(void)
```

戻り値　位置許容誤差の値．単位 [m]
説　明　位置許容誤差を取得する．

```
const RMatrix& RMatrixX(double t)
const RMatrix& RMatrixY(double t)
const RMatrix& RMatrixZ(double t)
```

戻り値　RMatrix 型の回転行列
説　明　それぞれ，x 軸，y 軸，z 軸の周りに t[rad] だけ回転するための回転行列を取得する．

Note

- この関数は，RMatrix クラスの RotationX() メンバー関数等の別名として，グローバル関数として定義されている．

```
const RMatrix& CRMatrixX(double t)
const RMatrix& CRMatrixY(double t)
const RMatrix& CRMatrixZ(double t)
```

戻り値　RMatrix 型の回転行列
説　明　それぞれ，x 軸，y 軸，z 軸の周りに t[rad] だけ座標回転するための回転行列を取得する．

Note

- この関数は，RMatrix クラスの CRotationX() メンバー関数等の別名として，グローバル関数として定義されている．

3.1.8　その他の関数

```
InterpolFunc GetInterpolFunc(Interpol interpol)
```

戻り値 InterpolFunc 型の関数ポインタ

説　明 WaveField クラスにおいて Interpol 列挙型で指定する補間関数の関数ポインタを取得する関数.

```
void SetRandomSeed(int s)
```

戻り値 無し

説　明 WaveField クラスの SetRandomPhase() メンバー関数や ModRandomPhase() メンバー関数で用いる乱数ジェネレータのシードを s に設定する. s = 0 の場合は, 乱数ジェネレータはシステム稼働時間から毎回異なったシードを用いる.

3.1.9 データ型の定義

```
InterpolFunc
```

WaveField クラスでサンプル点間の補間を行う関数に対する関数ポインタを表すデータ型. GetInterpolFunc() 関数を参照. データ型の定義は次のとおり.

```
typedef Complex (WaveField::*InterpolFunc)(double x, double y) const
```

3.1.10 定数の定義

```
const double Pi
```

円周率 π として定義された定数.

Example

```
double angle = Pi/2;    // 角度90度をラジアン単位で表した値
```

```
PI
```

円周率 π として定義されたマクロ.

```
const double Deg
```

角度単位の度 (degree) をラジアンに変換するための定数. π/180 に等しい.

Example

```
Phase phs = 15*Deg          // 位相角15度
double angle = 90*Deg;      // 角度90度をラジアン単位で表した値
```

DEG

角度単位の度 (degree) をラジアンに変換するための定数として定義されたマクロ．$\pi/180$ に等しい．

3.2 データ型と列挙型

3.2.1 列挙型の定義

FftLib

WaveField クラスで用いる FFT パッケージの識別.

MKL ： Intel の商用ライブラリパッケージである Math Kernel Library に基づく FFT.
OOURA ： 京都大学の大浦拓哉氏が作成・公開されている Split-Radix 型 FFT のフリーソースに基づく FFT.

ComplexForm

WaveField オブジェクトは，実空間での複素振幅分布の場合とフーリエ空間での複素スペクトルの場合がある．この列挙型はそれを識別するために用いる.

COMPLEX ： 実空間での複素振幅分布を直交形式 ($c = a + bi$) で表している.
SPECTRUM ： 複素振幅分布をフーリエ空間で表現したスペクトルとなっている．この場合は本来 m 単位の物理座標 (x, y) を m^{-1} 単位の空間周波数 (u, v) で読み替える.
POLAR ： 実空間での複素振幅分布を極形式 ($c = A\exp(i\theta)$) で表わしている．内部的に使われている形式なので，実際には使用しないこと.
POLAR_SPECTRUM ： フーリエ空間で表現したスペクトルを極形式で表わしている．内部的に使われている形式なので，実際には使用しないこと.

WindowFunc

サンプリング窓に用いる窓関数を識別する列挙型.

NONE ： 特に窓関数を用いない.
RECTANGLE ： 矩形窓関数
BARTLETT ： バートレット窓関数.
HAMMING ： ハミング窓関数.
HANNING ： ハニング窓関数.

Mode

WaveField オブジェクトを画像として表現するした場合の分布を識別する.

INTENSITY ： 強度分布.
REAL ： 実部の分布.
IMAGINARY ： 虚部の分布.
PHASE ： 位相分布.

AMPLITUDE ： 振幅分布.

Note

- WaveField クラスの SaveAsBmp() メンバー関数等で用いられる.

ColorMode

カラー画像を読み込む際の変換やカラープレーン，3D モデルのカラーチャンネル等を識別する.

GRAY_SCALE ： グレイスケール.
RED ： 赤プレーン，赤チャンネル.
GREEN ： 緑プレーン，緑チャンネル.
BLUE ： 青プレーン，青チャンネル.

Note

- WaveField クラスの SaveAsBmp() メンバー関数等で用いられ，PSL の TfbUvMapping クラス，TfbLambertShading クラス等でも用いられる.

Interpol

補間方法を識別する.

LINEAR ： 線形補間.
CUBIC ： Cubic 補間法.
CUBIC4 ： 4 点 Cubic 補間法.
CUBIC6 ： 6 点 Cubic 補間法.
CUBIC8 ： 8 点 Cubic 補間法.
SINC ： sinc 補間法.
ADJACENT ： 隣接サンプル点で補間.
BILINEAR ： バイリニア補間法.
BICUBIC ： バイキュービック補間法.
NEAREST_NEIGHBOR ： ニアレストネイバ補間法 (=ADJACENT).

Axis

座標軸や方向を識別する.

X_AXIS ： x 軸や x 軸方向.
Y_AXIS ： y 軸や y 軸方向.

Note

- WaveField クラスの SaveAsCsv() メンバー関数で用いられる.

ErrorCode

WaveFieldLib で標準的に定義するエラーコード.

WFL_NO_ERROR	:	エラーなし.
MEMORY	:	メモリが不足.
FILEOPEN	:	ファイルがオープンできない.
PARAMETER	:	パラメータ (引数) が不正.
FILEWRITE	:	ファイル書き込みエラー.
FILEREAD	:	ファイル読み込みエラー.
SOMETHINGWRONG	:	未分類のエラー. その他のエラー.
COMPLEX_FORM	:	ComplexForm 列挙型の直交形式/極形式が不適合.
INVALID_TYPE	:	ComplexForm 列挙型の実空間/フーリエ空間が不適合.
INTERNAL	:	内部エラー. 機能の未実装など.

ErrorHandling

エラー発生時の処理方法.

CONSOLE	:	コンソール (コマンドライン) にエラーメッセージを出力する.
EXCEPTION	:	例外を発生する.

Gradation

WaveField クラスのオブジェクトを画像として保存する際の階調表現.

GRAY	:	グレイスケール.
COLOR	:	カラースケール.

Note

- WaveField クラスの SaveAsBmp() メンバー関数等で用いられる.

Platform

プラットホームの識別

WIN32	:	32bit Edition の Windows.
X64	:	64bit Edtion の Windows.
CLR	:	Windows の Common Language Runtime.
CLI	:	Windows 以外の Common Language Infrastructure.
LINUX32	:	32bit 用の Linux.

CpuType

CPU の識別．現時点では 1 種類のみ．

`SSE1` ： Streaming SIMD Extension 1 を実装する Intel/AMD 系の CPU.

DevelopEnv

開発環境 (コンパイラ) の識別．

`ICC` ： Intel C++ コンパイラ．
`MSC` ： Microsoft C++ コンパイラ．
`GCC` ： GNU gcc コンパイラ．

3.3 FieldParam クラス

PointArray クラス ⇒ Plane クラス ⇒ FieldParam クラス

FieldParam クラスは 2 次元平面上で等間隔サンプリングされた複素振幅分布 (光波分布) に関するパラメータを保持するためのクラスである．平面上のサンプリング窓を表すため Plane クラスを継承しており，PointArray クラスから継承されたオーバーロード演算子により移動や回転操作が行える．このクラスは WaveField クラスに継承されており，そのメソッドは主に WaveField クラスで呼び出される．そのため，ここでは詳細は述べない．

FieldParam クラス内で定義されている主なパラメータとその変更メソッドは次の通りである．

波長・波数

GetWavelength() メンバー関数，SetWavelength() メンバー関数，GetWavenumber() メンバー関数，SetWavenumber() メンバー関数

サンプリング数

N() メンバー関数，GetN() メンバー関数，GetNx() メンバー関数，GetNy() メンバー関数，SetNx() メンバー関数，SetNy() メンバー関数

サンプリング間隔

GetPx() メンバー関数，GetPy() メンバー関数，SetPx() メンバー関数，SetPy() メンバー関数

サンプリング窓平面の位置と方向 (主として Plane クラスからの継承)

GetOrigin() メンバー関数，SetOrigin() メンバー関数，GetNormalVector() メンバー関数，SetNormalVector() メンバー関数

その他

以上に示した以外に，ウィンドウ領域や複素振幅が実空間にあるかフーリエ空間にあるか等の状態を保持している．

3.4　WaveField クラス

WaveField クラスは 2 次元平面上で等間隔サンプリングされた複素振幅と波長等それに付随する情報をカプセル化している．WaveField オブジェクトは，それぞれ独自のローカル座標系を有しており，またローカル座標系原点のグローバル座標系内での位置およびサンプリング平面の法線ベクトルを保持している．この概念を図 3.1 に示す．

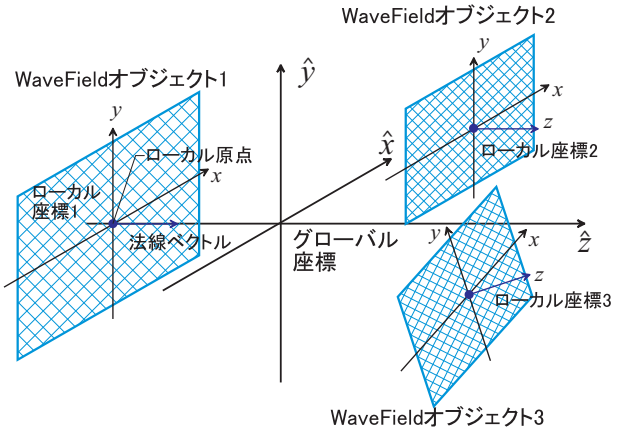


図 3.1　WaveField オブジェクトのモデル

WFL3 以降では，WaveField クラスは，FieldParam クラスを継承しており FieldParam クラスは Plane クラスを継承している．継承関係は次のとおりになっている．

PointArray クラス ⇒ Plane クラス ⇒ FieldParam クラス ⇒ WaveField クラス

このため WFL3 以降では，Plane クラスと同様，+=オーバーロード演算子による位置の変更や*=演算子による回転が可能である．なお，これらの演算はサンプリング窓の位置を変化するだけであり，光波の伝搬計算に基づいてサンプリング窓内での光波分布を求めるものではない．

3.4.1　ローカル座標系

図 3.2 に WaveField オブジェクトのローカル座標系を示す．WaveField オブジェクトでは，二つのローカル座標系を使用する．ひとつは，サンプル点を 2 次元の配列として見たときの，整数値による座標系である．これを整数座標系あるいは (i, j) 座標系と呼ぶ．整数座標系では，2 次元配列の左下のサンプル点を $(0, 0)$ とし，右上を $(N_x - 1, N_y - 1)$ としている．ここで N_x と N_y はそれぞれ，x 方向と y 方向のサンプル点数である．もうひとつは，物理的なローカル座標系で，メートルを単位としている．これを物理座標系あるいは (x, y) 座標系と呼ぶ．(i, j) 座標から (x, y) 座標への変換は，

$$x = \left(i - \frac{N_x}{2}\right) P_x \tag{3.1}$$
$$y = \left(i - \frac{N_y}{2}\right) P_y$$

で表され，逆変換は

$$i = \left\lfloor \frac{x}{P_x} \right\rfloor + 0.5 + \frac{N_x}{2} \tag{3.2}$$

$$j = \left\lfloor \frac{y}{P_y} \right\rfloor + 0.5 + \frac{N_y}{2}$$

で表される．ここで，$\lfloor \xi \rfloor$ は ξ を超えない整数を表す．

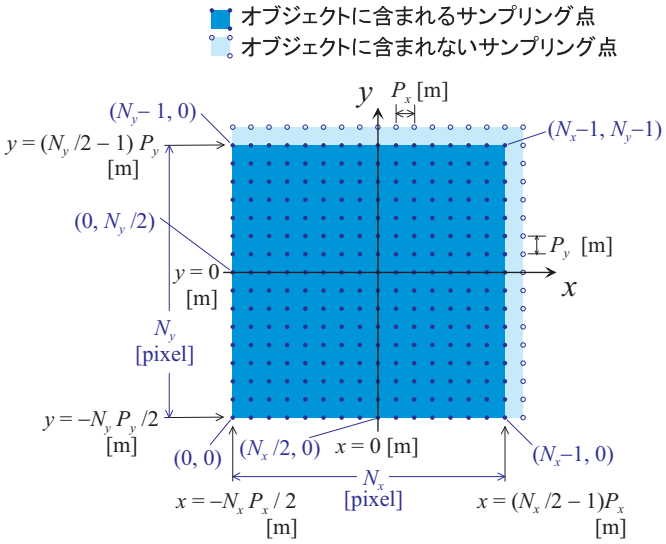


図 3.2　ローカル座標系の定義

Note

- 整数座標値 (i, j) からローカル物理座標値 (x, y) への変換には `X()` メンバー関数と `Y()` メンバー関数を用いる．
- ローカル物理座標値 (x, y) から整数座標値 (i, j) への変換には `I()` メンバー関数と `J()` メンバー関数を用いる．
- ローカル原点の座標 (グローバル座標値 $(\hat{x}, \hat{y})$) の取得・設定には `GetOrigin()` メンバー関数と `SetOrigin()` メンバー関数を用いる．
- ローカル座標系の z 軸は常にサンプリング平面の法線と同じ向きである．
- 法線ベクトルの取得・設定には `GetNormalVector()` メンバー関数と `SetNormalVector()` メンバー関数を用いる．
- デフォルトでは，ローカル原点はグローバル原点と一致し，法線ベクトルはグローバル座標系の $\hat{z}$ 軸の向きに一致する．

3.4.2　ウィンドウ領域

WaveField オブジェクトは，誤差等の計測や部分的なサンプリングアレイを切り出すための矩形ウィンドウ領域を保持している．ウィンドウ領域は (i, j) 座標系で保持されており，下図のとおり，`top`, `bottom`, `right`, `left` でその領域を参照する．図 3.3 にウィンドウ領域の定義を示す．

Note

- `bottom` と `left` の座標値のサンプル点は領域に含まれているが，`top` と `right` については，その座標値そのもののサンプル点は領域外であり，`top` -1 と `right` -1 のサンプル点が領域に含まれている．

ウィンドウには，以下のような操作が可能である (ウィンドウを操作するためのメンバー関数 参照)．

- ウィンドウ領域の設定

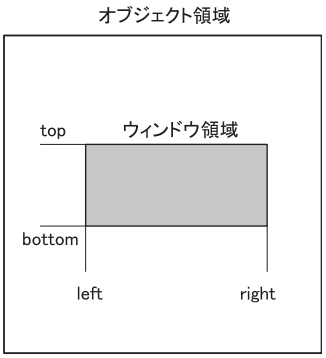


図 3.3　矩形ウィンドウ領域の定義

- 他のオブジェクトのウィンドウ領域の転送
- ウィンドウ領域をサブオブジェクトとして切り出し

3.4.3　コンストラクタ・デストラクタと初期化関連メンバー関数

```
WaveField(void)
WaveField(int nxy)
WaveField(int nx, int ny)
WaveField(int nx, int ny, double pxy)
WaveField(int nx, int ny, double px, double py)
WaveField(int nx, int ny, double px, double py, double wavelength)
```

戻り値　なし

説　明　コンストラクタ．1 番目の形式では全てデフォルト値のオブジェクトを生成する．2 番目の形式では x 方向サンプル点数と y 方向サンプル点数は同じ `nxy` となる．3 番目の形式では x 方向サンプル点数 `nx`，y 方向サンプル点数を `nx` とし，x, y 方向サンプリングが同じ間隔 `p` [単位:m] のオブジェクトを生成する．4 番目の形式では x 方向と y 方向サンプリング間隔を `px` と `py` で個別に設定する．5 番目の形式ではさらに，波長を `wavelength` [単位:m] に設定する．

Note

- 初期状態では省略された引数のデフォルト値 (既定値) は次のとおりである．
  サンプル点数　$N_x = N_y = 256$
  サンプリング間隔　$P_x = P_y = 1\mu\mathrm{m}$
  波長　$\lambda = 632.8\mathrm{nm}$
- これらのデフォルト値はデフォルト値を設定・取得するメンバー関数を用いて変更できる．
- サンプル点数は必ず，2 の累乗でなければならない．2 の累乗でないサンプル点数を指定した場合は，指定したサンプル点数が格納できる最小の 2 の累乗サイズでオブジェクトが生成される．

Example

```
//すべてデフォルト値のオブジェクトを宣言する
```

```
WaveField wf1;

//サンプル点数を512x512としたオブジェクトを宣言する
WaveField wf2(512);

//1024x1024でサンプリング間隔がx，y方向で同じ10umのオブジェクトを宣言する
WaveField wf3(1024, 1024, 10e-6);

//2048x2048でサンプリング間隔がx方向5um，y方向10umのオブジェクトを宣言する
WaveField wf3(2048, 2048, 5e-6, 10e-6);
```

```
WaveField(const WaveField& wf);
```

戻り値 なし

説　明 コピーコンストラクタ．

```
~WaveField(void)
```

戻り値 なし

説　明 WaveField オブジェクトを明示的に削除する．対象オブジェクトで使用していたメモリは解放される．

```
void Dispose(void)
```

戻り値 なし

説　明 WaveField オブジェクトで使用していたメモリのみが解放される．オブジェクト自体は削除されない．

```
WaveField& Init(Complex* d = NULL)
```

戻り値 対象オブジェクトへの参照

説　明 WaveField オブジェクトの以前のサンプリングデータを破棄し，d が NULL または省略された場合には新たにメモリブロックを取得する．以前のサンプリングデータは削除される．d が NULL でない場合には，d が示すメモリブロックをデータ配列として用いる．

Note

- データ配列は初期化されない．データ配列の初期化 (ゼロクリア) には Clear() メンバー関数を用いる．
- x 方向のサンプリング数が 4 以下のフィールドは生成できず，必ず 4 を超えるように修正される．
- 一度 d でメモリブロックを指定した後，再度 d=NULL でこの関数を呼び出した場合，以前に指定したメモリブロックを使用しようとする．この時，そのメモリブロックが $N_x \times N_y$ のデータに対して不足する場合はエラーを発生する．
- WaveField のメンバー関数には SSE 命令を使うものがあるため，メモリブロックは 16 バイト境界で整列しなければならない．
- wfl::Malloc() 関数で取得したメモリブロックのアライメントは 16 バイトになる．

```
WaveField& InitNxNy(void)
```

戻り値　対象オブジェクトへの参照

説　明　フィールドのサンプリング数の設定のみを 2 の整数乗に正規化する．データメモリの再取得は行わない．

Note

- データメモリは再取得されない．データメモリの再取得には Init() メンバー関数を用いる．
- x 方向のサンプリング数が 4 以下のフィールドは生成できず，必ず 4 を超えるように修正される．

```
Complex* GetDataPointer(void)
```

戻り値　データ配列メモリブロックの先頭へのポインタ

説　明　データ配列メモリブロックの先頭ポインタを取得する．

```
WaveField& Clear(void)
```

戻り値　対象オブジェクトへの参照

説　明　WaveField オブジェクトのサンプリングデータをゼロクリアする．実部虚部とも 0 が設定される．

```
WaveField& CopyParam(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　WaveField オブジェクトのサンプルデータとサンプル点数 $N_x \times N_y$ 以外の全てのパラメータを source からコピーする．

Note

- サンプル点数 $N_x \times N_y$ はコピーされない．サンプル点数変更には SetNx() メンバー関数，SetNy() メンバー関数を用いること．
- データメモリは確保されていない．必ず，直後に Init() メンバー関数を用いてメモリ確保を行う必要がある．

```
WaveField& CopyParamAll(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　WaveField オブジェクトのサンプルデータ以外の (サンプル点数 $N_x \times N_y$ を含む) 全てのパラメータを source からコピーする．

Note

- このメソッドだけではデータメモリは確保されていない．必ず，直後に Init() メンバー関数を用いてメモリ確

保を行う必要がある．

```
WaveField& CopyData(const WaveField& source)
```

戻り値 対象オブジェクトへの参照

説　明 WaveField オブジェクトのサンプルデータのみを source からコピーする．サンプル点数 $N_x \times N_y$ 以外のパラメータはコピーされない．

Note

- サンプル点数 $N_x \times N_y$ はサンプルデータとともにコピーされる．

3.4.4 デフォルト値を設定・取得するメンバー関数

以下の関数を用いるとコンストラクタで引数が省略された際のデフォルト値 (規定値) が設定または取得できる．なお以下の関数はすべて静的メンバー関数であるので，関数名の前に “WaveField::” を付加する必要がある．サンプルを参照．デフォルト値を設定する以前の初期値については，WaveField のコンストラクタ参照．

```
static void SetDefault(long nxy)
static void SetDefault(long nx, long ny)
static void SetDefault(long nx, long ny, double pxy)
static void SetDefault(long nx, long ny, double px, double py)
static void SetDefault(long nx, long ny, double px, double py, double lmd)
```

戻り値 なし

説　明 1 番目の形式では，nxy を N_x と N_y の両方のデフォルト値として設定する．2 番目の形式では，nx と ny をそれぞれ N_x と N_y のデフォルト値として設定する．3 番目の形式では，2 番目に加えて pxy を P_x と P_y の両方のデフォルト値として設定する．4 番目の形式では，2 番目に加えて px と py をそれぞれ P_x と P_y のデフォルト値として設定する．5 番目の形式では，4 番目に加えて lmd を波長のデフォルト値として設定する．

```
static void SetDefaultWavelength(double lmd)
```

戻り値 なし

説　明 lmd を波長のデフォルト値として設定する．

```
static double GetDefaultWavelength(void)
```

戻り値 デフォルト波長の値

説　明 波長のデフォルト値を取得する

```
static void SetDefaultNx(long nx)
static void SetDefaultNy(long ny)
```

戻り値　なし

説　明　それぞれ，nx と ny をサンプル点数 N_x と N_y のデフォルト値として設定する．

```
static long GetDefaultNx(void)
static long GetDefaultNy(void)
```

戻り値　デフォルトのサンプリング数

説　明　それぞれ，サンプル点数 N_x と N_y のデフォルト値を取得する．

```
static void SetDefaultPx(double px)
static void SetDefaultPy(double py)
```

戻り値　なし

説　明　それぞれ，px と py をサンプリング間隔 P_x と P_y のデフォルト値として設定する．

```
static double GetDefaultPx(void)
static double GetDefaultPy(void)
```

戻り値　デフォルトのサンプリング間隔

説　明　それぞれ，サンプリング間隔 P_x と P_y のデフォルト値を取得する．

Example デフォルト値の変更

```
// 長さ1024の1次元分布をデフォルトとする
WaveField::SetDefault(1024, 1);

//wf1, wf2は両方とも1024x1の1次元分布で，サンプリング間隔1e-6x1e-6，波長632.8nm
WaveField wf1, wf2;

// 1024x1024の2次元関数をサンプル数をデフォルトに変更する
WaveField::SetDefault(1024, 1024);
WaveField a;                    // aとbは1024x1024，サンプリング間隔1e-6x1e-6，波長632.8nm
WaveField b;

WaveField::SetDefualtPy(10e-6);
WaveField c(512, 1024);         // cは512x1024，サンプリング間隔1e-6x10e-6，波長632.8nm

WaveField::SetDefualtWavelength(532e-9);
WaveField c, d, e;              // c, d, eは1024x1024，サンプリング間隔1e-6x10e-6，波長532nm
```

3.4.5　オブジェクトの状態を取得するメンバー関数

```
bool IsRealSpace(void) const
bool IsComplexAmplitude(void) const
```

戻り値 判定結果

説　明 対象オブジェクトが実空間にある場合 (従って複素振幅分布である場合) に true になる．二つの関数はどちらも同じ結果を与える．

```
bool IsFourierSpace(void) const
bool IsSpectrum() const
```

戻り値 判定結果

説　明 対象オブジェクトがフーリエ空間にある場合 (従って複素スペクトルである場合) に true になる．二つの関数はどちらも同じ結果を与える．

```
bool IsComplexForm(void) const
```

戻り値 判定結果

説　明 複素形式が直交形式であるばあいに true になる．

Note

- ComplexForm 列挙型参照．
- 内部的に使用されるので使用しない．

```
bool IsPolarForm(void) const
```

戻り値 判定結果

説　明 複素形式が極形式であるばあいに true になる．

Note

- ComplexForm 列挙型参照．
- 内部的に使用されるので使用しない．

3.4.6 オブジェクトのパラメータへアクセスするメンバー関数

```
double GetWavelength(void) const
```

戻り値 波長

説　明 WaveField オブジェクトの波長を取得する．

```
LightWave& SetWavelength(double lambda)
```

戻り値　対象オブジェクトへの参照

説　明　lambda を WaveField オブジェクトの波長として設定する．

```
double GetWavenumber(void) const
```

戻り値　波数

説　明　WaveField オブジェクトの波数を取得する．

```
WaveField& SetWavenumber(double k)
```

戻り値　対象オブジェクトへの参照

説　明　k を WaveField オブジェクトの波数として設定する．

```
float& k(void)
```

戻り値　波数データメンバーへの参照

説　明　WaveField オブジェクトの波数データメンバーへの参照を取得する．左辺値として設定も可能．

Note

- LightWave との互換性確保のための関数．原則として使用しない．

```
long long N(void) const
long long GetN(void) const
```

戻り値　サンプル点の総数

説　明　WaveField オブジェクトのサンプル点の総数を取得する．次項の関数を用いた GetNx()*GetNy() の結果と同じになる．

```
const long& GetNx(void) const
const long& GetNy(void) const
```

戻り値　x 方向または y 方向のサンプル点数

説　明　x 方向または y 方向のサンプル点数を取得する．

```
WaveField& SetNx(long n)
WaveField& SetNy(long n)
```

戻り値 対象オブジェクトへの参照

説　明 n を x 方向または y 方向のサンプル点数として設定する．

Note

- この関数によるサンプル点数設定後は，Init() メンバー関数により，オブジェクトを初期化する必要がある．

Example

```
WaveField wf;    //オブジェクトwfの定義
wf.SetNx(512);   //x方向サンプル点数を512に設定
wf.SetNy(1024); //y方向サンプル点数を1024に設定
wf.Init();       //wfを再初期化
```

```
long& Nx(void)
long& Ny(void)
```

戻り値 サンプル点数への参照

説　明 x 方向または y 方向のサンプル点数を表す int 型データメンバーへの参照を取得する．左辺値として設定も可能．

Note

- LightWave との互換性確保のための関数．原則として使用しない．
- この関数によるサンプル点数設定後は，Init() メンバー関数により，オブジェクトを初期化する必要がある．

```
const double& GetPx(void) const
const double& GetPy(void) const
```

戻り値 サンプリング間隔

説　明 x 方向または y 方向のサンプリング間隔を取得する．

```
WaveField& SetPx(double pitch)
WaveField& SetPy(double pitch)
```

戻り値 対象オブジェクトへの参照

説　明 pitch を x 方向または y 方向のサンプリング間隔として設定する．

Example

```
WaveField wf;        //オブジェクトwfの定義
wf.SetPx(100e-6);    //x方向サンプリング間隔を100umに設定
wf.SetPx(50e-6);     //y方向サンプリング間隔を50umに設定
```

```
double& Px(void)
double& Py(void)
```

戻り値　サンプリング間隔を表す double 型データメンバーへの参照

説　明　x 方向または y 方向のサンプリング間隔への参照を取得する．左辺値として設定も可能．

Note

- LightWave との互換性確保のための関数．原則として使用しない．

3.4.7　サンプル値へのアクセスするメンバー関数

```
Complex& operator[](int i)
```

戻り値　サンプル値への参照

説　明　オーバーロードされたオペレータ．1 次元配列で表した i 番目のサンプル値への参照を取得する．

Example

```
WaveField wf;
Complex a;
a = wf[10];              //10番目のサンプル値をaに代入
wf[15]=Comp(1.0, 0.5);   //15番目のサンプル値として，1.0+0.5iを代入
```

```
Complex GetValue(int i, int j) const
Complex GetPixel(int i, int j) const【廃止予定】
```

戻り値　Complex 型のサンプル値

説　明　整数座標 (i, j) のサンプル値を取得する．

Example

```
WaveField wf;
Complex c = wf.GetValue(10, 15);     //(10,15)のサンプル値を取得
```

```
WaveField& SetValue(int i, int j, Complex val)
WaveField& SetPixel(int i, int j, Complex val)【廃止予定】
```

戻り値　対象オブジェクトへの参照

説　明　整数座標 (i, j) に Complex 型サンプル値 val を設定する．

Example

```
WaveField wf;
wf.SetValue(10, 15, Complex(1.0, 0.5);  //(10,15)のサンプル値として，1.0+0.5iを代入
```

```
Complex& Value(int i, int j)
Complex& Pixel(int i, int j)【廃止予定】
```

戻り値 Complex 型サンプル値への参照

説　明 整数座標 (i, j) のサンプル値への参照を返す．左辺値として代入も可能．

Note

- LightWave との互換性確保のための関数．原則として使用しない．

Example

```
WaveField wf;
wf.Value(10, 15) = Complex(1.0, 0.5);  //(10,15)のサンプル値として，1.0+0.5iを代入
```

```
double GetReal(int i, int j) const
double GetImag(int i, int j) const
```

戻り値 サンプル値の実部または虚部の値

説　明 整数座標 (i, j) のサンプル値の実部または虚部の値を取得する．

Example

```
WaveField wf;
double re = wf.GetReal(10, 50);   //(10,50)の実部を取得
double im = wf.GetImag(10, 50);   //(10,50)の虚部を取得
```

```
WaveField& SetReal(int i, int j, double val)
WaveField& SetImag(int i, int j, double val)
```

戻り値 対象オブジェクトへの参照

説　明 整数座標 (i, j) のサンプル値の実部または虚部に val を設定する．

Example

```
WaveField wf;
wf.SetReal(10, 50, 0.53);   //(10,50)の実部に0.53を設定
wf.SetImag(10, 50, 0.0);    //(10,50)の虚部に0を設定
```

```
float& Real(int i, int j)
float& Imag(int i, int j)
```

戻り値　サンプル値の実部または虚部への参照

説　明　整数座標 (i, j) のサンプル値の実部または虚部への参照を取得する．左辺値として設定も可能．

Note

- LightWave との互換性確保のための関数．原則として使用しない．

Example

```
float g = wf.Real(10, 50);   //(10,50)の実部を読み取り
wf.Real(23, 1) = 10 * g;     //(32,1)の実部に書き込み
wf.Imag(23, 1) *= 3;         //(23,1)の虚部を3倍にする．
```

```
double GetAmplitude(int i, int j)
```

戻り値　振幅値

説　明　整数座標 (i, j) のサンプル値の振幅値を取得する．

```
void SetAmplitude(int i, int j, double val)
```

戻り値　なし

説　明　整数座標 (i, j) のサンプル値の振幅値 val を設定する．ただし，位相は以前の値のまま変化しない．

```
Phase GetPhase(int i, int j)
```

戻り値　Phase 型の位相値

説　明　整数座標 (i, j) のサンプル値の位相角を取得する．

Note

- 位相角の範囲は $-\pi \sim +\pi$ である．

```
void SetPhase(int i, int j, double val)
```

戻り値　なし

説　明　整数座標 (i, j) のサンプル値の位相角 val を設定する．ただし，振幅は以前の値のまま変化しない．

```
double GetIntensity(int i, int j)
```

戻り値 光強度 (複素値の絶対値の 2 乗)

説　明 整数座標 (i, j) のサンプル値の光強度 (複素値の絶対値の 2 乗) を取得する.

```
void SetIntensity(int i, int j, double val)
```

戻り値 なし

説　明 整数座標 (i, j) のサンプル値の光強度 (複素値の絶対値の 2 乗) を設定する. ただし, 位相は以前の値のまま変化しない.

3.4.8 サンプル値へ定数を設定するメンバー関数

```
WaveField& SetConst(const Complex& c)
WaveField& SetConstReal(double a)
WaveField& SetConstImag(double b)
WaveField& SetConstAmplitude(double A)
WaveField& SetConstPhase(double p)
```

戻り値 対象オブジェクトへの参照

説　明 対象オブジェクトのすべてのサンプル値に対して同じ定数を設定する.

```
WaveField& SetConstWin(const Complex& c)
WaveField& SetConstWinReal(double a)
WaveField& SetConstWinImag(double b)
WaveField& SetConstWinAmplitude(double A)
WaveField& SetConstWinPhase(double p)
```

戻り値 対象オブジェクトへの参照

説　明 対象オブジェクトのウィンドウ領域内のサンプル値に対して同じ定数を設定する.

3.4.9 ローカルな物理座標と整数座標に関するメンバー関数

```
double X(int i) const
```

戻り値 x 座標値

説　明 整数座標の i を物理座標の x 座標値に変換する. i が有効な範囲にあるかどうかはチェックされない.

```
double Y(int j) const
```

戻り値　y 座標値

説　明　整数座標の j を物理座標の y 座標値に変換する．j が有効な範囲にあるかどうかはチェックされない．

```
int I(double x) const
```

戻り値　i 座標値

説　明　物理座標の x 座標値を整数座標の i に変換する．x が有効な範囲にあるかどうかはチェックされない．

```
int J(double y) const
```

戻り値　j 座標値

説　明　物理座標の y 座標値を整数座標の j に変換する．y が有効な範囲にあるかどうかはチェックされない．

```
int IJ(int i, int j) const
```

戻り値　1 次元データ配列のインデックス

説　明　整数座標の i と j から，1 次元データ配列のインデックスを計算する．

```
double GetMaxX(void) const
double GetMinX(void) const
double GetMaxY(void) const
double GetMinY(void) const
```

戻り値　物理座標 (ローカル座標) の有効な最大値または最小値

説　明　対象オブジェクトのローカルな物理座標の有効な最大値または最小値を取得する．

```
double GetWidth(void) const
```

戻り値　サンプリング領域の x 方向の物理的な幅 (単位:m)

説　明　対象オブジェクトのローカルな物理 x 座標の最大値から最小値を減算し，対象オブジェクト物理的な幅を求める．

```
double GetHeight(void) const
```

戻り値 サンプリング領域の y 方向の物理的な高さ (単位:m)

説　明 対象オブジェクトのローカルな物理 y 座標の最大値から最小値を減算し，対象オブジェクト物理的な高さを求める．

```
double GetMaxX(void) const
double GetMinX(void) const
double GetMaxY(void) const
double GetMinY(void) const
```

戻り値 物理座標 (ローカル座標) の有効な最大値または最小値

説　明 対象オブジェクトのローカルな物理座標の有効な最大値または最小値を取得する．

3.4.10 ローカル座標のシフトや画像的回転に関するメンバー関数

```
WaveField& ShiftZeroFill(Axis axis, double dxy)
WaveField& ShiftZeroFill(double dx, double dy)
```

戻り値 対象オブジェクトへの参照

説　明 ローカル座標の x 軸方向または y 軸方向にローカル座標の原点をシフトする．従ってオブジェクト内の複素振幅分布はその逆方向に移動したように見える．移動した結果，空白となった部分はゼロ値で埋められる．1 番目の形式ではシフトする方向を Axis 列挙型 axis で指定し，距離 dxy シフトする．2 番目の形式では x 方向に dx，y 方向に dy シフトする．

```
WaveField& ImageRotation(int n)
```

戻り値 対象オブジェクトへの参照

説　明 オブジェクトを複素画像として右回りに 90 度 (n=1)，180 度 (n=2)，または 270 度 (n=3) 回転する．

Note

- この関数は Rotate() メンバー関数や RotateFs() メンバー関数と異なり，単純に象限を回転させる．

```
WaveField& SwitchQuadrant(void)
```

戻り値 対象オブジェクトへの参照

説　明 第 1 象限 ⇔ 第 3 象限間，第 2 象限 ⇔ 第 4 象限間の象限の入れ替えを行う．

Note

- 通常の FFT 計算パッケージ等と象限の位置を合わせるための入れ替えを行う．離散フーリエ変換や離散畳み込みの処理で用いる．

```
WaveField& Transpose(bool interval = true)
```

戻り値　対象オブジェクトへの参照

説　明　x 軸方向と y 軸方向のサンプル点を入れ替える転置を行う．interval=true の場合はデータだけではなく，サンプリング間隔も交換する．

3.4.11　ウィンドウを操作するためのメンバー関数

```
WFL_RECT& Window(void)
```

戻り値　ウィンドウ領域を保持している WFL_RECT 構造体への参照．

説　明　オブジェクト内にはウィンドウ領域を保持している WFL_RECT 構造体がある．この関数はその WFL_RECT 構造体へのアクセスを提供する．使用法は以下を参照．

Example

```
WaveField wf(64, 64);

/* ウィンドウ領域の設定 */
wf.Window().left = 10;
wf.Window().right = 20;
wf.Window().top = 10;
wf.Window().bottom = 5;

/* ウィンドウ領域の読み取り例 */
int width = wf.Window().right - wf.Window().right;
int height = wf.Window().top - wf.Window().bottom;
```

```
WaveField& SetWindow(const WFL_RECT& r)
```

戻り値　対象オブジェクトへの参照

説　明　WFL_RECT 構造体 r で r.left, r.right, r.top, r.bottom として，ウィンドウの領域を設定する．WFL_RECT 構造体については，MSDN ライブラリを参照．

Example

```
WaveField wf(64, 64);
WFL_RECT r;
r.left =10; r.right 20; r.top = 10; r.bottom = 5;
wf.SetWindow(r);
```

```
WaveField& SetWindow(int left, int right, int bottom = 0, int top = 0)
```

戻り値　対象オブジェクトへの参照

説　明　(i, j) 整数座標系で，ウィンドウ領域を直接設定する．

```
WaveField& SetWindow(double left, double right, double bottom = 0.0, double top = 0.0)
```

戻り値　対象オブジェクトへの参照
説　明　(x, y) 物理座標系で，ウィンドウ領域を直接設定する．

```
WaveField& SetWindowMax(void)
```

戻り値　対象オブジェクトへの参照
説　明　ウィンドウ領域を最大化し，オブジェクト全体の領域と等しく設定する．

```
WaveField& SetWindowCommon(WaveField& signal)
```

戻り値　対象オブジェクトへの参照
説　明　対象オブジェクトと signal オブジェクトのウィンドウの共通領域を求め，それをオブジェクトの新たな領域として設定する．

```
WaveField& TransferWindow(WaveField& source)
```

戻り値　対象オブジェクトへの参照
説　明　source オブジェクトから対象オブジェクトにウィンドウ領域を転送する．

Note

- この関数はウィンドウ領域内のデータを転送するのではなく，ウィンドウ領域の位置情報のみを対象オブジェクトに転送する．

3.4.12　単純算術演算子

以下の演算子では，サンプル値間の算術演算を単純に行う．グローバル空間内ので WaveField オブジェクトの位置はチェックされない．グローバル空間内の位置を考慮した演算は位置依存演算のためのメンバー関数を用いる．

```
WaveField& operator=(const WaveField& wf)
WaveField& operator*=(const WaveField& wf)
WaveField& operator/=(const WaveField& wf)
WaveField& operator+=(const WaveField& wf)
WaveField& operator-=(const WaveField& wf)
```

戻り値　対象オブジェクトへの参照

説　明　WaveField オブジェクトでオーバーロードされている 2 項演算子．いずれも，C 言語の標準的代入演算子と同様の動作を二つのオブジェクトのサンプリングデータ同士すべてに対して行う．

Note

- 二つの WaveField オブジェクトのサンプル点数 $N_x \times N_y$ が等しくない場合はエラーとなり，エラーメッセージをコンソールに出力してプログラムを終了する．
- 二つの WaveField オブジェクトのサンプリング間隔や波長はチェックされない．サンプリング間隔や波長が等しくなくてもエラーにならない．
- 代入演算子でない 2 項演算子は提供されない．これは不必要なメモリ消費を抑えるためである．

Example

```
WaveField wf1, wf2;
wf1 *= wf2;          // wf1とwf2の各サンプルデータ同士をすべて乗算し，wf1に代入する
```

```
WaveField& operator*=(double val)
WaveField& operator*=(const ComplexDouble& val)
WaveField& operator/=(double val)
WaveField& operator/=(const ComplexDouble& val)
WaveField& operator+=(const ComplexDouble& val)
WaveField& operator-=(const ComplexDouble& val)
```

戻り値　対象オブジェクトへの参照

説　明　いずれも，各オブジェクトのサンプリングデータすべてに対して，double 型や ComplexDouble 型の値 val を標準の C 演算子と同様に演算する．

Example

```
WaveField wf;
wf *= Complex(3.0, 2.0);     // wfのサンプルデータすべてに3.0+2.0iを乗算する
```

3.4.13　ローカル座標に関する情報を取得・設定するためのメンバー関数

```
WaveField& SetNormalVector(Vector v)
```

戻り値　対象オブジェクトへの参照

説　明　Vector 型のベクトル v をオブジェクトの法線ベクトルとして設定する．

```
Vector GetNormalVector(void) const
```

戻り値 Vector 型の法線ベクトル
説　明 オブジェクトの法線ベクトルを取得する．

```
void SetCenter(Point p)
void SetOrigin(Point p)
```

戻り値 無し
説　明 フィールドの中心を Point 型の点 p に設定する．

```
Point GetCenter(void) const
Point GetOrigin(void) const
```

戻り値 Point 型の点オブジェクト
説　明 フィールドの中心点座標を取得する．

```
Vector GetUnitVectorX(void) const
Vector GetUnitVectorY(void) const
```

戻り値 Vector 型の単位ベクトル
説　明 それぞれローカル x 軸とローカル y 軸の単位ベクトルを取得する．

```
RMatrix GetCRMatrixLG(void) const
```

戻り値 RMatrix 型の回転行列
説　明 対象オブジェクトがグローバル座標と平行になるように座標回転するための回転行列を取得する．

```
RMatrix GetCRMatrixGL(void) const
```

戻り値 RMatrix 型の回転行列
説　明 グローバル座標と平行なある座標系が，対象オブジェクトのローカル座標と並行になるように座標回転するための回転行列を取得する．

```
Point GetLocalPosition(Point p) const
```

戻り値　ローカル座標値
説　明　グローバル座標で表された Point 型の点 p のローカル座標値を取得する．

```
Point GetGlobalPosition(Point p) const
```

戻り値　グローバル座標値
説　明　ローカル座標で表された Point 型の点 p のグローバル座標値を取得する．

```
Plane GetPlane(void) const
```

戻り値　Plane 型の平面オブジェクト
説　明　対象オブジェクトのサンプリング平面を含む平面オブジェクトを取得する．

```
Line GetOpticalAxis(void) const
```

戻り値　Line 型の直線オブジェクト
説　明　対象オブジェクトのローカル原点を通りサンプリング平面に垂直な光軸を取得する．

```
PointArray GetFrame(int n = 0) const
```

戻り値　PointArray 型の点群
説　明　対象オブジェクトのサンプリング領域 (フレーム) を取得する．n = 0 の場合にはフレームの四角を取得する，n ≥ 1 の場合には，フレームの各辺につき 2^n 個の点が得られる．

Note

- この関数はオブジェクトが実空間にあるかフーリエ空間にあるかで多少動作が異なる．
- オブジェクトが実空間にある場合は，ローカル原点や法線ベクトルから求めたグローバル空間中でのフレームが得られる．
- オブジェクトがフーリエ空間にある場合は，各点の x 値と y 値はそれぞれ単位 [1/m] のフーリエ周波数 u と v であり，z 値は式 (3.10) から求めた w が代入される．また，ローカル原点や法線ベクトルは考慮されない．

3.4.14　位置依存演算のためのメンバー関数

以下の演算子では，グローバル空間内ので WaveField オブジェクトの位置に依存した演算を行う．従って，一般に同一平面上に無いオブジェクト同士の演算はエラーとなる．グローバル空間内での位置に無関係にサンプル点同士の演算を行う場合はオーバーロード演算子を用いる．

```
WaveField& Add(const WaveField& wf)
WaveField& Multiply(const WaveField& wf)
```

戻り値 対象オブジェクトへの参照

説　明 それぞれ，対象オブジェクトに別のオブジェクト wf を加算または乗算する．この時，これらの関数では補間を用いずもっとも近いサンプル点同士を演算する．そのため，高速に演算されるが最大でサンプリング間隔の 2 分の 1 に等しい位置ずれが生じる場合がある．

Note

- 対象オブジェクトと source オブジェクトの波長は一致していなければならない．
- 対象オブジェクトと wf オブジェクトのサンプリング間隔は一致しなければならない．一致しない場合エラーとなる．サンプリング間隔が一致しないオブジェクトの演算の行う場合は ResamplingAdd() メンバー関数や ResamplingMultiply() メンバー関数を用いる．
- 対象オブジェクトと wf **オブジェクト**はグローバル空間内で同一の平面上になければならない．同一の平面上でない場合エラーとなる．
- 同一平面であっても二つオブジェクトに重なりが無い場合，Mutiply() では対象オブジェクトの全サンプル点が 0 となり，Add() では演算前と同一の結果になる．

```
WaveField& ResamplingAdd(const WaveField& wf, Interpol ip = CUBIC8)
WaveField& ResamplingMultiply(const WaveField& wf, Interpol ip = CUBIC8)
```

戻り値 対象オブジェクトへの参照

説　明 それぞれ，対象オブジェクトに別のオブジェクト wf を加算または乗算する．この時，これらの関数では Interpol 列挙型引数 ip で指定する補間法を用いて再サンプリングを行う．そのため，計算速度は Add() メンバー関数や Multiply() メンバー関数ほど速くないが，位置ずれは生じない．

Note

- 対象オブジェクトと source オブジェクトの波長は一致していなければならない．
- 対象オブジェクトと wf オブジェクトのサンプリング間隔が一致している必要はない．ただし，サンプリング間隔が一致している場合は Add() メンバー関数や Multiply() メンバー関数の方が処理が高速である．
- 対象オブジェクトと wf オブジェクトはグローバル空間内で同一の平面上になければならない．同一の平面上でない場合エラーとなる．
- 同一平面であっても二つオブジェクトに重なりが無い場合，Multiply() メンバー関数や Add() メンバー関数と同様の結果になる．
- これらの関数では補間に伴う誤差が生じる場合がある．

```
WaveField& ResamplingCopy(const WaveField& source, Interpol ip = CUBIC8, bool clear =
true)
```

戻り値　対象オブジェクトへの参照

説　明　対象オブジェクトに source オブジェクトをコピーする．この時 Interpol 列挙型引数 ip で指定する補間法を用いる．そのため，サンプリング間隔が異なっていてもよい．clear=true の時，対象オブジェクトと source の共通部分以外はゼロクリアされる．

Note

- 対象オブジェクトのサンプリング数やサンプリング間隔は変化しない．
- 対象オブジェクトと source オブジェクトの波長は一致していなければならない．
- 対象オブジェクトと source オブジェクトのサンプリング間隔が一致している必要はない．
- 対象オブジェクトと source オブジェクトはグローバル空間内で同一の平面上になければならない．同一の平面上でない場合エラーとなる．
- clear=true の場合，同一平面であっても二つオブジェクトに重なりが無い場合，対象オブジェクトの全サンプル点が 0 となる．
- この関数では補間に伴う誤差が生じる場合がある．

3.4.15　ファイルのロード・セーブのためのメンバー関数

```
void SaveAsGrayBmp(const char* fname, Mode mode, int depth = 8, double gamma = 2.2)
const
```

戻り値　なし

説　明　ファイル名が fname でピクセルビット数が depth のグレイスケール BMP 形式ファイルを作成し，フィールドのサンプル値をセーブする．関数終了時にファイルは閉じられる．Mode 列挙型 mode がフィールドをセーブするときのモードを指定する．モードとして，光強度・位相・振幅・実部・虚部などの指定が可能．セーブ時にはガンマ値 gamma でガンマ補正処理が行われる．

Note

- この関数では，SaveAsBmp() メンバー関数とは異なり，ガンマ補正が行われる．そのため，INTENSITY モードでセーブした画像が自然な像として見える．
- ピクセルビット数 depth として，1, 4, 8 ビットが指定でき，depth = 1 ではバイナリ画像になる．これら以外を設定した場合はエラーとなる．
- モード毎の変換は表 3.1 で与えらえる．
- オブジェクトのサンプリング間隔 P_x と P_y は BMP ファイルに解像度として設定される．

```
void SaveAsBmp(const char* fname, Mode mode, Gradation cs = GRAY) const
void SaveAsBmp(const char* fname, Mode mode, Gradation cs, int cw) const
void SaveAsBmp(const char* fname, Mode mode, Gradation cs, int cw, double max, double
min = 0.0) const
```

戻り値 なし

説　明 ファイル名 fname の 8 ビットグレイスケール BMP 形式ファイルを作成し，オブジェクトのサンプル値をセーブする．関数終了時にファイルは閉じられる．BMP に複素振幅そのものを保存することはできないため，WaveField オブジェクトをセーブするときのモードを Mode 列挙型 mode として指定しなければならない．モードとして，光強度・位相・振幅・実部・虚部などの指定が可能．これらの分布は画像のグラデーションとして Gradation 列挙型 cs の指定によってグレイスケールまたはカラースケールとして保存される．1 番目の形式で cs を指定しない場合は，デフォルトでチャート無しのグレイスケールで保存される．cs として COLOR を指定した場合には，カラーチャート付きのカラースケールで保存される．2 番目の形式ではチャートの幅 cw がピクセル単位で指定できる．3 番目の形式ではさらにグラデーションへの変換の上限・下限を指定できるので，特定の分布を強調表示できる．

Note

- この関数では，SaveAsGrayBmp() メンバー関数とは異なり，ガンマ補正が行われない．そのため，INTENSITY モードでセーブした画像が自然な像として見えないことに注意．
- デフォルトでは，Gradation 列挙型 cs = GRAY の場合にはチャート無し．cs = COLOR の場合には画像の右端に 10 ピクセル幅のチャートが示される．
- min あるいは max 引数が指定されている場合は，min 値以下をレベル 0(黒)，max 値以上をレベル 255(白) になるように変換する．
- min，max 引数が指定されていない場合の変換は表 3.1 となる．
- オブジェクトのサンプリング間隔 P_x と P_y は BMP ファイルに解像度として設定される．

表 3.1　SaveAsBmp() におけるデフォルトのグラデーション

Mode 列挙型	表示範囲 (レベル 0〜255 に対応する範囲)	備考
INTENSITY	0.0 ～ +1.0	強度変換後にスケーリング
AMPLITUDE	0.0 ～ +1.0	振幅変換後にスケーリング
PHASE	$-\pi$ ～ $+\pi$	
REAL	−1.0 ～ +1.0	
IMAGINARY	−1.0 ～ +1.0	

Example

```
WaveField wf(1024, 1024, 10e-6);
wf.SetGaussian(5e-3);
wf.SaveAsBmp("Gaussian1.bmp", PHASE);    //位相分布をグレイスケールで保存
wf.SaveAsBmp("Gaussian2.bmp", AMPLITUDE, COLOR, 0);
                                         //振幅分布をカラーチャート無しのカラースケールで保存
```

```
void WinSaveAsGrayBmp(const char* fname, Mode mode, int depth = 8, double gamma = 2.2)
const
```

戻り値　なし

説　明　ウィンドウ領域内をグレイスケール BMP ファイルとしてセーブする．引数等は SaveAsGrayBmp() メンバー関数と同じ．

```
void WinSaveAsBmp(const char* fname, Mode mode, Gradation cs = GRAY) const
void WinSaveAsBmp(const char* fname, Mode mode, Gradation cs, int cw) const
void WinSaveAsBmp(const char* fname, Mode mode, Gradation cs, int cw, double max, double
min = 0.0) const
```

戻り値　なし

説　明　ウィンドウ領域内を BMP ファイルとしてセーブする．引数等は SaveAsBmp() メンバー関数と同じ．

```
void SaveAsRgbBmp(const char* fname, const WaveField& green, const WaveField& blue,
const Mode mode, double min = 0.0, double max = 0.0) const
```

戻り値　なし

説　明　対象オブジェクトを赤プレーン，green を緑プレーン，blue を青プレーンとし，Mode 列挙型 mode のモードでファイル名 fname の 32 ビットカラー BMP 形式ファイルとして光波をセーブする．モードとして，SaveAsBmp() メンバー関数と同様，光強度・位相・振幅・実部・虚部などの指定が可能．これらの分布は各色のレベルとして保存される．

Note

- 対象オブジェクト (赤プレーン)，green，bule の各フィールドのサンプリング数が異なる場合はエラーとなる．
- 対象オブジェクト (赤プレーン)，green，bule の各フィールドの波長，サンプリング間隔等その他のすべてのパラメータは無視されるため，異なっていても良い．
- min あるいは max 引数が指定されている場合は，min 値以下がレベル 0，max 値以上がレベル 255 になるように変換する．
- min，max 引数が指定されていない場合の変換は表 3.1 と同様になる．
- 対象オブジェクトのサンプリング間隔 P_x と P_y が BMP ファイルに解像度として設定される．

```
void WinSaveAsRgbBmp(const char* fname, const WaveField& green, const WaveField& blue,
const Mode mode, double min = 0.0, double max = 0.0) const
```

戻り値　なし

説　明　対象オブジェクトを赤プレーン，green を緑プレーン，blue を青プレーンとし，Mode 列挙型 mode のモードでファイル名 fname の 32 ビットカラー BMP 形式ファイルとしてウィンドウ領域内の光波をセーブする．基本的な機能は SaveAsRgbBmp() メンバー関数と同じである．

```
void SaveAsSegBmp(const char* fname, Mode mode, int mx, int my, Gradation cs = GRAY)
const
```

戻り値 なし

説 明 対象オブジェクトをファイル名 fname，セグメント数 mx×my，グラデーション Gradation 列挙型 cs の分割 BMP ファイルとして保存する．基本的な機能は SaveAsBmp() メンバー関数と同じである．

Note

- グラデーションを cs = COLOR とした場合もカラーチャートは表示されない．
- 分割 BMP ファイルのファイル名形式については SegWaveField クラスの SaveAsSegBmp() メンバー関数の *Note* を参照．
- この関数では画像ガンマの補正をしないため，ガンマは 1 である．

```
WaveField& LoadBmp(const char* fname, Mode mode, Complex backg = Comp(0, 0), double
gamma = 1.0, ColorMode cm = GRAY_SCALE)
```

戻り値 対象オブジェクトへの参照

説 明 ファイル名 name の BMP 形式ファイルをロードする．BMP は複素振幅では無いため，WaveField オブジェクトにロードするときのモードを Mode 列挙型の mode 引数で指定しなければならない．光強度・位相・振幅・実部・虚部などへの変換が可能．backg はオブジェクトとぴったり一致しないサイズの画像を読み込んだ際，背景になるサンプル点の複素数値である．gamma は読み込む画像のガンマ値を指定する．ColorMode 列挙型の cm として GRAY_SCALE を指定すると，カラー画像を 8 ビットグレイスケール画像に変換した上で読み込む．それ以外の RED，GREEN，BLUE 等 を指定するとその 8 ビットカラープレーンを読み込む．

Note

- WFL Rel 3.5 より引数 gamma が追加された．それ以前の WFL を用いたプログラムは再コンパイルが必要な場合がある．
- 写真画像などガンマ補正された画像を読み込む場合は gamma として 2.2 を設定する．
- BMP ファイルの形式
  2 ビット，4 ビット，8 ビットのカラーテーブル付画像，また 24 ビット，32 ビットのフルカラー画像を読み込むことができる．ただし，圧縮されているものは読み込めない．
- サンプル点数
  この関数の適用後は，適用された WaveField オブジェクトのサンプル点数は，読み込む BMP ファイルのピクセル数で置き換えられ，データ領域もそれに応じて自動的に拡大/縮小される．従って，BMP をロードするオブジェクトのサイズは，あらかじめ BMP のピクセル数と一致しておく必要は無く，任意のサイズでよい．
- 解像度
  BMP ファイルの解像度は無視される．すなわち，BMP ファイルロード後も WaveField オブジェクトのサンプリング間隔はロード以前の値が保持されている．
- BMP 画像のサイズが 2 の累乗ではない場合
  その画像が格納できる最小の 2 の累乗サイズになるようにオブジェクトが自動的に更新される．

- サンプル値への変換
  BMP ファイルのピクセルビット数に応じて，表 3.2 の範囲を WaveField オブジェクトのサンプル値に変換する．
- 既定の引数
  backg と cm を省略した場合，既定では背景色黒のグレイスケール画像として読み込む．

表 3.2　LoadBmp() 関数で読み込まれる画像のピクセル値からサンプル値への変換

ピクセル深さ	ピクセル値	
8	$0 \sim 255$	
4	$0 \sim 16$	
2	$0, 1$	
Mode	サンプル値の範囲	備考
INTENSITY	$0.0 \sim +1.0$	実部に代入．虚部は 0
AMPLITUDE	$0.0 \sim +1.0$	実部に代入．虚部は 0
PHASE	$-\pi \sim +\pi$	振幅は 1 で一定
REAL	$-1.0 \sim +1.0$	虚部は 0 で一定
IMAGINARY	$-1.0 \sim +1.0$	実部は 0 で一定

Example

```
WaveField wf, r, g, b;
wf.LoadBmp("photo.bmp", INTENSITY);                    //グレイスケール画像に変換して強度として読み込む
r.LoadBmp("photo.bmp", INTENSITY, 0, RED);      //赤プレーンを強度として読み込む
g.LoadBmp("photo.bmp", INTENSITY, 0, GREEN);    //緑プレーンを強度として読み込む
b.LoadBmp("photo.bmp", INTENSITY, 0, BLUE);     //青プレーンを強度として読み込む
```

```
void SaveAsCsv(const char* fname, Axis axis = X_AXIS, int ij = 0) const
```

戻り値　なし

説　明　整数座標の ij で指定した x 軸方向または y 軸方向の 1 列のサンプル値を CSV 形式テキストとしてファイル名 fname のファイルにセーブする．セーブされたファイルは，直接 Excel で読み込んでグラフ化が可能である．Axis 列挙型の axis 引数はセーブする方向を指定するために用いられる．デフォルトでは x 軸方向に沿った $j = 0$ がセーブされる．なお，各列の最初の行には，データ値の意味を示すラベル文字列が付加される．

Note

- セーブされるのは，ウィンドウ領域範囲内のみである．全体をセーブしたい場合はこの関数の呼び出し前に，SetWindowMax() メンバー関数を呼び出すこと．

Example SaveAsCsv() コードサンプル

```
#include    <wfl.h>
using namespace wfl;
void main(void)
{
    Start();
    WaveField a(512, 512, 10e-6);
    a.Clear();
    a.SetConstAmplitude(1.0);        // 振幅を1.0に設定
```

```
    a.SetQuadraticPhase(200e-3);     // 焦点距離200mmのレンズ位相(2次位相)を設定
    a.SaveAsCsv("Lens1.csv");        // x軸方向にj=0の一列のデータを保存
    a.SaveAsCsv("Lens2.csv", Y_AXIS, a.J(0.0));
                                     // y軸方向にx=0.0(j=Nx/2)の一列のデータを保存
}
```

```
void SaveAsWf(const char* fname) const
```

戻り値 無し

説　明 対象オブジェクトの複素振幅分布やそれに付随するパラメータ (波長，サンプリング平面の法線ベクトル，サンプリング領域の中心座標その他) をファイル名 fname の WF 形式ファイルとして保存する．

Note

- WF 形式は WaveField の標準的な光波ファイル形式である．
- WF 形式は古い LW 形式に比べてより多くのパラメータを保存できる．
- マルチパート WF 形式や分割 WF 形式などのマルチセグメントのバリエーションがある．

```
void WinSaveAsWf(const char* fname) const
```

戻り値 無し

説　明 ウィンドウ領域内をファイル名 fname の WF 形式ファイルとして保存する．

```
WaveField& LoadWf(const char* fname)
```

戻り値 対象オブジェクトへの参照

説　明 ファイル名 fname の WF 形式ファイルを対象オブジェクトに読み込む．

```
void SaveAsMpWf(const char* fname, unsigned short int ix) const
```

戻り値 無し

説　明 ファイル名 fname のマルチパート WF 形式ファイルのインデックス ix に対象オブジェクトを保存する．

Note

- マルチパート WF 形式は，複数の光波を一つのファイルとしてまとめて保存する形式である．光波は 1 次元インデックスで管理される．
- マルチパート形式では波長，サンプリング数などの光波パラメータは個別に保持されている．
- マルチパート形式のファイルは，保存する前にまず CreateFileMpWf() 関数を用いてファイルの作成を行わなければならない．CreateFileMpWf() 関数で作成されていないファイルに保存することはできない．
- マルチパート形式のファイルに存在する光波の数は，GetSizeMpWf() 関数で取得できる．
- マルチパート形式のファイルに存在する光波のパラメータは，LoadParamMpWf() メンバー関数で読み込むことができる．

```
WaveField& LoadMpWf(const char* fname, unsigned short int ix, bool* exist = NULL)
```

戻り値　対象オブジェクトへの参照

説　明　ファイル名 fname のマルチパート WF 形式ファイルのインデックス ix の光波を対象オブジェクトに読み込む．bool 型変数のアドレス exist を指定した場合，ix の光波がファイル内に存在しない時には関数終了後*exist == false となっている．

Note

- *exist == false の場合，対象オブジェクトの内容は変化しない．

```
WaveField& LoadParamMpWf(const char* fname, unsigned short int ix, bool* exist = NULL)
```

戻り値　対象オブジェクトへの参照

説　明　ファイル名 fname のマルチパート WF 形式ファイルのインデックス ix の光波のパラメータのみを対象オブジェクトに読み込む．bool 型変数のアドレス exist を指定した場合，ix の光波がファイル内に存在しない時には関数終了後*exist == false となっている．

Note

- 光波の複素振幅自体は読み込まれない．

```
void SaveAsSegWf(const char* fname, unsigned short int ii, unsigned short int jj) const
```

戻り値　無し

説　明　ファイル名 fname の分割 WF 形式ファイルのインデックス (ii, jj) に対象オブジェクトを保存する．

Note

- 分割 WF 形式 (Segmented WF 形式) は，大きな光波をセグメントに分割して保存する形式である．光波は左下を (0,0) とする 2 次元のインデックスで管理される．
- 分割 WF 形式では，波長，サンプリング数などの光波パラメータは各セグメントで同一である．
- 分割 WF 形式のファイルは，保存する前にまず CreateFileSegWf() 関数を用いてファイルの作成を行わなければならない．CreateFileSegWf() 関数で作成されていないファイルに保存することはできない
- 分割 WF 形式のファイルに存在する光波の数は，GetSizeSegWf() 関数で取得できる．
- 分割 WF 形式のファイルに存在する光波のパラメータは，LoadParamSegWf() メンバー関数で読み込むことができる．
- 最初のセグメントを保存する際にパラメータがセーブされる．二つ目以降のセグメントの保存時のパラメータがそれ以前のパラメータと異なる場合はエラーになる．
- 上記のパラメータチェックでは，対象のオブジェクトの原点位置はチェックされない．

```
void SaveParamSegWf(const char* fname) const
```

戻り値 対象オブジェクトへの参照

説　明 ファイル名 fname の分割 WF 形式ファイルに対象オブジェクトのパラメータを書き込む.

Note

- すでにファイル fname に設定されているパラメータは上書きされる.
- 光波の複素振幅自体はセーブされない.
- 分割 WF 形式のファイルは，保存する前にまず CreateFileSegWf() 関数を用いてファイルの作成を行わなければならない．CreateFileSegWf() 関数で作成されていないファイルに保存することはできない

```
WaveField& LoadSegWf(const char* fname, unsigned short int ii, unsigned short int jj,
bool* exist = NULL)
```

戻り値 対象オブジェクトへの参照

説　明 ファイル名 fname の分割 WF 形式ファイルのインデックス (ii, jj) の光波を対象オブジェクトに読み込む．bool 型変数のアドレス exist を指定した場合，インデックス (ii, jj) の光波がファイル内に存在しない時には関数終了後*exist == false となっている.

Note

- *exist == false の場合，対象オブジェクトの内容は変化しない.

```
WaveField& LoadParamSegWf(const char* fname, unsigned short int ii, unsigned short int
jj, bool* exist = NULL)
```

戻り値 対象オブジェクトへの参照

説　明 ファイル名 fname の分割 WF 形式ファイルのインデックス (ii, jj) の光波のパラメータのみを対象オブジェクトに読み込む．bool 型変数のアドレス exist を指定した場合，インデックス (ii, jj) の光波がファイル内に存在しない時には関数終了後*exist == false となっている.

Note

- インデックス (ii, jj) の光波がファイル内に存在しない場合でもパラメータが読み出される.
- 光波の複素振幅自体は読み込まれない.

```
void SaveAsLw(const char* fname, OptionHead* ohh = NULL) const
```

戻り値 なし

説　明 オブジェクトをファイル名 fname の LW 形式ファイルとしてセーブする．LW 形式と互換性のない WaveField データメンバーは捨てられる．ohh はオプションヘッダ構造体へのポインタであり，NULL でない ohh 引数が与えられる場合は，オプションヘッダを LW ファイル内へ埋め込む.

Note

- LW 形式は古いファイル形式であり後方互換のために残されている．光波を保存する場合は WF 形式で保存する SaveAsWf() メンバー関数を用いる方が良い．
- LW 形式と互換性のない WaveField データメンバーは捨てられる．

```
void WinSaveAsLw(const char* fname, OptionHead* ohh = NULL) const
```

戻り値　なし

説　明　ウィンドウ領域内を LW ファイルとしてセーブする．引数等は SaveAsLw() メンバー関数と同じ．

```
WaveField& LoadLw(const char* fname, OptionHead* ohh = NULL)
```

戻り値　対象オブジェクトへの参照

説　明　LW 形式のファイル名 fname のファイルをロードする．ohh はオプションヘッダ構造体へのポインタであり，NULL でない ohh 引数が与えられ，かつ読み込む LW 形式ファイル内にオプションヘッダが存在する場合には，ohh で指定されるアドレスにオプションヘッダ構造体が読み込まれる．

Note

- LW 形式は古いファイル形式であり後方互換のために残されている．光波は WF 形式で保存し，LoadWf() メンバー関数を用いて読み込む方が良い．
- LW 形式と互換性のない WaveField データメンバーには 0 が設定させる．
- ロードされるファイルのサンプル点数が 2 の累乗ではない場合，そのファイルデータが格納できる最小の 2 の累乗サイズになるようにオブジェクトが自動的に更新される．
- ohh で指定されるメモリ領域に，実際オプションヘッダを読み込むための領域が確保されているかどうかはこの関数内ではチェックしない．したがって，この関数の呼び出し時には，呼び出し側でオプションヘッダに十分なメモリを確保しなければならない．

```
void SaveAsText(const char* fname, int j = 0) const
```

戻り値　対象オブジェクトへの参照

説　明　整数座標の j で指定した行のサンプル値をプレーンテキストとしてファイル名 fname のファイルにセーブする．セーブされたファイルは，直接 Excel で読み込んでグラフ化が可能である．

Note

- セーブされるのは，ウィンドウ領域範囲内のみである．全体をセーブしたい場合はこの関数の呼び出し前に，SetWindowMax() メンバー関数を呼び出すこと．
- 旧バージョンとの互換性維持のための関数．廃止予定なので使用しないこと．

```
void SaveAsPTF(const char* fname) const
void SaveAsDE(const char* fname) const
WaveField& LoadDE(const char* fname)
void SaveAsCA(const char* fname) const
WaveField& LoadCA(const char* fname)
void SaveAsHBI(...)  const
WaveField& LoadHBI(...)
```

戻り値　対象オブジェクトへの参照

説　明　いずれも，旧バージョンと互換性のあるファイル形式でロードセーブするための関数．廃止予定なので使用しないこと．

```
void SaveAsText(const char* fname, int j = 0) const
```

戻り値　対象オブジェクトへの参照

説　明　整数座標の j で指定した一列のサンプル値をプレーンテキストとしてファイル名 fname のファイルにセーブする．セーブされたファイルは，直接 Excel で読み込んでグラフ化が可能である．

Note

- セーブされるのは，ウィンドウ領域範囲内のみである．全体をセーブしたい場合はこの関数の呼び出し前に，SetWindowMax() メンバー関数を呼び出すこと．
- 旧バージョンとの互換性維持のための関数．廃止予定なので使用しないこと．

3.4.16　複素振幅の変換に関連するメンバー関数

```
WaveField& ConvToPolarForm(void)
```

戻り値　対象オブジェクトへの参照

説　明　複素形式の複素数を極形式に変換する．複素数表現の形式を参照.

```
WaveField& ConvToComplexForm(void)
```

戻り値　対象オブジェクトへの参照

説　明　極形式の複素数を複素形式に変換する．複素数表現の形式を参照.

```
WaveField& ConvToIntensity(void)
```

戻り値　対象オブジェクトへの参照

説　明　すべてのサンプル点の複素数値の絶対値の二乗を計算して，実部データとして格納する．また虚部には 0.0

を格納する．

Note

- 元の複素数値を復旧することはできない．

```
WaveField& ConvToLogIntensity(void)
```

戻り値　対象オブジェクトへの参照

説　明　視覚的に知覚される光強度に変換する関数．そのため，サンプル点の複素数値の絶対値二乗の対数値を計算して，実部データとして格納する．また虚部には 0.0 を格納する．即ち，として変換した後のを実部データとして格納する．

Note

- 元の複素数値を復旧することはできない．

```
WaveField& ConvToAbsolute(void)
```

戻り値　対象オブジェクトへの参照

説　明　すべてのサンプル点の複素数値の絶対値を計算して，実部データとして格納する．また虚部には 0.0 を格納する．元の複素数値を復旧することはできない．

```
WaveField& ConvToConjugate(void)
```

戻り値　対象オブジェクトへの参照

説　明　すべてのサンプル点の複素数値をその複素共役で置き換える．

```
WaveField& Normalize(float max = 1.0)
```

戻り値　対象オブジェクトへの参照

説　明　サンプル点の絶対値の最大値が max なるようにすべてのサンプル点を正規化する．

```
WaveField& NormalizeWin(float max = 1.0)
```

戻り値　対象オブジェクトへの参照

説　明　ウィンドウ領域内のサンプル点の絶対値の最大値が max なるようにすべてのサンプル点を正規化する．

Note

- この関数で検出最大値はウィンドウ領域内の最大値であるが，正規化はオブジェクト全体に対して行う．

3.4.17 離散 (高速) フーリエ変換のためのメンバー関数

```
WaveField& Fft(int s)
```

戻り値 対象オブジェクトへの参照

説 明 オブジェクトを離散フーリエ変換する．具体的には，この関数により次の変換が行われる．

$$F(k\Delta u, l\Delta v) = \sum_{m}^{M}\sum_{n}^{N} f(m\Delta x, n\Delta y)\exp\left[\mathtt{s}\ i2\pi\left(\frac{km}{M}+\frac{ln}{N}\right)\right] \tag{3.3}$$

ここで本関数の引数 s は上式での符号を表す -1 または $+1$ の整数値である．また上式では Δx と Δy は変換前のサンプリング間隔であり，Δu と Δv は変換後のサンプリング間隔である．これらの間には，

$$\begin{aligned}\Delta u &= \frac{1}{M\Delta x}\\ \Delta v &= \frac{1}{N\Delta y}\end{aligned} \tag{3.4}$$

の関係がある．オブジェクトのサンプリング間隔 P_x と P_y は本関数適用後に上式の Δu と Δv になる．また，M と N はそれぞれ x 方向と y 方向のサンプル点数 N_x と N_y である．従って，この関数の使用前と使用後では次のとおりにサンプリング間隔が変化する．

$$\begin{aligned}P_x &\leftarrow \frac{1}{P_x N_x}\\ P_y &\leftarrow \frac{1}{P_y N_y}\end{aligned} \tag{3.5}$$

なお，数学的に完全な離散逆フーリエ変換を求めるためには，この関数を適用後，各サンプル点をさらに $N_x \times N_y$ で除する必要がある．

Note

- 象限交換は内部で実行されている．
- ComplexForm 列挙型で表される状態が COMPLEX のオブジェクトに適用した場合は SPECTRUM，SPECTRUM に適用した場合は COMPLEX に変化する．
- オブジェクトの状態のチェックには IsComplexAmplitude() メンバー関数や IsSpectrum() メンバー関数を用いる．
- 実際の FFT 処理は，市販のパッケージライブラリやフリーソースに基づくルーチンで処理され，幾つか種類がある．利用する FFT ルーチンの選択や情報取得には，SetFftLib() 関数や GetFftLib() 関数を用いる．

Example

```
WaveField wf;
wf.LoadWf("sample.wf")
wf.SaveAsBmpP("sample.bmp", PHASE);      //位相をグレイスケールで保存
wf.Fft(-1);                              //離散フーリエ変換する
wf.SaveAsBmp("sampleS.bmp", PHASE);      //スペクトルの位相を保存
wf.Fft(1);                               //逆フーリエ変換する
wf /= wf.GetN();                         //数学的に完全な逆離散フーリエ変換
wf.SaveAsBmp("sample1.bmp", PHASE);      //位相を再び保存
```

WaveField& FakeFft()

戻り値　対象オブジェクトへの参照

説　明　オブジェクトを離散フーリエ変換した場合と同じ効果をそのパラメータにだけ与える関数．実際に離散フーリエ変換するわけではないので，実行時間はゼロに等しい．

Note

- フーリエ変換でも逆フーリエ変換でも結果は同じであるので引数はない．
- サンプリング間隔は式 (3.5) に従って変換される．
- ComplexForm 列挙型で表される状態が COMPLEX のオブジェクトに適用した場合は SPECTRUM，SPECTRUM に適用した場合は COMPLEX に変化する．
- オブジェクトの状態のチェックには IsComplexAmplitude() メンバー関数や IsSpectrum() メンバー関数を用いる．

WaveField& ScaledFft(double ax, double ay)

戻り値　対象オブジェクトへの参照

説　明　Scaled FFT によりオブジェクトを離散フーリエ変換する [3]．具体的には，この関数により次の変換が行われる．

$$F(k\Delta u, l\Delta v) = \sum_{m}^{M}\sum_{n}^{N} f(m\Delta x, n\Delta y)\exp\left[i2\pi\left(\frac{km}{M}\mathtt{ax} + \frac{ln}{N}\mathtt{ay}\right)\right] \tag{3.6}$$

本関数では Fft() メンバー関数と異なり，引数 ax と ay は任意の実数でよい．また変換前のサンプリング間隔 Δx，Δy と変換後のサンプリング間隔 Δu，Δv の間には，

$$\begin{aligned}\Delta u &= \frac{\mathtt{ax}}{M\Delta x}\\ \Delta v &= \frac{\mathtt{ay}}{N\Delta y}\end{aligned} \tag{3.7}$$

の関係がある．オブジェクトのサンプリング間隔 P_x と P_y は本関数適用後に上式の Δu と Δv になる．M と N はそれぞれ x 方向と y 方向のサンプル点数 N_x と N_y である．従って，この関数の使用前と使用後では次のとおりにサンプリング間隔が変化する．

$$\begin{aligned}P_x &\leftarrow \frac{\mathtt{ax}}{P_x N_x}\\ P_y &\leftarrow \frac{\mathtt{ay}}{P_y N_y}\end{aligned} \tag{3.8}$$

Note

- ax = ay = −1 または ax = ay = 1 とした場合，通常の Fft() メンバー関数と同一の結果になる．
- サンプル点数そのものは変換前後で変化しない．
- この関数の実行には，サイズ $4N_xN_y$ の FFT が 3 回用いられている．
- 内部的には，Fft() メンバー関数を呼び出して処理をしている．したがって，利用する FFT ルーチンの選択や情報取得には，SetFftLib() 関数や GetFftLib() 関数を用いる．
- ComplexForm 列挙型で表される状態が COMPLEX のオブジェクトに適用した場合は SPECTRUM，SPECTRUM に適用した場合は COMPLEX に変化する．

- オブジェクトの状態のチェックには IsComplexAmplitude() メンバー関数や IsSpectrum() メンバー関数を用いる.

```
WaveField& RawFft(int s)
```

戻り値 対象オブジェクトへの参照

説　明 象限交換無しで符号 s の FFT を実行する.

Note

- Fft() メンバー関数の象限交換無しバージョン.
- 利用する FFT ルーチンの選択や情報取得には，SetFftLib() 関数や GetFftLib() 関数を用いる.

3.4.18 光波の回折伝搬計算に関するメンバー関数

```
WaveField& AsmProp(double d)
```

戻り値 対象オブジェクトへの参照

説　明 高速畳み込みを用いた帯域制限角スペクトル法 [2] により，距離 d [m] だけ光波を伝搬 (回折) する．Fresnel 近似のような伝搬距離 d に関する制約はないが，この関数では円状畳み込みであるため，数 10cm 以上伝搬するとエリアシング誤差を生じる場合が多い.

Note

- より正確な伝搬結果が必要な場合はこの関数の呼び出し前に Embed() メンバー関数でサンプリング領域を 4 倍拡張して呼出し後に Extract() メンバー関数を用いるか (サンプルコード参照)，これらを一括して行う ExactAsmProp() メンバー関数を用いる.
- ExactAsmProp() メンバー関数は正確な伝搬結果を与えるが，一方この関数の方が演算が高速であり，必要なメモリ容量も約 1/4 という利点がある.
- この計算法では伝搬後のサンプリング間隔は伝搬前と変化しない.

Example 高速・コンパクトだが誤差が多い計算

```
WaveField wf;
wf.LoadWf("sample.wf");
wf.AsmProp(10e-3);          //角スペクトル法による回折計算
wf.Normalize();
wf.SaveAsWf("sample1.wf");  //正規化して保存
```

Example 正確な計算

```
WaveField wf;
wf.LoadWf("sample.wf");
wf.Embed(1);                //４倍拡張
wf.AsmProp(10e-3);          //角スペクトル法による回折計算
wf.Extract(1)               //４分の１縮小
wf.Normalize();
wf.SaveAsWf("sample1.wf");  //正規化して保存
```

```
WaveField& AsmPropFs(double d)
```

戻り値　対象オブジェクトへの参照

説　明　帯域制限角スペクトル法 [2] により，フーリエ空間にあるオブジェクト (スペクトル状態) を距離 d [m] だけ伝搬 (回折) する．したがってオブジェクトはスペクトルでなければならない．フーリエ空間で追加の処理が必要な場合にはこの関数が便利である．Fresnel 近似のような伝搬距離 d に関する制約はないが，数 10cm 以上伝搬するとエリアシング誤差を生じる場合が多い．

Note

- AsmProp() メンバー関数と同様，より正確な伝搬結果が必要な場合は FFT の呼び出し前に Embed() メンバー関数でサンプリング領域を拡張して逆 FFT の後に Extract() メンバー関数を用いる．
- この計算法では伝搬後のサンプリング間隔は伝搬前と変化しない．

Example

```
WaveField wf;
wf.LoadWf("sample.wf");
wf.Fft(-1);                    //フーリエスペクトルに変換する
wf.AsmPropFs(10e-3);           //角スペクトル法による回折計算
wf.Fft(1);                     //複素振幅に戻す
wf.Normalize();
wf.SaveAsWf("sample1.wf");   //正規化して保存
```

```
WaveField& ExactAsmProp(double d)
```

戻り値　対象オブジェクトへの参照

説　明　高速畳み込みを用いた帯域制限角スペクトル法 [2] により，距離 d [m] だけ光波を伝搬 (回折) する．Fresnel 近似のような伝搬距離 d に関する制約はない．また AsmProp() メンバー関数と異なり，この関数では線状畳み込みをおこなうため伝搬に伴うエリアシング誤差は生じない．

Note

- この関数は AsmProp() メンバー関数に比べて正確な伝搬結果を与えるが，演算中に一時的に必要なメモリ容量が 4 倍であり，その分計算も遅い問題点があり，ラフな計算では AsmProp() メンバー関数で十分であることも多い．
- この計算法では伝搬後のサンプリング間隔は伝搬前と変化しない．

Example

```
WaveField wf;
wf.LoadWf("sample.wf");
wf.EactAsmProp(10e-3);         //正確な角スペクトル法による回折計算
wf.Normalize();
wf.SaveAsWf("sample1.wf");   //正規化して保存
```

```
WaveField& ShiftedAsmProp(const WaveField& source, int prec = 1)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトを精度指数 prec のシフテッド角スペクトル法 [4] で伝搬した結果を対象のオブジェ

クトに代入する．

Note

- この関数では二つのオブジェクトの原点のグローバルな x, y 位置がずれていてもかまわない．
- 伝搬距離は source オブジェクトと対象オブジェクトのローカル原点のグローバル z 値の差から自動的に計算される．
- サンプリング間隔 P_x，P_y とサンプリング数は同一でなければならない．
- この関数は，ShiftedFresnelProp() メンバー関数と異なり，近距離の伝搬計算でも正しい結果を得ることができる．その半面，ある程度距離の長い伝搬では ShiftedFresnelProp() メンバー関数の方が計算精度が高いことが多い．
- 精度指数 prec は通常 1 である．1 以上の値を設定した場合，計算精度は向上するが，2 の prec 乗のサンプリング数拡張を行うので，メモリ消費量が増加し計算速度が低下する．

Example

```
WaveField a, b;
a.LoadWf("input.wf");
b.CopyParam(a);
a.SetOrigin(Vector(0, 0, 0));              //aの原点はグローバル座標(0, 0, 0)
b.SetOrigin(Vector(10e-3, 10e-3, 50e-3)); //bの原点はグローバル座標(10, 10, 50)[mm]
b.ShiftedAsmProp(a);                       //aをbの位置に伝搬した結果をbに入れる．
b.SaveAsWf("result.wf");
```

```
WaveField& ShiftedAsmPropAdd(const WaveField& source, int prec = 1)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトを精度指数 prec のシフテッド角スペクトル法 [4] で伝搬した結果を対象のオブジェクトに加算する．

Note

- 詳細は ShiftedAsmProp() メンバー関数参照．
- サンプリング間隔 P_x，P_y とサンプリング数は同一でなければならない．

```
WaveField& ShiftedAsmPropEx(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをシフテッド角スペクトル計算法により伝搬した結果を対象のオブジェクトに代入する．サンプリング数可変型．

Note

- この関数では ShiftedAsmProp() メンバー関数とは異なり，伝搬元の source オブジェクトと伝搬先の対象オブジェクトは異なったサンプリング数でもよい．

```
WaveField& ShiftedAsmPropAddEx(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをシフテッド角スペクトル法 [4] により伝搬した結果を対象のオブジェクトに加算する．サンプリング数可変型．

Note

- この関数では ShiftedAsmPropAdd() メンバー関数とは異なり，伝搬元の source オブジェクトと伝搬先の対象オブジェクトは異なったサンプリング数でもよい．

```
WaveField& FresnelProp(double d)
```

戻り値　対象オブジェクトへの参照

説　明　Fresnel 近似により，距離 d [m] だけ光波を伝搬 (回折) する．伝搬距離 d には Fresnel 近似の制約がある．また，伝搬後のサンプリング間隔は伝搬前に比べて大きくなる．

```
WaveField& FourierProp(double f)
```

戻り値　対象オブジェクトへの参照

説　明　焦点距離 f [m] のレンズを用いた $2f$ セットアップによるフーリエ回折により光波を伝搬 (回折) する．回折後のサンプリング間隔は回折前とは異なる．

```
WaveField& BackFourierProp(double f)
```

戻り値　対象オブジェクトへの参照

説　明　焦点距離 f [m] のレンズを用いた $2f$ セットアップによるフーリエ回折により光波を後方伝搬 (逆回折) する．回折後のサンプリング間隔は回折前とは異なる．

```
WaveField& ShiftedFresnelProp(const WaveField& source)
WaveField& ShiftedFresnelProp(const WaveField& source, const ShiftedFresnelPropDescriptor&
sfpd)
WaveField& ShiftedFresnelProp(const Vector& origin, double px, double py)
WaveField& ShiftedFresnelProp(const Vector& origin, double px, double py, const
ShiftedFresnelPropDescriptor& sfpd)
```

戻り値　対象オブジェクトへの参照

説　明　第 1 と第 2 の形式では source オブジェクトをシフテッドフレネル型計算法 [5] により伝搬した結果を対象のオブジェクトに代入する．この場合 source オブジェクトの内容は変化しない (out-place 型)．第 3 と第 4 の形式では対象オブジェクトそのものを位置 origin，サンプリング間隔 px，py に伝搬した結果に変化させる

(in-place 型)．第 2 と第 4 の形式では ShiftedFresnelPropDescriptor 型デスクリプタ sfpd を用いた事前計算により高速化を行う．

Note

- この関数では伝搬元と伝搬先の原点のグローバルな x, y 位置がずれていてもかまわない．またそれぞれの P_x, P_y が異なっていてもよい．
- out-place 型での伝搬距離は source オブジェクトと対象オブジェクトのローカル原点のグローバル z 値の差から自動的に計算される．in-place 型での伝搬距離は伝搬前の対象オブジェクトの原点と origin から計算される．
- この関数では伝搬元と伝搬先のサンプリング数は等しくなければならない．サンプリング数を変えたい場合は ShiftedFresnelPropEx() メンバー関数あるいは ShiftedFresnelAdd() メンバー関数を用いる．

Example out-place 型

```
WaveField a, b;
a.LoadWf("input.wf");
a.SetOrigin(Vector(0, 0, 0));              //aの原点はグローバル座標(0, 0, 0)
b.SetOrigin(Vector(10e-3, 10e-3, 100e-3); //bの原点はグローバル座標(10, 10, 100)[mm]
b.ShiftedFresnelProp(a);                   //aをbの位置に伝搬した結果をbに入れる．
b.SaveAsWf("result.wf");
```

Example out-place 型でデスクリプタを利用

```
WaveField a;
a.LoadWf("input.wf");
a.SetOrigin(Vector(0, 0, 0));              //aの原点はグローバル座標(0, 0, 0)

double d = 100e-3, px = 5e-6, py = 3e-6;
SetDefault(a.GetNx(), a.GetNy(), px, py, a.GetWavelength());
WaveField b, c, d;
b.SetOrigin(Vector(10e-3, 0, d));
c.SetOrigin(Vector(-10e-3, 0, d));
d.SetOrigin(Vector(0, 10e-3, d));

//以下で伝搬計算．重複する無駄な計算を行わないので複数伝搬で高速化
ShiftedFresnelDescriptor sfpd(d, px, py, a);//デスクリプタ構築
b.ShifteFresnelProp(a, sfpd);                //aをbに伝搬
c.ShifteFresnelProp(a, sfpd);                //aをcに伝搬
d.ShifteFresnelProp(a, sfpd);                //aをdに伝搬

b.SaveAsWf("result-b.wf");
c.SaveAsWf("result-c.wf");
d.SaveAsWf("result-d.wf");
```

Example in-place 型

```
WaveField a;
a.LoadWf("input.wf");
a.SetOrigin(Vector(0, 0, 0));                //aの原点はグローバル座標(0, 0, 0)
a.ShiftedFresnelProp(Vector(10e-3, 10e-3, 100e-3, a.GetPx()/2, a.GetPx());
                                             //aそのものを伝搬する
a.SaveAsWf("result.wf");
```

```
WaveField& ShiftedFresnelPropAdd(const WaveField& source, const
ShiftedFresnelPropDescriptor& sfpd)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをシフテッドフレネル型計算法により伝搬しそ，その結果を対象のオブジェクトに加算する．

```
WaveField& ShiftedFresnelPropEx(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをシフテッドフレネル型計算法により伝搬した結果を対象のオブジェクトに代入する．サンプリング数可変型．

Note

- この関数では ShiftedFresnelProp() メンバー関数とは異なり，伝搬元の source オブジェクトと伝搬先の対象オブジェクトは異なったサンプリング数でもよい．

```
WaveField& ShiftedFresnelPropAddEx(const WaveField& source)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをシフテッドフレネル型計算法により伝搬しそ，その結果を対象のオブジェクトに加算する．サンプリング数可変型．

Note

- この関数の第 1 の形式では，ShiftedFresnelPropAdd() メンバー関数とは異なり，伝搬元の source オブジェクトと伝搬先の対象オブジェクトは異なったサンプリング数でもよい．

```
WaveField& Rotate(const WaveField& source, RMatrix& crmat, SFrequency* c = NULL,
Interpol ip = CUBIC8, bool Jacobian = false)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトを回転変換し，対象オブジェクトに代入する．RMatrix 型の crmat が座標回転の回転行列を与える．キャリア周波数がゼロになるように，回転変換後は変換前の光波のキャリヤ信号成分が取り除かれている．取り除いたキャリア信号成分の周波数は，SFrequency 型のポインタ c が NULL でない場合に*c に設定される．回転変換で用いる補間方法は Interpol 列挙型 ip で指定する．また，Jacobian = true の場合はヤコビアンを用いた厳密計算を行う．

Note

- 回転変換の理論については文献 [6, 7] を参照．

```
WaveField& RotateFs(const WaveField& source, RMatrix& crmat, SFrequency* c = NULL,
Interpol ip = CUBIC8, bool Jacobian = false)
```

戻り値　対象オブジェクトへの参照

説　明　source オブジェクトをフーリエ空間で回転変換 [6, 7] し，対象オブジェクトに代入する．source オブジェクトと対象オブジェクトはいずれもフーリエ空間になければならない．RMatrix 型の crmat が回転行列

を与える．回転変換後はキャリア周波数がゼロになるようにスペクトルがシフトしており，このシフト量は SFrequency 型のポインタ c が NULL でない場合に*c に設定される．回転変換で用いる補間方法は Interpol 列挙型 ip で指定する．また，Jacobian = true の場合はヤコビアンを用いた厳密計算を行う．

Example

```
RMatrix rm = CRMatrixY(15*Deg);          //座標をｙ軸の周りに１５度回転する行列
WaveField a, b;
a.LoadWf("input.wf");
a.Fft(-1);                               //ａをフーリエ変換
b.FakeFft();                             //ｂも形だけフーリエ領域に移す
b.RotateFs(a, rm);                       //ａを回転変換してｂに入れる
b.Fft(1);                                //ｂを逆フーリエ変換
b/=b.GetN();
b.SaveAsWf("result.wf");
```

3.4.19 基本的な光波・位相・開口を生成するメンバー関数

```
WaveField& SetGaussian(double w, double n = 2.0, double a = 1.0)
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　ちょうどウェストの位置にある基本ガウスビームを生成する関数．従って位相は定数の 0 である．w [m] は 1/e ビーム半径，n は次数，a は中心部の振幅である．これらの関係は次式で表される．

$$f(x,y) = \mathtt{a}\ \exp\left[-\left(\frac{\sqrt{x^2+y^2}}{\mathtt{w}}\right)^{\mathtt{n}}\right]$$

なお，n = 2 の場合が通常の基本ガウシアンである．n > 2 の場合は通常のガウシアン関数ではないが，これはスーパーガウシアンと呼ばれることもある．

```
WaveField& SetEllipticGaussian(double rx, double ry, double n = 2.0, double a = 1.0)
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　ちょうどウェストの位置にある楕円ガウスビームを生成する関数．従って位相は定数の 0 である．rx [m] と ry [m] はそれぞれ x 方向と y 方向の 1/e ビーム半径，n は次数，a は中心部の振幅である．これらの関係は次式で表される．

$$f(x,y) = \mathtt{a}\ \exp\left[-\left(\sqrt{\left(\frac{x}{\mathtt{rx}}\right)^2+\left(\frac{y}{\mathtt{ry}}\right)^2}\right)^{\mathtt{n}}\right]$$

なお，n = 2 の場合が通常の基本ガウシアンである．

```
WaveField& SetSeparableGaussian(double wx, double wy, double nx = 2.0, double ny = 2.0,
double a = 1.0)
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　非円筒対称の変数分離型ガウス関数を生成する関数．wx [m] と wy [m] はそれぞれ x 方向と y 方向の 1/e

ビーム半径，nx と ny はそれぞれの方向の次数，a は中心部の振幅である．これらの関係は次式で表される．

$$f(x,y) = \mathtt{a}\ \exp\left[-\left(\frac{x}{\mathtt{wx}}\right)^{\mathtt{nx}} - \left(\frac{y}{\mathtt{wy}}\right)^{\mathtt{ny}}\right]$$

なお，n = 2 の場合が一般的なガウス関数であり，n > 2 の場合はスーパーガウス関数と呼ばれる．

```
WaveField& SetRect(double wx, double wy, double a = 1.0);
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　矩形開口を生成する関数．開口の中心は物理座標の原点に位置し，x 方向の幅が wx [m]，y 方向の幅が wy [m] ある．開口部の複素振幅の実部は a，虚部は 0 となる．非開口部は実部虚部とも 0 となる．

```
WaveField& AddSphericalWave(Point p, Phase phs, double a, WindowFunc w)
WaveField& AddSphericalWave(Point p, Phase phs, double a)
WaveField& AddSphericalWave(Point p)
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　点光源からの球面波を生成する関数．Point 型のローカル座標 p に位置する振幅 a の点光源からの球面波を計算し，対象オブジェクトに追加 (重畳) する．このとき，球面波の初期位相を Phase 型の phs で指定し，またエイリアス誤差を防ぐため，WindowFunc 列挙型の w で指定する窓関数を使用して描画範囲を制限する．第 1 の形式では，これらの全てを明示的に指定する．第 2 の形式では窓関数は矩形関数となる．第 3 の形式ではそれに加え，初期位相が 0 となる．

```
WaveField& AddSphericalWaveSqr(const SphericalWaveDescriptor& swd,
        double x, double y, double z, double a = 1.0, double InitPhase = 0,
        WindowFunc w = RECTANGLE)
```

戻り値　対象オブジェクトへの参照を戻す．

説　明　点光源からの球面波を高速に生成する関数．SphericalWaveDescriptor クラスのディスクリプタ swd を用いて，物理座標 (x, y, z) に位置する振幅 a で初期位相が InitPhase($= 0 \sim 2\pi$) の点光源からの球面波を計算し，対象オブジェクトに追加 (重畳) する．この関数では三角関数の表参照を用いて，高速計算をおこなう．このとき，エイリアス誤差を防ぐため，WindowFunc 列挙型 w で指定する窓関数を使用して描画範囲を制限する (現バージョンでは，RECTANGLE 以外の窓関数は不可)．

Note

- InitPhase は 0 以上 2π 以下の範囲で与えなければならない．負の InitPhase に対しては正常に動作しない．
- この関数を呼び出す前に，必ず，SphericalWaveDescriptor クラスのディスクリプタ swd を生成しておく必要がある．

Example

```
WaveField wf(512, 512);
```

```
//デスククリプタ生成
SphericalWaveDescriptor sfd(wf.GetWavelength(), wf.GetPx(), wf.GetPy();

wf.Celar();
wf.AddSphericalWaveRF(sfd, 0.1e-3, -0.1e-3, -1e-3);    // 球面波を追加
wf.AddSphericalWaveRF(sfd, -0.1e-3, 0.1e-3, -1e-3);    // 球面波を追加
```

```
WaveField& MultiplySphericalWave(Point p, Phase phs, double a, WindowFunc w, double sign
= +1.0)
WaveField& MultiplySphericalWave(Point p, Phase phs, double a, double sign = +1.0)
WaveField& MultiplySphericalWave(Point p, double sign = +1.0)
```

戻り値　対象オブジェクトへの参照を戻す.

説　明　Point 型のローカル座標 p に位置する振幅 a の点光源からの球面波を対象オブジェクトに乗算する．このとき，球面波の初期位相を Phase 型の phs で指定し，またエイリアス誤差を防ぐため，WindowFunc 列挙型の w で指定する窓関数を使用して描画範囲を制限する．sign $= -1$ とすると，収れんする球面波，あるいは発散球面波の共役となる．第 1 の形式では，これらの全てを明示的に指定する．第 2 の形式では窓関数は矩形関数となる．第 3 の形式ではそれに加え，初期位相が 0 となる.

Note

- sign の整合性はチェックされない．sign $= \pm 1$ 以外の値も指定できる.
- 計算式：

$$g(x,y) = \mathrm{a}\exp[i(\mathrm{s}k\sqrt{(x-x_0)^2+(y-y_0)^2+z_0^2}+\mathrm{phs})]$$

x_0, y_0, z_0: ローカル座標 p で指定される球面波の中心.

```
WaveField& SetRandomPhase(int m = 1)
```

戻り値　対象オブジェクトへの参照

説　明　乱数位相の光波を設定する．振幅は 1 になる．m = 1 のとき，メルセンヌ・ツイスタ (Mersenne twister) アルゴリズムを用いて乱数を発生する．それ以外の場合は，C ランタイムの rand() 関数を用いる.

Note

- 乱数ジェネレータのシードを設定するためには，wfl::SetRandomSeed() 関数を用いる.

```
WaveField& ModRandomPhase(int m = 1)
```

戻り値　対象オブジェクトへの参照

説　明　オブジェクトの位相を乱数化する．この関数では SetRandamPhase() メンバー関数と異なり，振幅は影響を受けず，関数適用前の振幅が残されている．スペクトル形式に適用した場合エラーとなる．m = 1 のとき，メルセンヌ・ツイスタ (Mersenne twister) アルゴリズムを用いて乱数を発生する．それ以外の場合は，C ランタ

イムの rand() 関数を用いる．

Note

- 乱数ジェネレータのシードを設定するためには，wfl::SetRandomSeed() 関数を用いる．

```
WaveField& SetQuadraticPhase(double f)
```

戻り値　対象オブジェクトへの参照

説　明　焦点距離 f のレンズとして機能する 2 次の位相分布を設定する．振幅分布は変化しない．

```
WaveField& MultiplyQuadraticPhase(double f)
```

戻り値　対象オブジェクトへの参照

説　明　焦点距離 f のレンズとして機能する 2 次の位相分布を乗算する．振幅分布は変化しない．

```
WaveField& MultiplyPlaneWave(double CosA, double CosB, Phase phs = 0.0)
WaveField& MultiplyPlaneWave(Vector dir, Phase phs = 0.0)
WaveField& MultiplyPlaneWave(SFrequency f, Phase phs = 0.0)
```

戻り値　対象オブジェクトへの参照

説　明　第 1 の形式では，方向余弦が x 軸に対して CosA で y 軸に対して CosB，初期位相が phs の平面波が乗算される．第 2 の形式では，方向ベクトルが dir で初期位相が phs の平面波が乗算される．第 3 の形式では，SFrequency 型の空間周波数が f で初期位相が phs の平面波が乗算される．

Note

- 第 2 の形式における方向ベクトル dir は，その方向のみが計算に用いられ，長さは無視される．
- 本関数の平面波はグローバル原点での初期位相が phs でグローバル空間中に「固定された」平面波であり，対象オブジェクトがグローバル原点以外の位置に移動している場合も位相関係が狂うことなく計算される．しかし，対象オブジェクトが傾いている場合は正しい計算結果が得られない．

```
WaveField& SetPlaneWave(double CosA, double CosB, Phase phs = 0.0)
WaveField& SetPlaneWave(Vector dir, Phase phs = 0.0)
WaveField& SetPlaneWave(SFrequency f, Phase phs = 0.0)
```

戻り値　対象オブジェクトへの参照

説　明　第 1 の形式では，方向余弦が x 軸に対して CosA で y 軸に対して CosB，初期位相が phs の平面波を設定する．第 2 の形式では，方向ベクトルが dir で初期位相が phs の平面波を設定する．第 3 の形式では，SFrequency 型の空間周波数が f で初期位相が phs の平面波を設定する．

Note

- この関数を適用する以前の光波は消失し，振幅が 1 の平面波で置き換わる．
- 第 2 の形式における方向ベクトル dir は，その方向のみが計算に用いられ，長さは無視される．
- 本関数の平面波はグローバル原点での初期位相が phs でグローバル空間中に「固定された」平面波であり，対象オブジェクトがグローバル原点以外の位置に移動している場合も位相関係が狂うことなく計算される．しかし，対象オブジェクトが傾いている場合は正しい計算結果が得られない．

3.4.20 誤差・効率などを計測するためのメンバー関数

ウインドウの概要

```
double GetSquareSum(void) const
```

戻り値 サンプル点の光強度 (絶対値の 2 乗) の総和.
説　明 ウィンドウ領域内のサンプル点の光強度 (絶対値の 2 乗) の総和を計算する．

```
double GetSquareSumWhole(void) const
```

戻り値 サンプル点の光強度 (絶対値の 2 乗) の総和.
説　明 ウィンドウ領域に関係なく，全てのサンプル点の光強度 (絶対値の 2 乗) の総和を計算する．

```
double GetSquareAverage(void) const
```

戻り値 サンプル点の光強度 (絶対値の 2 乗) の総和の平均.
説　明 ウィンドウ領域内のサンプル点の光強度 (絶対値の 2 乗) の総和の平均値を計算する．

```
double GetAbsoluteSum(void) const
double GetAmplitudeSum(void) const
```

戻り値 サンプル点の振幅値 (絶対値) の総和.
説　明 ウィンドウ領域内のサンプル点の振幅値 (絶対値) の総和を計算する．

```
double GetAbsoluteAverage(void) const
double GetAmplitudeAverage(void) const
```

戻り値 サンプル点の振幅値 (絶対値) の平均.
説　明 ウィンドウ領域内のサンプル点の振幅値 (絶対値) の平均値を計算する．

```
double GetAbsoluteMax(void) const
double GetAmplitudeMax(void) const
```

戻り値　サンプル点の振幅値 (絶対値) の最大値.

説　明　ウィンドウ領域内のサンプル点の振幅値 (絶対値) の最大値を求める.

```
double GetAmplitudeMaxWhole(void) const
```

戻り値　サンプル点の振幅値 (絶対値) の最大値.

説　明　ウィンドウ領域に関係なく，全てのサンプル点の振幅値 (絶対値) の最大値を求める.

```
double GetAmplitudeVariance(void) const
```

戻り値　サンプル点の振幅値 (絶対値) の分散値.

説　明　ウィンドウ領域内のサンプル点の振幅値 (絶対値) の分散値を求める.

```
ComplexDouble GetScaleFactor(const WaveField& signal) const
```

戻り値　スケールファクター

説　明　signal オブジェクトに対する対象オブジェクトのスケールファクターを計算する．スケールファクターとは，対象関数 $f(x,y)$ とシグナル関数 $f_{sig}(x,y)$ に対して，総和

$$\sum_{(m,n)\in W} |f(x_m,y_n) - \alpha f_{sig}(x_m,y_n)|^2$$

を最小にするような複素数 α であり，

$$\alpha = \frac{\sum_{(m,n)\in W} f(x_m,y_n) f^*_{sig}(x_m,y_n)}{\sum_{(m,n)\in W} |f_{sig}(x_m,y_n)|^2}$$

で定義される．ここで，W はウィンドウ領域である．従って，計算は対象オブジェクトと signal オブジェクトのウィンドウ領域に対して行われる．両者のウィンドウサイズが異なる場合には，両者のウィンドウ領域の左下角を一致させ，signal オブジェクトのウィンドウサイズで計算が行われる (図 3.4 参照).

```
double GetSnr(const WaveField& signal, const ComplexDouble& scaleFactor) const
```

戻り値　S/N 比

説　明　signal オブジェクトに対する S/N 比を，スケールファクター scaleFactor を用いて計算する．計算は対

象オブジェクトと signal オブジェクトのウィンドウ領域に対して行われる．両者のウィンドウサイズが異なる場合には，両者のウィンドウ領域の左下角を一致させ，signal オブジェクトのウィンドウサイズで計算が行われる (図 3.4)．

Note

- スケールファクターの計算には GetScaleFactor() メンバー関数を用いる．

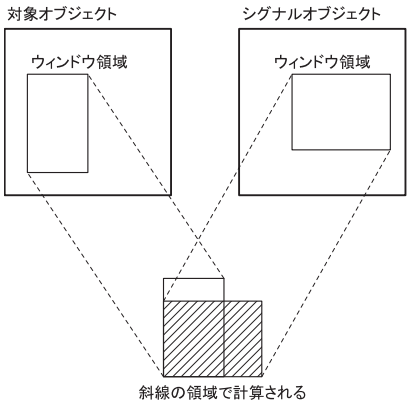


図 3.4 GetSnr() のウィンドウ領域

```
double GetEnergyRatioInWindow(void) const
```

戻り値 エネルギーの比率 (最大 1.0)

説　明 オブジェクトの全体のエネルギーに対するウィンドウ領域内エネルギーの比率を計算して戻す．

```
WaveField& CreateRealHistogram(int level = 512) const
```

戻り値 1 次元 WaveField オブジェクト

説　明 オブジェクトの実部サンプル値のヒストグラムを level 段階で計算し，それを格納した 1 次元 WaveField オブジェクトを生成して戻す．この関数は，2 値振幅変調型ホログラム作成における閾値の決定を支援するための関数である．

Note

- ヒストグラムを生成する元になったオブジェクトはそのまま変化せず，1 次元 WaveField オブジェクトが新規に作成される．
- 生成した 1 次元 WaveField オブジェクトの Px には level 値の逆数が設定される．

Example

```
WaveField wf(512, 512);              // ヒストグラム作成の対象となるオブジェクト
wf.SetSphericalWave(0, 0, 10e-3);    // 点光源の球面波をセット
WaveField hist;                      // ヒストグラムを格納するオブジェクト
hist = wf.CreateRealHistogram();     // ヒストグラムの作成
float max = hist.Real(0,0);          // 以下，ヒストグラムの最大値を求めるループ
int i;
```

```
for (i = 0; i < hist.Nx(); i++)
    if (max > hist.Real(i, 0))
        max = hist.Real(i, 0);
```

```
double GetOverlapFactor(const WaveField& sig) const
```

戻り値　強度分布のオーバーラップ係数

説　明　オブジェクトと sig オブジェクトの全体を比較して，光強度分布の重なりの程度を示すオーバーラップ係数を計算する．この係数が 1 のときは，光強度分布は完全に同一である．

3.4.21　サンプル値を量子化するためのメンバー関数

```
WaveField& HardClipPhase(int level)
```

戻り値　対象オブジェクトへの参照

説　明　単純閾値法により，位相を level 段階に量子化する．このとき，位相は最も近いレベル値に丸められる．また，$-\pi$ に近い位相値は π に丸められる．

```
WaveField& HardClipAmplitude(int level)
```

戻り値　対象オブジェクトへの参照

説　明　単純閾値法により，振幅を level 段階に量子化する．振幅は最も近いレベルに丸められる．なお，振幅は $0 \sim 1.0$ の範囲内にあるものとして量子化する．したがって，1.0 以上の振幅は必ず 1.0 に丸められる．

```
WaveField& PartialClipPhase(int level, double epsilon)
```

戻り値　対象オブジェクトへの参照

説　明　段階的量子化法により，位相を部分的に level 段階に量子化する．このとき，位相は最も近いレベル値に丸められる．また，$-\pi$ に近い位相値は π に丸められる．epsilon は図 3.5 に示すように段階的量子化法の量子化程度を表す ε である．

Note

- $\varepsilon = 0.5$ が完全な量子化である．

```
WaveField& PartialClipAmplitude(int level, double epsilon)
```

戻り値　対象オブジェクトへの参照

説　明　段階的量子化法により，振幅を部分的に level 段階に量子化する．振幅は最も近いレベルに丸められる．

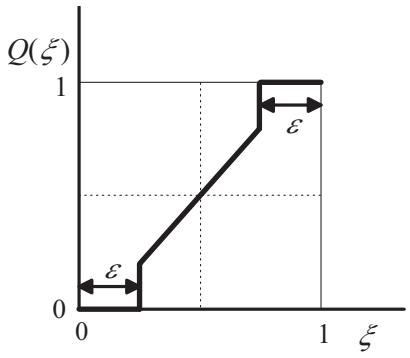


図 3.5 段階的量子化

なお，振幅は 0〜1.0 の範囲内にあるものとして量子化する．したがって，1.0 以上の振幅は必ず 1.0 に丸められる．epsilon は図 3.5 に示すように段階的量子化法の量子化程度を表す ε である．

Note

- $\varepsilon = 0.5$ が完全な量子化である．

3.4.22 補間のためのメンバー関数

以下のメンバー関数では補間を行って，サンプル点の間の複素値を取得する．

```
Complex GetInterpolVal(double x, double y, Interpol ip = CUBIC8) const
Complex GetInterpolVal(double x, double y, InterpolFunc intfunc = &Cubic8Interpol) const
```

戻り値 補間値

説 明 1 番目の形式では，Interpol 列挙型引数 ip が示す補間ルーチンを用いてローカル座標点 (x, y) における複素値を取得する．2 番目の形式では，InterpolFunc 型 intfunc の関数ポインタが示す補間ルーチンを用いて複素値を取得する．

Note

- デフォルトでは Interpol 列挙型 CUBIC8 [8] の補完ルーチンを用いる．
- 関数ポインタの取得については GetInterpolFunc() 関数を参照．

```
Complex Cubic8Interpol(double x, double y) const
Complex Cubic6Interpol(double x, double y) const
Complex Cubic4Interpol(double x, double y) const
```

戻り値 補間値

説 明 それぞれ，8 点，6 点，4 点の Cubic 補間法 [8] を用いてローカル座標点 (x, y) における複素値を取得する．

Note

- 対応する Interpol 列挙型の値はそれぞれ CUBIC8, CUBIC6, CUBIC4 である.

```
Complex LinearInterpol(double x, double y) const
```

戻り値　補間値

説　明　線形補間法を用いてローカル座標点 (x, y) における複素値を取得する.

Note

- 対応する Interpol 列挙型の値は LINEAR である.

```
Complex AdjacentInterpol(double x, double y) const
```

戻り値　サンプル値

説　明　補間を行わずローカル座標点 (x, y) に最も近いサンプル値を取得する.

Note

- 対応する Interpol 列挙型の値は ADJACENT である.

```
Complex BiLinear(double x, double y) const
```

戻り値　サンプル値

説　明　バイリニアー補間法を用いてローカル座標点 (x, y) における複素値を取得する.

Note

- 対応する Interpol 列挙型の値は BILINEAR である.

```
Complex BiCubic(double x, double y) const
```

戻り値　サンプル値

説　明　バイキュービック補間法を用いてローカル座標点 (x, y) における複素値を取得する.

Note

- 対応する Interpol 列挙型の値は BICUBIC である.

```
Complex NearestNeighbor(double x, double y) const
```

戻り値　サンプル値

説　明　補間を行わずローカル座標点 (x, y) に最も近いサンプル値を取得する.

Note

- 対応する Interpol 列挙型の値は NEAREST_NEIGHBOR である.

3.4.23 サンプル点数を拡大縮小するメンバー関数

以下のメンバー関数はサンプル点の多重化や反復により，オブジェクトのサンプル点数を 2 の m 乗倍にする．例えば，デフォルト引数 mxy=1 の場合にはいずれの関数もサンプル点数を $2(=2^1)$ 倍にし，mxy=2 の場合には $4(=2^2)$ 倍に変化させる.

```
WaveField& Replicate(int mxy = 1)
WaveField& Replicate(int mx, int my)
```

戻り値 対象オブジェクトへの参照

説 明 1 番目の形式では関数実行前のオブジェクトサンプル値のパターンを x 方向にも y 方向にも 2 の mxy 乗回繰り返すことにより，オブジェクトを拡大する (図 3.6)．2 番目の形式では x 方向に 2 の mx 乗回，y 方向には 2 の my 乗回繰り返すことにより，オブジェクトを拡大する.

Note

- オブジェクトのサンプリング間隔は変化しない.

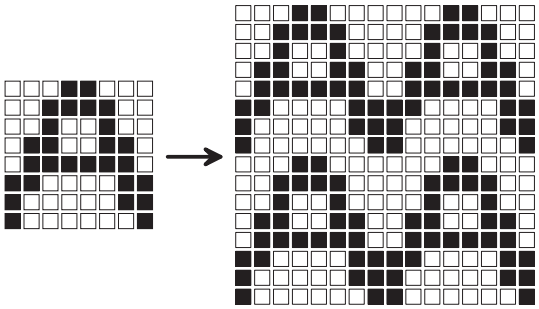


図 3.6 Replicate() の効果

```
WaveField& EnlargeZeroFill(int mxy = 1)
WaveField& EnlargeZeroFill(int mx, int my)
```

戻り値 対象オブジェクトへの参照

説 明 1 番目の形式では関数実行前のオブジェクトサンプル値のパターンを x 方向にも y 方向にも 2 の mxy 乗倍に拡大する．このとき，新たに付け加えられたサンプル点を 0 で埋める (図 3.7)．2 番目の形式では x 方向に 2 の mx 乗倍，y 方向には 2 の my 乗倍に拡大する.

Note

- オブジェクトのサンプリング間隔は変化しない.

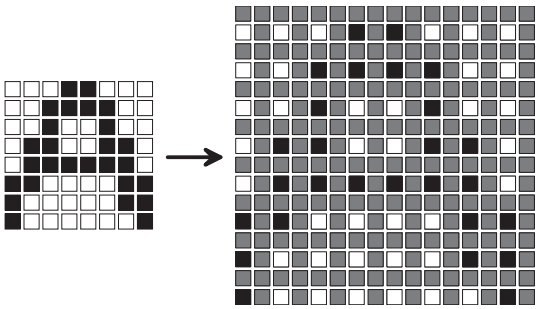


図 3.7　EnlargeZeroFill() の効果

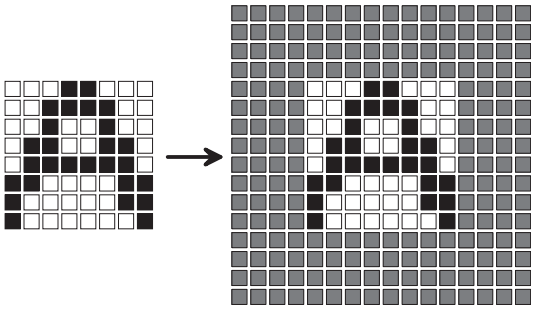


図 3.8　Embed() の効果

```
WaveField& Embed(int mxy = 1)
WaveField& Embed(int mx, int my)
```

戻り値　対象オブジェクトへの参照

説　明　1 番目の形式では関数実行前のオブジェクトサンプル値のパターンを x 方向にも y 方向にも 2 の mxy 乗倍に拡大する．このとき，元のサンプル点の周辺に均等に 0 値を埋め込む (図 3.8)．2 番目の形式では x 方向に 2 の mx 乗倍，y 方向には 2 の my 乗倍に拡大する．

Note

- オブジェクトのサンプリング間隔は変化しない．

```
WaveField& Extract(int mxy = 1)
WaveField& Extract(int mx, int my)
```

戻り値　対象オブジェクトへの参照

説　明　1 番目の形式では関数実行前のオブジェクトサンプル値のパターンを x 方向にも y 方向にも 2 の mxy 乗分の 1 に縮小する (図 3.9)．2 番目の形式では x 方向に 2 の mx 乗分の 1，y 方向には 2 の my 乗分の 1 に縮小する．

Note

- オブジェクトのサンプリング間隔は変化しない．

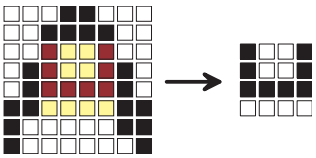


図 3.9 Extract() の効果

```
WaveField& ReduceByAverage(int mxy = 1)
WaveField& ReduceByAverage(int mx, int my)
```

戻り値 対象オブジェクトへの参照

説 明 平均化処理による縮小．1 番目の形式では関数実行前のオブジェクトのサンプル点数を x 方向にも y 方向にも 2 の mxy 乗分の 1 に縮小する．2 番目の形式では x 方向に 2 の mx 乗分の 1，y 方向には 2 の my 乗分の 1 に縮小する．

Note

- オブジェクトの物理サイズが変化しないようにサンプル点数のみを縮小するため，オブジェクトのサンプリング間隔は mxy 倍，または mx 倍と my 倍に拡大する．

```
WaveField& ReduceByThinning(int mxy = 1)
WaveField& ReduceByThinning(int mx, int my)
```

戻り値 対象オブジェクトへの参照

説 明 サンプル点の間引き処理による縮小．1 番目の形式では，関数実行前のオブジェクトのサンプル点数を x 方向にも y 方向にも 2 の mxy 乗分の 1 に縮小する．2 番目の形式では x 方向に 2 の mx 乗分の 1，y 方向には 2 の my 乗分の 1 に縮小する．

Note

- オブジェクトの物理サイズが変化しないようにサンプル点数のみを縮小するため，オブジェクトのサンプリング間隔は mxy 倍，または mx 倍と my 倍に拡大する．

```
WaveField& ExtractWindow(const WaveField& source, Complex backg = Complex(0.0, 0.0))
```

戻り値 対象オブジェクトへの参照

説 明 source オブジェクトのウィンドウ領域を切り出し，その部分のみを対象オブジェクトにコピーする．対象オブジェクトの各サンプル点はあらかじめゼロで埋められている．

Note

- source のウィンドウ領域全体が格納できる最小の 2 の累乗サイズのオブジェクトとして対象オブジェクトを再生成して，対象オブジェクトの中央部にウィンドウ領域のサンプルデータをコピーする．
- 対象オブジェクトのウィンドウ領域はコピーした範囲に設定される．
- source オブジェクトのパラメータからサンプル点数 $N_x \times N_y$ 以外のパラメータが対象オブジェクトのパラメータにコピーされる．

```
WaveField& Subdivide(int mxy = 1)
WaveField& Subdivide(int mx, int my)
WaveField& Pixelation(int mxy = 1)
WaveField& Pixelation(int mx, int my)
```

戻り値　対象オブジェクトへの参照

説　明　1 番目の形式ではサンプリング数が x 方向にも y 方向にも 2 の mxy 乗倍になるようにサンプル点を多重化する．この関数では，オブジェクト全体の物理的大きさは変化せず，サンプリング数が増加した割合でサンプリング間隔が減少する (図 3.10)．2 番目の形式ではサンプリング数が x 方向に 2 の mx 乗倍，y 方向には 2 の my 乗倍になるようにサンプル点を多重化する．

Note

- 1 番目の形式ではサンプリング間隔は x 方向にも y 方向にも 2 の mxy 乗分の 1 になる．2 番目の形式ではサンプリング間隔が x 方向に 2 の mx 乗分の 1，y 方向には 2 の my 乗分の 1 になる．

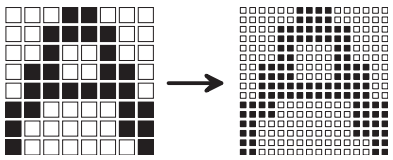


図 3.10　Subdivide() & Pixelation() の効果

3.4.24　複素振幅の図形を描くためのメンバー関数

```
WaveField& DrawLine(int x1, int y1, int x2, int y2, const Complex& val = Comp(1.0, 0.0))
```

戻り値　対象オブジェクトへの参照

説　明　オブジェクトをキャンバスとして，整数座標 (x1, y1) から (x2, y2) へ複素数値 val の直線を描く．

```
WaveField& Paint(int x, int y, const Complex& val = Comp(1.0, 0.0))
```

戻り値　対象オブジェクトへの参照

説　明　ローカル座標の (x, y) 点を含む閉領域を複素数値 val で塗りつぶす．

Note

- 閉領域とは val 以外の値を持つサンプル点で囲われた閉じた領域である.
- オブジェクトの境界は閉領域の境界とは見なされない.
- 領域が閉じていない場合の動作は未定義.
- 凹図形は処理できないので注意.

```
WaveField& PaintTriangle(int x1, int y1, int x2, int y2, int x3, int y3, const Complex&
val = Comp(1.0, 0.0))
WaveField& PaintTriangle(const PointArray& pa, const Complex& val = Comp(1.0, 0.0))
```

戻り値 対象オブジェクトへの参照

説　明 第 1 の形式では，ローカル座標の点 (x1, y1), (x2, y2), (x3, y3) で指定される三角形領域を複素数値 val で塗りつぶす．第 2 の形式では PointArray 型 pa の最初の 3 点の (x, y) 座標値で指定される三角形領域を複素数値 val で塗りつぶす．なおこの時，z 値は無視される．

Note

- いずれの座標値も対象オブジェクトのローカル座標値として解釈される.
- 第 1 の形式では 3 点がオブジェクトに含まれているかどうかはチェックされない．含まれていない場合の動作は未定義.
- 第 2 の形式では 3 点がオブジェクトの領域に含まれていない場合はクリッピング処理される.

```
WaveField& MultiplyTriangle(const PointArray& pa, const Complex& val)
```

戻り値 対象オブジェクトへの参照

説　明 PointArray 型 pa の最初の 3 点の (x, y) 座標値で指定される三角形領域内に複素数値 val を乗算する.

Note

- pa の座標値は対象オブジェクトのローカル座標値として解釈される.
- pa の z 座標値は無視される.
- 3 点がオブジェクトの領域に含まれていない場合はクリッピング処理される.

```
WaveField& MultiplyTriangleAperture(const PointArray& pa, const Complex& va =
Complex(0,0))
```

戻り値 対象オブジェクトへの参照

説　明 PointArray 型 pa の最初の 3 点の (x, y) 座標値で指定される三角形領域の外側に複素数値 val を乗算する.

Note

- 座標値は対象オブジェクトのローカル座標値として解釈される.

- pa の z 座標値は無視される．
- 3 点がオブジェクトの領域に含まれていない場合はクリッピング処理される．

```
WaveField& PaintPolygonShape(const PointArray& polygon, const Complex& val)
```

戻り値　対象オブジェクトへの参照

説　明　ローカル座標の PointArray 型の点群 polygon で指定される領域を複素数値 val で塗りつぶす．

Note

- いずれの座標値も対象オブジェクトのローカル座標値として解釈される．
- z 値は無視される．

```
WaveField& MultiplyPolygonShape(const PointArray& polygon, const Complex& val)
```

戻り値　対象オブジェクトへの参照

説　明　ローカル座標の PointArray 型の点群 polygon で指定される領域無いに複素数値 val を乗算する．

Note

- この関数では四角形を超える多角形は処理できない．polygon が 5 点以上だとエラーになる．
- いずれの座標値も対象オブジェクトのローカル座標値として解釈される．
- z 値は無視される．
- クリッピング処理される．

3.5 Complex/ComplexDouble クラス

単精度の複素数型が Complex クラス，また倍精度の複素数型が ComplexDouble クラスである．これらのクラスは基本的の同じメンバー関数を有している．

3.5.1 メンバー関数

以下では，Complex クラスのメンバー関数のみしか記されない場合でも基本的に ComplexDouble にも適用可能である．

```
Complex(const float& a = 0.0, const float& b = 0.0)
ComplexDouble(double a = 0.0, double b = 0.0)
```

戻り値 なし
説　明 標準のコンストラクタ

```
Complex(const ComplexDouble& c)
ComplexDouble(const Complex& c)
```

戻り値 なし
説　明 Complex と ComplexDouble を相互変換するためのコンストラクタ．

```
void SetReal(double a)
```

戻り値 なし
説　明 複素数の実部の値 a を設定する．

```
double GetReal(void) const
```

戻り値 実部の値
説　明 複素数の実部の値を取得する．

```
float& real(void)
double& real(void)
```

戻り値 実部への参照
説　明 複素数の実部への参照を戻す．したがって，左辺値として設定も可能．

```
void SetImag(double b)
```

戻り値　なし
説　明　複素数の虚部の値 b を設定する．

```
double GetImag(void) const
```

戻り値　虚部の値
説　明　複素数の虚部の値を取得する．

```
float& imag(void)
double& imag(void)
```

戻り値　虚部への参照
説　明　複素数の虚部への参照を戻す．したがって，左辺値として設定も可能．

```
double GetAmplitude(void)
```

戻り値　振幅値
説　明　複素数の振幅値を取得する．

```
void SetAmplitude(double val)
```

戻り値　なし
説　明　複素数の振幅値を設定する．この時，位相は以前の値のまま変化しない．

```
Phase GetPhase(void)
```

戻り値　Phase 型の位相値
説　明　複素数の位相角を取得する．

Note

- 位相角の範囲は $-\pi \sim +\pi$ である．

```
void SetPhase(double val)
```

戻り値 なし
説　明 複素数の位相角を設定する．この時，振幅は以前の値のまま変化しない．

```
double GetIntensity(void)
```

戻り値 光強度 (複素値の絶対値の 2 乗)
説　明 光強度 (複素値の絶対値の 2 乗) を取得する．

```
void SetIntensity(double val)
```

戻り値 なし
説　明 光強度 (複素値の絶対値の 2 乗) を設定する．この時，位相は以前の値のまま変化しない．

```
void Comp(float a = 0.0, float b = 0.0)
```

戻り値 なし
説　明 複素数の実部，虚部にそれぞれ値 a と b を代入する．

```
Complex Conjugate(void) const
```

戻り値 複素数の共役値
説　明 共役な複素数値を取得する．

3.5.2 オーバーロード演算子

```
Complex& operator=(const Complex& c)
Complex& operator=(const ComplexDouble& c)
ComplexDouble& operator=(const Complex& c)
ComplexDouble& operator=(const ComplexDouble& c)
```

戻り値 左辺値の参照
説　明 代入演算子

```
Complex& operator+=(const Complex& c)
Complex& operator-=(const Complex& c)
ComplexDouble& operator+=(const ComplexDouble& c)
ComplexDouble& operator-=(const ComplexDouble& c)
```

戻り値　左辺値の参照
説　明　加減算代入演算子

```
Complex& operator*=(const Complex& c)
Complex& operator*=(double c)
ComplexDouble& operator*=(const ComplexCouble& c)
ComplexDouble& operator*=(double c)
```

戻り値　左辺値の参照
説　明　乗算代入演算子

```
Complex& operator/=(const Complex& c)
Complex& operator/=(double c)
ComplexDouble& operator/=(const ComplexCouble& c)
ComplexDouble& operator/=(double c)
```

戻り値　左辺値の参照
説　明　除算代入演算子

```
Complex operator+(const Complex& rhs) const
Complex operator-(const Complex& rhs) const
ComplexDouble operator+(const ComplexCouble& rhs) const
ComplexDouble operator-(const ComplexCouble& rhs) const
```

戻り値　演算結果
説　明　加減算 2 項演算子

```
Complex operator*(const Complex& rhs) const
Complex operator*(double rhs) const
ComplexDouble operator*(const ComplexCouble& rhs) const
ComplexDouble operator*(double rhs) const
```

戻り値　演算結果

説　明　乗算 2 項演算子

```
Complex operator/(const Complex& rhs) const
Complex operator/(double rhs) const
ComplexDouble operator/(const ComplexCouble& rhs) const
ComplexDouble operator/(double rhs) const
```

戻り値　演算結果
説　明　除算 2 項演算子

```
Complex operator-(void) const
ComplexDouble operator-(void) const
```

戻り値　演算結果
説　明　単項演算子

```
bool operator==(const Complex& rhs) const
bool operator==(const ComplexCouble& rhs) const
```

戻り値　演算結果
説　明　関係演算子

3.5.3　Complex/ComplexDouble に関連したグローバル関数

```
double abs(ComplexDouble c)
```

戻り値　絶対値
説　明　$\mathrm{c} = a + ib$ に対して絶対値 $\sqrt{a^2 + b^2}$ を計算する関数.

```
Phase arg(ComplexDouble c)
```

戻り値　Phase 型の位相. 単位：ラジアン.
説　明　$\mathrm{c} = a + ib$ に対して位相角 $Arg(\mathrm{c}) = \tan^{-1} b/a$ を計算する関数.

```
double norm(ComplexDouble c)
```

戻り値　絶対値の 2 乗

説　明　c $= a + ib$ に対して絶対値の 2 乗 $a^2 + b^2$ を計算する関数.

```
ComplexDouble euler(double t)
```

戻り値　ComplexDouble 型の複素数値

説　明　Eular の公式により，実数 t に対して複素数 $\exp(it) = \cos t + i \sin t$ を計算する関数.

```
ComplexDouble exp(ComplexDouble c)
```

戻り値　ComplexDouble 型の複素数値

説　明　複素数 c $= a + ib$ に対して，複素数 $\exp(c) = \exp(a)(\cos b + i \sin b)$ を計算する関数.

```
ComplexDouble polar(double A, double t)
```

戻り値　ComplexDouble 型の複素数値

説　明　実数 A と t に対して，複素数 $A \exp(it) = A(\cos t + i \sin t)$ を計算する関数.

```
Complex Comp(float r = 0.0, float i = 0.0)
```

戻り値　Complex 型の複素数値

説　明　Complex 型の複素数値を作成し，それを戻すグローバル関数.

3.6 Phase クラス

2π の周期性を有する位相値 φ を表わすクラス．本質的には double 型と同じであり相互に代入できるが，Phase オブジェクトでは必ず $-\pi \leq \varphi < +\pi$ の範囲にあるように折り畳まれる．これを “wrap” と呼ぶ．

3.6.1 メンバー関数と演算子

```
Phase(void)
Phase(double phase)
Phase(float phase)
Phase(_Phase phase)
```

戻り値 なし

説　明 コンストラクタ．1 番目の形式では位相値は 0 に初期化される．2 番目と 3 番目は，それぞれ double と float からの変換．3 番目の形式は定義域への折り畳み (wrap) が不必要な場合に使用される．_Phase クラス参照.

```
operator double() const
```

戻り値 double 型に変換された位相値

説　明 double 型へ変換するためのキャスト演算子

Example

```
Phase ph = 3.1415;      //double型からPhase型への変換
double pd = (double) ph;//Phase型からdouble型への変換
```

```
Phase operator+(Phase p)
Phase operator-(Phase p)
```

戻り値 Phase 型の位相値

説　明 Phase 型位相値同士の加減算．従って結果の値は wrap される．

3.6.2 _Phase クラス

すでに wrap されている位相値に対して再 wrap の無駄を防ぐために定義されている．キャストのためのクラス．

```
_Phase(double phase)
_Phase(float phase)
```

戻り値　なし

説　明　コンストラクタ．1 番目の形式は double から，2 番目の形式は float からの変換を定義する．ただしこれらの変換は型の変換のみであり数値は何も変換されない．

```
operator double() const
```

戻り値　double 型に変換された位相値

説　明　double 型へ変換するためのキャスト演算子

Example

```
Phase p = (_Phase) PI/2;    //double型からPhase型への変換．wrap処理が行われない．
```

3.7 Vector/Point クラス

物理的な空間ベクトルまたは 3 次元空間の点を表すクラス．WFL では，Point クラスと Vector クラスは同一のクラスの別名である．

3.7.1 位置許容誤差

位置許容誤差とは，Vector クラス/Point クラスにおいて，同一の点を識別する際に許されるメートル単位の誤差のことである．すなわち，ある 2 点間の距離がこの誤差以下である場合は，その 2 点は同一の点であると識別される．この誤差は SetPositionTolerance() 関数と GetPositionTolerance() 関数を用いて取得・設定できる．この位置許容誤差は，Vector クラス/Point クラス以外においても様々な判定基準として用いられている．

Note

- (Rel 3.2.0 以降) デフォルトでは位置許容誤差は **1 ナノメートル**である．
- (Rel 3.1.0 以前) デフォルトでは位置許容誤差は **1 マイクロメートル**である．

3.7.2 メンバー関数

```
Vector(double x = 0, double y = 0, double z = 0)
Point(double x = 0, double y = 0, double z = 0)
```

戻り値 なし
説　明 コンストラクタ．要素 (x, y, z) のベクトルまたは座標 (x, y, z) の点を生成する．

```
double GetX() const
double GetY() const
double GetZ() const
```

戻り値 成分または座標値
説　明 それぞれ，x，y，z 方向の成分または座標値を取得する．

```
void SetX(double val)
void SetY(double val)
void SetZ(double val)
```

戻り値 なし
説　明 それぞれ，x，y，z 方向の成分または座標値を設定する．

```
double& X()
double& Y()
double& Z()
```

戻り値　成分または座標値

説　明　それぞれ，x，y，z 方向の成分または座標値への参照を取得する．

```
static Vector I(void)
static Vector J(void)
static Vector K(void)
```

戻り値　単位ベクトル

説　明　それぞれ，x，y，z 方向の単位ベクトルを取得する．

Example

```
Vector v;
v = Vector::I()*3 + Vector::J()*2 + Vector::K()*4; // ベクトル v=(3,2,4)
```

```
double GetLength(void) const
```

戻り値　ベクトルの長さ

説　明　ベクトルの長さを取得する．

```
Vector& SetLength(double len)
```

戻り値　対象ベクトルへの参照

説　明　ベクトルの方向を保ったまま長さ len を設定する．

```
double GetSqure(void) const
```

戻り値　ベクトルの長さの 2 乗

説　明　ベクトルの長さの 2 乗を取得する．

```
Vector& Normalize(void)
```

戻り値　対象ベクトルへの参照

説　明　対象ベクトルを単位ベクトルへ変換する．

```
Vector GetNormalized(void) const
```

戻り値 単位ベクトル

説　明 対象ベクトルの単位ベクトルを取得する．

```
double GetDistance(Point p) const
double GetDistance(const Line& line)const
double GetDistance(const Plane& plane)const
```

戻り値 距離

説　明 それぞれ，点 p，Line 型の直線 line，Plane 型の平面 plane との距離を取得する．

```
bool IsInline(const Line& line) const
```

戻り値 判定結果

説　明 Line 型の直線 line 上にあるかどうかを判定する．

Note

- この判定は，対象オブジェクトの line からの距離を計算して，それが位置許容誤差内であるかどうかを調べることによって行われる．
- 位置許容誤差の取得と設定にはそれぞれ GetPositionTolerance() 関数と SetPositionTolerance() 関数を用いる．

```
bool IsInplane(const Plane& plane) const
```

戻り値 判定結果

説　明 Plane 型の plane 上にあるかどうかを判定する．

Note

- この判定は，対象オブジェクトの平面 plane からの距離を計算して，それが位置許容誤差内であるかどうかを調べることによって行われる．
- 位置許容誤差の取得と設定にはそれぞれ GetPositionTolerance() 関数と SetPositionTolerance() 関数を用いる．

```
bool IsEmpty()
```

戻り値 判定結果

説　明 空のオブジェクトかどうかを判定する．

3.7.3　オーバーロード演算子

```
Vector operator-(void) const
```

戻り値　ベクトル
説　明　向きを逆転したベクトルを取得する単項演算子.

```
Vector operator*(double rhs) const
Vector operator/(double rhs) const
```

戻り値　演算結果のベクトル
説　明　スカラー rhs との乗除算を行う 2 項演算子.

```
Vector& operator*=(double rhs)
Vector& operator/=(double rhs)
```

戻り値　代入されたベクトルへの参照
説　明　スカラー rhs との乗除算を行う代入演算子.

```
Vector operator-(const Vector& rhs) const
Vector operator+(const Vector& rhs) const
```

戻り値　演算結果のベクトル
説　明　ベクトル rhs との加減算を行う 2 項演算子.

```
Vector& operator-=(const Vector& rhs)
Vector& operator+=(const Vector& rhs)
```

戻り値　代入されたベクトルへの参照
説　明　ベクトル rhs との加減算を行う代入演算子.

```
double operator*(const Vector& vec)
```

戻り値　内積の結果 (スカラー)
説　明　ベクトル vec との内積を行う 2 項演算子.

```
Vector operator&(const Vector& vec) const
```

戻り値 外積の結果 (ベクトル)

説　明 ベクトル vec との外積を行う 2 項演算子.

```
Vector& operator&=(const Vector& vec)
```

戻り値 代入されたベクトルへの参照

説　明 ベクトル vec との外積を行う代入演算子.

```
Vector& operator*=(const RMatrix& mat)
```

戻り値 代入されたベクトルへの参照

説　明 3×3 正方行列 mat を左からベクトルに乗算する代入演算子. 例えば,

```
v *= M;
```

は $\boldsymbol{v} \leftarrow \boldsymbol{M}\boldsymbol{v}$ に相当する演算を行う.

Note

- 行列とベクトルの乗算演算子で代入演算子ではないタイプは RMatrix クラスの乗算演算子として定義されている.

Example

```
Vector vec(1, 0, 0), vv;
RMatrix mat = RMatrixX(30*Deg);
vec *= mat;          // Vectorクラスの乗算演算子
vv = mat * vec;      // RMatrixクラスの乗算演算子
```

```
bool operator==(const Vector& rhs) const
bool operator!=(const Vector& rhs) const
```

戻り値 判定結果

説　明 ベクトル rhs と一致または不一致であれば true になる.

Note

- この判定は, 2 点 (位置ベクトル) の間の距離を計算して, それが位置許容誤差内であるかどうかを調べることによって行われる.
- 位置許容誤差の取得と設定にはそれぞれ GetPositionTolerance() 関数と wfl::SetPositionTolerance() 関数を用いる.

3.8　PointArray クラス

Point クラスの点オブジェクトの可変長配列．内部的には，3 点以下の場合には通常の配列となるように実装されているためメモリ利用効率が高い．3 点を超える場合には STL の vector<Point>として扱われる．この時の取り扱いの変化はシームレスであるため，通常それを意識する必要はない．この配列は可変長であるため，任意の位置に要素を挿入削除できる．

3.8.1　メンバー関数

```
PointArray(int n = 0)
PointArray(const Point& p)
PointArray(const LineSegment& ls)
```

戻り値　なし

説　明　コンストラクタ．第 1 の形式では点数 n の Point 型の配列を生成する．各点は空の点に初期化される．第 2 の形式では，Point 型の点オブジェクト p の 1 点だけを要素とする配列を生成する．第 3 の形式では，LineSegment 型の線分の始点と終点の 2 点を要素とする配列を生成する．

```
int GetN(void) const
```

戻り値　点数

説　明　点数を取得する．

```
const Point& GetPoint(int i) const
```

戻り値　Point 型の点オブジェクト

説　明　インデックス i の点オブジェクトへの参照を取得する．

Note

- i が GetN() メンバー関数で取得される点数以上の場合はエラーとなる．
- このメンバー関数で取得される参照には書き込みできない．従って左辺値にはなれない．

```
Point& At(int i) const
```

戻り値　Point 型の点オブジェクト

説　明　インデックス i の点オブジェクトへの参照を取得する．

Note

- i が GetN() メンバー関数で取得される点数以上の場合はエラーとなる．

- このメンバー関数で取得される参照には書き込みできるため左辺値になれる.

Example

```
PointArray pa(pp);
pa.At(0) *= 3.0;    // 先頭の要素に3を乗算する
```

```
void SetPoint(int i, const Point& p)
```

戻り値 なし

説　明 インデックス i の要素として Point 型の点オブジェクト p を設定する.

Note

- i が GetN() メンバー関数で取得される点数以上の場合はエラーとなる.
- 点数は増加しない.
- 点を追加する (点数を増加する) 場合には, Insert() メンバー関数を用いる.

```
void Insert(const Point& p)
void Insert(int i, const Point& p)
```

戻り値 なし

説　明 第 1 の形式では, 配列の末尾に Point 型の点オブジェクト p を挿入する. 第 2 の形式では, インデックス i の要素の直前に点オブジェクト p を挿入する.

Note

- i が GetN() メンバー関数で取得される点数を超える場合はエラーとなる.
- 配列の長さ (点数) は自動的に増加される.

```
void Remove(void)
void Remove(int i)
```

戻り値 なし

説　明 第 1 の形式では, 配列の末尾から点オブジェクトを一つ削除する. 第 2 の形式では, インデックス i の点オブジェクトを削除する.

Note

- i が GetN() メンバー関数で取得される点数以上の場合はエラーとなる.
- 配列の長さ (点数) は自動的に減少する.

```
void Clear(void)
```

戻り値　なし

説　明　全点を削除し，点数 0 の点群にする．

```
PointArray& Rotate(const RMatrix& r)
```

戻り値　対象オブジェクトへの参照

説　明　RMatrix 型の回転行列 r でこのオブジェクトを回転する．

```
PointArray GetBoundingBox(void) const
```

戻り値　外接直方体の対角点である 2 点の PointArray 型オブジェクト

説　明　全点をその中に含む外接直方体を求め，その対角点を戻す．戻り値の PointArray 型オブジェクトでは，x, y, z 座標値が最小の頂点がインデックス 0 の点，またこれらが最大の頂点がインデックス 1 の点となっている．

Note

- 点数が 0 個の場合はエラーになる．

Example

```
// PointArrayのpa1に点群が入っている
PointArray pa2 = pa1.GetBoundingBox();
cout << "最小座標値の点:" << pa2[0] << endl;
cout << "最大座標値の点:" << pa2[1] << endl;
cout << "中心点:" << (pa2[0] + pa2[1])/2.0 << endl;
```

```
Vector GetCenter(void) const
```

戻り値　Vector 型の位置座標

説　明　全点をその中に含む外接直方体の中心座標を取得する．

```
void SetCenter(const Vector& p)
```

戻り値　なし

説　明　全点をその中に含む直方体の中心座標が Vector 型の位置 p になるように点群を平行移動する．

```
void Localize(void)
```

戻り値　なし

説　明　全点をその中に含む直方体の中心座標が原点になるように点群を平行移動する．

```
double GetMaxX(void) const
double GetMaxY(void) const
double GetMaxZ(void) const
```

戻り値 最大座標値

説　明 それぞれ，点群の中での座標の最大値を取得する．

Note

- 点数が多い場合，この関数を個別に呼び出すとパフォーマンスが悪い．最大最小値が必要な場合は代わりに GetBoundingBox() メンバー関数を用いる．

```
double GetMinX(void) const
double GetMinY(void) const
double GetMinZ(void) const
```

戻り値 最小座標値

説　明 それぞれ，点群の中での座標の最小値を取得する．

Note

- 点数が多い場合，この関数を個別に呼び出すとパフォーマンスが悪い．最大最小値が必要な場合は代わりに GetBoundingBox() メンバー関数を用いる．

```
double GetWidth(void) const
double GetHeight(void) const
double GetDepth(void) const
```

戻り値 空間的な長さ

説　明 それぞれ，全点をその中に含む外接直方体の x 方向の幅，y 方向の高さ，z 方向の奥行きを取得する．

Note

- 点数が 0 個の場合はエラーになる．

```
Plane GetPlane(void) const
```

戻り値 Plane 型の平面オブジェクト

説　明 この点群の最初の 3 点を含む平面を取得する．

```
PointArray& ProjectionOn(const Plane& plane)
PointArray& ProjectionOn(const Plane& plane, const Line& line)
```

戻り値　対象オブジェクトへの参照

説　明　第 1 番目の形式では，この点群を Plane 型の平面 plane へ正投影する．第 2 番目の形式では，この点群を通り Line 型の直線 line に平行な直線に沿って，この点群を Plane 型の平面 plane へ投影する．

```
bool IsInplane(void) const
```

戻り値　判定結果

説　明　この点群の点が全て同一平面上にあるかどうか判定する．

```
void SaveAsCsv(const char* fname) const
```

戻り値　なし

説　明　ファイル名 fname の CSV 形式ファイルとして点群の座標を保存する．

3.8.2　オーバーロード演算子

```
Point& operator[](int i) const
```

戻り値　Point 型の点オブジェクト

説　明　インデックス i の点オブジェクトへの参照を取得する．左辺値として代入可能．

```
PointArray& operator=(const PointArray& rhs)
```

戻り値　対象オブジェクトへの参照

説　明　rhs に含まれる全ての点オブジェクトを対象オブジェクトにコピーする代入演算子．

```
PointArray& operator-=(const Vector& rhs)
PointArray& operator+=(const Vector& rhs)
```

戻り値　対象オブジェクトへの参照

説　明　含まれる全ての点オブジェクトに Vector 型のベクトル rhs を加減算する代入演算子．

```
PointArray& operator*=(double m)
PointArray& operator*=(const RMatrix& mat)
```

戻り値 Point 型の点オブジェクト

説　明 乗算代入演算子．第 1 の形式では，含まれる全ての点オブジェクトにスカラー値 m を乗算する．第 2 の形式では含まれる全ての点オブジェクトに左側より RMatrix 型の行列 mat を乗算する．

```
bool operator==(const PointArray& rhs)
```

戻り値 判定結果

説　明 等号演算子．含まれる全ての点オブジェクトが rhs の全ての点オブジェクトに等しいとき true になる．

```
bool operator!=(const PointArray& rhs)
```

戻り値 判定結果

説　明 不等号演算子．含まれる点オブジェクトの一つでも rhs の点オブジェクトと異なるとき true になる．

3.9 LineSegment クラス

PointArray クラスの派生クラス．長さが有限の線分オブジェクトを表す．このクラスは PointArray クラスのメンバー関数と演算子を継承しているため，回転や平行移動ができる．無限長の直線を表すためには Line クラスを用いる．

3.9.1 メンバー関数

```
LineSegment()
LineSegment(const Point& begin, const Point& end)
```

戻り値 なし

説　明 コンストラクタ．第 1 の形式では空の線分を生成する第 2 の形式では，始点が Point 型の点 begin で終点 end の線分を生成する．

```
const Point& GetBegin(void) const
const Point& GetEnd(void) const
```

戻り値 Point 型の点オブジェクト

説　明 それぞれ，始点と終点を取得する．

```
double GetLength(void) const
```

戻り値　線分の長さ
説　明　線分の長さを取得する.

```
Vector GetVector(void)const
```

戻り値　Vector 型のベクトル
説　明　始点から終点へのベクトルを取得する.

```
Vector GetUnitVector(void) const
```

戻り値　Vector 型のベクトル
説　明　単位方向単位ベクトルを取得する.

```
PointArray& Rotate(const RMatrix& r)
```

戻り値　対象オブジェクトへの参照
説　明　PointArray クラスから継承しているメンバー関数. RMatrix 型の回転行列 r で線分を回転する.

```
bool IsParallel(const Plane& plane) const
```

戻り値　平行なら true
説　明　対象の線分が Plane 型の plane と平行かどうか判定する.

```
bool Intersect(const Plane& plane) const
```

戻り値　交差していれば true
説　明　対象の線分が Plane 型の plane と交差しているかどうか判定する.

```
bool IsEmpty(void) const
```

戻り値　空の線分なら true
説　明　空の線分かどうか判定する.

Note

- 長さが位置許容誤差以下なら空と判定する.

3.9.2 オーバーロード演算子

```
bool operator==(const LineSegment& ls)
```

戻り値 判定結果
説　明 対象の線分が線分 ls と同一の線分かどうか判定する．

Note

- 始点と終点が逆転していても正しく判定する．

```
bool operator!=(const LineSegment& ls)
```

戻り値 判定結果
説　明 対象の線分が線分 ls と異なった直線かどうか判定する．

```
PointArray& operator=(const PointArray& rhs)
PointArray& operator-=(const Vector& rhs)
PointArray& operator+=(const Vector& rhs)
PointArray& operator*=(double m)
PointArray& operator*=(const RMatrix& mat)
```

戻り値 そのオブジェクトへの参照
説　明 PointArray クラスから継承している演算子．

3.10 Line クラス

LineSegment クラスの派生クラス．長さが無限の直線オブジェクトを表す．このクラスは PointArray クラスと LineSegment クラスのメンバー関数と演算子を継承しているため，回転や平行移動ができる．有限長の線分を表すためには LineSegment クラスを用いる．

3.10.1 メンバー関数

```
Line(void)
Line(const Point& p, const Vector& n)
Line(const LineSegment& ls)
```

戻り値 なし
説　明 コンストラクタ．第 1 の形式ではグローバル座標で $(0, 0, z)$ を通る光軸に対応する直線を生成する．第 2

の形式では，Point 型の点 p を通り，Vector 型のベクトル n のに沿った直線を生成する．第 3 の形式では，LineSegment 型の線分 ls を含む直線を生成する．

```
const Point& GetPoint(void) const
```

戻り値　点

説　明　この直線オブジェクトを定義している Point 型の点オブジェクトを取得する．

```
const Vector GetUnitVector(void) const
```

戻り値　ベクトル

説　明　LineSegment クラスの GetUnitVector() メンバー関数の継承．この直線オブジェクトの Vector 型の単位方向ベクトルを取得する．

Note

- 直線は，点 $\boldsymbol{p}$ と単位方向ベクトル $\boldsymbol{n}$，助変数 t に対して $\boldsymbol{r} = t\boldsymbol{n} + \boldsymbol{p}$ で定義される点 $\boldsymbol{r}$ の集合として定義されている．

```
double GetDistance(const Point& p) const
```

戻り値　距離

説　明　Point 型の点 p との距離を取得する．

```
const Point& GetVerticalPoint(const Point& p) const
```

戻り値　垂線の足

説　明　Point 型の点 p からの垂線の足となる Point 型の点オブジェクトを取得する．

```
Point GetIntersectionPoint(const Plane& plane) const
```

戻り値　交点

説　明　Plane 型の平面 plane との交点である Point 型の点オブジェクトを取得する．

```
bool IsParallel(Plane& plane) const
```

戻り値　判定結果

説　明　対象の直線が Plane 型の平面オブジェクト plane と平行どうか判定する．

Note

- 対象直線の方向ベクトルと平面の法線ベクトルの内積の平方根が位置許容誤差以下の場合に true と判定する．

```
bool Include(const Point& p) const
```

戻り値 判定結果

説　明 対象の直線が Point 型の点オブジェクト p を含むかどうか判定する．

Note

- 直線と点オブジェクト p の距離が位置許容誤差以下の場合に true と判定する．

```
PointArray& Rotate(const RMatrix& r)
```

戻り値 対象オブジェクトへの参照

説　明 PointArray クラスから継承しているメンバー関数．RMatrix 型の回転行列 r で線分を回転する．

```
bool IsEmpty(void) const
```

戻り値 判定結果

説　明 LineSegment クラスの IsEmpty() メンバー関数の継承．空のオブジェクトかどうかを判定する．

Note

- 方向ベクトルの長さが位置許容誤差以下の場合に空と判定する．

3.10.2 オーバーロード演算子

```
bool operator==(const Vector& rhs) const
```

戻り値 判定結果

説　明 rhs が同一の直線かどうか判定する．

Note

- 対象直線と rhs の点オブジェクト p の距離が位置許容誤差以下で，かつ両者の方向ベクトルが等しいまたは反対向きで等しい場合に true と判定する．

```
bool operator!=(const Vector& rhs) const
```

戻り値 判定結果

説　明　rhs が異なった直線かどうか判定する．

```
PointArray& operator=(const PointArray& rhs)
PointArray& operator-=(const Vector& rhs)
PointArray& operator+=(const Vector& rhs)
PointArray& operator*=(double m)
PointArray& operator*=(const RMatrix& mat)
```

戻り値　そのオブジェクトへの参照

説　明　PointArray クラスから継承している演算子．

3.10.3　静的メンバー関数

```
static Line X(void)
static Line Y(void)
static Line Z(void)
```

戻り値　Line 型の直線オブジェクト

説　明　それぞれ，x 軸，y 軸，z 軸の直線オブジェクトを取得する．

3.11　Plane クラス

PointArray クラスの派生クラス．無限の広がりを有する平面オブジェクトを表す．このクラスは PointArray クラスのメンバー関数と演算子を継承しているため，回転や平行移動ができる．

WFL3 以降では，平面に裏表が定義されている．GetNormalVector() メンバー関数で取得される法線ベクトルは表側の面からの法線ベクトルである．

3.11.1　メンバー関数

```
Plane(void)
Plane(const Point& p1, const Point& p2, const Point& p3)
Plane(const Point& p, const Vector& n)
```

戻り値　なし

説　明　コンストラクタ．第 1 の形式では $(x, y, 0)$ 平面を生成する．第 2 の形式では，Point 型の 3 点 p1, p2, p3 を含む平面を生成する．第 3 の形式では，Vector 型の法線ベクトルが n で 1 点 p を含む平面を生成する．

Note

- 第 1 の形式の場合，3 点の順序が右ねじの法則に合うように面の表側 (法線ベクトル) が設定される．

- p1, p2, p3 の 2 点以上が同一点の場合や，n がゼロベクトルの場合はエラーとなる．

```
const Vector& GetNormalVector(void) const
```

戻り値 Vector 型のベクトル
説　明 平面の表側の法線ベクトルを取得する．

Note

- WFL3 以降では，法線ベクトルは表側の面からのそれとなり，平面の位置等では変化しない．PointArray クラスの operator*=() メンバー関数による回転や，Reverse() メンバー関数による反転で表面の方向を変えることができる．
- WFL3 以前では， 平面の法線ベクトルは，その平面の原点からの距離が正値になるように定義される．すなわち，法線ベクトル $\boldsymbol{n}$ とすると，

$$\boldsymbol{n} \cdot \boldsymbol{r} = n_x x + n_y y + n_z z = d \tag{3.9}$$

  において，$d \geq 0$ となるように $\boldsymbol{n}$ が定義される．そのため，平面の位置の変化により法線ベクトルが反転することがある．
- 上式の d は GetDistance() メンバー関数で取得できる．

```
void SetNormalVector(const Vector& n)
```

戻り値 無し
説　明 Vector 型のベクトル n を平面の法線ベクトルとして設定する．

```
void Reverse(void)
```

戻り値 無し
説　明 平面の向きを逆転する．

```
int GetSide(const Point& p) const
```

戻り値 点 p の位置．-1: 裏側の半空間，0: 面内，1: 表側の半空間
説　明 この関数は，点 p が平面のどちら側に位置しているかを検出する．

```
double GetDistance(void) const
double GetDistance(const Point& p) const
```

戻り値 距離

説　明　第 1 の形式では，原点からの距離を取得する．すなわち式 (3.9) の d を取得する．第 2 の形式では，Point 型の点 p からの距離を取得する．

```
double GetX(double y, double z) const
double GetY(double x, double z) const
double GetZ(double x, double y) const
```

戻り値　座標値

説　明　二つの座標値から平面上の点の残りの一つの座標値を取得する．

Note

- 求めたい座標値の軸と対象平面が平行の場合にはエラーとなる．

```
Point GetIntersectionPoint(const Line& line) const
```

戻り値　Point 型の点オブジェクト

説　明　Line 型の直線 line との交点を取得する．

```
Line GetIntersectionLine(const Plane& plane) const
```

戻り値　Line 型の直線オブジェクト

説　明　Plane 型の平面 plane との交線を取得する．

```
Point GetVerticalPoint(const Point& p) const
```

戻り値　Point 型の点オブジェクト

説　明　Point 型の点 p からの垂線の足となる点を取得する．

```
PointArray& Rotate(const RMatrix& r)
```

戻り値　対象オブジェクトへの参照

説　明　PointArray クラスから継承しているメンバー関数．RMatrix 型の回転行列 r で平面を回転する．

```
bool IsParallel(const Line& line) const
bool IsParallel(const Plane& plane) const
```

戻り値　判定結果

説　明　Line 型の直線 line または Plane 型の平面 plane と平行かどうか判定する．

```
Include(const Point& p) const
Include(const Plane& plane) const
```

戻り値　判定結果

説　明　Point 型の点 p または Plane 型の平面 plane が対象平面に含まれているかどうか判定する．

Note

- 点と平面の距離が位置許容誤差以下なら true となる．
- Plane 型に適用した場合，双方の面の裏表に関係なく，平面が重なっている場合に true を返す．面の表裏も含めて同一平面かどうかを判定するためには==演算子を用いる．

```
bool IsEmpty(void) const
```

戻り値　判定結果

説　明　空のオブジェクトかどうか判定する．

Note

- 法線ベクトルの長さが位置許容誤差以下なら true となる．

3.11.2　オーバーロード演算子

```
bool operator==(const Plane& plane) const
```

戻り値　判定結果

説　明　対象の平面が plane と同一の平面かどうか判定する．

Note

- 法線ベクトルが等しく，原点からの距離が位置許容誤差以下で等しい場合に true となる．
- WFL2 以前では幾何的に重なる面であれば true となったが，WFL3 以降では平面の表裏も含めて等しい場合に true となる．表裏に無関係に幾何的に重なっていることを検出するためには Include() メンバー関数を用いる．

```
bool operator!=(const Plane& plane) const
```

戻り値　判定結果

説　明　対象の平面が plane と異なった平面かどうか判定する．

```
PointArray& operator=(const PointArray& rhs)
PointArray& operator-=(const Vector& rhs)
PointArray& operator+=(const Vector& rhs)
PointArray& operator*=(double m)
PointArray& operator*=(const RMatrix& mat)
```

戻り値　そのオブジェクトへの参照

説　明　PointArray クラスから継承している演算子．+=演算子や-=演算子で移動，*=演算子で面の回転ができる．

3.11.3　静的メンバー関数

```
static Plane XY(void)
static Plane YZ(void)
static Plane ZX(void)
```

戻り値　Plane 型の平面オブジェクト

説　明　それぞれ，$(x, y, 0)$ 平面，$(0, y, z)$ 平面，$(x, 0, z)$ 平面の平面オブジェクトを取得する．

3.12　SFrequency クラス

Vector クラスの派生クラス．平面上のフーリエ周波数空間における座標 (u, v, w) を表現するクラス．フーリエ周波数空間では，

$$u^2 + v^2 + w^2 = \lambda^{-2} \tag{3.10}$$

の関係があるため，座標 u, v, w は互いに独立ではないことに注意しなければならない．特に，Vector クラスの派生クラスであることから，Vector クラスの演算子やメンバー関数を用いることができるが，その場合は通常 w は正しい値になっていない．u と v に応じて w を更新するためには Normalize() メンバー関数を用いなければならない．

3.12.1　メンバー関数

```
SFrequency()
SFrequency(double lmd, double u, double v)
```

戻り値　なし

説　明　コンストラクタ．第 1 の形式では空のオブジェクトを生成する．第 2 の形式では，波長 lmd で，x 方向周波数 u, y 方向周波数 v のフーリエ座標オブジェクトを生成する．この場合，z 方向周波数の w は u と w から式 (3.10) を用いて自動的に計算される．

```
double GetWavelength(void) const
```

戻り値 波長
説　明 波長を取得する．

```
void SetWavelength(double val)
```

戻り値 なし
説　明 val を波長として設定する．

```
SFrequency& SetU(double val)
SFrequency& SetV(double val)
```

戻り値 対象オブジェクトへの参照
説　明 それぞれ，val を x 方向または y 方向の空間周波数として設定する．z 方向周波数の w は式 (3.10) を用いて自動的に計算され設定される．

```
SFrequency& SetUV(double u, double v)
```

戻り値 なし
説　明 u を x 方向周波数，また v を y 方向周波数として同時に設定する．z 方向周波数の w は式 (3.10) を用いて自動的に計算され設定される．

```
double GetU(void) const
double GetV(void) const
```

戻り値 空間周波数値．単位 $[\mathrm{m}^{-1}]$
説　明 それぞれ，x 方向と y 方向の空間周波数を取得する．

```
double GetW(void) const
```

戻り値 空間周波数値．単位 $[\mathrm{m}^{-1}]$
説　明 あらかじめ計算済みの z 方向の空間周波数を取得する．SetU() メンバー関数，SetV() メンバー関数，SetUV() メンバー関数等の関数を用いた直後は正しく設定された値が取得できるが，回転行列の乗算などの演算後は w 値が正しくないため，この関数を呼び出す前に Normalize() メンバー関数を実行する必要がある．

```
double CalcW(void) const
```

戻り値　空間周波数値．単位 $[\mathrm{m}^{-1}]$

説　明　計算して z 方向の空間周波数を取得する．回転行列の乗算などの演算後でも正しい値が取得できる．ただし，対象オブジェクト内の w 値は更新されない．

```
void Normalize(void)
```

戻り値　なし

説　明　z 方向の空間周波数を計算して対象オブジェクト内の w 値を更新する．

```
SFrequency& Rotate(const RMatrix& r)
```

戻り値　対象オブジェクトへの参照

説　明　Vector クラスから継承しているメンバー関数．RMatrix 型の回転行列 `r` で空間周波数座標点を回転する．

```
bool IsEvanescent(void)
```

戻り値　空間周波数がエンバネッセント領域にあるなら true

説　明　対象オブジェクトの空間周波数がエバネッセント領域にあるか確認し $(u^2 + v^2 > \lambda^{-2})$，そうであれば `true` を返す．

3.12.2　オーバーロード演算子

```
SFrequency operator-(const SFrequency& rhs)
SFrequency operator+(const SFrequency& rhs)
```

戻り値　SFrequency 型の演算結果

説　明　空間周波数オブジェクト同士の加減算を行う演算子．演算後の w 値 (z 方向周波数) も式 (3.10) を用いて計算され設定される．

3.13　RMatrix クラス

SqrMatrix クラスの派生クラス．回転や座標回転のための回転行列として用いる 3×3 の正方行列のクラス．

3.13.1　メンバー関数

```
RMatrix(void)
RMatrix(const Vector& dest, const Vector& source)
```

戻り値 なし

説　明 コンストラクタ．1 番目の形式では 3 × 3 のゼロ行列を生成する．2 番目の形式では，ロドリゲスの回転公式 (Rodrigues' rotation formula) を用いて，Vector 型のベクトル source の方向を Vector 型のベクトル dest の方向に一致させるための回転行列を生成する．

```
RMatrix GetInverse(void)
```

戻り値 RMatrix 型の行列

説　明 逆行列を取得する．

3.13.2 オーバーロード演算子

```
Vector operator*(const Vector& p) const
```

戻り値 Vector 型の演算結果

説　明 この行列を Vector 型のベクトル p に左から乗算した結果を求める乗算演算子．

```
RMatrix operator*(const RMatrix& rhs) const
```

戻り値 RMatix 型の演算結果

説　明 この行列を RMatrix 型の行列 rhs に左から乗算した結果を求める乗算演算子．

```
SFrequency operator*(const SFrequency& sf) const
```

戻り値 SFrequency 型の演算結果

説　明 この行列を SFrequency 型のフーリエ空間座標オブジェクト sf に左から乗算した結果を求める乗算演算子．

Note

- 演算結果に対して SFrequency クラスの Normalize() メンバー関数の処理は行われないので，そのままでは演算結果の w 値は正しい値ではない．

3.13.3 静的メンバー関数

```
static const RMatrix& RotationX(double t)
static const RMatrix& RotationY(double t)
static const RMatrix& RotationZ(double t)
```

戻り値　RMatrix 型の 3×3 回転行列

説　明　それぞれ，x 軸，y 軸，z 軸の周りに t[rad] だけ回転するための回転行列を取得する．

Note

- 同じ関数はグローバル関数の RMatrixX() 関数等としても定義されている．

```
static const RMatrix& CRotationX(double t)
static const RMatrix& CRotationY(double t)
static const RMatrix& CRotationZ(double t)
```

戻り値　RMatrix 型の 3×3 回転行列

説　明　それぞれ，x 軸，y 軸，z 軸の周りに t[rad] だけ座標回転するための回転行列を取得する．

Note

- 同じ関数はグローバル関数の CRMatrixX() 関数等としても定義されている．

3.14 StopWatch クラス

StopWatch はストップウォッチの概念を実装したクラスであり，プログラム中のある区間の実行時間やある特定の処理の実行時間を計測することに適した軽量なクラスである．使用方法についてはサンプルを参照.

3.14.1 メンバー関数

```
StopWatch(void)
```

戻り値 なし
説　明 コンストラクタ．ストップウォッチオブジェクトを生成し時間をリセットする.

```
void Reset(void)
```

戻り値 なし
説　明 ストップウォッチをリセットする．計測値はゼロクリアされる.

```
void Start(void)
```

戻り値 なし
説　明 ストップウォッチのスタートボタンを押す動作．通常のストップウォッチの動作をシミュレートするので，スタートボタンを押した時にリセット状態であればゼロから計測を始める．スタートを押した時にストップ状態であれば一時停止状態から復帰し，計測を再開する.

```
void Stop(void)
```

戻り値 なし
説　明 ストップウォッチのストップボタンを押す動作．通常のストップウォッチの動作をシミュレートするので，計測がスタートしている場合は一時停止する．計測がストップしている場合は何も起こらない.

```
double LapTime(void)
```

戻り値 経過時間 (単位：秒)
説　明 前回 LapTime() 関数が実行されてからの経過時間を秒単位で返す.

```
double Read(void)
```

戻り値　秒単位の計測時間

説　明　ストップウォッチの計測値を読み出す.

Example StopWatch クラスの利用法

```
StopWatch a, b;
WaveField wfa(256, 256), wfb(1024, 1024);
int i;
for (i = 0; i < 10000; i++)
{
    b.Start();      // ↓ b スタート
    wfb.Fft(-1);    // wfbのフーリエ変換
    b.Stop();       // ↑ b ストップ

    a.Start();      // ↓ a スタート
    //**********
    オブジェクトwfaに関する何かの処理
    ************//
    a.Stop();       // ↑ a ストップ

    b.Start();      // ↓ b スタート
    wfb.Fft(1);     // wfbの逆フーリエ変換
    b.Stop();       // ↑ b ストップ
}

a.Start();          // ↓ a スタート
//**********
オブジェクトwfaに関する追加の処理
************//
a.Stop();           // ↑ a ストップ

Printf("wfbのフーリエ変換・逆変換の時間:%f\n", b.Read());
Printf("wfaの処理にかかった時間:%f\n", a.Read());
```

3.15 ディスクリプタ

WaveFieldLib に実装された関数の中には，必要な定数や参照表などの計算を事前に行って高速に処理する関数がある．何度もそのような関数を実行する際に定数を毎回計算することは無駄であるので，このような定数・参照表等は一度だけ計算して再利用する．それを保持するのがディスクリプタである．従って，ディスクリプタは一度作成しておけば，同じ計算を繰り返す場合にもう一度作成する必要はない

3.15.1 SphericalWaveDescriptor クラス

WaveField クラスの AddSphericalWaveSqr() メンバー関数等で用いる球面波の高速計算のためのディスクリプタ．

```
SphericalWaveDescriptor(double wavelength, double px, double py, Vector ref=Vector(0, 0, 1.0))
```

戻り値 なし

説　明 SphericalWaveDescriptor を生成するコンストラクタ．波長 wavelength，球面波の高速計算する対象となる WaveField クラスのサンプリング間隔 (px, py) を引数としてディスクリプタを生成する．Vector 型 ref はホログラムとして干渉縞を作成する場合に想定される参照光の波動ベクトル (方向のみ必要) である．

```
bool IsSameAs(double wavelength, double px, double py, Vector ref=Vector(0, 0, 1.0))
```

戻り値 判定結果

説　明 生成済みのディスクリプタのパラメータをチェックする．同一のパラメータであれば true となる．パラメータはコンストラクタのパラメータと同じ．

Note

- 同一パラメータのディスクリプタは再利用できる．

```
bool operator==(const SphericalWaveDescriptor& swd)
```

戻り値 判定結果

説　明 対象ディスクリプタと swd が同一であるかを判定する．

3.15.2 ShiftedFresnelPropDescriptor クラス

WaveField クラスの ShiftedFresnelProp() メンバー関数等で用いる球面波の高速計算のためのディスクリプタ．

```
ShiftedFresnelPropDescriptor(double distance, double dpx, double dpy, double spx, double
spy, double wavelength, int nx, int ny)
ShiftedFresnelPropDescriptor(double distance, double destPx, double destPy, const
WaveField& source)
```

戻り値　なし

説　明　ShiftedFresnelPropDescriptor を生成するコンストラクタ．第 1 の形式では，伝搬距離 distance，伝搬先のサンプリング間隔 dpx, dpy，伝搬元のサンプリング間隔 spx，spy，波長 wavelength，サンプリング数 nx，ny のシフトフレネル伝搬を計算するためのディスクリプタを生成する．第 2 の形式では，伝搬元のサンプリング間隔，波長，サンプリング数のパラメータは source から取得する．

Note

- 同一パラメータのディスクリプタは互換性があり，再利用できる．
- 伝搬距離，伝搬先のサンプリング間隔，伝搬元のサンプリング間隔，波長，サンプリング数が等しい場合に同一パラメータとなる．

```
bool IsCompatibleWith(double distance, double dpx, double dpy, double spx, double spy,
double wavelength, int nx, int ny) const
bool IsCompatibleWith(double distance, double dpx, double dpy, const WaveField& source)
const
```

戻り値　判定結果

説　明　生成済みのディスクリプタのパラメータをチェックし，デスクリプタの互換性があれば true となる．パラメータはコンストラクタのパラメータと同じ．

```
bool operator==(const ShiftedFresnelPropDescriptor& sfpd) const
```

戻り値　判定結果

説　明　生成済みのディスクリプタのパラメータをチェックし，デスクリプタの互換性があれば true となる．

第 II 部

PolygonSource ライブラリ (PSL)

第 4 章

開発環境

4.1 PolygonSource ライブラリを用いる開発環境

4.1.1 開発環境とコンパイラ

PolygonSource ライブラリを用いてプログラム開発ができる開発環境とコンパイラは WaveFieldLib と基本的に同じである.

4.1.2 ディレクトリ構造とファイル

注意
PolygonSource ライブラリ 1.0 Beta2 以降では，それ以前のバージョンとインストールポイントが変更されている.

PolygonSource ライブラリのインストーラは次のディレクトリに必要なファイルをインストールする．なおデフォルトでは < インストールディレクトリ >=c:\WaveFieldTools である.

4.1.2.1 ヘッダ

ヘッダファイルは 64 ビット版と 32 ビット版で共通である.
< インストールディレクトリ >\include\
< インストールディレクトリ >\include\psl\
ファイル：*.h

4.1.2.2 インポートライブラリ

PolygonSource ライブラリを呼び出すプログラムのビルド時には以下のフォルダにあるインポートライブラリが必要である.
< インストールディレクトリ >\lib\win32\
32 ビット版のインポートライブラリ
< インストールディレクトリ >\lib\
64 ビット版のインポートライブラリ
ファイル：psl.lib

4.1.2.3 ダイナミックリンクライブラリ

PolygonSource ライブラリを呼び出すプログラムの実行時には以下のフォルダにあるダイナミックリンクライブラリが必要である．これは PATH の通ったフォルダーか実行ファイル (.exe ファイル) と同じフォルダーに置かれている必要がある.
< インストールディレクトリ >\bin\win32\
　32 ビット版のダイナミックリンクライブラリ
< インストールディレクトリ >\bin\
　64 ビット版のダイナミックリンクライブラリ
ファイル：psl.dll,

4.1.2.4 データ

ポリゴン光波を生成するために必要なデータやモデルファイル等が保持されている.

＜インストールディレクトリ＞\SampleData\
ファイル：Diffuser1024x1024.wf，物体モデルファイル等

4.1.2.5 サンプルコード

チュートリアルのサンプリコードの一部は以下に存在する．
＜インストールディレクトリ＞\SampleCode\
ファイル:*.cpp

第 5 章

チュートリアル

5.1 PolygonSource ライブラリを用いたプログラミングの基本

5.1.1 単一のポリゴンの光波を計算する

PolygonSource ライブラリを用い，ポリゴン法 [9, 10] により単一のポリゴン (正三角形) の光波を計算するプログラムのソースコードを次に示す[*1].

Example 傾いた正三角形ポリゴンの光波計算 (ソース: ExTrianglePolygon.cpp)

```
#include    <psl.h>
using namespace wfl;
using namespace psl;

int main(void)
{
    Start();

    // x-z平面上で正三角形のポリゴンデータ作成
    psl::Polygon triangle(3);
    triangle[1].SetX(0.5);
    triangle[1].SetZ(-sqrt(3.0) / 2.0);
    triangle[2] = triangle[1];
    triangle[2].SetX(-0.5);

    // ポリゴンの大きさ，位置，傾き設定
    triangle.Localize();            //ポリゴンの中心を原点にする
    triangle *= 2e-3;               //1辺の長さを設定．単位:m
    triangle *= RMatrixY(15 * Deg)*RMatrixX(30 * Deg);  //傾き

    // フレームバッファの設定
    double px = 4e-6, py = 4e-6;    //サンプリング間隔
    int nx = 1024, ny = 1024;       //サンプリング数
    WaveField frame(nx, ny, px, py);//フレームバッファの位置は(x,y,0)面
    frame.Clear();

    //シェーダーの設定
    //WaveField diffuser(0);        //特殊な拡散位相を用いる場合
    //diffuser.LoadWf("diffuser1024x1024.wf"); //拡散位相読み込み
    Vector light(-1.0, -1.0, -1.0); //照明光の方向
    double env = 0.3;               //環境光の割合
    double gam = 0.5;               //補正制限値
    TfbFlatShading flatShader(/*diffuser,*/ gam, light, env); //乱数位相によるフラットシェーディング

    // SurfaceBuilderの設定
    SurfaceBuilder sb(frame);
    sb.SetShader(flatShader);       //上で用意したシェーダーを組み込む

    // ポリゴンからの光波を計算してフレームバッファに加算
    sb.SetCurrentPolygon(triangle);
    sb.AddPolygonField(frame);

    frame.SaveAsWf("frame.wf");
}
```

[*1] 2010/04/22 SurfaceBuilder クラスの大幅な仕様変更により，サンプルコードが書き直されている．

5.1.1.1　ヘッダファイル

ヘッダファイルとして psl.h をインクルードする．PolygonSource ライブラリのヘッダファイルは多数のファイルに分かれているが，psl.h をインクルードすることにより関連するヘッダファイルがすべて読み込まれる．

5.1.1.2　名前空間

PolygonSource ライブラリの関数やクラスは全て PolygonSourceLib 名前空間内で定義されている．psl はその別名である．従って，名前空間 psl の利用を宣言するのが一般的である．

Note

- Polygon クラスにおいてのみ，特別に名前空間 psl を明示的に示す必要がある．これは名前空間無しで定義された Polygon 識別子との区別が必要なためである．従って，ソースリストの 10 行目で示したとおり
  psl::Polygon triangle(3)
  の様に記述する必要がある．

5.1.1.3　初期化関数

PolygonSource ライブラリは WaveField ライブラリ (WFL) をベースとして作成されているので，原則としてまず初めに WFL の wfl::Start() 関数を実行する．

Note

- ビルド時にはライブラリファイル psl.lib をリンクするが，これを明示的に設定する必要はない．「追加インストール作業」が正しく行われていれば，psl.h の読み込みによって自動的に psl.lib がリンクされる．
- 実行時には，PATH の通ったディレクトリか，実行ファイル (*.EXE) と同じディレクトリに psl.dll が必要である．

ソースリストの 10–14 行目では，まず計算するポリゴンの形状を設定している．ここでは三角形ポリゴンであるので，10 行目で Polygon クラスのコンストラクタで 3 点のポリゴンを作成している．各点の座標値は $(x, y, z) = (0, 0, 0)$ で初期化される．この例では最初の点 triangle[0] を原点に置いているので初期値のまま設定していない．この三角形は正三角形であり，まず $(x, 0, z)$ 平面上で座標値設定を行うため，残りの 2 点の x と z 座標のみを設定している．なお，この段階では各辺の長さは 1[m] としている．

次にソースリストの 17–19 行目では，上で設定した座標のポリゴンに対して物理的な大きさや傾き，位置を設定している．17 行目では PointArray クラスの Localize() メンバー関数を呼び出し，ポリゴンを含む外接矩形の中心を原点に設定しなおしている．また，18 行目ではオーバーロード演算子を用いてサイズを変更している．ここでは各辺の長さが 2[mm] になるようにしている．19 行目ではやはり RMatrix クラスとオーバーロード演算子を用いてポリゴンの傾きを変えている．この例では，まず x 軸周りに 30 度回転し，次に y 軸周りに 15 度回転している．

22–25 行目では，ポリゴン光波を格納するフレームバッファを用意している．ここで，フレームバッファとは WaveField クラスのオブジェクトであり，一般に光軸 (グローバル z 軸) に垂直な平面上での複素振幅分布である．フレームバッファの物理的な位置は，一般に計算したいポリゴンの近傍に設定する方が計算が速い．フレームバッファの位置とポリゴンの位置が離れていると長い計算時間を要する場合や計算に失敗する場合がある．この例ではフレームバッファは $(x, y, 0)$ 面にあり，ポリゴンは，17 行目の Localize() メンバー関数の呼び出しによりほぼ原点付近にあるので，フレームバッファと重なる位置にポリゴンが存在している．

28–33 行目では，シェーダーオブジェクト作成と設定を行っている．シェーダーオブジェクトはポリゴンをどのようにレンダリングするかを決定するオブジェクトであり，シェーディング方法やテクスチャマッピング等の設定を行うものである．ここでは，最も単純なフラットシェーディングを設定している．28-29 行目の拡散位相の読み込みは，ここではコメントアウトしてある．これは，特殊なケース以外では拡散位相を用いるより十分精度が高い乱数位相を用いる

ほうが簡単なためである．特殊な性質を備えた拡散位相を読み込む場合はここをコメントインする．30–32 行目で照明光の方向 (照明の光が進行する方向を表すベクトル)，環境光の割合，さらに補正制限値を設定している．33 行目では，やはり拡散位相の指定がコメントアウトしてある．このようにすることにより，自動的に乱数位相が用いられる．特定の性質の拡散位相を用いる場合は，33 行目のコメントアウトを取り除く．

なお，照明光の方向と環境光の割合はどのようなデザインをしたいかで決まるが，補正制限値はホログラムを作成/再生する方法により決まる．

36–37 行目では，SurfaceBuilder クラスのオブジェクト作成と設定を行っている．SurfaceBuilder クラスは，ポリゴン光波を計算するために中心的な役割を果たすクラスである (6.4 節参照)．必要に応じて，計算精度に関係するパラメータ (6.3.3 節参照) を設定する必要があるが，ここではそれらは全て既定値とし，37 行目でシェーダーオブジェクトを組み込んでいるだけである．

34 行目では，SurfaceBuilder 型 sb に計算するポリゴンを設定しており，35 行目でポリゴン光波を計算してフレームバッファ frame に加算している．複数のポリゴンを処理してフレームバッファに加算するためには，この SetCurrentPolygon() メンバー関数と AddPolygonField() メンバー関数を交互に繰り返して呼び出す．

5.1.2 物体モデルファイルを読み込んでその光波を計算する

PolygonSource ライブラリを用い，CG 用のモデラー等で作成した物体モデルファイルを読み込んでその光波を計算するプログラムのソースコードを次に示す．

Example 物体光波を計算する (ソース: ExCgModel.cpp)

```
1  #include    <psl.h>
2  using namespace wfl;
3  using namespace psl;
4
5  int main(void)
6  {
7      Start();    wfl::SetNumThreads();    //最大限にプロセッサコアを用いる
8
9      // 物体モデルファイルの読み込みと設定
10     IndexedFaceSet model;
11     model.LoadWrl("cube.wrl");           //読み込み
12     model.Localize();                    //物体を原点付近に置く
13     model.SetWidth(2e-3);                //外接矩形の横幅が2mmになるように物体サイズを設定
14
15     // フレームバッファの設定
16     double px = 4e-6, py = 4e-6;         //サンプリング間隔
17     int nx = 1024, ny = 1024;            //サンプリング数
18     WaveField frame(nx, ny, px, py);     //フレームバッファの位置は(x,y,0)面
19     frame.Clear();
20
21     //シェーダーの設定
22     //WaveField diffuser(0);             //特殊な拡散位相を用いる場合
23     //diffuser.LoadWf("diffuser1024x1024.wf"); //拡散位相の読み込み
24     Vector light(-1.0, -3.0, -1.0);      //照明光の方向
25     double env = 0.1;                    //環境光の割合
26     double gamma = 0.3;                  //補正制限値
27     TfbFlatShading flatShader(/*diffuser, */ gamma, light, env); //フラットシェーディング
28
29     // SurfaceBuilderの設定
30     SurfaceBuilder sb(frame);            //帯域制限のため，サンプル数とサンプル間隔等を正しく設定
31     sb.SetBandLimitMethod(3);            //帯域制限をレベル3にする．
32     sb.SetCenter(Point(0, 0, 0));        //帯域制限のため，ホログラムの中心位置(0,0,0)を設定
33     sb.SetDiffractionRatio(0.9);         //回折率設定
34     sb.SetCullingRate(0.6);              //カリング率設定
35     sb.SetShader(flatShader);            //上で用意したシェーダーを組み込む
36
37     // 物体モデルmodelからの光波を計算してフレームバッファに加算
38     sb.AddObjectField(frame, model);
39
40     //フレネルホログラム用の物体光波を得るためホログラムの位置まで伝搬計算する
41     //frame.ExactAsmProp(50e-3);         //31行で帯域制限をオンにしたため，4倍拡張は不要
42     frame.AsmProp(50e-3);
43
44     frame.SaveAsWf("frame.wf");
45 }
```

この例ではマシンのプロセッサコアを最大限に用いるため，7 行目で wfl::SetNumThreads() 関数を用いて最大限の並列処理を設定している．10–13 行目では，モデルファイルを読み込んでその位置とサイズを設定している．IndexedFaceSet クラスは，モデルの構造を表す基本的なクラスである．11 行目で，IndexedFaceSet クラスの LoadWrl() メンバー関数を用いて，VRLM2.0 形式ファイルを読み込んでいる．IndexedFaceSet クラスは PointArray クラスを継承しているので，12-13 行目では，PointArray クラスの Localize() メンバー関数により物体モデルの外接矩形 (Bounding Box) の中心が原点と一致するように物体の位置を移動し，IndexedFaceSet クラスの SetWidth() メンバー関数によりその物体の外接矩形の横幅が 2mm になるように物体サイズを変えている．

16–35 行目は概ね前節のサンプルコードと同じであるが，ここでは，SurfaceBuilder クラスの SetBandLimitMethod() メンバー関数を用いてポリゴン光波を帯域制限している．これにより，42 行目の伝搬計算で，4 倍拡張が不要になるだけでなく，計算速度が向上する．ただし，帯域制限を行うためには，ホログラムの位置と大きさが必要であるため，必ず 30 行目のようにサンプリング間隔とサンプル数を，また 32 行目のように最終伝搬位置 (ホログラム面) での中心位置を SurfaceBuilder クラスに設定する必要がある．ポリゴン光波の帯域制限については 6.3.6 節を参照．

この例では，さらに，SurfaceBuilder クラスの SetDiffractionRate() メンバー関数や SetCullingRate() メンバー関数を用いて計算精度を既定値から変更している．これらの変更は計算精度のみならず計算速度にも影響を与える．なお，前節のサンプルコードと同様に，フレームバッファの位置は物体を切断するような物体近傍の位置に置くことが重要である．

Note

- ホログラムから離れた位置に物体を配置するフレネル型ホログラムの場合もフレームバッファの位置を物体位置から離してはならない．41 行目または 42 行目に示すように，ホログラムの位置での光波を得るためには物体光波の計算後に伝搬計算を行う．

38 行目では，SurfaceBuilder クラスの AddObjectField() メンバー関数を用いて物体光波の計算を行っている．モデルの複雑さやサイズ，フレームバッファの大きさによっては計算時間が長時間になる場合がある．そのため AddObjectField() メンバー関数には幾つかの形式が用意されており，マルチスレッド処理を指定することもできる．また，既定では一つのポリゴンを処理するたびにコンソールにメッセージが出力されるが，このメッセージは SurfaceBuilder クラスの SetCallback() メンバー関数を用いて変更することができる．例えば，SurfaceBuilder の設定に，次の 1 行を追加することにより，メッセージ出力を停止することができる．

```
sb.SetCallback(NULL);
```

これはリアルタイム計算が必要な場合には重要である．

最後に，41 行目で WaveField クラスの ExactAsmProp() メンバー関数を用いて物体光波を伝搬計算し，ホログラム面での光波を得ることが必要である．これはフレネル型ホログラムの光波を計算する場合であり，イメージ型の場合は必要ない．なお，この例のように帯域制限を行う場合には，ExactAsmProp() メンバー関数ではなく 42 行目のように AsmProp() メンバー関数を用いることができる．

5.2　テクスチャマッピング

5.2.1　正投影テクスチャマッピング

正投影テクスチャマッピングは z 軸と直交する平面上に配置したテクスチャ画像を物体表面に正投影することによって行うテクスチャマッピングである．正投影によるテクスチャマッピングの概念を図 5.1 に示す．

正投影テクスチャマッピングの基本的な処理のサンプルソースを次に示す．

Example　正投影テクスチャマッピング (ソース: OrthoTexureMapping.cpp)

```
1 #include    <psl.h>
```

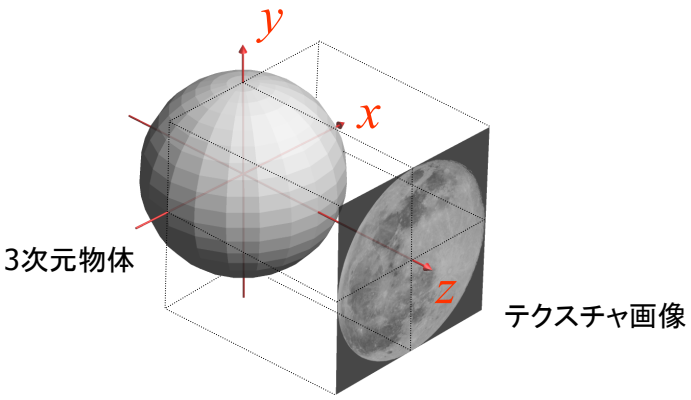


図 5.1　正投影によるテクスチャマッピング

```
2  using namespace wfl;
3  using namespace psl;
4
5  int main(void)
6  {
7      Start();     wfl::SetNumThreads();    //最大限にプロセッサコアを用いる
8
9      //モデルの設定
10     double objectSize = 2.5e-3;           //物体の実サイズを2.5mmとする
11     Point posObject(0.5e-3, -0.3e-3, 0);//物体の中心位置
12
13     //物体モデルファイルの読み込みと設定
14     IndexedFaceSet model;
15     model.LoadDxf("Sphere200.dxf");       //球体の読み込み
16     model.Localize();                     //物体を一時的に原点に置く
17     model.SetWidth(objectSize);           //物体サイズを設定
18     model += posObject;                   //物体位置を設定
19     model.AutoNormalVector();             //グーローシェーディングのための準備
20
21     // フレームバッファの設定
22     double px = 4e-6, py = 4e-6;          //サンプリング間隔
23     int nx = 1024, ny = 1024;             //サンプリング数
24     WaveField frame(nx, ny, px, py);      //フレームバッファの位置は(x,y,0)面
25     frame.Clear();
26
27     //シェーダーの設定
28     //WaveField diffuser(0);                 //特殊な拡散位相を用いる場合
29     //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相の読み込み
30     Vector light(-1.0, -1.0, -0.3);       //照明光の方向
31     double env = 0.1;                     //環境光の割合
32     double gamma = 0.01;                  //補正制限値
33     TfbGouraudShading gouraudShader(/*diffuser, */ gamma, light, env);//グーローシェーディング準備
34
35     //テクスチャ画像の読み込みと設定
36     Texture texImage;                                    //テクスチャ画像読み込み用
37     texImage.LoadBmp("MoonTexture.bmp", INTENSITY, 0, 2.2); //テクスチャ画像をガンマ値2.2で読み込む
38     texImage.SetWidth(objectSize);                      //物体の横サイズと画像横サイズを合わせる
39     texImage.SetHeight(objectSize);                     //物体の縦サイズと画像縦サイズを合わせる
40     texImage.SetCenter(posObject);                      //物体の位置と画像の位置を合わせる
41     TfbOlthoProjectMapping moonTexture(texImage);    //正投影テクスチャマッピングの準備
42
43     //SurfaceBuilderの設定
44     SurfaceBuilder sb(frame);
45     sb.SetDiffractionRatio(0.9);          //回折率設定
46     sb.SetCullingRate(0.6);               //カリング率設定
47     sb.SetShader(gouraudShader);          //上で用意したシェーダーを組み込む
48     sb.SetTexture(moonTexture);           //上で用意したテクスチャ画像を組み込む
49
50     //物体モデルmodelからの光波を計算してフレームバッファに加算
51     sb.AddObjectField(frame, model);      //物体光波計算
52     frame.SaveAsWf("object.wf");          //デバッグ用にセーブ
53
54     //フレネルホログラム用の物体光波を得るためホログラムの位置まで伝搬計算する
55     frame.ExactAsmProp(50e-3);
56
57     frame.SaveAsWf("frame.wf");           //物体光波の保存
58 }
```

5.1.2 節のサンプルソースを書き直したものとなっているので，変更点を中心に解説する．なおこのソースでは，テクスチャマッピングだけではなく，TfbGouraudShading クラスを用いたグーローシェーディングも行っている．

10–11 行では物体のサイズと位置を定義している．テクスチャマッピングでは物体のサイズと位置が正確に合う必要があるため，この定義は重要である．この例では，例示のためにわざと原点からずらした位置に物体を配置している．

14–19 行では物体を読み込んでいる．この部分は以前のサンプルソースとほぼおなじであるが，18 行目で物体の位置を設定していることと，グーローシェーディングの準備のために IndexedFaceSet クラスの AutoNormalVector() メンバー関数を用いて法線ベクトルを自動設定しているところが異なっている．

28–33 行のシェーダーの設定も大きな違いはないが，TfbGouraudShading クラスのコンストラクタを用いてシェーダーオブジェクトを作成している．

37–42 行ではテクスチャ画像の設定を行っている．37–38 行に示すとおり，テクスチャ画像は Texture クラスのオブジェクトにガンマ値 2.2 の INTENSITY モードで読み込む．テクスチャ画像は，補間してテクスチャマッピングされるため，そのピクセルピッチは重要ではない．しかし，図 5.1 に示すとおり，3 次元物体モデル表面に正投影されるため，その物理サイズと位置を物体モデルと一致させなければならない．そのため，39–40 行では，物理サイズが物体モデルと一致するように Texture クラスの SetWidth() メンバー関数や SetHeight() メンバー関数を用いてピクセルピッチを調整している．また，41 行目では画像のグローバル座標を物体モデルとあわせている．この時，画像は z 軸に沿って正投影されるため，z 位置はどのような値でも良い．

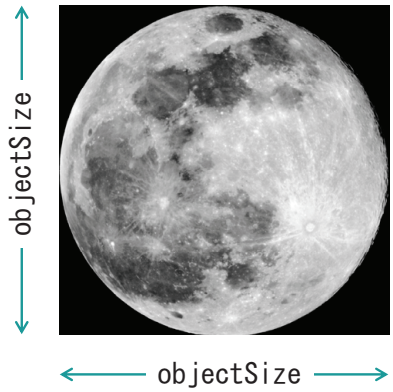


図 5.2　テクスチャ画像のサイズ

この例では，図 5.2 に示すとおり，テクスチャ画像は初めから縦横が物体モデルと一致するように準備してあるため，サイズの設定は簡単であるが，テクスチャ画像の一部のみをマッピングしたい場合等はピクセルピッチと画像位置の調整は複雑になる．

42 行目では，TfbOlthoProjectMapping クラスのオブジェクトを作成し，テクスチャマッピングの準備を行っている．ここで準備したマッピングオブジェクトを，49 行目で SurfaceBuilder クラスの SetTexture() メンバー関数を用いて SurfaceBuilder クラスのインスタンスに設定することにより，AddObjectField() メンバー関数実行時にテクスチャマッピングが実行される．

5.2.2　UV テクスチャマッピング

CG と同様の UV マッピングによるテクスチャマッピングを行うことができる．ただし，UV マッピング情報は，モデラーで作成したモデルファイルに含まれている必要がある．PSL では現在のところ，メタセコイア標準の MQO 形式ファイルの UV マップのみを読み込むことができる．なお，Rel 1.1 以降では二つ以上のテクスチャ画像を用いた

マッピングにも対応しており，スイッチバック法との併用も考慮されている．そのため，Rel 1.1 で UV テクスチャマッピングの実装が変更されており，Rel 1.0 までのテクスチャマッピング (by 山下裕士) とはソースの互換性がなくなっているので，注意が必要である．

UV マッピングによるテクスチャマッピングの概念を図 5.3 に示し，UV テクスチャマッピングの基本的な処理の Rel 1.1 以降のサンプルソースを次に示す．

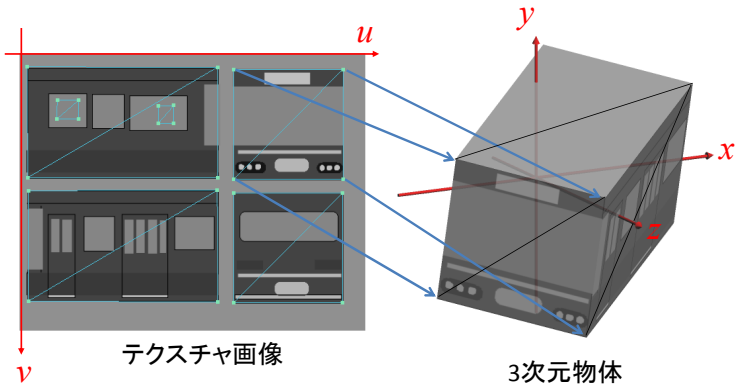


図 5.3　UV テクスチャマッピング

Example　UV テクスチャマッピングのサンプルソース (Rel 1.1 以降) (ソース: ExUvTextureMapping2.cpp)

```
#include <psl.h>
using namespace wfl;
using namespace psl;

void main(void)
{
    Start();    wfl::SetNumThreads();    //最大限にプロセッサコアを用いる

    //モデルの設定
    double objectSize = 6e-3;            //物体の実サイズを6mmとする
    Point posObject(0, 0, -50e-3);       //物体の中心位置

    //物体モデルファイルの読み込みと設定
    IndexedFaceSet model;
    model.LoadMqo("bus.mqo");            //物体データの読み込み．UVマッピングデータも読み込まれる
    model.Localize();                    //物体を一時的に原点に置く
    model.SetWidth(objectSize);          //物体サイズを設定
    model += posObject;                  //物体位置を設定
    model.AutoNormalVector();            //グーローシェーディングのための準備

    //フレームバッファの設定
    double px = 10e-6, py = 10e-6;       //サンプリング間隔
    int nx = 1024, ny = 1024;            //サンプリング数
    WaveField frame(nx, ny, px, py);
    frame.SetOrigin(posObject);          //フレームバッファの位置は物体モデルの中心
    frame.Clear();

    //シェーダーの設定
    //WaveField diffuser(0);             //特殊な拡散位相を使う場合
    //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相の読み込み
    Vector light(-1.0, -1.0, -0.3);      //照明光の方向
    double env = 0.1;                    //環境光の割合
    double gamma = 0.01;                 //補正制限値
    TfbGouraudShading gouraudShader(/*diffuser,*/ gamma, light, env);//グーローシェーディング

    //UVテクスチャマッピングの設定
    std::vector<char*> texNames;                 //テクスチャ画像ファイル名のリスト
    GetTextureNamesMqo("bus.mqo", texNames);     //ファイル名リストを読み込む
    TfbUvMapping busTexture(texNames, ".\\", 2.2, wfl::GRAY_SCALE);  //UVマッピングオブジェクト
                                                 //ファイルの場所としてカレントディレクトリを指定
    //SurfaceBuilderの設定
    SurfaceBuilder sb(frame);            //帯域制限のため，サンプル数とサンプル間隔等を正しく設定
    sb.SetBandLimitMethod(3);            //帯域制限をレベル3にする．
    sb.SetCenter(Point(0, 0, 0));        //帯域制限のため，ホログラムの中心位置(0,0,0)を設定
    sb.SetDiffractionRatio(0.9);         //回折率設定
    sb.SetCullingRate(0.6);              //カリング率設定
```

```
47      sb.SetShader(gouraudShader);          //上で用意したシェーダーオブジェクトを組み込む
48      sb.SetTexture(busTexture);            //上で用意したＵＶマッピングオブジェクトを組み込む
49
50      //物体モデルｍｏｄｅｌからの光波を計算してフレームバッファに加算
51      sb.AddObjectField(frame, model);      //物体光波計算
52      frame.SaveAsWf("object.wf");          //デバッグ用にセーブ
53
54      //フレネルホログラム用の物体光波を得るためホログラムの位置(z=0)まで伝搬計算する
55      //frame.ExactAsmProp(-posObject.GetZ());
56      frame.AsmProp(-posObject.GetZ());     //43行で帯域制限をオンにしたため，4倍拡張は不要
57
58      frame.SaveAsWf("frame.wf");           //物体光波の保存
59  }
```

5.2.1 節のサンプルソースを書き直したものとなっているので，変更点について解説する．14-15 行ではモデルファイルを読み込んでいる．UV マッピングを使用するためには IndexedFaceSet クラスの LoadMqo() メンバー関数を使用しなければならない．この関数によって IndexedFaceSet 型の model にモデル形状が読み込まれ，各頂点座標に図 5.4 に示したテクスチャ画像の UV 座標値が設定される．36-39 行ではマッピングするテクスチャ画像の設定を行っている．メタセコイア MQO 形式ファイルには，テクスチャ画像ファイルのファイル名が記録されているため，GetTextureNamesMqo() 関数を用いてその一覧を texNames に取得する．このテクスチャ画像のリストは，39 行目で TfbUvMapping クラスのインスタンス busTexture を生成する際の引数となる．なおここでは，TfbUvMapping クラスの第 4 の形式を用いており，モデルファイル bus.mqo に保存されたテクスチャ画像のディレクトリ情報を無視して，カレントディレクトリにテクスチャ画像を置いてこのプログラムを実行することを仮定している．また，テクスチャ画像をガンマ値 2.2 のグレイスケール (ColorMode 列挙型) で読み込みことを指定している．

このサンプルでは，43 行で SurfaceBuilder クラスの SetBandLimitMethod() メンバー関数を用いてポリゴン光波を帯域制限している．これにより，56 行の伝搬計算で，4 倍拡張が不要になるだけでなく，計算速度が向上する．ただし，帯域制限を行うためには，ホログラムの位置と大きさが必要であるため，必ず 42 行のようにサンプル間隔とサンプル数を SurfaceBuilder クラスに設定し，さらに 44 行のように最終伝搬位置 (ホログラム面) の中心位置を SurfaceBuilder クラスに設定する必要がある．ポリゴン光波の帯域制限については 6.3.6 節を参照．

48 行で SurfaceBuilder クラスのインスタンス sb に設定したテクスチャマッピングオブジェクト busTexture を登録している．これにより，51 行の AddObjectField() メンバー関数によって物体光波を生成する際にこの UV テクスチャマッピングが適用される．

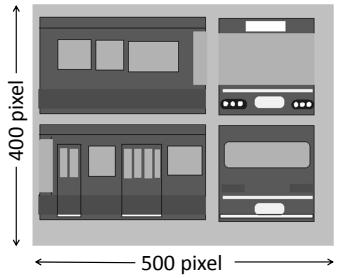


図 5.4　UV マッピングするテクスチャ画像

5.3 光波遮蔽 (隠面消去)

5.3.1 物体単位のシルエット法で背景の光波を遮蔽する

3D シーンに二つ以上の物体が存在する場合，手前の物体がファントムイメージとなるのを防ぐために遮蔽処理をする必要がある．以下は，そのために必要なシルエット法 [11] に遮蔽処理のサンプルコードである．

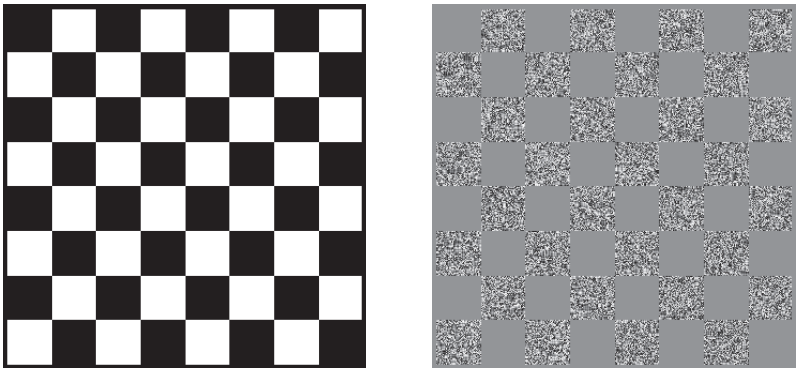
Example 壁紙の光を遮蔽処理する (ソース: ExSimpleSilhouette.cpp)

```
#include    "psl.h"
using namespace wfl;
using namespace psl;

int main(void)
{
    Start();    wfl::SetNumThreads();        //最大限にプロセッサコアを用いる

    //壁紙・物体位置
    Point posWallpaper(0, 0, -100e-3);       //壁紙はz=-100mmの位置
    Point posObject(0, 0, -50e-3);           //物体はz=-50mmの位置

    //フレームバッファの設定
    double px = 4e-6, py = 4e-6;             //サンプリング間隔
    int nx = 1024, ny = 1024;                //サンプリング数
    WaveField frame(nx, ny, px, py);         //フレームバッファの位置は(x,y,0)面
    frame.Clear();
    frame.SetOrigin(posWallpaper);           //フレームバッファの初期位置は壁紙の位置

    //壁紙(背景)の設定
    double width = 4e-3, height = 4e-3;      //壁紙のサイズ
    WaveField wallPaper;
    wallPaper.LoadBmp("Checker(256x256).bmp", AMPLITUDE);//壁紙画像を壁紙光波の振幅とする
    wallPaper.SetPx(width / wallPaper.GetNx()); //壁紙の幅と一致するようサンプリング間隔を設定
    wallPaper.SetPy(height / wallPaper.GetNy());//壁紙の高さと一致するようサンプリング間隔を設定
    wallPaper.SetOrigin(posWallpaper);       //壁紙の位置設定

    //壁紙の光波を発生してフレームバッファに加算
    WaveField wpFrame = frame;               //壁紙用のフレームバッファ
    wpFrame.Clear();
    wpFrame.ResamplingAdd(wallPaper, NEAREST_NEIGHBOR);//壁紙画像を補完して壁紙バッファに加算

    //--以下は特殊な拡散位相を使う場合
    //WaveField diffuser;                        //拡散位相
    //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相読み込み
    //MultiplyDiffuser(wpFrame, diffuser);     //壁紙画像に拡散位相の乗算

    wpFrame.ModRandomPhase();                //通常は拡散位相を使わずに位相乱数化
    frame += wpFrame;                        //フレームバッファに壁紙光波(壁紙バッファ)を加算
    wpFrame.Dispose();                       //壁紙バッファはもう必要ないので廃棄
    frame.SaveAsWf("frame0.wf");             //デバッグ用にセーブ
    frame.ExactAsmProp(posObject.GetZ() - posWallpaper.GetZ());
    //壁紙光波を物体平面まで伝搬. 位置はz=-50mmになる

    //物体モデルファイルの読み込みと設定
    IndexedFaceSet model;
    model.LoadWrl("cube.wrl");               //読み込み
    model.Localize();                        //物体を原点付近に置く
    model.SetWidth(2e-3);                    //外接矩形の横幅が2mmになるように物体サイズを設定
    model += posObject;                      //物体モデルの位置設定

    //シルエットマスク処理
    PaintObjectSilhouette(frame, model);     //シルエット部分を黒(0)に塗る
    frame.SaveAsWf("frame1.wf");             //デバッグ用にセーブ

    //シェーダーの設定
    Vector light(-1.0, -2.0, -1.0);          //照明光の方向
    double env = 0.2;                        //環境光の割合
    double gamma = 0.3;                      //補正制限値
    TfbFlatShading flatShader(/*diffuser, */ gamma, light, env); //乱数位相でフラットシェーディング準備

    // SurfaceBuilderの設定
    SurfaceBuilder sb(frame);
    sb.SetDiffractionRatio(1.0);             //回折率設定
    sb.SetCullingRate(0.6);                  //カリング率設定
    sb.SetShader(flatShader);                //上で用意したシェーダーを組み込む

    //物体モデルmodelからの光波を計算してフレームバッファに加算
```

```
69      sb.AddObjectField(frame, model);
70      frame.SaveAsWf("frame2.wf");              //デバッグ用にセーブ
71
72      //フレネルホログラム用の物体光波を得るためホログラムの位置まで伝搬計算する
73      frame.ExactAsmProp(0 - posObject.GetZ());
74
75      frame.SaveAsWf("frame.wf");               //ホログラム位置での物体光波
76  }
```

10 行目に示すように, この例では 2 次元平面物体の「壁紙」が $z = -100$mm の位置にあり, 3 次元物体が $z = -50$mm の位置に置かれた 3D シーンの物体光波合成を示している．3D シーンの奥から順番に計算を進めるため，18 行目で設定してるとおり，フレームバッファの初期値は壁紙と同じ $z = -100$mm である．

壁紙の光波を作成するため，まず 22 行目で Texture クラスのインスタンスを作成し，23 行目で壁紙の画像を読み込んでいる．この時，ガンマ値 2.2 の INTENSITY モードで読み込む．次に 24–25 行目で壁紙の物理的なサイズを設定し，さらに 26 行目でその位置を設定している．



(a) 振幅分布　　(b) 位相分布

図 5.5　壁紙光波 (frame0.wf) の振幅と位相

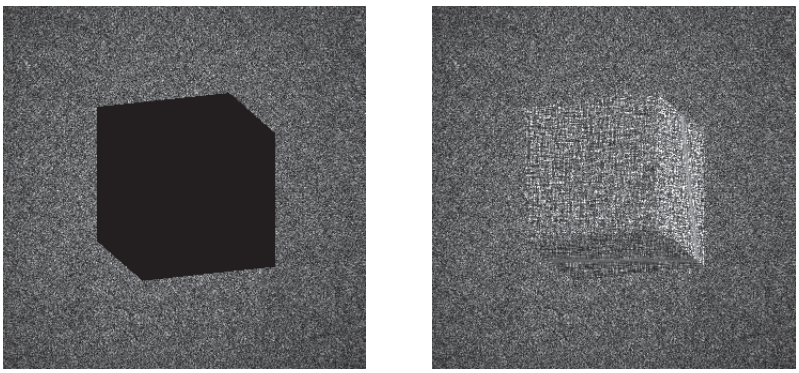
29–30 行目で壁紙光波用のフレームバッファとして wpFrame を準備している．このフレームバッファは，一般に，ホログラムと同じサイズで同じサンプリング間隔である．31 行目ではこの壁紙光波用フレームバッファに壁紙画像をコピーしている．ここで，壁紙画像のピクセルピッチは壁紙光波用フレームバッファのそれと一致しないため，ResamplingCopy() メンバー関数を用いて再サンプリングしてコピーしている．33–36 行目は特殊な拡散位相を用いる場合であり，この場合は MultiplyDiffuser() 関数を用いて壁紙光波用フレームバッファに拡散位相を乗算し，39 行目でフレームバッファに加算する．しかし，一般には特殊な拡散位相は必要ないので，38 行目に示すように，ModRandomPhase() メンバー関数を用いて位相を乱数化するだけでよい．この時点での壁紙光波を図 5.5 に示している．こうして作成したこの壁紙光波は，さらに 49 行目で物体の位置 ($z = -50$mm) まで伝搬計算される．

46–49 行目では，前節と同様，物体モデルの読み込みとそのサイズ設定を行っている．50 行目ではさらに，物体位置を $z = -50$mm に設定している．

シルエットマスク処理を行っているのが 53 行目である．PaintObjectSilhouette() 関数を用いてフレームバッファにシルエット形状を描いている．その結果を図 5.6 (a) に示す．

57–69 行目は前節のサンプルコードとほぼ同じで，シルエットマスク処理された壁紙光波に物体光波を重畳している．ただし，ここでは，64–66 行目で物体光波計算のパラメータを多少調整している．物体光波を重畳した合成光波を図 5.6 (b) に示す．

最後に，73 行目でホログラムの位置まで光波を伝搬計算している．



(a) fram1.lw (振幅分布)　　(b) frame2.lw (振幅分布)

図 5.6 マスキング処理された壁紙光波 (frame1.wf) と物体光波を重畳した合成光波 (frame2.wf)

5.3.2 スイッチバック法によるポリゴン単位のシルエット法

物体単位のシルエット法では，物体そのものに自己オクルージョンがあるとその部分が透ける部分的なファントムイメージになる．また，傾いた扁平な物体などではシルエットを入れる位置によってはシルエットマスクの黒い影が見える再生像となる．これらの現象はポリゴン単位でシルエット遮蔽を行うことで防ぐことができる．しかし，通常のシルエット法をそのままポリゴン単位に適用すると計算時間が長大になり実用的でない．

そこで，ポリゴン単位シルエット遮蔽処理を高速化するスイッチバック法を用いる．スイッチバック法を用いたソースを次に示す．

Example スイッチバック法のサンプルソース (ソース: ExSwitchBack1.cpp)

```
#include <psl.h>
using namespace wfl;
using namespace psl;

void main(void)
{
    Start();    wfl::SetNumThreads();   //最大限にプロセッサコアを用いる

    //モデルの設定
    double objectSize = 6e-3;           //物体の実サイズを6mmとする
    Point posObject(0, 0, -50e-3);      //物体の中心位置

    //物体モデルファイルの読み込みと設定
    IndexedFaceSet model;
    model.LoadMqo("bus.mqo");           //物体データの読み込み．UVマッピングデータも読み込まれる
    model.Localize();                   //物体を一時的に原点に置く
    model.SetWidth(objectSize);         //物体サイズを設定
    model += posObject;                 //物体位置を設定
    model.AutoNormalVector();           //グーローシェーディングのための準備

    //フレームバッファの設定
    double px = 10e-6, py = 10e-6;      //サンプリング間隔
    int nx = 1024, ny = 1024;           //サンプリング数
    WaveField frame(nx, ny, px, py);
    frame.SetOrigin(posObject);         //フレームバッファの位置は物体モデルの中心
    frame.Clear();

    //シェーダーの設定
    //WaveField diffuser(0);            //特殊な拡散位相を使う場合
    //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相の読み込み
    Vector light(-1.0, -1.0, -0.3);     //照明光の方向
    double env = 0.1;                   //環境光の割合
    double gamma = 0.01;                //補正制限値
    TfbGouraudShading gouraudShader(/*diffuser,*/ gamma, light, env);//グーローシェーディング

    //UVテクスチャマッピングの設定
    std::vector<char*> texNames;                //テクスチャ画像ファイル名のリスト
```

```
38      GetTextureNamesMqo("bus.mqo", texNames);    //ファイル名リストを読み込む
39      TfbUvMapping busTexture(texNames, ".\\", 2.2, wfl::GRAY_SCALE);  //UVマッピングオブジェクト
40                                                  //ファイルの場所としてカレントディレクトリを指定
41      //SurfaceBuilderの設定
42      SurfaceBuilder sb(frame);               //帯域制限のため，サンプル数とサンプル間隔等を正しく設定
43      sb.SetBandLimitMethod(3);               //帯域制限をレベル3にする．
44      sb.SetCenter(Point(0, 0, 0));           //帯域制限のため，ホログラムの中心位置(0,0,0)を設定
45      sb.SetDiffractionRatio(0.9);            //回折率設定
46      sb.SetCullingRate(0.6);                 //カリング率設定
47      sb.SetShader(gouraudShader);            //上で用意したシェーダーオブジェクトを組み込む
48      sb.SetTexture(busTexture);              //上で用意したUVマッピングオブジェクトを組み込む
49
50      //物体モデルmodelからの光波を計算してフレームバッファに加算
51      sb.AddObjectField(frame, model);        //物体光波計算
52      frame.SaveAsWf("object.wf");            //デバッグ用にセーブ
53
54      //フレネルホログラム用の物体光波を得るためホログラムの位置(z=0)まで伝搬計算する
55      //frame.ExactAsmProp(-posObject.GetZ());
56      frame.AsmProp(-posObject.GetZ());   //43行で帯域制限をオンにしたため，4倍拡張は不要
57
58      frame.SaveAsWf("frame.wf");             //物体光波の保存
59  }
```

この例では，http://innoce.nobody.jp/からダウンロードさせて頂いた初音ミクの 3D データを用いている．psl では現在のところ BMP 形式の画像しか読み込めないため，事前準備として，テクスチャ画像全てを 8 ビット BMP に変換し，カレントディレクトリ直下の miku1 ディレクトリに MQO ファイルと画像ファイルを配置している．

この例は 5.2.2 節のサンプルソースを書き直したものとなっているので，変更点について解説する．この例での実質的な変更点は 51 行目の物体光波生成関数を前述の AddObjectField() メンバー関数から AddObjectFieldSb() メンバー関数に変えただけである．その他の注意点として，39 行で TfbUvMapping クラスのインスタンス texture を生成する際のディレクトリ指定引数においては，パス文字列の最後を “\\” としてディレクトリであることを明示する必要がある．またここでは，テクスチャ画像のガンマ値として 2.2 を指定し，グレイスケールモードで (ColorMode 列挙型) で画像を読み込みことを指定している．

このサンプルでは，43 行で SurfaceBuilder クラスの SetBandLimitMethod() メンバー関数を用いてポリゴン光波を帯域制限している．これにより，56 行目の伝搬計算で，4 倍拡張が不要になるだけでなく，計算速度が向上する．ただし，帯域制限を行うためには，ホログラムの位置と大きさが必要であるため，必ず 42 行目のようにサンプル間隔とサンプル数を SurfaceBuilder クラスに設定し，さらに 44 行のように最終伝搬位置 (ホログラム面) の中心位置を SurfaceBuilder クラスに設定する必要がある．ポリゴン光波の帯域制限については 6.3.6 節を参照．

セーブされた物体光波 Object.wf を振幅像で表示した結果を図 5.7 に示す．

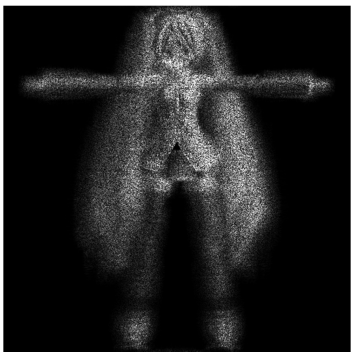


図 5.7　物体光波 Object.wf の振幅像

5.4 ランバートシェーディングによる物体光波のフルカラーレンダリング

5.4.1 ランバートシェーダ

TfbLambertShading クラスを用いて，物体光波をフルカラーでレンダリングするためのサンプルソースを次に示す．

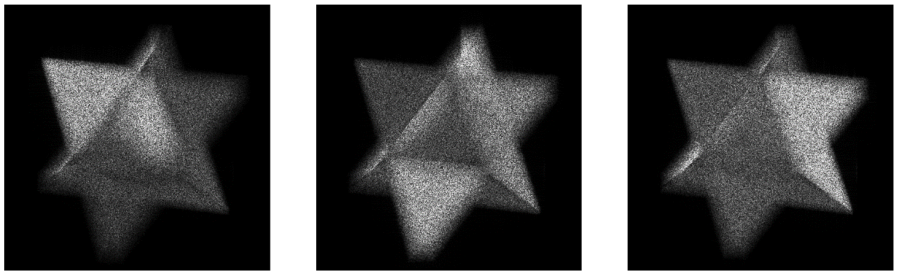
Example ランバートシェーダによるフルカラーレンダリング (ソース: ExFullColorRendering-Lambert.cpp)

```
#include <psl.h>
using namespace wfl;
using namespace psl;

void main(void)
{
    Start();    wfl::SetNumThreads();                    //最大限にプロセッサコアを用いる

    //モデルの設定
    double objectSize = 3e-3;                            //物体の実サイズ
    Point posObject(0, 0, -20e-3);                       //物体の中心位置

    //物体モデルファイルの読み込みと設定
    IndexedFaceSet model;
    model.LoadMqo("testmodel\\star8.mqo");
    model.Localize();                                    //物体を一時的に原点に置く
    model.SetWidth(objectSize);                          //物体サイズ(幅)を設定
    model += posObject;                                  //物体位置を設定
    model.AutoNormalVector();                            //グーローシェーディングのための準備

    //フレームバッファの設定
    double lambda[] = { 630e-9, 540e-9, 460e-9 };        //フルカラー用に用いる波長(R, G, B)
    double px = 1e-6, py = 1e-6;                         //サンプリング間隔
    int nx = 4096, ny = 4096;                            //サンプリング数
    WaveField frame(nx, ny, px, py);

    //シェーダーの設定
    Vector light(-0.408, -0.408, -0.816);                //照明光の方向（この方向はMqoのデフォルト）
    double gamma = 0.01;                                 //補正制限値

    //物体光波のレンダリング
    for (int RGBcounter = 1; RGBcounter <= 3; RGBcounter++){

        Printf("\n\n%d色目の計算開始\n", RGBcounter);

        frame.Clear();
        frame.SetWavelength(lambda[RGBcounter - 1]);     //各色の波長設定
        frame.SetOrigin(posObject);                      //フレームバッファの位置は物体モデルの中心

        TfbLambertShading shading(gamma, light, (ColorMode)RGBcounter);//ランバートシェーダ

        //SurfaceBuilderの設定
        SurfaceBuilder sb(frame);                        //帯域制限のため，サンプル数とサンプル間隔等を正しく設定
        sb.SetBandLimitMethod(3);                        //帯域制限をレベル3にする．
        sb.SetCenter(Point(0, 0, 0));                    //帯域制限のため，ホログラムの中心位置(0,0,0)を設定
        sb.SetDiffractionRatio(0.9);                     //回折率設定
        sb.SetCullingRate(0.6);                          //カリング率設定
        sb.SetShader(shading);                           //上で用意したシェーダーオブジェクトを組み込む

        //物体モデルmodelからの光波を計算してフレームバッファに加算
        sb.AddObjectFieldSb(frame, model, 1);            //スイッチバック法で物体光波計算

        //デバッグ用にセーブ
        if (RGBcounter == RED)      frame.SaveAsWf("object-R.wf");
        if (RGBcounter == GREEN)    frame.SaveAsWf("object-G.wf");
        if (RGBcounter == BLUE)     frame.SaveAsWf("object-B.wf");

        //フレネルホログラム用の物体光波を得るためホログラムの位置(z = 0)まで伝搬計算する
        frame.ExactAsmProp(-frame.GetOrigin().GetZ());

        //物体光波の保存
        if (RGBcounter == RED)      frame.SaveAsWf("frame-R.wf");
        if (RGBcounter == GREEN)    frame.SaveAsWf("frame-G.wf");
        if (RGBcounter == BLUE)     frame.SaveAsWf("frame-B.wf");
    }
}
```

この例は 5.3.2 節のサンプルソースを書き直したものとなっているので，主に変更点について解説する．この

例の 32 行目以降は，赤，緑，青に対応した波長で物体光波をそれぞれ計算する処理となっている．40 行目では TfbLambertShading クラスのコンストラクタを用いてシェーダオブジェクトを作成している．このクラスでは TfbFlatShading クラスや TfbGouraudShading クラスとは異なる引数 (ColorMode 列挙型の mode) を持つ．mode は，ある物体モデルについて物体光波を計算する際，その物体モデルのどのカラーチャンネルを用いて輝度の計算を行うかを指定している．一方，計算する光波の波長は 43 行の SurfaceBuilder クラスのコンストラクタによって決まるため，例えば赤のチャンネルで波長 532 nm(緑色) の物体光波を計算することも可能である．しかし，一般には，指定したカラーチャンネルに対応する波長で光波を計算することで，3D モデルの有する色を正しく再現することができる．

テクスチャを読み込む場合は，5.3.2 節のサンプルソース同様 TfbUvMapping クラスの第 3 の形式を用いて，各波長に応じて読み込むテクスチャ画像のカラーチャンネルをやはり ColorMode 列挙型で指定すればよい．セーブされた物体光波 object-R.wf，object-G.wf，object-B.wf を振幅像で表示した結果を図 5.8 に示す．



(a) object-R.wf　(b) object-G.wf　(c) object-B.wf

図 5.8　フルカラー物体光波の振幅像

5.5　セグメント分割を用いた大規模計算

メモリにすべてをロードできないような大規模な光波の計算では，**セグメント分割**を行う必要がある [10]．セグメント分割光波を効率良く扱うためのクラスが SegWaveField クラスである．本節では SegWaveField クラスを用いた処理の実例を示すが，本節を読む前に，まずセグメント分割光波の概念や構造，その用語を理解するために 6.11.1 節のセグメント分割光波 (Segemented wave-field) の構造と概念を参照していただきたい．

5.5.1　セグメント分割光波プログラミングの基本

SegWaveField クラスを用いてセグメント分割光波 [10] を扱うための基本的な処理のサンプルソースを次に示す．

Example　セグメント分割光波プログラミングの基本 (ソース: ExSegmentation.cpp)

```
1  #include    "psl.h"
2  using namespace wfl;
3  using namespace psl;
4
5  int main()
6  {
7      Start(3);
8      SegWaveField sw(4, 3, 256, 256);
9      sw.SetWavelength(532e-9);            //波長の設定
10     sw.SetPx(4e-6);                      //サンプリング間隔の設定
11     sw.SetPy(4e-6);
12     sw.SyncParam()  ;                    //セグメント全体でパラメータを同期
13     sw.SetCenter(Point(1e-3, 0, 0));     //光波全体のセンターを設定
14
15     int i, j;
16     for (j = 0; j < sw.GetMy(); j++)     //カレントセグメントを一巡させるループ
17     {
18         for (i = 0; i < sw.GetMx(); i++)
```

```
19          {
20              //カレントセグメントを(i,j)に設定(セグメントファイルからロード)
21              sw.Segment(i, j);
22
23              //カレントセグメント処理(WaveFieldクラスの関数の使用)
24              sw.AddSphericalWave(Point(0, 0, -50e-3));
25
26             //カレントセグメントをセグメントファイルにセーブ
27              sw.SaveSeg();
28          }
29      }
30      sw.SaveAsSegBmp("test.bmp", PHASE);
31      sw.SaveAsCombinedBmp("test-c.bmp", PHASE);
32  }
```

ここで，第 8 行は SegWaveField クラスのコンストラクタであり，セグメント数が $4 \times 3 = 12$ セグメントで，一つのセグメントのサンプリング数 (セグメントサンプリング数) が 256×256 点のセグメント分割光波のインスタンスを生成している[*2]．このとき，必要なメモリは一つのセグメント分の複素振幅分布に必要なメモリのみである．なお，この例題のコンストラクタでは，セグメントファイルのファイル名を指定していない．この場合，一時的なファイル名が自動的に生成され，そのファイル名のセグメントファイルが作成される (一時ファイル名についてはコンストラクタの *Note* を参照)．

SegWaveField クラスでは非常に多くのパラメータが必要であるため，コンストラクタのみではそれをすべて指定できない．そのため，WaveField クラスをテンプレートとして用いるコンストラクタやデフォルトパラメータを用いる方法 (SetDefault() メンバー関数) が用意されている．ここでは，それらを用いずに WaveField クラスから継承した WaveField クラスの SetPx() メンバー関数や SetPy() メンバー関数，SetWavelength() メンバー関数を用いてパラメータを設定している．12 行目では，SyncParam() メンバー関数を用いて設定をセグメントファイルと同期している．

Note

- カレントセグメントに対してサンプリング間隔と波長の設定・変更を行った場合には，SyncParam() メンバー関数を用いてセグメントファイルとカレントセグメントの設定を同期しなければならない．
- セグメント数やセグメントサンプリング数を設定・変更した場合は，Init() メンバー関数を用いてそれを有効化しなければならない．

13 行目では，SetCenter() メンバー関数を用いて分割光波全体の中心位置を設定している．

16 行目からのループはカレントセグメントを分割光波全体で一巡させるためのループである．セグメントのインデックスは $0 \leq i < M_x$，$0 \leq j < M_y$ であり，セグメント数 M_x と M_y は GetMx() メンバー関数と GetMy() メンバー関数によって取得できる．

この例では一つの点光源の球面波を分割光波全体に設定している．そのため，ループ内での処理ではまず，Segment() メンバー関数を用いてカレントセグメントを (i, j) に設定している．Segment() メンバー関数は，セグメントファイルからカレントセグメントをメモリ上にロードする関数である．ただし，セグメントファイル内にまだそのセグメントの光波が生成されていない場合は，メモリ上に光波の領域を作成してそれをクリアする．この例では，Segment() メンバー関数はこの動作を行っている．次に，24 行目ではカレントセグメントに対して WaveField クラスの AddSphericalWave() メンバー関数を呼び出して，球面波 (の一部) を設定している．

一つのセグメントの処理が終わったら，SaveSeg() メンバー関数を呼び出し，カレントセグメントをセグメントファイルにセーブする．すなわち，カレントセグメントの処理の初めでは Segment() メンバー関数を用い，終わりでは SaveSeg() メンバー関数を呼び出すのが，分割光波プログラミングの基本となる．

Note

[*2] ここでは，例題として小さな光波を用いているが，実際にはこのような小規模な光波をセグメント分割する必要はない．原則として，セグメント分割をする場合はしない場合に比べて処理が遅くなるため，不要なセグメント分割はするべきではない．

- 分割光波を参照するだけの場合，すなわち分割光波自体を変化する必要が無い場合は，SaveSeg() メンバー関数を呼び出す必要はない．Segment() メンバー関数と SaveSeg() メンバー関数は必ずセットで用いなければならないわけではない．
- カレントセグメントをセグメントファイルからロードあるいはセーブする処理は I/O 処理であるため，非常に時間を要することが多い．従って，可能なかぎり SaveSeg() メンバー関数や Segment() メンバー関数を呼び出す回数を減らすことが望ましい．
- Rel1.3 以降では，引数 (インデックス) が一つだけの第 2 形式 Segment() メンバー関数が用意されている．これを用いると，ループを 2 重にする必要がない単純な処理の場合，$0 \leq m < M_x M_y$ の範囲の 1 重ループで全セグメントを処理できる．なお，$M_x M_y$ は GetM() メンバー関数で取得できる．

最後に，2 種類の方法で計算結果を BMP ファイルとして保存している．30 行目では SaveAsSegBmp() メンバー関数を用いて個々のセグメント毎の BMP ファイルとして保存している．この場合，test[ii][jj].bmp というファイル名の BMP ファイルがセグメント数と同じ数だけ作成される．こちらが標準的な方法である．

31 行目の SaveAsCombinedBmp() メンバー関数では，すべてのセグメントを統合して一つの BMP ファイルを作成する．一般的にこの関数で作成される BMP ファイルは巨大になるため，通常の画像アプリケーションでは表示できない場合が多いため注意が必要である．

5.5.2　セグメント分割を用いた物体光波の計算

SegWaveField クラスを用いてセグメント分割光波でポリゴン法による物体光波を計算するためのサンプルソースを次に示す．このソースは，5.1.2 節のサンプルソースを SegWaveField クラスを用いて書き直したものである．

Example　セグメント分割を用いた物体光波の計算例 (ソース: ExSegObjectField.cpp)

```
1  #include    "psl.h"
2  using namespace wfl;
3  using namespace psl;
4
5  int main(void)
6  {
7      Start(3);
8
9      // 物体モデルファイルの読み込みと設定
10     IndexedFaceSet model;
11     model.LoadWrl("cube.wrl");          //読み込み
12     model.Localize();                   //物体を原点付近に置く
13     model.SetWidth(6e-3);               //外接矩形の横幅が6mmになるように物体サイズを設定
14
15     // フレームバッファの設定
16     double px = 4e-6, py = 4e-6;        //サンプリング間隔
17     int nx = 1024, ny = 1024;           //サンプリング数
18     WaveField temp(nx, ny, px, py, 532e-9);//フレームバッファのテンプレート
19     SegWaveField frame(3, 2, temp);     //テンプレートから3x2の分割光波を作成
20
21     //シェーダーの設定
22     //WaveField diffuser(0);            //特殊な拡散位相を持ちる場合
23     //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相の読み込み
24     Vector light(-1.0, -3.0, -1.0);     //照明光の方向
25     double env = 0.1;                   //環境光の割合
26     double gamma = 0.3;                 //補正制限値
27     TfbFlatShading flatShader(/*diffuser, */gamma, light, env); //フラットシェーディング準備
28
29     // SurfaceBuilderの設定
30     SurfaceBuilder sb(frame);
31     sb.SetDiffractionRatio(0.9);        //回折率設定
32     sb.SetCullingRate(0.6);             //カリング率設定
33     sb.SetShader(flatShader);           //上で用意したシェーダーを組み込む
34
35     //物体モデルmodelからの光波を計算してフレームバッファに加算
36     for (int j = 0; j < frame.GetMy(); j++)
37     {
38         for (int i = 0; i < frame.GetMx(); i++)
39         {
40             frame.Segment(i, j);
41             sb.AddObjectField(frame, model);//各セグメントに物体光波を加算
```

```
42              frame.SaveSeg();
43          }
44      }
45
46      //フレネルホログラム用の物体光波を得るためホログラムの位置まで伝搬計算する
47      SegWaveField hologram(3, 2, temp);  //ここではホログラムのセグメント数は物体光波と同じ
48      hologram.SetCenter(Point(0, 0, 50e-3)); //ホログラム(伝搬先)の位置は50mm進んだ位置
49      for (int j = 0; j < hologram.GetMy(); j++)  //伝搬先の全セグメントでループする
50      {
51          for (int i = 0; i < hologram.GetMx(); i++)
52          {
53              hologram.Segment(i, j);
54              hologram.ShiftedAsmProp(frame); //伝搬元の全セグメントからの光波を求める
55              hologram.SaveSeg();
56          }
57      }
58
59      hologram.SaveAsSegWf("hologram.wf");          //保存のためセーブ
60      hologram.SaveAsCombinedWf("hologram.wf");     //確認のためセーブ(デバッグ用)
61  }
```

以前のサンプルソースから書き換わったのは，まず 18，19 行目である．ここでは，WaveField クラスのテンプレートを作成し，それを用いて SegWaveField クラスのコンストラクタを呼び出している．

次に 36 行目から 44 行目では，以前のサンプルソースにおいて SurfaceBuilder クラスの AddObjectField() メンバー関数を一度だけ呼び出していたのを，全セグメントに対して AddObjectField() メンバー関数を呼び出すように書き換えている．

次に，求めた物体光波をホログラムの位置まで伝搬計算することによりフレネルホログラムを計算するが，SegWaveField クラスによる分割光波処理ではそれ自体を伝搬計算によって進めることはできず，伝搬先と伝搬元それぞれについて SegWaveField クラスの分割光波が必要である．そのため，47 行目では，18 行目で作成したテンプレートを再び用いてホログラム面での分割光波 hologram を作成している．48 行目では，この hologram の中心位置を物体面から z 方向に 50mm 進めた位置に設定している．

続いて 49〜57 行目において，伝搬元の分割光波 frame から伝搬先の分割光波 hologram を計算している．SegWaveField クラスの ShiftedAsmProp() メンバー関数は，伝搬元の全セグメントから伝搬先のカレントセグメントへの伝搬計算をシフト角スペクトル法によって行うための関数である．分割光波における伝搬計算の概念を図 6.3 に示す．ShiftedAsmProp() メンバー関数は内部的に WaveField クラスの ShiftedAsmPropAddEx() メンバー関数を呼び出しているため，伝搬元と伝搬先のセグメントサンプリング数が異なっていても良い．シフトフレネル法が必要な場合 (サンプリング間隔を変えたい場合など) は SegWaveField クラスの ShiftedFresnelProp() メンバー関数を用いることもできる．この例では，伝搬先の全セグメントについて光波を求める必要があるため，カレントセグメントが全セグメントを一巡するようにループを回している．

Note

- Rel1.3 以降では，ShiftedAsmPropWhole() メンバー関数や ShiftedAsmFresnelWhole() メンバー関数を用いることができる．これらでは内部的にループを回しているため，上記の 49 行目から 57 行目のループは省略でき，ソースが簡単になる．
- Rel1.3 以降ではさらに，ShiftedAsmProp() メンバー関数や ShiftedAsmFresnel() メンバー関数の第 2 の形式があり，中心位置を変えずに z 方向に伝搬計算をする場合には，これらを用いると移動距離を引数に与えるだけで，対象オブジェクトを z 方向に伝搬させることができる．

なお, 伝搬元の 1 セグメントから伝搬先の 1 セグメントの光波を求めるためには, それぞれのセグメントをカレントセグメントに設定してから，通常の WaveField クラスの ShiftedAsmProp() メンバー関数や ShiftedFresnelProp() メンバー関数を用いて計算することができる．

最後に，59 行目では SegWaveField クラスの SaveAsSegWf() メンバー関数を用いて，WF 形式で全光波をファイルに保存している．なお，保存した後でメモリ上の光波が不要である場合には SegWaveField クラスの

Dispose() メンバー関数を用いた方が処理速度が速くなる．この例では，さらに 60 行目で SegWaveField クラスの SaveAsCombinedWf() メンバー関数を用いて全光波を統合して WF 形式でも保存している．ただし，この関数の現在の実装では，一時的に全光波を収容するメモリサイズが必要であるため，実際の大規模計算では実用的ではない (全光波を収容できるメモリサイズがあるなら，そもそも分割光波自体が不要である)．

5.5.3　シルエット法による光波遮蔽付き物体光波の計算

SegWaveField クラスを用いてセグメント分割光波でシルエット法による光波遮蔽を行なった上，ポリゴン法による物体光波を計算するためのサンプルソースを次に示す．このソースは，5.3.1 節のサンプルソースを SegWaveField クラスを用いて書き直したものである．

Example　セグメント分割を用い壁紙の光を遮蔽処理する例 (ソース: ExSegSilObjectField.cpp)

```
#include    "psl.h"
using namespace wfl;
using namespace psl;

int main(void)
{
    Start();

    //壁紙・物体位置
    Point posWallpaper(0, 0, -100e-3);       //壁紙はz=-100ｍｍの位置
    Point posObject(0, 0, -50e-3);           //物体はz=-50ｍｍの位置

    //デフォルトパラメータの設定
    double px = 4e-6, py = 4e-6;             //サンプリング間隔
    double lambda = 532e-9;                  //波長
    int nx = 1024, ny = 1024;                //サンプリング数
    WaveField::SetDefault(nx, ny, px, py, lambda);

    //壁紙(背景)の設定
    double width = 8e-3, height = 8e-3;      //壁紙のサイズ
    Texture wallPaper;
    wallPaper.LoadBmp("Checker(256x256).bmp", AMPLITUDE);//壁紙画像を壁紙光波の振幅とする
    wallPaper.SetWidth(width);               //壁紙の幅を設定
    wallPaper.SetHeight(height);             //壁紙の高さを設定
    wallPaper.SetOrigin(posWallpaper);       //壁紙の位置を設定

    //壁紙の光波を発生
    SegWaveField wpFrame(2, 2);              //2x2セグメントの壁紙用フレームバッファ
    wpFrame.SetCenter(posWallpaper);         //壁紙光波の位置を設定
    //WaveField diffuser(0);                   //拡散位相を使う場合
    //diffuser.LoadWf("diffuser1024x1024.wf");//拡散位相を使う場合：拡散位相読み込み
    for (int j = 0; j < wpFrame.GetMy(); j++)
    {
        for (int i = 0; i < wpFrame.GetMx(); i++)
        {
            wpFrame.Segment(i, j);
            wpFrame.ResamplingAdd(wallPaper, NEAREST_NEIGHBOR);//壁紙画像を補完して加算
            //MultiplyDiffuser(wpFrame, diffuser);//拡散位相を使う場合：壁紙画像に拡散位相の乗算
            wpFrame.ModRandomPhase();        //通常は拡散位相を使わず乱数化
            wpFrame.SaveSeg();
        }
    }

    //物体モデルファイルの読み込みと設定
    IndexedFaceSet model;
    model.LoadWrl("cube.wrl");               //読み込み
    model.Localize();                        //物体を原点付近に置く
    model.SetWidth(6e-3);                    //外接矩形の横幅を6ｍｍに設定
    model += posObject;                      //物体モデルの位置設定

    //シェーダーの設定
    Vector light(-1.0, -2.0, -1.0);          //照明光の方向
    double env = 0.2;                        //環境光の割合
    double gamma = 0.3;                      //補正制限値
    TfbFlatShading flatShader(/*diffuser, */gamma, light, env); //フラットシェーディング準備

    // SurfaceBuilderの設定
    SurfaceBuilder sb;                       //デフォルトパラメータで生成
    sb.SetDiffractionRatio(1.0);             //回折率設定
    sb.SetCullingRate(0.6);                  //カリング率設定
    sb.SetShader(flatShader);                //上で用意したシェーダーを組み込む

```

```
63      //物体平面のフレームバッファ作成と，伝搬計算，シルエットマスク処理，物体光波計算
64      SegWaveField frame(3, 2);                //3x2セグメントの物体平面フレームバッファ
65      frame.SetCenter(posObject);              //物体平面の位置設定
66      for (int j = 0; j < frame.GetMy(); j++)
67      {
68          for (int i = 0; i < frame.GetMx(); i++)
69          {
70              frame.Segment(i, j);
71              frame.ShiftedAsmProp(wpFrame);       //壁紙光波全体をカレントセグメントに伝搬
72              PaintObjectSilhouette(frame, model);//シルエットマスク処理
73              sb.AddObjectField(frame, model, 4); //物体モデルの光波計算(4スレッド使用)
74              frame.SaveSeg();
75          }
76      }
77      wpFrame.Dispose("wallpaper.wf");         //壁紙フレームバッファを保存してメモリ廃棄
78
79      //ホログラム用のフレームバッファ作成と伝搬計算
80      SegWaveField hologram(3, 2);             //ホログラム用フレームバッファ
81      hologram.SetCenter(Point(0, 0, 0));      //ホログラム面は位置設定
82      for (int j = 0; j < hologram.GetMy(); j++)
83      {
84          for (int i = 0; i < hologram.GetMx(); i++)
85          {
86              hologram.Segment(i, j);
87              hologram.ShiftedAsmProp(frame); //伝搬元の全セグメントからの光波を求める
88              hologram.SaveSeg();
89          }
90      }
91      frame.Dispose("object.wf");              //物体面フレームバッファを保存してメモリ廃棄
92      hologram.SaveAsSegWf("hologram.wf");     //ホログラム面での物体光波を保存
93      hologram.SaveAsCombinedLw("debug.lw");   //縮小モデルによるデバッグ時のみ使用
94  }
```

この例では，前節のサンプルと異なり，テンプレートを用いずに 14–17 行目でデフォルトパラメータを設定して用いている．またこの例では，壁紙が $z = -100$ mm，物体平面が $z = -500$ mm にあり，ホログラムは原点にあると考えている．

以下，以前のサンプルソースから書き換わった点を中心に解説する．19–25 行目の壁紙の設定はほとんど以前のサンプルソースから変更されていない．

28–42 行目の壁紙光波の発生では，まず 28 行目で壁紙光波を 2×2 セグメントとしている．次に，32–42 行目のループで全セグメントを処理しており，この時ループ内では壁紙画像を光波の振幅としてコピーし，さらに拡散位相を乗算している．

次に非常に重要な処理が 63–77 行目の処理である．この部分は物体平面での光波を求める部分である．物体平面の分割フレームバッファは 3×2 のセグメント数を有しており，ここでは，全セグメントを一巡する一つの 2 重ループで三つの処理を行っている．まず，71 行目で物体平面のカレントセグメントを伝搬先として伝搬元の壁紙光波の全セグメントからの伝搬計算を行う．次に，物体モデルのシルエットをカレントセグメントに書き込むシルエットマスク処理を 72 行目で行っている．さらに 73 行目では物体光波を計算して加算している．この様に，一つのループ内に 3 種類の処理を集中することにより，カレントセグメントをセグメントファイルからロードあるいはセーブする際の I/O 処理を最小限に抑えることができる．

77 行目では，この時点で不要になった壁紙光波のフレームバッファを廃棄すると共に，念のため，これをファイルに保存している．ただし，ファイルの保存は必ずしも必要ではない．

79–90 行目の処理は，ほとんど前節のサンプルと同じであり，物体平面からホログラム面への伝搬計算を行っている．ここでは，ホログラムは物体平面と同じ 3×2 のセグメント数を有している．

5.6　フィールドデータをカラー画像に変換する

フィールドの波長が可視波長域である場合，カラー画像としてそれを観察したい場合がある．このようなカラー化を取り扱うクラスとして ColorImage クラスがある．カラー画像を得たい場合，一般に複数の波長で計算した結果を合成

する必要があるが，ColorImage クラスは等色関数を用いてこのような合成を行う．なお，ColorImage クラスでは，原則としてピクセルカラーを CIE1931 XYZ 表色系と呼ばれる色で扱っているので，必要に応じて他の表色系に変換したり，sRGB に変換してファイルの保存を行うことができる．

以下は，3 種類の波長で計算した WaveField 型のフィールドからカラー画像を生成する例である．用いる波長数 (フィールド数) には制限はなく，単に AddSpectralImage() メンバー関数または '+=' 演算子を用いて，既存の ColorImage 型のオブジェクトに追加するだけである．

Example　ソース：ExColorImage1

```
1  #include "psl.h"
2
3  using namespace psl;
4  using namespace wfl;
5
6  int main(void)
7  {
8      Start();
9      WaveField::SetDefault(512, 512, 10e-6); //フィールドのデフォルト値を設定
10     WaveField wf1, wf2, wf3;
11
12     // 各フィールドに波長を設定する
13     wf1.SetWavelength(633e-9);
14     wf2.SetWavelength(532e-9);
15     wf3.SetWavelength(488e-9);
16
17     ////////////////////////////////////////////////////////
18     // ここで，wf1でなんらかのシミュレーション等を行なう．
19     ////////////////////////////////////////////////////////
20
21     ColorImage colImage(wf1);        //wf1の強度像をベース画像としてカラー画像を生成
22
23     ////////////////////////////////////////////////////////
24     // ここで，wf2でなんらかのシミュレーション等を行なう．
25     ////////////////////////////////////////////////////////
26
27     colImage.AddSpectralImage(wf2); //wf2の強度像を既存の画像に追加する
28
29     ////////////////////////////////////////////////////////
30     // ここで，wf3でなんらかのシミュレーション等を行なう．
31     ////////////////////////////////////////////////////////
32
33     colImage.AddSpectralImage(wf3); //wf3の強度像を既存の画像に追加する
34
35     colImage.NormalizeXYZ();         //ハレーションしないように正規化
36     colImage.SaveAsBmpSRGB("Image.bmp");     //sRGB画像として保存
37 }
```

ここで，第 9 行目では WaveField クラスの SetDefault() メンバー関数を用いてサンプリング数とサンプリング間隔にデフォルト値を設定している．色合成を行う際，各波長のフィールドのサンプリング数は同じである必要があるため，この例ではこの様にしている．13-15 行目では各フィールドにそれぞれの波長を設定している．以後，設定された波長でシミュレーション等が行われるとする．

21 行目では，シミュレーションの最初の結果である wf1 から，コンストラクタを用いて，ColorImage クラスのインスタンスを生成している．この時，縦横のピクセル数として wf1 のサンプリング数が設定されるため，以後それ以外のサイズのフィールドを追加することはできない．なお，サンプリング間隔はフィールド毎に異なっていてもエラーにはならないが，原則として異なったサンプリング間隔を用いると正しい結果にならない．なお，カラー画像として合成されるのは，常にフィールドの強度像である．

次に 27 行目では，波長 532 nm の wf2 で計算を行い，それを AddSpectralImage() メンバー関数を用いて既存の colImage インスタンスに追加している．同様に，33 行目では wf2 の結果を追加している．

この様に，次々と波長ごとの強度像を追加していくが，当然，多くのフィールドを追加すると輝度が増加し，カラー画像として保存した際にハレーションが生じる．これを防ぐために，この例では，35 行目で NormalizeXYZ() メンバー関数を用いて正規化している．これは XYZ カラーの $(X + Y + Z)$ 値を用いて正規化する関数であるが，そのほかに輝度値 (Y 値) で正規化する NormalizeY() メンバー関数や XYZ ベクトルの長さで正規化する NormalizeVec()

メンバー関数がある．どれが最も良い結果を与えるかは一概に言えないので，正規化の値 (Norimalize() 関数の引数) の設定を含めて，種々試す必要がある．

得られた結果は 36 行目で SaveAsBmpSRGB() メンバー関数を用いて sRGB 画像として保存している．この時，この関数の内部では ColorSRGB() 関数を呼び出している．なお，この例のようにわずかな数の波長でシミュレーションし，それを sRGB に変換する場合，しばしば sRGB の色域では表せない結果となるため，これらの関数では sRGB 色空間で最も近い色に丸める処理を行っている．しかし，この処理は完全ではないので，微妙に異なった色合いに変換されることがある．

なお前述のように，ColorImage クラスでは，各ピクセルの色を CIE XYZ カラーとして扱っている．これらのピクセル値は GetColorXYZ() メンバー関数や GetColorXYY() メンバー関数を用いて取り出すことができるため，色度座標を定量的に取り扱いたい場合はこれらの関数を用いる．また，この様な目的では，グローバル定義されているユーティリティ関数として等色関数 ColorMatchingCIE1931XYZ() 関数や，XYZ から sRGB に変換する ColorSRGB() 関数，XYZ カラーから xyY カラーを求める ColorXYY() 関数等も用いることができる．

第 6 章

リファレンス

6.1 psl 名前空間内でグローバルな定義

6.1.1 グローバル関数

```
Vector ColorMatchingCIE1931XYZ(double lambda)
```

戻り値 Vector 型で表された XYZ カラー値

説　明 CIE1931 XYZ 表色系の等色関数を用い，波長 lambda に対応する XYZ カラー値を求める．

Note

- 変換可能な波長範囲は，400nm $\leq$ lambda $\leq$ 700nm であり 5 nm 単位である．補間は行っておらず，中間的な値は 5 nm 単位で丸められる．
- lambda が上の範囲にない場合はエラーになる．

```
Color ColorSRGB(Vector XYZ)
```

戻り値 Color 型で表された RGB カラー値．

説　明 CIE1931 XYZ 表色系の XYZ 値を sRGB の RGB 値に変換する．

Note

- Color クラスでは，各色は 0.0〜1.0 の範囲で表されているが，この関数で変換した結果の輝度が高く，この範囲から外れる場合がある．そのような場合，この関数では，色味を変えずに輝度を限界値に切り下げる処理を行っている．すなわち，RGB カラー値のいずれかが 1.0 を超える場合，RGB ベクトルの長さ全体を縮める形で 1.0 になるように変換している．
- XYZ 値が sRGB で表現可能な色空間から外れる場合は，D65 白色と XYZ 色の (x, y) 色度座標を結ぶ直線が sRGB 色空間の端と交わる値に変換している．
- この関数では sRGB で定義されたガンマ補正を行っている．

```
Vector ColorXYY(Vector XYZ)
```

戻り値　Vector 型で表された xyY カラー値.

説　明　Vector 型の XYZ 値を同じ Vector 型で表した xyY カラー値に変換する.

Note

- Vector クラスの x, y 成分が xyY カラーの (x, y) 値を表し，z 要素が Y 値 (輝度値) を表している.

```
void MultiplyDiffuser(WaveField& fb, const WaveField& diffuser)
```

戻り値　なし

説　明　fb に拡散位相 diffuser をサンプル点単位に乗算する．この関数が WaveField クラスの乗算関数や演算子と異なるのは，fb と diffuser のサイズが同じでない場合 (diffuser が fb より小さい場合)，diffuser のパターンが繰り返し乗算されることである.

Note

- この関数では fb と diffuser のサンプリング間隔は検査されない．つまり，サンプリング間隔が異なっていてもサンプル点単位に乗算される.

```
double CorrectedPolygonAmplitude(const WaveField& pfb, const WaveField& tfb, double brt,
double gamma = 0)
```

戻り値　輝度補正を行った表面関数の振幅値

説　明　明るさ (輝度)brt のポリゴンを傾斜フレームバッファ tfb に描画する際の振幅値を求める．ここで，引数 pfb は平行フレームバッファ，gamma は補正制限値である．これらは補正振幅値を求めるために必要である.

Note

- 引数 pfb はサンプリング間隔のみが参照される.

```
void PaintPolygonShape(WaveField& fb, const Polygon& polygon, Complex amp)
```

戻り値　なし

説　明　fb に Polygon 型の polygon の形状を振幅値 amp で描画する．この時，polygon の頂点座標は fb におけるローカル座標とみなされ，また頂点座標の z 値は無視される.

```
void PaintObjectSilhouette(WaveField& fb, const IndexedFaceSet model)
```

戻り値　なし

説　明　fb に IndexedFaceSet 型のモデル model を正投影して (z 座標値を無視して) そのシルエットを描画する.

Note

- fb と model の位置関係は，これらのオブジェクトが有するグローバル座標で評価される．

```
int GetTextureNamesMqo(const char *fname, std::vector<char*>& texNames)
int GetTextureNamesMqo(const wchar_t *fname, std::vector<char*>& texNames)
```

戻り値　テクスチャファイルの数

説　明　メタセコイア MQO 形式モデルファイル fname に設定されているテクスチャファイル名のリストを texNames に取得する．

```
int GetNumOfTextureMqo(const char *fname)
int GetNumOfTextureMqo(const wchar_t *fname)
```

戻り値　テクスチャファイルの数

説　明　メタセコイア MQO 形式モデルファイル fname に設定されているテクスチャファイルの数を取得する．

```
int GetTextureNamesMtl(const char *fname, std::vector<char*>& texNames)
int GetTextureNamesMtl(const wchar_t *fname, std::vector<char*>& texNames)
```

戻り値　テクスチャファイルの数

説　明　MTL 形式マテリアルファイル fname に設定されているテクスチャファイル名のリストを texNames に取得する．

```
int GetAlphaMapNamesMqo(const wchar_t* fname, std::Mtlvector<char*>& texNames)
int GetAlphaMapNamesMqo(const char *fname, std::vector<char*>& texNames)
```

戻り値　アルファマップの数

説　明　メタセコイア MQO 形式モデルファイル fname に設定されているアルファマップファイル名のリストを texNames に取得する．

6.1.2　データ型の定義

```
AofCallback
```

SurfaceBuilder クラスの AddObjectField() メンバー関数で進捗状況を報告するためのコールバック関数の関数ポインタを表すデータ型．SurfaceBuilder クラスの SetCallback() メンバー関数によるコールバック関数の設定に用いる．AddObjectField() メンバー関数は，ポリゴンを一つ処理するたびにこの関数を呼び出す．データ型の定義は次のとおり．

```
typedef bool (*AofCallback)(int,int,const psl::Polygon&)
```

ここで，最初の int 型引数はポリゴン番号，二つ目の int 型引数はスレッド番号，また psl::Polygon 型の引数は処理されたポリゴンである．

struct Statics

IndexedFaceSet クラスの GetDepthStatistics() メンバー関数および GetAreaStatistics() メンバー関数で戻り値である統計情報を表すデータ型．データ型の定義は次のとおり．

```
struct Statistics
{
    double Average;        //平均値
    double Sigma;          //標準偏差
    double Min;              //最小値
    double Max;            //最大値
}
```

6.1.3　列挙型の定義

SurfaceType

ポリゴンが平面であるか曲面であるかを識別するために用いる．

FLAT　　：平面
SMOOTH_GOURAUD：グーロー補間を用いる曲面
SMOOTH_PHONG　：フォン補間を用いる曲面

MaterialType

マテリアルの種類を表す列挙型．マテリアルは用いる CG ソフトに依存するためそれを識別するために用いる．

BASE　　：特にソフトを指定しない一般的なマテリアル
MQO　　：メタセコイア
METASEQUOIA：メタセコイア
BLENDER　：ブレンダー

ShadingModel

マテリアルのシェーディングモデルを識別するために用いる．

CONSTANT：コンスタント
LAMBERT　：ランバート
PHONG　　：フォン
BLINN　　：ブリン

SilhouetteMaskOption

シルエットマスクの位置を指定するための列挙型.

AUTO : 自動. 最も適切と思われる位置にシルエットマスクを設定する.
CENTER : ポリゴンの外接矩形の中央にシルエットマスクを設定する.
MANUAL : シルエットマスク位置を別の方法で指定する.

MaskType

オクルージョン処理に用いるマスクのタイプを指定するための列挙型.

SILHOUETTE : シルエットマスク.
SURFACE : サーフェースマスク.

6.2 IndexedFaceSet クラス

6.2.1 概要

IndexedFaceSet クラスは，図 6.1 に示すように番号付けした頂点座標を用いて，3 頂点あるいは 4 頂点からなるポリゴンによって表された物体モデルを格納するためのクラスである．このクラスは PointArray クラスから派生している．このクラスを用いて様々な物体形状ファイルからデータを読み込むことができる．

ポリゴン表

ポリゴン番号 (インデックス)	頂点番号			
	p1	p2	p3	p4
0	0	2	3	4
1	2	3	4	x
2	1	3	5	x
3	6	7	5	3
.				
.				

頂点表

頂点番号 (インデックス)	頂点座標		
	x	y	z
0	3.23	0.00	1.05
1	2.25	3.25	0.40
2	1.25	3.00	5.00
3	6.25	7.23	5.00
.			
.			

図 6.1 IndexedFaceSet の概念・構造

6.2.2 メンバー関数

6.2.2.1 基本的なメソッド

```
IndexedFaceSet()
IndexedFaceSet(const PointArray& pa)
```

戻り値 なし

説 明 コンストラクタ．第 1 の形式ではポリゴン数ゼロの空の表を作成する．第 2 の形式では，PointArray 型の

オブジェクト pa に含まれる全点を頂点表にコピーするが，ポリゴン数はゼロの空の表を作成する．

```
int GetNumberOfPolygon(void) const
```

戻り値　ポリゴン数
説　明　この物体モデルのポリゴン数を取得する．

```
int GetNumberOfVertex(int n) const
int GetNumberOfPoint(int n) const
```

戻り値　頂点数
説　明　ポリゴン番号 n のポリゴンの頂点数を取得する．

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数でなければならない．
- この物体モデルの全頂点数を取得するには PointArray クラスの GetN() メンバー関数を用いる．

```
int GetVertexIndex(int n, int m) const
int GetPointIndex(int n, int m) const
```

戻り値　頂点のインデックス
説　明　ポリゴン番号 n の m 番目 ($m \geq 0$) の頂点に対応する頂点表の頂点番号を取得する．

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数，またインデックス m は GetNumberOfVertex() メンバー関数が返す頂点数未満の整数でなければならない．

```
void Insert(const Point& p)
```

戻り値　なし
説　明　Point 型の頂点座標を頂点表の末尾に挿入する．

```
void Clear(void)
```

戻り値　なし
説　明　頂点数とポリゴン数がゼロの空のオブジェクトにする．

```
const Point& GetPoint(int n, int m) const
const Point& GetVertex(int n, int m) const
Point& PointAt(int n, int m)
Point& VertexAt(int n, int m)
```

戻り値 Point 型の頂点座標の参照

説　明 ポリゴン番号 n のポリゴンの m 番目 ($m \geq 0$) の頂点座標を示す Point 型オブジェクトの参照を取得する．Get で始まる関数では取得した参照を用いた頂点座標の変更は不可．

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数，またインデックス m は GetNumberOfVertex() メンバー関数が返す頂点数未満の整数である．

```
void InsertPolygon(int p1, int p2, int p3, int p4 = -1)
void InsertPolygon(Polygon polygon)
```

戻り値 なし

説　明 第 1 の形式では，頂点番号 p1〜p4 で構成されるポリゴンをポリゴン表の末尾に挿入する．第 2 の形式では，Polygon 型のポリゴン polygon の頂点を頂点表の末尾に挿入し，そのポリゴンをポリゴン表の末尾に挿入する．

Note

- 第 1 の形式で p4 は省略可能．p4 を省略した場合は 3 角形ポリゴンを挿入することになる．
- 第 2 の形式では，polygon からコピー可能な全ての属性 (カラー，テクスチャマップ等) がコピーされる．

```
void RemovePolygon(int n)
```

戻り値 なし

説　明 ポリゴン番号 n のポリゴンを削除する．

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数でなければならない．

```
void InsertPoints(const PointArray& pa)
```

戻り値 なし

説　明 4 点以下の PointArray 型点列 pa を頂点表末尾に挿入し，面表末尾に新しいポリゴンを挿入する．

```
Polygon GetPolygon(int n) const
```

戻り値　psl::Polygon 型のポリゴン
説　明　ポリゴン番号 n のポリゴンを psl::Polygon クラスのオブジェクトとして取得する.

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数でなければならない.

```
void SetColor(int n, const Color& cl)
```

戻り値　なし
説　明　ポリゴン番号 n のポリゴンに Color 型のカラー cl を設定する

```
const Color& GetColor(int n) const
```

戻り値　Color 型の色
説　明　ポリゴン番号 n のポリゴンの色を取得する.

Note

- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数でなければならない.

```
void SetTexMap(int n, int m, const TexMap& uv)
```

戻り値　なし
説　明　ポリゴン番号 n のポリゴンの頂点番号 m の頂点に，TexMap 型のテクスチャ UV 座標 uv を設定する.

```
TexMap GetTexMap(int n, int m) const
```

戻り値　TexMap 型の UV 座標
説　明　ポリゴン番号 n のポリゴンの頂点番号 m の頂点のテクスチャ UV 座標を取得する.

Note

- UV 座標が未定義の頂点についてこの関数を呼び出してもエラーにはならないが，取得された TexMap 型座標では $U = V = -1$ の未定義状態となっている.

```
void SetTexNum(int n, int num)
```

戻り値　なし
説　明　ポリゴン番号 n のポリゴンに，テクスチャ番号 num を設定する.

Note

- 複数テクスチャを用いたテクスチャマッピングに用いる.

```
int GetTexNum(int n) const
```

戻り値 テクスチャ番号

説　明 ポリゴン番号 n のポリゴンのテクスチャ番号を取得する.

Note

- 複数テクスチャを用いたテクスチャマッピングに用いる.

```
void SetAlphaMapNum(int n, int num)
```

戻り値 なし

説　明 ポリゴン番号 n のポリゴンに，アルファマップ番号 num を設定する.

Note

- 複数アルファマップを用いたアルファチャマッピングに用いる.

```
int GetAlphaMapNum(int n) const
```

戻り値 テクスチャ番号

説　明 ポリゴン番号 n のポリゴンのアルファマップ番号を取得する.

Note

- 複数アルファマップを用いたアルファチャマッピングに用いる.

```
void SetNormalVector(int n, const Vector& vec)
```

戻り値 なし

説　明 頂点番号 n の頂点に Vector 型の法線ベクトル vec を設定する

Note

- インデックス n は PointArray クラスの GetN() メンバー関数が返す点数未満の整数である.

```
const Vector& GetNormalVector(int n) const
```

戻り値 Vector 型の法線ベクトル

説　明 頂点番号 n の頂点の Vector 型の法線ベクトル vec を取得する.

Note

- インデックス n は PointArray クラスの GetN() メンバー関数が返す点数未満の整数である．

```
void AutoNormalVector(void)
```

戻り値　なし

説　明　スムースシェーディングの準備として，隣接するポリゴンから自動的に頂点の法線ベクトルを設定する．(西)

Note

- TfbGouraudShading クラスを用いたグーローシェーディングを行う場合，IndexedFaceSet クラスで表現されるモデルに対して事前にこの関数を呼び出しておく必要がある．呼び出していない場合はエラーになる．
- TfbFlatShading クラスを用いたフラットシェーディングを行う場合は本関数を呼び出しておく必要はない．呼び出しても実害はないが，現在の実装ではこの関数の実行にはかなりの時間を要するため注意が必要である．
- PSL Rel1.2 以前では，この関数内部で EliminateDuplicatePoint() メンバー関数を呼び出して重複点を統合していた．Rel1.3 以降では EliminateDuplicatePoint() メンバー関数を呼び出していない．これは，両面化されているポリゴンがある場合．頂点を統合すると頂点の法線ベクトルがゼロになってしまうためである．

```
void SetWidth(double v)
void SetHeight(double v)
void SetDepth(double v)
```

戻り値　なし

説　明　その物体モデルを含む外接矩形について，それぞれ，高さ，幅，奥行きを v に設定する．

Note

- これらの関数はモデル全体の大きさを比例的に変えるだけであり，物体モデルの形状を変形しない．すなわち，高さ，幅，奥行きのいずれかを設定すると他の値はそれに比例して変化する．
- 現在の高さ，幅，奥行きを取得するためには，それぞれ，PointArray クラスの GetWidth() メンバー関数，GetHeight() メンバー関数，GetDepth() メンバー関数を用いる．

```
int EliminateDefectPolygon(void)
```

戻り値　除去したポリゴン数

説　明　全てのポリゴンをスキャンし，3 点に満たない不完全なポリゴンを除去する．

```
int EliminateDuplicatePoint(void)
```

戻り値　除去した点数

説　明　全てのポリゴンをスキャンし，頂点表の重複点を発見して除去した後にポリゴン表を修正する．

Note

- このメソッドを用いた後は，同じ頂点は同じ頂点番号を持つことが保証される．
- 同じ頂点であるか否かの判定には Vector クラスの==演算子が用いられる．

```
void BreakQuad(int n, int method = 0)
```

戻り値　無し

説　明　インデックス n の四角形ポリゴンを method の方法で三角形に分割する．

Note

- ポリゴンの頂点が (p_0, p_1, p_2, p_3) であるとき，method $= 0$ では，(p_0, p_1, p_2) と (p_0, p_2, p_3) の三角形に分割する．一方，method $\neq 0$ では，(p_0, p_1, p_3) と (p_1, p_2, p_3) の三角形に分割する．
- インデックス n は GetNumberOfPolygon() メンバー関数が返すポリゴン数未満の整数である．
- このメソッドを用いるとポリゴン数が増加する．

```
void BreakQuadAll(int method = 0)
```

戻り値　無し

説　明　全てのの四角形ポリゴンを method の方法で三角形に分割する．

Note

- method については BreakQuad() メンバー関数を参照．
- このメソッドを用いるとポリゴン数が増加する．

```
void Add(const IndexedFaceSet& ifs)
```

戻り値　無し

説　明　対象オブジェクトに別の IndexedFaceSet 型のモデル ifs を統合する．

```
void Split(IndexedFaceSet& front, IndexedFaceSet& back, double z)
```

戻り値　なし

説　明　奥行き位置 z の z 軸に垂直な断面で対象オブジェクトのメッシュ構造を分割し，手前側を front に，奥側を back に格納する．(東)

```
void AutoSplit(IndexedFaceSet& front, IndexedFaceSet& back)
```

戻り値　なし

説　明　z 軸に垂直な断面の面積が最大となる位置で対象オブジェクトのメッシュ構造を分割し，手前側を front に，奥側を back に格納する．(東)

```
void DivideByDepth(IndexedFaceSet& front, IndexedFaceSet& back, double z) const
```

戻り値　なし

説　明　対象オブジェクトを奥行き位置 z より奥側の部分と手前側の部分に分割し，分割したモデルをそれぞれ front と back に格納する．

Note

- ポリゴン自体は分割されない．前後に振り分けられるだけである．

```
void SortByDepth(void)) const
```

戻り値　なし

説　明　対象オブジェクトのポリゴンを z 座標の小さいものから大きなものへ順にソートする．

Note

- z 座標値の最も小さなポリゴンがポリゴン番号 0 となる．
- スイッチバック法を用いる関数 AddObjectFieldSb(), AddObjectFieldSbUnity() では内部でこの関数を呼び出している．

```
Statistics GetAreaStatistics(void) const
```

戻り値　Statics 型の面積統計情報

説　明　ポリゴンの面積統計情報を取得する．(永江)

Note

- 面積統計情報の単位はポリゴンの頂点座標を定義している単位によって決まる．

```
void TessellationByDepth(double depth)
```

戻り値　無し

説　明　全ポリゴンの奥行長さが depth 以下になるまでテッセレーション処理を行う．(永江)

```
void TessellationByArea(double area)
```

戻り値 無し

説　明 全ポリゴンの最大面積が area 以下になるまでテッセレーション処理を行う．(永江)

```
void GetLineTable(std::vector<LineSegment>& ls)
```

戻り値 無し

説　明 稜線の表を LineSegment クラスの配列 ls に取得する．

```
int AddWireFrameField(WaveField& fb, double density, double amp = 1.0, int from = 0, int
to = -1)
```

戻り値 追加した点光源数

説　明 対象オブジェクトのワイヤフレームモデルの物体光波を点光源密度 density $[1/\mathrm{m}]$ でフレームバッファ fb に計算する．この時，GetLineTable() メンバー関数の稜線番号 from から to までの稜線の光波を振幅 amp で計算する．from と to を省略した場合はすべての稜線が計算される．

Note

- 暫定仕様として，三角関数の表参照を行った点光源法で光波を計算する．

```
static IndexedFaceSet SquareObject(double Aspect, int nx, int ny)
```

戻り値 平面物体のポリゴンメッシュ

説　明 nx × ny のポリゴンに分割した平面物体のポリゴンメッシュを作成する．主として平行投影テクスチャマッピングにより背景の壁紙等を作成することを目的としている．Aspect は高さ／幅のアスペクト比を設定する．

Note

- 物体の実サイズは SetWidth() メンバー関数等を，また位置などは SetCenter 等 (PointArray クラスから継承) を用いて設定する．
- この関数は四角形ポリゴンを生成する．三角形ポリゴンが必要な場合は BreakQuadAll() メンバー関数を用いる．

Example

```
IndexedFaceSet wallObject = SquareObject(0.8, 50, 50);
wallObject.SetWidth(50e-3);                                    //物体幅を50mmに設定
wallObject.SetCenter(Point(0, 0, -100e-3)); //中心位置を-100mmに設定
```

6.2.2.2 計測を行うメソッド

```
double GetArea(unsigned int n) const
```

戻り値　面積

説　明　インデックス n のポリゴンの面積を取得する．(永江)

Note

- 面積の単位はポリゴンの頂点座標を定義している単位によって決まる．

```
Statistics GetDepthStatistics(void) const
```

戻り値　Statics 型の奥行き統計情報

説　明　ポリゴンの奥行き統計情報を取得する．(永江)

Note

- 奥行き統計情報の単位はポリゴンの頂点座標を定義している単位によって決まる．

6.2.2.3　モデルのマテリアル（柳谷）

```
bool HasMaterial(void) const
bool HasMaterial(unsigned int n) const
```

戻り値　マテリアル表があれば true

説　明　第 1 の形式では，この物体モデルが MaterialList クラスのマテリアル表を持つか判定する．第 2 の形式では，インデックス n の面が MaterialList クラスのマテリアル表を持つか判定する．

```
unsigned int GetMaterialNum(unsigned int n) const
```

戻り値　インデックス n の面に対するマテリアル番号

説　明　この物体モデルが持つ MaterialList クラスのマテリアル表から，インデックス n の面に対するマテリアル番号を取得する．

```
const Material* GetMaterial(unsigned int n) const
```

戻り値　インデックス n の面に対するマテリアルのポインタ

説　明　この物体モデルが持つ MaterialList クラスのマテリアル表から，インデックス n の面に対する Material クラスのポインタを取得する．

```
const psl:SurfaceType GetSurfaceType(unsigned int n) const
```

戻り値 インデックス n の面の種類
説　明 インデックス n の面の種類を SurfaceType 列挙型で取得する．

```
double GetSmoothingAngle(unsigned int n) const
```

戻り値 インデックス n の面のスムージング角度．単位：度
説　明 インデックス n の面のスムージング角度を取得する．

6.2.2.4 モデルファイルの読み込み

```
int LoadWrl(const char* fname)
int LoadWrl(const wchar_t* fname)
int LoadDxf(const char* fname)
int LoadDxf(const wchar_t* fname)
```

戻り値 読み込んだポリゴン数
説　明 それぞれ，WRL(VRML) 形式，DXF 形式のファイルから含まれる全ての物体モデルの形状データを読み込む．

Note

- WRL 形式は VRML Ver 2.0 のみ読み込める．

```
void SaveAsDxf(const char* fname, double mul = 1000.0)
```

戻り値 なし
説　明 DXF 形式ファイル fname として全ての物体形状データを保存する．このとき拡大率 mul を座標値に乗算する．マルチバイト文字列版．

Note

- 現在のバージョンでは保存されるのは形状データのみである．

```
void LoadPov(const char* fname)
void LoadPov(const wchar_t* fname)
void LoadPov(const char* fname, int n)
void LoadPov(const wchar_t* fname, int n)
```

戻り値 無し
説　明 1 番目と 2 番目の形式では，POV 形式のファイルから物体モデル全体の形状データを読み込む．3 番目と 4

番目の形式では，n 番目 ($n \geq 0$) の物体のみを読み込む．

```
void LoadMqo(const char *fname)
void LoadMqo(const wchar_t *wfname)
```

戻り値　無し

説　明　モデラーソフトのメタセコイア標準ファイル形式である MQO フォーマットの物体モデルを読み込む．TfbUvMapping クラスと組み合わせることで UV マッピングができる．

Note

- 実際の UV マッピングは，TfbUvMapping クラスを用いて行う．
- テクスチャファイル名リストを取得するには GetTextureNamesMqo() 関数を，またテクスチャファイル数を取得するには GetNumOfTextureMqo() 関数を用いる．

```
void LoadObj(const char *fname)
void LoadObj(const char *fname, const char *mname)
void LoadObj(const wchar_t *fname)
void LoadObj(const wchar_t *fname, const wchar_t *mname)
```

戻り値　無し

説　明　ファイル名 fname の OBJ フォーマットの物体モデルを読み込む．2 番目と 4 番目の形式では，さらに，ファイル名 mname の MTL 形式マテリアルファイルを読み込む．この場合，UV マッピングやマテリアルを反映したレンダリングができる．

Note

- UV マッピングには，TfbUvMapping クラスを用いて行う．
- テクスチャファイルの数とファイル名リストを取得するには GetTextureNamesMtl() 関数を用いる．
- マテリアルを反映したシェーディングについては，TfbLambertShading クラスと TfbPhongSpecularShading クラスを参照．

6.3　SurfaceBuilder クラス

SurfaceBuilder クラスはポリゴン形状の面光源の光波 (ポリゴン光波) を計算するためのクラスである．SurfaceBuilder クラスは FieldParam クラスを継承しているため，サンプリング間隔やサンプリング数，波長など通常の WaveField 型のフィールドと同じパラメータを保持している．

6.3.1　基本的なパラメータを設定取得するメンバー関数

FieldPram クラスから，以下のメンバー関数を継承している (主要な関数だけを記載)．これらの使用方法は WaveField クラスを参照．

SetNx, SetNy, GetNx, GetNy, SetPx, SetPy, GetPx, GetPy, SetWavelength, GetWavelength, SetOrigin,

GetOrigin

6.3.2 コンストラクタ

```
SurfaceBuilder(int nx, int ny, double px, double py, double wavelength)
SurfaceBuilder(double wavelength, double px, double py)
SurfaceBuilder(const WaveField& wf)
SurfaceBuilder(void)
```

戻り値 なし

説　明 コンストラクタ．第 1 の形式では，サンプル点数 nx×ny，波長 wavelength，サンプリング間隔 px×py の物体光波合成用のインスタンスを作成する．第 2 の形式ではサンプル点数がゼロ ($N_x = N_y = 0$) となる．第 3 の形式では，波長とサンプリング間隔が WaveField クラスのオブジェクト wf と一致する SurfaceBuilder オブジェクトを生成する．第 4 の形式では WaveField クラスのデフォルト値のパラメータでオブジェクトを生成する．

Note

- ポリゴン光波の帯域制限 (6.3.6 節参照) を用いないのであれば，SurfaceBuilder オブジェクトで必要とするパラメータは，計算結果を得るフレームバッファのサンプリング間隔と波長のみである．従って，第 1 の形式であっても WaveField クラスのサンプル数等のパラメータは用いられない．
- ポリゴン光波の帯域制限 (6.3.6 節参照) を用いる場合は，SurfaceBuilder オブジェクトがホログラム面での物体光波の位置と大きさを表している見なされる．従って，サンプリング間隔と波長に加えて，物体光波のサンプル数とホログラム面の中心位置を SurfaceBuilder オブジェクトに正しく設定しなければならない．

6.3.3 ポリゴン光波計算精度に関係するパラメータを設定取得するメンバー関数

```
void SetDiffractionRate(double df)
double GetDiffractionRate(void)
```

戻り値 なし / 回折率

説　明 回折率を設定/取得する．回折率とは，最大回折範囲に対して実際の回折範囲として扱う範囲の割合であり，0 ～ 1.0 の値である．回折率 1.0 に設定した場合は，ポリゴン光波が平行フレームバッファのサンプリング間隔で定まる最大回折角で回折するとしてその回折範囲を決定する．

Note

- 既定値は 1.0 である．
- Rel 1.9 以前では関数名が SetDiffractionRatio()/GetDiffractionRatio() であった．現在でもこれらの関数名は使用可能であるが使用しないことが望ましい．

```
void SetCullingRate(double val)
double GetCullingRate(void)
```

戻り値　なし / カリング率

説　明　カリング率を設定/取得する．CG では通常，法線がスクリーンと反対向きになっているポリゴンは処理を省略する背面カリング処理を行う．CGH では，最大回折角が大きい場合には，ホログラムに対して垂直を超えて反対向きになったポリゴンでもその光がホログラムに届く場合がある．そのため，どこまでを処理範囲とするかの設定が必要である．カリング率とは，対象ポリゴンがどの程度背面向きの場合に処理を省略 (背面カリング) するかの割合であり，値が大きいほど処理されるポリゴンが減少する．カリング率は概ね次の定義に従う．

1.0　ホログラム全体に光が到達すると推定されるポリゴンのみを処理する．部分的にしか光が到達しないポリゴンは全て背面カリング処理を行う．

0.5　概ね垂直より裏向きのポリゴンの場合は背面カリング処理を行う．

0.0　光が全くホログラムに到達しないと推定されるポリゴンのみ背面カリング処理を行う．

Note

- 既定値は 0.5 である．

```
void SetPfbExtension(bool onoff)
void SetTfbExtension(bool onoff)
```

戻り値　なし

説　明　それぞれ onoff が true の時に平行フレームバッファ (PFB) と傾斜フレームバッファ (TFB) の 4 倍拡張を設定する．4 倍拡張を設定すると計算精度が高くなる．一方，メモリを 4 倍消費し計算速度は遅くなる．

Note

- 既定では平行フレームバッファの拡張は**オフ**になっている．
- 既定では傾斜フレームバッファの拡張は**オン**になっている．

```
void SetDefaultInterpol(Interpol interpol)
Interpol GetDefaultInterpol(void)
```

戻り値　なし / Interpol 列挙型の補間方法

説　明　デフォルトの補間方法を設定/取得する．interpol は Interpol 列挙型の補間方法.

Note

- AddPolygonField() メンバー関数，AddObjectField() メンバー関数，AddPolygonFieldSb() メンバー関数，AddObjectFieldSb() メンバー関数，AddObjectFieldSbUnity() メンバー関数等の関数ではこの関数を用いて補間方法の設定取得を行う．

6.3.4 ポリゴン光波計算に用いるメンバー関数

```
void SetShader(TfbPaint& sh)
TfbPaint* GetShader(void)
```

戻り値 なし/TfbPaint 型オブジェクトへの参照

説　明 TfbPaint クラスのシェーダーオブジェクト sh を登録/取得する．SurfaceBuilder は，TfbPaint クラスの PaintTfb() メンバー関数を継承している sh のメンバー関数を使用して傾斜フレームバッファを描画する．これによりポリゴンをシェーディングすることができる．

Note

- シェーダ (シェーダオブジェクト) については，6.4.2 節を参照．
- テクスチャマッピングを行う場合もシェーダーオブジェクトは必ず設定しなければならない．シェーダーオブジェクトを設定せずに SurfaceBuilder クラスを用いることはできない．
- シェーディング無しでテクスチャマッピングを行う場合は，TfbFlatShading クラスの第 1 の形式のコンストラクタで生成したオブジェクトをシェーダーオブジェクトに用いる．

```
void SetTexture(TfbPaint& sh)
TfbPaint* GetTexture(void)
```

戻り値 なし/TfbPaint 型オブジェクトへの参照

説　明 TfbPaint クラスのテクスチャマッピングオブジェクト sh を登録/取得する．SurfaceBuilder は，TfbPaint クラスの PaintTfb() メンバー関数を継承している sh のメンバー関数を使用して傾斜フレームバッファにテクスチャ画像を乗算する．これによりテクスチャマッピングすることができる．

Note

- テクスチャマッパー (テクスチャマッピングオブジェクト) については，6.4.3 節を参照．
- テクスチャマッピングを行わない場合は，テクスチャマッピングオブジェクトを登録する必要は無い．
- シェーディング無しでテクスチャマッピングを行う場合は，TfbFlatShading クラスの第 1 の形式のコンストラクタで生成したオブジェクトをシェーダーオブジェクトに用いる．

```
void SetAlphaMap(TfbPaint& sh)
TfbPaint* GetAlphaMap(void)
```

戻り値 なし/TfbPaint 型オブジェクトへの参照

説　明 TfbPaint クラスのアルファマッピングオブジェクト sh を登録/取得する．SurfaceBuilder は，TfbPaint クラスの PaintTfb() メンバー関数を継承している sh のメンバー関数を使用してサーフェースマスクの透過度を変化する．これによりアルファチャマッピングを実行することができる．

Note

- アルファマッピングを行う場合，SetMaskType() メンバー関数を用いてマスクタイプをサーフェースマスク (MaskType::SURFACE) に変更した上，スイッチバック法を用いる必要がある．
- アルファマッピングを行わない場合は，アルファマッピングオブジェクトを登録する必要は無い．
- アルファマッピングオブジェクトについては，6.4.4 節を参照．

```
void SetCurrentPolygon(Polygon& p)
```

戻り値　なし

説　明　Polygon クラスのポリゴンデータ p を処理対象として設定する．

```
const Polygon& GetPolygon(void) const
```

戻り値　Polygon 型のポリゴン

説　明　SurfaceBuilder オブジェクトないで保持しているポリゴンデータを取得する．

Note

- この関数では SurfaceBuilder オブジェクトに保持しているポリゴンを取得するが，このポリゴンは下記のように処理のステージによって変化する．
- AddPolygonField() メンバー関数，AddObjectField() メンバー関数，SetupTfb() メンバー関数実行以前では，SetCurrentPolygon() メンバー関数で設定したポリゴンデータがそのまま得られる．
- AddPolygonField() メンバー関数，AddObjectField() メンバー関数，SetupTfb() メンバー関数実行以後では，SetCurrentPolygon() メンバー関数で設定したポリゴンデータそのままではなく，ホログラムと平行になるように回転されたポリゴンデータが得られる．

```
void AddPolygonField(WaveField& frameBuff)
void AddPolygonField(WaveField& frameBuff, Interpol ip)
```

戻り値　なし

説　明　SetCurrentPolygon() メンバー関数を用いて設定したポリゴンの光波をフレームバッファ frameBuff 上で計算し，フレームバッファに加算する．第 1 の形式ではポリゴン光波計算にデフォルトの補間方法を用いる (Rel1.0 以前では BICUBIC を用いる)．第 2 の形式では Interpol 列挙型の ip で用いる補間法を指定する．

Note

- この関数呼び出し前に SetCurrentPolygon() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．
- この関数内では，背面カリング処理は行われているが，カレントポリゴンの光がフレームバッファに到達するかどうかのチェックは行われていない．すなわち，カレントポリゴンの光がフレームバッファに到達しない場合も計算は実行される．しかし，この場合の光波はフレームバッファに加算されないため，計算時間が無駄になる．
- 第 1 の形式では補間方法の設定/取得は，SetDefaultInterpol() メンバー関数 / GetDefaultInterpol() メンバー関数で行われる．

```
void AddObjectField(WaveField& frameBuff, IndexedFaceSet& model)
void AddObjectField(WaveField& frameBuff, IndexedFaceSet& model, Interpol ip)
void AddObjectField(WaveField& frameBuff, IndexedFaceSet& model, int t)
void AddObjectField(WaveField& frameBuff, IndexedFaceSet& model, int t, Interpol ip)
```

戻り値 なし

説　明 IndexedFaceSet 型の物体モデル model の光波をフレームバッファ frameBuff 上で計算しフレームバッファに加算する．第 1 と第 2 の形式では実行しているマシンのプロセッサコアを同じ数のポリゴンを並列処理するのに対して第 3 と第 4 の形式では，t 個のポリゴンを並列処理する．また，第 1 と第 3 の形式ではデフォルトの補間方法を用いる (Rel1.0 以前では BICUBIC を用いる) のに対して，第 2 と第 3 の形式では Interpol 列挙型の ip で用いる補間法を指定する．

Note

- この関数内では，マシンのプロセッサコアを最大限に使うように WFL のスレッド数を自動制御している．すなわち，この関数を実行している間は，WFL のスレッド数は wfl::SetNumThreads() 関数により設定されているスレッド数から変更される．そのため，この関数を用いたジョブは，原則として一つのマシン内で他のジョブと共存することはできない．
- この関数終了後には，WFL のスレッド数は関数実行前に wfl::SetNumThreads() 関数により設定されていたスレッド数に戻される．
- 使用するメモリ量はポリゴンの並列処理数に比例するため，第 1 あるいは第 2 の形式ではメモリ数が不足する場合がある．実行時に使用メモリ量を観察し，メモリを使い切るようであれば第 3 あるいは第 4 の形式を用いて並列処理数を制限する．
- 第 3・第 4 の形式で設定する並列処理数をマシンの総プロセッサコア数の約数に設定した場合にのみ，マシンの全コアが利用できる．
- この関数を終了後は，WFL のスレッド数は関数実行前のスレッド数に戻る．
- この関数では背面カリング処理が行われている．また各々のポリゴンの光がフレームバッファに到達するかどうかもチェックされており，到達不能なポリゴン光波は計算されない．
- 第 1 と第 2 の形式では補間方法の設定/取得は，SetDefaultInterpol() メンバー関数 / GetDefaultInterpol() メンバー関数で行われる．

```
void AddObjectFieldMt(WaveField& frameBuff, IndexedFaceSet& model, Interpol ip =
wfl::BICUBIC))
void AddObjectFieldMt(WaveField& frameBuff, IndexedFaceSet& model, int threads, Interpol
ip = wfl::BICUBIC)
```

戻り値 なし

説　明 IndexedFaceSet 型の物体モデル model の光波をフレームバッファ frameBuff 上で計算しフレームバッファに加算する．第 1 の形式では計算は psl::SetNumThreads() 関数で設定されている個数のスレッドを用いてポリゴンの並列計算を行い，デフォルトでバイキュービック法で補間処理する．第 2 の形式では，threads 個のスレッドを用いて計算を行い，デフォルトでバイキュービック法で補間処理する．

Note

- この関数では WFL 関数のスレッド数の制御は行っていない．WFL 関数や一部の PSL 関数 (wfl::GetNumThreads() 関数によって得られるスレッド数で実行される関数) のスレッド数は，wfl::SetNumThreads() 関数によって設定された数になる．
- 使用するメモリ量はポリゴンの並列処理数に比例する．
- この関数では背面カリング処理が行われている．また各々のポリゴンの光がフレームバッファに到達するかどうかもチェックされており，到達不能なポリゴン光波は計算されない．

```
void AddObjectFieldSm(WaveField& frame, const IndexedFaceSet& model, int ndiv = 10, bool
exact = false, bool backg = true, double pos = 0.5, int t = 0)
```

戻り値　なし

説　明　IndexedFaceSet 型の物体モデル model の光波をフレームバッファ frameBuff の位置で計算し，フレームバッファに加算する．この時，速度向上のために，物体モデルは奥行方向に ndiv 個のサブモデルに分割して処理される．backg = true の場合は背景光波の物体単位シルエットマスク処理を行う．pos は相対的マスク位置 (0.0: 最背面, 1.0: 最前面) を示す．またこの場合，初期状態のフレームバッファ frame を最初のサブモデルの計算開始位置まで伝搬計算する．backg = false の場合は，伝搬計算を行なわずフレームバッファ frame の位置を開始位置に設定しゼロクリアする．なお，exact = true であれば ExactAsmProp() メンバー関数を用いて伝搬計算を行ない，exact = false であれば AsmProp() メンバー関数を用いる．既定 (t=0) では，wfl::SetNumThreads() 関数で設定されている個数のスレッドを用いてポリゴンの並列計算を行うが，t != 0 の場合は t 個のポリゴンを並列処理する．

Note

- このメソッドは，AddObjectFieldSb() メンバー関数とほぼ同様の処理をスイッチバックを使わずに物体単位シルエット法で行う．

```
void SetCallback(AofCallback prog)
```

戻り値　なし

説　明　AddObjectField() メンバー関数で進捗状況を報告するためのコールバック関数として AofCallback 型の prog を設定する．AddObjectField() メンバー関数は，ポリゴンを一つ処理するたびにこのコールバック関数を呼び出す．この関数を用いてコールバック関数を設定しない場合，既定では次の関数が呼び出される．ここで，コールバック関数の最初の引数 n は処理されたポリゴンのインデックスであり，psl::Polygon 型の引数 polygon は，処理されたポリゴンである．また引数 t はそのポリゴンを処理したスレッド番号である．

Example デフォルトで設定されているコールバック関数

```
bool DefaultCallback(int n, int t, const psl::Polygon\& polygon)
{
    printf("%d(%d) ", n, t);
    return true;
}
```

Note

- コールバック関数によるメッセージ表示を止めるためには，次のように NULL を設定する．

```
SurfaceBuilder sb;
sb.SetCallback(NULL);
```

6.3.5 スイッチバック法を用いた隠面消去付きポリゴン光波計算に用いるメンバー関数

スイッチバック法では，隠面消去とポリゴン光波計算を同時に行い，自己オクルージョンを有する物体の隠面消去も行えるが，反面，並列計算が難しく物体サイズによっては計算時間が長くなる場合がある．

```
void SetMaskType(psl::MaskType mt)
```

戻り値 なし

説　明 スイッチバック法による隠面消去に用いるマスクタイプを MaskType 列挙型の mt に設定する．既定では MaskType::SILHOUETTE であり，シルエットマスクを隠面消去に用いる．MaskType::SURFACE を指定すると，スイッチバック法でサーフェースマスクを用い，ギャップの無い高精度な隠面消去を行うが，一方，計算時間と使用メモリ量が大きく増加するため，注意が必要．(中本)

```
MaskType GetMaskType(void)
```

戻り値 MaskType 列挙型のマスクタイプ．

説　明 MaskType 列挙型のマスクタイプを取得する．(中本)

```
void SetSilhouetteMaskOption(psl::SilhouetteMaskOption smo))
```

戻り値 なし

説　明 各ポリゴンのシルエットマスク位置を SilhouetteMaskOption 列挙型の smo で設定する．smo=AUTO の場合はポリゴンの向きの応じて最適なマスク位置が自動的に選択される．既定では smo=AUTO である．smo=CETER の場合は，ポリゴンの外接矩形の奥行きの中央にシルエットマスクが設定される．smo=MANUAL の場合は，SetSilhouetteMaskPos() メンバー関数でシルエットマスク位置をしていする．

Note

- SetMaskType() メンバー関数で，MaskType::SURFACE を指定している場合には，このメソッドによるシルエットマスク位置の指定は無視される．

```
SilhouetteMaskOption GetSilhouetteMaskOption(void)
```

戻り値 SilhouetteMaskOption 列挙型のマスクオプション．

説　明 SilhouetteMaskOption 列挙型のマスクオプションを取得する．

```
void SetSilhouetteMaskPos(double pos)
```

戻り値　なし

説　明　SetSilhouetteMaskOption() メンバー関数でマスク挿入位置が MANUAL に設定されている場合に，シルエットマスク挿入位置を pos で指定する．pos は区間 [0.0, 1.0] で定義される (0.0: 最背面, 1.0: 最前面).

```
double GetSilhouetteMaskPos(void)
```

戻り値　区間 [0.0, 1.0] のシルエットマスク挿入位置 (0.0: 最背面, 1.0: 最前面).

説　明　シルエットマスク挿入位置を取得する．

```
void AddPolygonFieldSb(WaveField& frame, const Polygon& polygon)
void AddPolygonFieldSb(WaveField& frame)
```

戻り値　なし

説　明　スイッチバック法によりフレームバッファ frameBuff からカレントポリゴンの位置に後方伝搬して光波遮蔽し，ポリゴン光波を加算する．第 1 の形式では polygon によってポリゴンを指定する．第 2 の形式では，SetCurrentPolygon() メンバー関数で設定されたポリゴンを処理する

Note

- Rel 1.9 以前は，この関数を呼び出す前に IsVisible() メンバー関数を呼び出しておく必要があったが，Rel 1.9 以降はこの関数内で IsVisible() メンバー関数を呼び出しているため，事前呼び出しは不要になった．
- 第 2 の形式では SetCurrentPolygon() メンバー関数を用いてポリゴンを設定する．
- この関数では，ポリゴン光波計算において既定の方法で補間処理する．補間方法を変えるには SetDefaultInterpol() メンバー関数を用いる．
- シェーディングやテクスチャマッピングは，AddPolygonField() メンバー関数と同じ方法で行うことができる．

```
void AddObjectFieldSb(WaveField& frame, IndexedFaceSet& model, int ndiv = 10, bool exact
= false, bool backg = true)
```

戻り値　なし

説　明　スイッチバック法によりフレームバッファ frame に IndexedFaceSet 型の物体モデル model の光波を加算する．物体モデルは ndiv 個のサブモデルに分割して処理される．exact スイッチを true とすると，サブモデル間の伝搬計算に ExactAsmProp() メンバー関数を用いる．

backg = true の場合は，初期状態のフレームバッファ frame をスイッチバック法開始位置まで伝搬計算する．この時，exact = true であれば ExactAsmProp() メンバー関数 を用いて伝搬計算を行なう．exact = false であれば AsmProp() メンバー関数 を用いる．backg = false の場合は，伝搬計算を行なわずフレームバッファ frame の位置を開始位置に設定しゼロクリアする．

Note

- この関数内で IndexedFaceSet クラスの SortByDepth() メンバー関数を呼び出しているので，事前にポリゴンを奥行き順にソートしておく必要はない．
- この関数終了後にはフレームバッファ frame の位置は最後にスイッチバック法を行った物体平面に移動している．
- この関数では，物体を奥行き方向に均等分割してから処理する．内部的には均等分割後に AddObjectFieldSbUnity() メンバー関数を呼び出している．
- 分割数 ndiv は物体の奥行きが深い場合には大きな値にする方が計算が高速化する．ただし，SegWaveField クラスを使った分割計算を行う場合など伝搬計算が遅い場合は，逆に計算時間が長くなる．デフォルトでは 10 分割となっている．
- 光波計算する対象オブジェクトに背面より入射する背景光波が本関数呼び出し前にフレームバッファ frame に設定されており，かつその光波が frame のサンプリング領域全体に広がっている場合には，exact = true とすることにより，背景光波のエイリアシングを防止することができる．ただし，その場合計算速度は遅くなる．
- 背景光波が無い場合や frame 全面に広がっていない場合など，上記に該当しない場合は，exact = false の方が良い．デフォルトでは，exact = false である．
- ポリゴン光波計算において既定の方法で補間処理する．補間方法を変えるには SetDefaultInterpol() メンバー関数を用いる．
- シェーディングやテクスチャマッピングは，AddObjectField() メンバー関数と同じ方法で行うことができる．

```
void AddObjectFieldSbUnity(WaveField& frame, IndexedFaceSet& model)
```

戻り値 なし

説　明 スイッチバック法によりフレームバッファ frameBuff に IndexedFaceSet 型の物体モデル model の光波を加算する．物体モデルはサブモデルに分割されず単一の物体として計算される．

Note

- この関数は AddObjectFieldSb() メンバー関数と違って物体を分割せずに計算を行う．独自の方法で物体を分割したい場合などに用いる．

6.3.6 ポリゴン光波の帯域制限に用いるメンバー関数

伝搬計算では一般にサンプリング数の 4 倍拡張が必要である．しかし，これは光波がサンプリング領域を超えて広がるためである．ポリゴン光波をホログラムの領域以上に広がらないように帯域制限することにより，この 4 倍拡張を不要にできる．また，それだけではなく，回転変換のリサンプリング領域を帯域内に制限することにより，ポリゴンの位置によっては計算を大幅に高速化できる．本節のメソッドでは，ポリゴン光波の帯域制限に関する機能を提供する．

Note

- 帯域制限を行うためには，SurfaceBuilder クラスにホログラム面での物体光波のサンプリング間隔やサンプリング数を設定しておく必要がある．SurfaceBuilder クラスは WaveField クラスを継承しているため，WaveField クラスの SetNx() メンバー関数や SetNy() メンバー関数，SetPx() メンバー関数，SetPy() メンバー関数などのメソッドが利用できる．また，SurfaceBuilder クラスのコンストラクタでもこれらの設定ができる．

- Rel 1.9 以降では，サンプリング間隔やサンプリング数を設定せずに帯域制限を行うとエラーになる．
- 帯域制限を用いたサンプルソース：AddObjectField() メソッド，UV テクスチャマッピング，スイッチバック法．

```
void SetBandLimitMethod(int method)
```

戻り値　なし

説　明　method が 0 の場合は帯域制限を行わない．1, 2, 3 のいずれかである場合は帯域制限を行うが，1 < 2 < 3 の順序で帯域制限が厳しくなり，4 倍拡張無しでもエイリアシングが生じにくくなる．一方，帯域制限が厳しいほうが，光波の広がりが制限され視域が減じる，周辺の視点から像を見た時に一部のポリゴンが欠ける等の症状が出る可能性がある．

Note

- 帯域制限を行うためには，SurfaceBuilder クラスにホログラム面での物体光波のサンプリング間隔やサンプリング数を設定しておく必要がある．SurfaceBuilder クラスは WaveField クラスを継承しているため，WaveField クラスの SetNx() メンバー関数や SetNy() メンバー関数，SetPx() メンバー関数，SetPy() メンバー関数などのメソッドが利用できる．また，SurfaceBuilder クラスのコンストラクタでもこれらの設定ができる．

```
GetBandLimitMethod(void) const
```

戻り値　帯域制限の手法 (レベル)

説　明　帯域制限の手法 (レベル) を取得する．手法については，SetBandLimitMethod() メンバー関数を参照．

```
BoundingBox BandForPoint(Point p) const
```

戻り値　BoundingBox 型で示される周波数帯域 (1/m 単位)

説　明　Point 型の座標 p にある点から出た光がちょうどホログラムを通過するのに必要な帯域を返す．

```
BoundingBox BandForPolygon(const PointArray& pa) const
```

戻り値　BoundingBox 型で示される周波数帯域 (1/m 単位)

説　明　現在設定されている手法に従って制限する帯域を返す．

```
BoundingBox BandForPolygon1(const PointArray& pa) const
```

戻り値　BoundingBox 型で示される周波数帯域 (1/m 単位)

説　明　PointArray 型の pa の個別の頂点について求めた帯域をすべて内包する帯域を返す．

```
BoundingBox BandForPolygon3(const PointArray& pa) const
```

戻り値 BoundingBox 型で示される周波数帯域 (1/m 単位)

説　明 PointArray 型の pa の個別の頂点について求めた帯域に共通な帯域を返す.

```
void BandLimiting(WaveField& wf, BoundingBox bb)
```

戻り値 なし

説　明 スペクトル状態の WaveField 型のフィールド wf を BoundingBox 型の bb で示される帯域で制限する.

```
void BandToWindow(WaveField& wf, BoundingBox bb) const
```

戻り値 なし

説　明 スペクトル状態の WaveField 型のフィールド wf に対して，BoundingBox 型の bb で示される帯域をウィンドウ領域として設定する.

```
void PolygonBandLimiting(WaveField& wf, const PointArray& polygon)
```

戻り値 なし

説　明 スペクトル状態の WaveField 型のフィールド wf を PointArray 型の polygon で示される頂点群に対して，SetBandLimitMethod() メンバー関数で指定した手法で帯域制限する.

6.3.7 サービスとして補助的に用いるメンバー関数

```
BoundingBox GetDiffractionRect(const PointArray& pa, double z)
```

戻り値 BoundingBox 型の外接矩形

説　明 z の位置にあるホログラムと平行な平面上での PointArray 型のポリゴン pa の回折範囲を求める.

Note

- BoundingBox 型の戻り値の z 値は回折範囲を求めた z 値になる．また外接矩形は奥行き方向の厚み 0 の 2 次元外接矩形となる.
- この関数で pa が psl::Polygon 型でない理由は，pa に含まれる点が同一平面上にある必要がないためである．引数 pa として psl::Polygon 型のオブジェクトを用いても問題なく実行できる.

```
int GetNumOfVisible(const IndexedFaceSet& model)
```

戻り値　計算対象のポリゴンの個数

説　明　IndexedFaceSet クラスのモデル model で，現在の設定で可視であり計算対象となるポリゴンの個数を取得する．

```
void BackFaceCulling(IndexedFaceSet& model)
```

戻り値　無し

説　明　IndexedFaceSet クラスのモデル model から，現在の設定で不可視の背面ポリゴンを削除する．

6.3.8　主に内部的に用いるメンバー関数

この節のメンバー関数は，AddPolygonField() メンバー関数や AddObjectField() メンバー関数を用いて計算する場合には，とくに使用する必要はない関数である．特殊なレンダリングをする場合のみに用いる．

```
bool IsVisible(void)
```

戻り値　処理対象のポリゴンが可視であれば true

説　明　SetCurrentPolygon() メンバー関数で設定・保持されているポリゴンが，種々の設定条件下で可視であり，処理対象であるかどうかを判定する，処理対象である場合には true を返す．

Note

- この関数呼び出し前に SetCurrentPolygon() メンバー関数と SetFrameZ() メンバー関数が呼び出されていない場合はエラーになる．
- 可視判定は，SetDiffractionRate() メンバー関数や SetCullingRate() メンバー関数，SetCircularRefPoint() メンバー関数等の設定によって変化する．
- この関数はポリゴン光波生成処理の起点となる．この関数を呼び出した後でなければ呼び出せない関数がいくつかある．

```
void SetupTfb(WaveField& tfb)
```

戻り値　なし

説　明　WaveField クラスの傾斜フレームバッファ tfb にサンプリング間隔，サンプリング数等の適切なパラメータを設定し，傾斜フレームバッファにポリゴン形状等を描画する準備を行う．

Note

- この関数呼び出し前に IsVisible() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．
- 傾斜フレームバッファのサンプリング間隔やサンプリング数，波長等のパラメータは対象となる SurfaceBuilder オブジェクトとポリゴンデータから決定される．

```
void SetupPfb(WaveField& pfb) const
```

戻り値 なし

説　明 WaveField クラスの平行フレームバッファ pfb にサンプリング間隔，サンプリング数等の適切なパラメータを設定する．

Note

- この関数呼び出し前に IsVisible() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．
- この関数は SetFrameZ() メンバー関数で設定された基準平面まで平行フレームバッファ内の光波を並進伝搬した際に，光波が回折する領域が平行フレームバッファのサンプリング範囲を超えないように pfb のサンプリング数を設定する．
- 平行フレームバッファのサンプリング間隔には対象となる SurfaceBuilder オブジェクトが保持している値が設定される．

```
void PaintPolygonShape(WaveField& tfb, double amp)
```

戻り値 なし

説　明 WaveField クラスの傾斜フレームバッファ tfb に，振幅値 amp のポリゴンを描く．

Note

- この関数呼び出し前に SetupTfb() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．

```
void RotateFs(WaveField& pfb, const WaveField& tfb, SideOption side) const
```

戻り値 なし

説　明 WaveField クラスの傾斜フレームバッファ tfb を回転変換して平行フレームバッファ pfb にその光波分布を求める．この時，SideOption 列挙型 side が ONESIDE であればポリゴンの正の面からの光波のみが得られるように回転変換し，BOTHSIDES であれば両面からの光波が得られるように回転変換する．

Note

- Rel 1.9 以降では，pfb のウィンドウ領域内のみでポリゴン光波が**加算**される．
- Rel 1.9 以前では，pfb の全領域にポリゴン光波が**代入**される．
- この関数呼び出し前に IsVisible() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．
- IsVisible() メンバー関数で決定される．回転中心の座標は GetRotationCenter() メンバー関数で取得できる．

Example

```
Point pp(1.0, 2.0, 3.0);
SurfaceBuilder pa(pp);
pa.At(0) *= 3.0;    // 先頭の要素に3を乗算する
```

```
void SetFrameZ(double fz)
double GetFrameZ(void)
```

戻り値　なし / 基準平面の z 座標

説　明　平行フレームバッファのサンプリング数を決定する際等で必要となる基準平面の z 座標を設定/取得する．

Note

- 既定値はない．必ず値を設定しなければならない．

```
const Point GetRotationCenter(void)
```

戻り値　回転中心の座標

説　明　回転中心の座標を取得する．

Note

- この関数呼び出し前に IsVisible() メンバー関数が呼び出されていなければならない．呼び出されていない場合はエラーになる．

```
void SetBalanceRotation(bool onoff)
```

戻り値　なし

説　明　このフラグがオン (true) の場合，回転変換の中心をポリゴンの重心にとる．オフ (false) の場合は，PFB の中心をポリゴン平面に写像した点を回転変換の中心とする．初期値オン．

Note

- オフの場合，回転変換の中心とポリゴンの位置が大きくずれる場合がある．
- オンの場合，PFB が大きくなる場合がある．
- 回転中心の決定は，IsVisible() メンバー関数の呼び出しで行われる．このフラグはそれに影響を与える．

```
void SetRemappingShift(SFrequency sh)
SFrequency GetRemappingShift(void)
```

戻り値　なし / スペクトル再マッピングのシフト量

説　明　スペクトル再マッピング (回転変換) における SFrequency 型のシフト量を設定/取得する

```
const PointArray& GetCulledReferencePoint(void) const
```

戻り値 PointArray 型の const 参照
説　明 背面カリングされた回転後の参照点を取得する．

Note

- この関数を呼び出す前に IsVisible() メンバー関数を呼び出す必要がある．

```
double GetTfbDensityRatio(void)
```

戻り値 TFB の密度
説　明 TFB の密度を取得する．

6.4 TfbPaint クラスとそれを継承するクラス

シェーディングやテクスチャマッピングには TfbPaint クラスを継承するクラスが用いられる．新しいシェーディング等のアルゴリズムを導入する際には，TfbPaint クラスを継承するクラスを定義する必要がある．

6.4.1 ベースクラス: TfbPaint クラス

```
TfbPaint(double gm)
```

戻り値 なし
説　明 コンストラクタ．補正制限値 gm を引数とする．

```
virtual void PaintTfb(WaveField& tfb, const psl::Polygon& polyL, const psl::Polygon&
polyG, const WaveField& pfb, SurfaceBuilder* sb = NULL) = 0
```

戻り値 なし
説　明 傾斜フレームバッファを描画する仮想メンバー関数．SurfaceBuilder クラスはこの関数を呼び出して傾斜フレームバッファ tfb に表面関数を設定する．呼び出し時には，引数としてローカル座標で頂点座標が設定された psl::Polygon クラスの polyL，グローバル座標で頂点座標が設定された polyG，平行フレームバッファ pfb，また呼び出した SurfaceBuilder クラスのオブジェクト自体へのポインタが与えられる．

Note

- この関数は「純粋仮想関数」であり，この関数を直接呼び出すことはできない．継承クラスでオーバーロードする目的で定義されている．
- TfbPaint クラスを継承するクラスは必ずこの関数をオーバーロードしなければならない．

```
double GetGamma(void)
void SetGamma(double gm)
```

戻り値　補正制限値/なし
説　明　補正制限値 gm を取得/設定する．

```
Vector GetDirection(void)
```

戻り値　Vector 型で表される照明光の進行方向ベクトル
説　明　設定されている照明光の進行方向ベクトルを取得する．

```
double GetEnvironment(void)
```

戻り値　環境光の比率
説　明　設定されている環境光比率を取得する．

6.4.2　シェーディング

シェーダは SurfaceBuilder クラスの SetShader() メンバー関数を用いて SurfaceBuilder クラスのインスタンスに登録することで有効になる．TfbPaint クラスを継承して定義されているシェーダの比較を表 6.1 に示す．TfbLambertShading クラスと TfbPhongSpecularShading クラスでは，モデルデータに設定されたマテリアル情報に基づいてシェーディングを行う．そのため，これらのシェーダでは，モデルにマテリアルが設定されていることが必要である．なお，TfbPhongSpecularShading は，TfbLambertShading の上位互換であり，Phong の反射モデルに基づいて，マテリアルの光沢度や鏡面反射率を反映したスペキュラーシェーディングを行う．

マテリアルが無く，テクスチャのみのモデルでシェーディングが不要な場合は，TfbFlatShading クラスを用い，仮想照明光無し (シェーディング無し) の設定にする．

表 6.1　シェーダクラスの比較

	拡散面		鏡面		
シェーダ	フラット	スムース	フラット	スムース	マテリアル
TfbFlatShading	○	×	×	×	不要
TfbGouraudShading	×	○	×	×	不要
TfbPhongShading	×	○	×	×	不要
TfbLambertShading	○	○ *1	×	×	必要
TfbPhongSpecularShading	○	○ *1	○	○ *1	必要

*1　隣接するポリゴンの法線が為す角度が「スムージング角度」以下の場合にスムースシェーディングとなる．

Note

- スペキュラーな反射光をモデルのテクスチャとしてベイクする (焼き込む) ことでも，見かけ上，スペキュラーなレンダリングができる．しかし，視点を変化することによってハイライトの位置が変化するといった「そこに

実物があるように見える」ホログラム本来の良さが失われるため，この様な方法は用いるべきではない．

```
TfbFlatShading(double gam)
TfbFlatShading(double gam, Vector dir, double env)
TfbFlatShading(const WaveField& diffuser, double gam)
TfbFlatShading(const WaveField& diffuser, double gam, Vector dir, double env)
```

戻り値 なし

説　明 拡散面のレンダリングとしてフラットシェーディングを実行する．第 1 の形式のコンストラクタを用いた場合には，補正制限値 gam を引数とし，ポリゴンカラーの輝度情報のみを利用してレンダリングを行う．そのため，照明光源によるシェーディングを行わない．第 2 の形式のコンストラクタを用いた場合には，補正制限値 gam，Vector 型の照明光方向ベクトル dir，環境光比率 env を用いてフラットシェーディングを行う．この場合，ポリゴンに設定された color は無視される．第 3 と第 4 の形式のコンストラクタを用いた場合には，拡散位相として diffuser を用いる．

Note

- 第 1 と第 2 の形式では拡散位相として乱数を用いる．
- 第 1 の形式を用い，かつポリゴンにマテリアル (Material クラス) が無い場合，輝度は規定値である 1 となり，シェーディングは一切行われない．シェーディング無しでテクスチャマッピングのみを行いたい場合には，この方法を用いる．
- このシェーダを用いてカラーモデルをレンダリングする場合，通常，テクスチャマッピングが必要である．テクスチャマッピングをしない場合，モデル全体が同じ色となり，その色は 3 波長に対して設定した輝度値の比率で決まる．
- シェーダーを SurfaceBuilder クラスのオブジェクトに登録するには SetShader() メンバー関数を用いる．
- 第 1 と第 3 の形式には隠れた (省略可能な)bool 型引数 cr があるが，2 種類のコンストラクタを識別するために用いられており意味はない．この引数を省略しないときは常に true としなければならない．

```
TfbGouraudShading(double gam, Vector dir, double env)
TfbGouraudShading(const WaveField& diff, double gam, Vector dir, double env)
```

戻り値 なし

説　明 Gouraud シェーディングを用いて拡散面のスムースシェーディングを行う．補正制限値 gam，Vector 型の照明光方向ベクトル dir，環境光比率 env を用いてシェーディングを行う．このシェーダでは，ポリゴンにマテリアルが設定されている必要は無く，設定されていても無視される．第 2 の形式では拡散位相として diff を用いる．(西)

Note

- 本クラスを用いたグーローシェーディングを行う場合，IndexedFaceSet クラスで表現される物体モデルに対して必ず事前に AutoNormalVector() メンバー関数を呼び出しておく必要がある．呼び出していない場合はエラーになる．
- このシェーダを用いてカラーモデルをレンダリングする場合，通常，テクスチャマッピングが必要である．テク

スチャマッピングをしない場合，モデル全体が同じ色となり，その色は 3 波長に対して設定した輝度値の比率で決まる．

```
TfbPhongShading(double gam, Vector dir, double env)
TfbPhongShading(const WaveField& diff, double gam, Vector dir, double env)
```

戻り値　なし

説　明　Phong 補間を用いて拡散面のスムースシェーディングを行う．補正制限値 gam，Vector 型の照明光方向ベクトル dir，環境光比率 env を用いてフォンシェーディングを行う．このシェーダでは，ポリゴンにマテリアルが設定されている必要は無く，設定されていても無視される．第 2 の形式では拡散位相として diff を用いる．(西)

Note

- 本クラスを用いたシェーディングを行う場合，IndexedFaceSet クラスで表現される物体モデルに対して必ず事前に AutoNormalVector() メンバー関数を呼び出しておく必要がある．呼び出していない場合はエラーになる．
- このシェーダを用いてカラーモデルをレンダリングする場合，通常，テクスチャマッピングが必要である．テクスチャマッピングをしない場合，モデル全体が同じ色となり，その色は 3 波長に対して設定した輝度値の比率で決まる．

```
TfbLambertShading(double gam, Vector dir, ColorMode mode)
TfbLambertShading(double gam, Vector dir, ColorMode mode, Color col)
TfbLambertShading(double gam, Vector dir, ColorMode mode, double env, Color col =
Color(1.0, 1.0, 1.0))
TfbLambertShading(const WaveField& diff, double gam, Vector dir, ColorMode mode, double
env = 0.25, Color col = Color(1.0, 1.0, 1.0))
```

戻り値　なし

説　明　ポリゴンが持つマテリアルのパラメータを用いて拡散面のランバートシェーディングを実行する．引数の ColorMode 列挙型の mode でレンダリングに用いるマテリアルの色チャンネルを指定する．第 1 の形式のコンストラクタを用いた場合には，補正制限値 gam，Vector 型の照明光方向ベクトル dir，照明光源色を白色としてランバートシェーディングを行う．環境光反射率と環境光色については，モデルの設定に従う．第 2 の形式では，照明光源色を Color 型の col で設定する．第 3 の形式では，環境光比率 env，照明光源色 col を用いてランバートシェーディングを行う．この時，env と col を乗算したカラーを環境光の計算に用いる．第 4 の形式では拡散位相として diff を用いる．(柳谷)

Note

- 使用方法については，ランバートシェーダによるフルカラーレンダリングを参照．
- 本クラスを用いたシェーディングを行う場合，IndexedFaceSet クラスで表現される物体モデルに対して必ず事前に AutoNormalVector() メンバー関数を呼び出しておく必要がある．呼び出していない場合はエラーになる．

- このシェーダでは，モデルデータが有するマテリアル情報に基づいて物体光波のレンダリングを行うため，テクスチャを持たない場合でも正しくモデルの色を再現することができる．
- ポリゴンがマテリアルを持たない場合はエラーになる．
- モデルデータのマテリアル情報は，Material クラスで表されており，Polygon クラスの GetMaterial() メンバー関数や IndexedFaceSet クラスの GetMaterial() メンバー関数等を用いて取得することができる．一方，モデルデータのマテリアル設定は，原則としてモデラ—ソフトウェアによって行う．
- ColorMode 列挙型の mode は，モデルデータのどの色チャンネルを用いてレンダリングを行うかを指定する．一方，レンダリングする物体光波の波長は SurfaceBuilder クラス等の設定によって決まる．そのため，例えば mode = RED を指定しておいて，532 nm (緑色) の光波をレンダリングすることも可能であるが，一般的には指定したチャンネルに相当する波長でレンダリングするべきである．これにより，モデルの色を再現することができる．
- このシェーダでは，モデルに設定されているスムージング角度に基づいて内部で平面と曲面の判定をしており，隣り合ったポリゴンの法線の角度がスムージング角度以下の場合，曲面としてスムースシェーディングする．逆にスムージング角度以上の場合，平面としてフラットシェーディングする．
- スムースシェーディングとなる場合は，TfbGouraudShading クラスを用いてグーローシェーディングを行うため，正しい結果を得るためには，事前に IndexedFaceSet クラスの AutoNormalVector() メンバー関数を呼び出しておく必要がある．
- モデルに設定されているスムージング角度は，IndexedFaceSet クラスの GetSmoothingAngle() メンバー関数を用いて取得することができる．なお，スムージング角度を変更したい場合は，そのモデルを作成したモデラーソフトを用いて変更しなければならない．
- 現在のバージョンでは, MQO 形式ファイル (LoadMqo() メンバー関数) と OBJ/MTL 形式ファイル (LoadObj() メンバー関数) を用いて読み込んだ場合のみマテリアルを反映したシェーディングができる．

```
TfbPhongSpecularShading(double gam, Vector dir, ColorMode mode)
TfbPhongSpecularShading(double gam, Vector dir, ColorMode mode, Color col)
TfbPhongSpecularShading(double gam, Vector dir, ColorMode mode, int n, double env =
0.25, Color col = Color(1.0, 1.0, 1.0))
```

戻り値 なし

説　明 TfbLambertShading クラスに Phong 反射モデルに基づく鏡面レンダリングを付加した上位互換シェーダ．モデル (ポリゴン) が持つマテリアルのパラメータを用いてレンダリングを実行する．引数の ColorMode 列挙型の mode でレンダリングに用いるマテリアルの色チャンネルを指定する．第 1 の形式のコンストラクタを用いた場合には，補正制限値 gam，Vector 型の照明光方向ベクトル dir，照明光源色を白色としてシェーディングを行う．これら以外は，ポリゴンのマテリアルに設定された光沢度や鏡面反射の比率に基づいてレンダリングを行う．環境光反射率と環境光色についても，モデルの設定に従う．第 2 の形式では，照明光源色を Color 型の col で設定する．第 3 の形式では，環境光比率 env，照明光源色 col を用いる．引数 n は GS アルゴリズムで位相を整形する時の反復回数であり，n = 0 とした場合は反復を行わない．(柳谷)

Note

- 拡散反射に関しては，内部的に TfbLambertShading クラスのメソッドをよびだしている．そのため，Phong 反射モデルの範囲で，鏡面/拡散面，フラット/スムースの全てのシェーディングを行う事ができる万能シェーダ

となっている．ただし，モデルにマテリアルが設定されている必要がある．

- 本クラスを用いたシェーディングを行う場合，IndexedFaceSet クラスで表現される物体モデルに対して必ず事前に AutoNormalVector() メンバー関数を呼び出しておく必要がある．呼び出していない場合はエラーになる．
- 詳細は，TfbLambertShading クラスの ***Note*** を参照．
- 使用方法は，ランバートシェーダによるフルカラーレンダリングと基本的に同じである．

```
void TfbPhongSpecularShading::SetCalcOption(unsigned int opt)
```

戻り値　なし

説　明　下記の計算オプションを設定する．

0:　高速化なし．位相の事前計算なしで FFT2 回 (デフォルト)

1:　事前計算した位相を用いて FFT2 回

2:　位相の事前計算なしで FFT1 回

6.4.3　テクスチャマッピング

テクスチャマッピングは，SurfaceBuilder クラスの SetTexture() メンバー関数を用いてテクスチャマッパーを SurfaceBuilder クラスのインスタンスに登録することによって実行される．TfbPaint クラスを継承して定義したテクスチャマッピング用クラス (テクスチャマッパー) を以下に示す．

```
TfbOlthoProjectMapping(const WaveField& texture)
```

戻り値　なし

説　明　z 軸に垂直な平面上のテクスチャ画像を物体に平行投影することによりテクスチャマッピングを行う．

Note

- WaveField 型のテクスチャ画像 texture の原点は，マッピングされるオブジェクトの正しい位置に画像が投影されるように設定されていなければならない．またそのピクセルピッチは画像の物理サイズに反映するので正しく設定されていなければならない．
- テクスチャ画像は補間されて投影される

```
TfbUvMapping(const std::vector<char*>& texNames double gamma = 1.0, wfl::ColorMode mode
= wfl::GRAY_SCALE)
TfbUvMapping(const std::vector<char*>& texNames, const char* dir, double gamma = 1.0,
wfl::ColorMode mode = wfl::GRAY_SCALE)
```

戻り値　なし

説　明　UV マッピングを行うためのクラス．第 1 の形式では，カレントディレクトリにある BMP 形式のテクスチャファイルのリスト texNames を用いて UV マッピングを行う．このとき，gamma でテクスチャのガンマ値

(一般に 2.2) を指定し，ColorMode 列挙型の mode で指定したカラーチャンネルをマッピングする．第 2 の形式では，ディレクトリ dir にある BMP 形式のテクスチャファイルのリスト texNames を用いて UV マッピングを行う．

Note

- 実際の UV マッピングの方法についてはサンプルソースを参照.
- 現在のバージョンでは，IndexedFaceSet クラスの LoadMqo() メンバー関数，および IndexedFaceSet クラスの LoadObj() メンバー関数を用いて読み込んだモデルで UV マッピングができる．
- MQO モデルファイルに設定されているテクスチャのリストを取得するためには，psl::GetTextureNamesMqo() 関数を用いる．
- MQO モデルファイルに設定されているテクスチャの数を取得するためには，psl::GetNumOfTextureMqo() 関数を用いる．
- OBJ/MTL モデルファイルに設定されているテクスチャの数を取得するためには，psl::GetNumOfTextureMtl() 関数を用いる．
- このオブジェクトを SurfaceBuilder クラスのオブジェクトに登録するには SetTexture() メンバー関数を用いる．
- 第 2 の形式では，texNames に記述されたディレクトリ情報を無視し，dir のディレクトリから該当テクスチャファイルを探してマッピングを行う．

```
std::vector<psl::Texture*> TfbUvMapping::GetTexes(void)
```

戻り値　Texture 型オブジェクトのポインタ配列

説　明　UV テクスチャマッピングで用いるテクスチャのリストを取得する．

Note

- このメソッドは，TfbUvMapping のインスタンスにすでに設定されているテクスチャのリストを取得する．モデルファイルに設定されているテクスチャのリストを取得するためには，psl::GetTextureNamesMqo() 関数を用いる．

6.4.4　アルファマッピング

アルファマッピングは，SurfaceBuilder クラスの SetAlphaMap() メンバー関数を用いてマッピングオブジェクトをアルファマッパーとして SurfaceBuilder クラスのインスタンスに登録することによって実行される．アルファマッピング専用のマッピングオブジェクトは定義されておらず，TfbUvMapping クラスのインスタンスをアルファマッパーとして SurfaceBuilder に登録して用いる．

Note

- 現在のバージョンでは，IndexedFaceSet クラスの LoadMqo() メンバー関数を用いて読み込んだモデルでのみアルファマッピングができる．
- モデルファイルに設定されているアルファマップの数とリストを取得するためには，psl::GetAlphaMapNamesMqo() 関数を用いる．

6.5　psl::Polygon クラス

PointArray クラスの派生クラス．ポリゴンの頂点及び色を保持するクラス．このクラスは PointArray クラスのメンバー関数と演算子を継承しているため，回転や平行移動ができる．このクラスに保持される点群は，PointArray クラスとは異なり，同一平面上にあることが保証される．

6.5.1　メンバー関数

```
Polygon()
Polygon(const IndexedFaceSet* mObj, unsigned int pNum)
explicit Polygon(const PointArray& points)
explicit Polygon(const IndexedFaceSet& model, int n)
```

戻り値　なし

説　明　コンストラクタ．第 1 の形式では空のポリゴンを生成する．第 2 の形式では，mObj の母体オブジェクトのインデックス pNum のポリゴンを生成する．第 3 の形式では，PointArray クラスの点群をポリゴンに変換する．第 4 の形式では，IndexedFaceSet クラスのモデル model のインデックス n の面を Polygon に変換する．

Note

- 識別子 Polygon は，Windows.h でデフォルト名前空間で用いられているため，「Polygon があいまい」のエラーを起こす場合がある．その場合は，psl::Polygon を用いる．
- 第 3 と第 4 の形式では点が全て同一平面にあるかチェックする．同一平面でなければエラーになる．

```
void Insert(const Point& p)
void Insert(int i, const Point& p)
```

戻り値　なし

説　明　第 1 の形式では末尾に Point 型の点 p を挿入する．第 2 の形式では i の直前に Point 型の点 p を挿入する．

Note

- 挿入時に同一平面かチェックする．同一平面でなければエラーになる．

```
Vector GetNormalVector(void) const
```

戻り値　Vector 型の法線ベクトル

説　明　法線ベクトルを取得する．

```
Point GetAveragePoint(void) const
```

戻り値 Point 型の平均点 (多角形の重心)
説　明 全点の平均点 (多角形の重心) を求める.

```
void SetBrightness(double b) 【Rel 2.0 より削除】
```

戻り値 なし
説　明 ポリゴンの輝度を設定する.

```
double GetBrightness(void) const
```

戻り値 輝度
説　明 ポリゴンが有するのマテリアルカラーから輝度を取得する．マテリアルが設定されていない場合，1.0 を返す.

```
void SetColor(const Color& c) 【Rel 2.0 より削除】
const Color& GetColor(void) const 【Rel 2.0 より削除】
```

戻り値 なし/Color 型の色
説　明 ポリゴンの Color 型の色を設定/取得する.

```
Color& Color(void) 【Rel 2.0 より削除】
```

戻り値 Color 型オブジェクトの参照
説　明 Color 型オブジェクトの参照を取得する．左辺値として設定可能.

```
void SetTexMap(int n, const TexMap uv) 【Rel 2.0 より削除】
```

戻り値 なし
説　明 頂点 n に TexMap 型のテクスチャ UV 座標 uv を設定する.

```
TexMap GetTexMap(int n) const
```

戻り値 TexMap 型の UV 座標
説　明 頂点 n のテクスチャ UV 座標を取得する.

Note

- UV 座標が未定義の頂点についてこの関数を呼び出してもエラーにはならないが，取得された TexMap 型座標では $U = V = -1$ の未定義状態となっている．

`void SetTexNum(int num)` **【Rel 2.0 より削除】**

戻り値　なし

説　明　このポリゴンに，テクスチャ番号 num を設定する．

Note

- 複数テクスチャを用いたテクスチャマッピングに用いる．

`int GetTexNum(void) const`

戻り値　テクスチャ番号

説　明　このポリゴンのテクスチャ番号を取得する．

Note

- 複数テクスチャを用いたテクスチャマッピングに用いる．

`int GetAlphaMapNum(void) const`

戻り値　アルファマップ番号

説　明　このポリゴンのアルファマップ番号を取得する．

Note

- 複数アルファマップを用いたアルファマッピングに用いる．

`double GetArea(void) const`

戻り値　ポリゴンの面積 (単位は頂点座標の単位と同じ)

説　明　三角形ポリゴンの面積を計算する．

Note

- 現在のバージョンでは三角形以外のポリゴンにこの関数を呼び出すとエラーになる．

`void ReverseVertexOrder(void)`

戻り値　なし

説　明　頂点の格納順序を逆転する.

```
bool HasMotherObject(void) const
```

戻り値　母体オブジェクトがあれば true
説　明　母体オブジェクトへのポインタが存在するか判定する.

```
bool HasTexture(void) const
```

戻り値　テクスチャがあれば true
説　明　ポリゴンが有するマテリアルにテクスチャが存在するか判定する.

```
int GetMaterialNum(void) const
```

戻り値　マテリアル番号
説　明　ポリゴンの有するマテリアル番号を取得する.

Note

- 複数のマテリアルを用いたシェーディングに用いる.

```
bool HasMaterial(void) const
```

戻り値　マテリアルがあれば true
説　明　ポリゴンがマテリアルを有するか判定する.

```
psl::Material* GetMaterial(void) const
```

戻り値　Material 型のマテリアルのポインタ
説　明　ポリゴン有する有するマテリアルのポインタを取得する.

```
psl::SurfaceType GetSurfaceType(void) const
```

戻り値　SurfaceType 列挙型の SurfaceType
説　明　そのポリゴンの SurfaceType を取得する.

```
double GetSmoothingAngle(void) const
```

戻り値　スムージング角度．単位：度
説　明　ポリゴンのスムージング角度を取得する．

```
std::ostream& operator<< (std::ostream& os, const Polygon& polygon)
```

戻り値　ストリームへの参照
説　明　内容をストリームへ出力

6.6　Color クラス

6.6.1　データメンバー

Color クラスには以下のデータメンバーが protected アクセスとして定義されている．

red　：赤色の割合．
Green：緑色の割合．
Blue　：青色の割合．
Alpha：アルファ値．

これらの値は原則として 0.0〜1.0 の範囲にあるものとして扱われる．アルファ値は 0.0：透明 〜1.0：不透明であり，この値が負の場合そのカラーが無効 (未定義) であることを示す (柳谷)．

6.6.2　メンバー関数

```
Color(void)
Color(double red, double green, double blue, double alpha = 1.0)
```

戻り値　なし
説　明　コンストラクタ．第 1 の形式では alpha が −1.0 である未定義のカラーを生成する．第 2 の形式ではデータメンバーの値をそれぞれ指定する．

```
static Color White(void)
```

戻り値　なし
説　明　白色を返す静的関数．

```
static Color WHITE(double g = 1.0)
static Color RED(double g = 1.0)
static Color GREEN(double g = 1.0)
static Color BLUE(double g = 1.0)
static Color YELLOW(double g = 1.0)
static Color CYAN(double g = 1.0)
static Color MAGENTA(double g = 1.0)
```

戻り値 なし
説　明 輝度 g の各色を返す静的関数.

```
void Set(double R, double G, double B, double A = 1.0)
```

戻り値 なし
説　明 RGBA カラーを設定する.

```
void SetGray(double g)
```

戻り値 なし
説　明 輝度 g の灰色を設定する．アルファ値は 1.0 に設定される.

```
void SetGrayLevel(double g)
```

戻り値 なし
説　明 色相を変えずにグレイ値を g に設定する．アルファ値は 1.0 に設定される.

```
double Red(void) const
double Green(void) const
double Blue(void) const
```

戻り値 輝度
説　明 各色の輝度を取得する.

```
double Alpha(void) const
```

戻り値 アルファ値
説　明 アルファ値を取得する.

```
double GrayLevel(void) const
```

戻り値　輝度
説　明　RGB を加重平均した輝度を取得する.

```
bool IsValid(void)
```

戻り値　有効なカラーであれば true
説　明　カラーが有効であるか無効であるかの判定を行う.

```
void Disable(void)
```

戻り値　なし
説　明　カラーを無効化 (未定義化) する.

```
void Enable(void)
```

戻り値　なし
説　明　カラーが無効であればカラーを有効化する．RGBA 値は現在設定されているものが残る．カラーが有効でなければなにもしない.

```
void SetAlpha(double A)
```

戻り値　なし
説　明　A をアルファ値として設定する.

```
double GetBrightness(void) const
```

戻り値　輝度
説　明　RGB を加重平均した輝度を取得する.

```
void SetBrightness(double brt)
```

戻り値　なし
説　明　RGB の比率を変えずに，全体の輝度のみを brt に設定する.

```
unsigned char GetRed8bit(void)
unsigned char GetGreen8bit(void)
unsigned char GetBlue8bit(void)
```

戻り値 明度

説　明 8 ビットカラーとして各色の明度を取得する.

Note

- 明度 0.0〜1.0 を 8 ビット明度 0〜255 に変換して取得する.
- 負の明度は 0，1.0 を超える明度は 255 になる.

6.6.3 オーバーロード演算子

```
Color& operator*=(const Color& col)
Color& operator*=(double val)
```

戻り値 左辺値の参照

説　明 乗算代入演算子

```
Color operator*(const Color& col) const
Color operator*(double val) const
```

戻り値 演算結果

説　明 乗算 2 項演算子

6.7 TexMap クラス

ポリゴンの頂点の UV 座標を表現するクラス.

6.7.1 フィールドとメンバー関数

```
TexMap()
TexMap(double u, double v)
```

戻り値 なし

説　明 コンストラクタ．第 1 の形式では $U = V = -1$ である空の UV 座標を生成する．第 2 の形式では，u が U 座標，v が V 座標となる.

Note

- 負値の座標は未定義を表す.

```
double U
double V
```

戻り値

説　明　フィールド．それぞれ U 座標と V 座標を表す.

6.8　Texture クラス

テクスチャマッピングを行う際にテクスチャ画像を保持するクラス．WaveField クラスを継承しており，そのメンバー関数を使うことができる．WaveField クラスでは画像を読み込んだ際に元のピクセル数情報が失われるのに対して，Texture クラスでは原画像のピクセル数が保持されている点が WaveField クラスと異なっている.

6.8.1　フィールドとメンバー関数

```
Texture()
```

戻り値　なし

説　明　コンストラクタ．空のインスタンスを生成する.

```
Texture& LoadBmp(const char* fname, Mode mode, Complex backg = Complex(0, 0), double
gamma = 1.0, ColorMode cm = GRAY_SCALE)
Texture& LoadBmp(const char* fname, double gamma = 1.0, ColorMode cm = GRAY_SCALE)
```

戻り値　オブジェクト自体への参照

説　明　ファイル名 name の BMP 形式ファイルをロードする．第 1 の形式ではロードするときのモードを Mode 列挙型の mode 引数で指定しなければならない．光強度・位相・振幅・実部・虚部などへの変換が可能．backg は 2 のべき乗サイズでない画像を読み込んだ際，周囲に埋め込まれるサンプル点の複素数値であり，gamma は読み込む画像のガンマ値である．これは通常 2.2 とする．ColorMode 列挙型の cm として GRAY_SCALE を指定すると，カラー画像を 8 ビットグレイスケール画像に変換した上で読み込む．それ以外の RED，GREEN，BLUE 等 を指定するとその 8 ビットカラープレーンを読み込む．第 2 の形式では，モードは INTENSITY で埋め込み値は 0 となり，gamma でガンマ値を指定して画像を読み込む.

Note

- コードサンプルは，WaveField クラスの LoadBmp() メンバー関数を参照.
- WaveField クラスの LoadBmp() メンバー関数とは異なり，この関数では読み込んだ BMP 画像ファイルのピクセルピッチがオブジェクトのサンプリング間隔として設定される.

```
int GetTexNx(void) const
int GetTexNy(void) const
```

戻り値 横方向のピクセル数 / 縦方向のピクセル数
説　明 それぞれ，読み込んだ画像の横方向と縦方向のピクセル数を返す．

Note

- WaveField クラスとしての x 方向，y 方向サンプリング数は GetNx() メンバー関数，GetNy() メンバー関数で取得できる．

```
void SetTexNx(int Nx)
void SetTexNy(int Ny)
```

戻り値 なし
説　明 読み込んだ画像の横方向と縦方向のピクセル数設定を変えたい場合に用いる．

Note

- 単に GetTexNx() メンバー関数と GetTexNx() メンバー関数で取得されるピクセル数が変化するだけで，画像自体は変化しない．

```
double GetWidth(void) const
double GetHeight(void) const
```

戻り値 メートル単位のサイズ
説　明 テクスチャ画像の横方向と縦方向の物理的なサイズを取得する．

```
void SetWidth(double s)
void SetHeight(double s)
```

戻り値 なし
説　明 テクスチャ画像の横方向と縦方向のメートル単位物理サイズ s を設定する．

Note

- これらの関数ではピクセルピッチを変更してサイズを変えている．

```
double GetAspectRatio(void) const
```

戻り値 アスペクト比

説　明　テクスチャ画像のアスペクト比を取得する．

```
void SetAspectRatio(double a)
```

戻り値　なし

説　明　テクスチャ画像のアスペクト比を a に設定する．

Note

- この関数では y 方向のピクセルピッチを変更してアスペクト比を変えている．

6.9　BoundingBox クラス

BoundingBox クラスは，3 次元あるいは 2 次元の物体や点集合に外接する直方体あるいは長方形の対角点を保持するクラスであり，PointArray クラスの派生クラスである．対角点の一方 (座標値の小さいほう) はインデックス 0 の点として保持され，もう一方はインデックス 1 の点として保持されている．

```
BoundingBox(void)
BoundingBox(const PointArray& pa)
```

戻り値　なし

説　明　コンストラクタ．第 1 の形式では空のインスタンスを生成する．第 2 の形式では PointArray 型の pa を BoundingBox クラスに変換する．

```
BoundingBox GetCommonRegion2D(const BoundingBox& bb)
```

戻り値　bb との共通領域．z 値は常にゼロ．

説　明　z 値を無視し，(x, y) の 2 次元領域で bb との共通領域を取得する．

```
bool IsEmpty2D(void)
bool IsEmpty2D(double criterion)
bool IsEmpty2D(double cx, double cy)
```

戻り値　z を無視した 2 次元領域で空の外接矩形なら true.

説　明　(x, y) のみの 2 次元領域で空の外接矩形かどうか検査する．第 2 の形式では，x，y 方向共に重なりが criterion 以下の場合に空と判定する．第 3 の形式では，x 方向には cx，y 方向には cy 以下の重なりの場合に空と判定する．

```
bool IsOverlap2D(const BoundingBox& bb)
bool IsOverlap2D(const BoundingBox& bb, double criterion)
bool IsOverlap2D(const BoundingBox& bb, double cx, double cy)
```

戻り値 z を無視した 2 次元領域で重なりがあれば true.

説　明 (x, y) のみで bb との重なりを判定する．第 2 の形式では，x または y 方向に criterion 以上の重なりがあれば true と判定する．第 3 の形式では，x 方向に cx 以上または y 方向に cy 以上の重なりがあるの場合に true と判定する．

```
PointArray GetFrame2D(int n)
```

戻り値 z を無視した 2 次元領域を囲む点の集合

説　明 z を無視した (x, y) の長方形領域の各辺上 (フレーム上) に 2^n 個の点の集合を取得する．従って，点の数は 4×2^n となる．

```
void Expand(double mx, double my, double mz)
```

戻り値 無し

説　明 中心位置を変えずに，それぞれの方向に外接矩形の大きさを拡大/縮小する．

6.10 ImagingViewer クラス

6.10.1 概要

ImagingViewer クラスは，複素振幅分布で表された光波に注視点を設定し，視点に置かれた瞳を通過する光を結像再生するためのクラスである．このクラスは WaveField クラスから派生している．注視点と視点の位置から，自動的にピントが合う．既定では，結像距離固定モードである．この場合，注視点と視点を設定すると，レンズと像面の距離 (結像距離) を一定としてレンズの焦点距離を変化してピントを合わせる．焦点距離固定モードに切り替えると，レンズの焦点距離を一定として，結像距離を変化してピントを合わせる．

6.10.2 メンバー関数

```
ImagingViewer()
```

戻り値 なし

説　明 デフォルト設定で結像を行うコンストラクタ．

Note

- デフォルト設定とその取得/設定方法は次のとおりである．
  モード　結像距離固定モード

焦点距離固定モードへの切り替え: SetFocalLength() メンバー関数

結像距離固定モードへの切り替え: SetImagingDistance() メンバー関数

視点位置　$(0, 0, +20\ \mathrm{mm})$

設定/取得方法: WaveField クラスの SetOrigin() メンバー関数 / GetOrigin() メンバー関数

像面サンプリング数　4096×4096 pixel

設定/取得: WaveField クラスの SetNx() メンバー関数, SetNy() メンバー関数/ GetNx() メンバー関数, GetNy() メンバー関数

設定後には WaveField クラスの Init() メンバー関数による初期化が必要.

像面サンプリング間隔　$1\ \mu\mathrm{m} \times 1\ \mu\mathrm{m}$

設定/取得: WaveField クラスの SetPx() メンバー関数, SetPy() メンバー関数/ GetPx() メンバー関数, GetPy() メンバー関数

結像距離　20 mm

設定/取得: SetImagingDistance() メンバー関数 / GetImagingDistance() メンバー関数

瞳の直径　4 mm

設定/取得: SetPupilDiameter() メンバー関数 / GetPupilDiameter() メンバー関数

```
void SetImagingDistance(double d)
```

戻り値　なし

説　明　結像距離固定モードに切り替え，結像距離 (結像レンズと像面の距離) を d に設定する.

```
double GetImagingDistance(Point p) const
```

戻り値　結像距離 (結像レンズと像面の距離) [単位:m]

説　明　Point 型の座標の p 点を注視点とした時の結像距離を取得する.

Note

- 結像距離固定モードでは設定済みの結像距離を返す.
- 焦点距離固定モードでは設定された焦点距離から必要な結像距離を計算して返す.

```
void SetPupilDiameter(double v)
double GetPupilDiameter(void) const
```

戻り値　なし/瞳の直径 [単位:m]

説　明　瞳の直径を設定/取得する.

```
void View(const WaveField& wf, Point p)
void View(SegWaveField& wf, Point p)
```

戻り値 なし

説　明 Point 型の p を注視点として，光波 wf を結像再生する．第 1 の形式では wf は WaveField 型の光波として与えられる．第 2 の形式では wf は SegWaveField 型の光波として与えられる．

Note

- 視点は，この ImagingViewer オブジェクトの中心となる．従って，視点を変更するためにはこのオブジェクトに対して WaveField クラスの SetCenter() メンバー関数を用いる．
- 正確には，視点位置は瞳の中心となっている．
- 結像結果は，このメソッド実行後にこのオブジェクトに対して WaveField クラスの SaveAsGrayBmp() メンバー関数等を用いて保存することができる．
- このメソッド実行後は，このオブジェクトの波長はフィールド wf の波長で置き換えられる．

```
void SpectralView(const WaveField& wf, Point p, ColorImage& image)
```

戻り値 なし

説　明 Point 型の p を注視点として，光波 wf を結像再生し，その強度分布を ColorImage 型の image にカラー画像として付け加える．

Note

- 一つの image に対して，異なった波長の光波を付け加え，ColorImage クラスの SaveAsBmpSRGB() メンバー関数でセーブすることにより，付け加えた波長に対応したフルカラー画像を得ることができる．
- ColorImage クラスの image の画素数は 2 の累乗に設定しておいてこの関数を呼び出すことが望ましい．この関数では結像再生像が正立像に変換され，image の画素数に一致するように平均化処理で縮小される．
- 視点等の詳細は，View() メンバー関数の Note を参照．

```
void MultiSpectralView(const vector<WaveField*>& wf, Point p, ColorImage& image)
void MultiSpectralView(const char* fname, Point p, ColorImage& image)
void MultiSpectralView(const vector<string> list, Point p, ColorImage& image);
```

戻り値 なし

説　明 いずれの形式でも，Point 型の注視点を p として，複数の異なった光波を結像再生して得られるカラー画像を ColorImage 型の image に合成する．

第 1 の形式では vector 型配列の wf として WaveField 型の複数の光波を与える．第 2 の形式ではファイル名 fname のマルチパート形式 wf ファイルとして，複数の光波を与える．第 3 の形式では vector 型ファイル名配列 list として複数の光波を与える．

Note

- image に保存されるカラー画像は ColorImage クラスの SaveAsBmpSRGB() メンバー関数でセーブすることに

より，フルカラー画像として確認することができる．

- ColorImage クラスの image の画素数は 2 の累乗に設定しておいてこの関数を呼び出すことが望ましい．この関数では結像再生像が正立像に変換され，image の画素数に一致するように平均化処理で縮小される．
- マルチパート形式 wf ファイルの生成については，WaveField クラスの SaveAsMpWf() メンバー関数を参照．
- 視点等の詳細は，View() メンバー関数の Note を参照．

```
void ViewWf(const char* fname, Point p)
```

戻り値　なし

説　明　Point 型の注視点を p として，ファイル名 fname の分割 WF 形式ファイルで与えられた光波を結像再生する．

Note

- 分割 WF 形式光波ファイルの書き出し・読み込みについては，WaveField クラスの SaveAsSegWf() メンバー関数，LoadSegWf() メンバー関数，LoadParamSegWf() メンバー関数等を参照．
- 視点等の詳細は，View() メンバー関数の Note を参照．

```
void ViewHologramSw(SegWaveField& holo, Point p, Point rp)
void ViewHologramSw(SegWaveField& holo, Point p, Point rp, double lambda)
```

戻り値　なし

説　明　SegWaveField 型のホログラム holo において Point 型の p 点を注視したときの結像再生像を得る．この時，ホログラムの照明光として Point 型の中心 rp 点の球面波を用いる．第 1 の形式では再生照明光の波長としてホログラム wf に設定された値を用いるが，第 2 の形式では lambda で再生照明光の波長を与える．

Note

- ホログラムが振幅型である場合は，位相分布はゼロまたは定数でなければならない．位相型の場合は，振幅が定数でなければならない．
- 視点等の詳細は，View() メンバー関数の Note を参照．

```
void ViewHologramPw(SegWaveField& holo, Point p, Vector kv)
```

戻り値　なし

説　明　SegWaveField 型のホログラム holo において Point 型の p 点を注視したときの結像再生像を得る．この時，ホログラムの照明光として Vector 型の波動ベクトル kv の平面波を用いる．再生照明光の波長は kv で与えられる．

Note

- ホログラムが振幅型である場合は，位相分布はゼロまたは定数でなければならない．位相型の場合は，振幅が定数でなければならない．

- 視点等の詳細は，View() メンバー関数の Note を参照.

```
void SaveAsReducedBmp(const char* fname, int nx = 1024, int ny = 1024) const
```

戻り値 なし

説　明 得られた再生像をサイズ nx × ny に平均化縮小した上，180 度画像回転し，振幅像を BMP ファイルとしてセーブする.

```
void SaveAsReducedWf(const char* fname, int nx = 1024, int ny = 1024) const
```

戻り値 なし

説　明 得られた再生像をサイズ nx × ny に平均化縮小し，180 度画像回転し，振幅像を WF ファイルとしてセーブする.

Note

- WF 形式ファイルとして保存されるが，位相データは削除されている．WaveFront 等により正規化レベルを調整して観察したい場合に用いる.

```
void Imaging(Point p)
```

戻り値 なし

説　明 このオブジェクトに事前に設定された光波を瞳への入射光として Point 型の p 点を注視するよう結像する.

```
void SetHumanEyeParam(void)
```

戻り値 なし

説　明 人間の眼に近いパラメータを設定する.

Note

- 設定パラメータは次のとおりである.

  像面サンプリング数　16384 × 16384 pixel

  像面サンプリング間隔　0.5 μm × 0.5 μm

  結像距離　24 mm

  瞳の直径　6 mm
- この関数はパラメータ設定後に自動的に WaveField クラスの Init() メンバー関数を呼び出して初期化を行う.
- この関数で設定されるパラメータのうち，像面サンプリング数やサンプリング間隔は本来の人の値ではない.

```
void SetFocalLength(double f)
```

戻り値　なし
説　明　焦点距離固定モードに切り替え，焦点距離を f に設定する．

Note

- 結像距離固定モードに切り替えるためには，SetImagingDistance() メンバー関数を用いる．

```
double GetFocalLength(Point p) const
```

戻り値　結像レンズの焦点距離
説　明　Point 型の座標の p 点を注視点とする時に用いられる結像レンズの焦点距離を取得する．

Note

- 焦点距離固定モードでは設定済みの焦点距離を返す．
- 結像距離固定モードでは設定されている結像距離から必要な焦点距離を計算して返す．

```
bool IsFixedFocalLength(void) const
```

戻り値　焦点距離固定モード：true，結像距離固定モード：false
説　明　焦点距離固定モードなら true を返す．

6.11　SegWaveField クラス

SegWaveField クラスはメモリに全体をロードできない大きな光波をセグメント分割して効率的に扱うためのクラスである．

6.11.1　セグメント分割光波 (Segmented wave-field) の構造と概念

図 6.2 にセグメント分割光波の構造と概念と概念を示す．光波のセグメント分割では，巨大な単一の光波を同サイズの矩形の**セグメント**に分割し，一つのセグメントだけをメモリ上にロードして各種の処理を行い，他のセグメントは 2 次記憶装置上のファイルとして保存する．セグメント分割光波ではメモリ上にロードされたセグメントを**カレントセグメント**と呼び，セグメント全体を収めたファイルを**セグメントファイル**と呼ぶ．

SegWaveField クラスは，メモリ上にある WaveField クラスのオブジェクトであるカレントセグメントとセグメントファイルを管理するためのクラスである．SegWaveField クラスそれ自体は WaveField クラスを継承しており，カレントセグメントそのものであると考えてよい．カレントセグメントは WaveField クラスのオブジェクトであるので，WaveField クラスのメンバー関数を用いてカレントセグメントを操作することができる．

一方，セグメントファイルは分割 WF フォーマットのファイルである．分割 WF 形式ファイルの詳細については，WaveField クラスの SaveAsSegWf() メンバー関数を参照されたい．ただし，SegWaveField クラスは分割 WF 形式ファイルの操作の大部分を隠蔽しているため，直接分割 WF 形式ファイルを操作する必要はほとんどない．

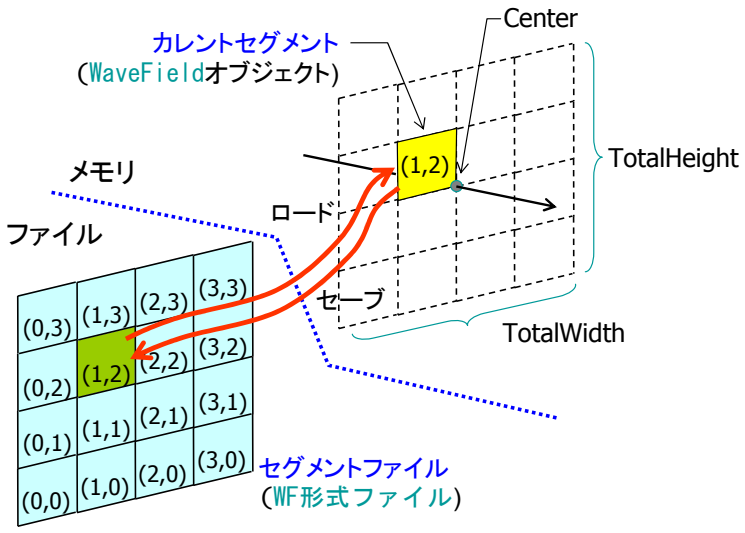


図 6.2 セグメント分割光波の構造と概念

6.11.2 コンストラクタと基本的なパラメータを設定取得するメンバー関数

```
SegWaveField(const char* fn = NULL)
SegWaveField(int mx, int my, const char* fn = NULL)
SegWaveField(int mx, int my, const WaveField& temp, const char* fn = NULL)
SegWaveField(int mx, int my, int nx, int ny, const char* fn = NULL)
SegWaveField(double width, double height, const char* fn = NULL)
SegWaveField(double width, double height, const WaveField& temp, const char* fn = NULL)
SegWaveField(const WaveField& wf, int mx, int my, const char* fn = NULL)
SegWaveField(const SegWaveField& swf, int cmx, int cmy, const char* fn = NULL)
SegWaveField(SegWaveField& swf)
```

戻り値 なし

説 明 コンストラクタ.

すべての形式に共通して fn はセグメントファイルのファイル名を表し，省略可能な引数である．ファイル名 fn を省略した場合は，一時的なファイル名が自動的に生成される．

第 1 の形式では SegWaveField クラスの SetDefault() メンバー関数で指定されるセグメント数，および WaveField クラスの SetDefault() メンバー関数で指定される波長，サンプリング間隔，サンプリング数 (一つのセグメントのサンプリング数) のインスタンスが生成される．

第 2 の形式では，セグメント数 mx×my のインスタンスが生成される．その他のパラメータは WaveField クラスの SetDefault() メンバー関数で指定される．

第 3 の形式では，セグメント数 mx×my のインスタンスが生成される．その他のパラメータは WaveField クラスの temp と同じ値が指定される．従って，総サンプリング数は (mx×temp.GetNx())×(my×temp.GetNy()) となる．

第 4 の形式では，セグメント数 mx×my，一つのセグメントのサンプリング数 nx×ny のインスタンスが生成される．その他のパラメータは WaveField クラスの SetDefault() メンバー関数で指定される．
第 5 の形式では，セグメント数以外のパラメータは WaveField クラスの SetDefault() メンバー関数で指定される．セグメント数は，これらのパラメータに基づいて，全光波の横方向の物理サイズが width 以上に，また縦方向の物理サイズが height 以上になるように自動的に設定される．
第 6 の形式では，セグメント数以外のパラメータには WaveField クラスの temp と同じ値が指定される．セグメント数は，これらのパラメータに基づいて，横方向の物理サイズが width 以上に，縦方向の物理サイズが height 以上になるように自動的に設定される．
第 7 の形式では，WaveField 型の wf をコピーしてインスタンスを生成する．ただし，セグメント数 mx×my のセグメントに分割されている．その他のパラメータは wf と同じ値が指定される．従って，総サンプリング数は wf のサンプリング数と等しくなる．
第 8 の形式では，swf の内容をコピーするが，セグメントの分割を cmx×cmy に変更する．セグメントンの分割を変えたい場合に使用する．
第 9 の形式は，コピーコンストラクタである．ただし，swf は const で指定されていない点に注意．

Note

- セグメントファイルのファイル名を指定するのは特殊な場合であり，一般には指定しない．
- もしもセグメントファイルのファイル名が指定され，かつそのファイルが存在する場合，そのファイルをそのままセグメントファイルとして使用する．したがって，例えば計算途上の分割光波を呼び出せることになる．もしも当該ファイルがセグメント wf 形式でない場合はエラーとなる．
- セグメントファイルのファイル名を指定しない場合に生成される一時ファイル名は “_SegTemp_xxxx.n.wf” の形式のファイル名である．‘xxxx’ と ‘n’ には自動的に生成される文字列と数字が入る．
- 一時ファイル名が自動的に生成されている場合は，セグメントファイルはデストラクタによって自動的に消去される．プログラムが異常終了しデストラクタが呼び出されなかった場合にはセグメントファイルはそのまま残ることになるので，手動で削除しなければならない．
- セグメントファイルのファイル名を明示的に指定した場合には，セグメントファイルはデストラクタによって消去されずにそのまま残る．
- SegWaveField クラスのインスタンスを生成した直後にはカレントセグメントは未定義であり，カレントセグメントのデータ領域も確保されていない状態である．

```
const char* GetFileName(void) const
```

戻り値　セグメントファイルのファイル名
説　明　セグメントファイルのファイル名を取得する．

```
void Dispose(void)
void Dispose(const char* fn)
```

戻り値　なし
説　明　第 1 の形式では，カレントセグメントのメモリを開放し，セグメントファイルを削除する．第 2 の形式で

は，カレントセグメントのメモリを開放し，セグメントファイルをファイル名 fn の分割 wf 形式ファイルとして保存する．

```
void Init(void)
```

戻り値 なし

説　明 カレントセグメントを初期化し，セグメントファイルを再作成する．

Note

- これが必要となるのは，SetMx() メンバー関数や SetMy() メンバー関数でセグメント数を変更した場合と，WaveField クラスの SetNx() メンバー関数や SetNy() メンバー関数でセグメントのサンプリング数を変更した場合である．
- この関数の呼び出しで，カレントセグメントは未定義状態となる．カレントセグメントのデータメモリも確保されていない．カレントセグメントは，Segment() メンバー関数の呼び出しで定義され，そのデータメモリも確保される．

Example セグメント数やセグメントサンプリング数の変更

```
SegWaveField a; //規定値で生成されるインスタンス
a.SetMx(3);          //セグメント数変更
a.SetMy(2);
a.SetNx(2048);  //セグメントのサンプリング数変更(WaveFieldから継承した関数)
a.SetNy(512);
a.Init();                //変更の有効化
```

```
static void SetDefault(int mx, int my)
```

戻り値 なし

説　明 セグメント数の既定値を mx×my に設定する．

Note

- この関数は静的関数であるので，SegWaveField::SetDefault() として呼び出す．

Example デフォルト値の変更

```
WaveField::SetDefault(1024, 512, 1.0e-6, 0.8e-6, 532e-9);
SegWaveField::SetDefault(3, 2);

SegWaveField a;
// インスタンスaは，セグメント数3×2で，各セグメントのサンプリング数1024×512，
// サンプリング間隔(1μm, 0.8μm)，波長532nmの分割光波になる
```

```
int GetMx(void) const
int GetMy(void) const
```

戻り値 セグメント数

説　明 横方向と縦方向のセグメントを取得する．

```
int GetM(void) const
```

戻り値　総セグメント数
説　明　総セグメント数を取得する．すなわち，横方向と縦方向のセグメントの積を返す．

```
void SetMx(int value)
void SetMy(int value)
```

戻り値　なし
説　明　横方向と縦方向のセグメントを設定する．

Note

- この関数でセグメント数を変更した後は，必ず Init() メンバー関数を用いて変更を有効化する必要がある．

```
double GetTotalWidth(void) const
double GetTotalHeight(void) const
```

戻り値　全光波の幅/高さ
説　明　それぞれ，全光波 (全セグメントを合わせた光波) の横方向と縦方向の物理サイズを取得する．

```
Point GetCenter(void)
```

戻り値　Point 型の座標値
説　明　全光波 (全セグメントを合わせた光波) の中心位置を取得する．

```
void SetCenter(Point c)
```

戻り値　Point 型の座標値
説　明　全光波 (全セグメントを合わせた光波) の中心位置を設定する．

```
double XX(int i)
double YY(int j)
```

戻り値　x 方向または y 方向の物理座標値
説　明　WaveField クラスのカレントセグメントにおける整数座標値 i または j から，フィールド全体の中心を原点とするローカル物理座標値 x または y を求める

Note

- カレントセグメント (WaveField クラス) における整数座標値からローカル物理座標への変換には X() メンバー関数または Y() メンバー関数を用いる.

```
int II(double x)
int JJ(double y)
```

戻り値 x 方向または y 方向の整数座標値

説　明 フィールド全体の中心を原点とするローカル物理座標値 x または y から，WaveField クラスのカレントセグメントにおける整数座標値 i または j を求める

Note

- カレントセグメント (WaveField クラス) におけるローカル物理座標から整数座標値への変換には I() メンバー関数または J() メンバー関数を用いる.

```
void SyncParam(void)
```

戻り値 なし

説　明 カレントセグメントとセグメントファイルとの間でパラメータの整合性を保つ.

Note

- WaveField クラスの SetPx() メンバー関数，SetPy() メンバー関数，SetWavelength() メンバー関数等でカレントセグメントのサンプリング間隔や波長等のパラメータを変更した場合，この関数を用いてセグメントファイル全体の整合性を保たなければならない.
- 特定のセグメントのみのサンプリング間隔や波長を変更することは意味が無く，また危険である.
- セグメント数とセグメントサンプリング数を変更する場合は Init() メンバー関数を用いる.

Example パラメータ変更の同期

```
SegWaveField a(2, 3, 1024, 512);
a.SetPx(0.8e-6);                    //パラメータ変更
a.SetWavelength(532e-9);
a.SyncParam();                      //セグメントファイルと整合性を維持
```

```
void CopyParam(const SegWaveField& swf)
```

戻り値 なし

説　明 セグメント数や各セグメントのサンプリング数を含め，swf のデータ本体以外の全てのパラメータをコピーする.

Note

- コピー先オブジェクトのセグメントファイルは一旦消去されて再生成される.
- セグメントファイルは再生成されるが，このメンバー関数実行後もセグメントファイル名は変化しない.

```
void CombineInto(WaveField& wf))
```

戻り値　なし

説　明　全セグメントを結合して一体化して WaveField 型の wf に格納する．

Note

- wf の全内容は一旦消去されて対象オブジェクトのパラメータと全データがコピーされる．
- この関数を実行するには，少なくとも対象オブジェクトの全セグメントプラス 1 セグメントを格納するメモリが必要．
- この関数実行前に対象オブジェクトのカレントセグメントが呼び出されている必要はない．カレントセグメントは未定義でもよい．

6.11.3　カレントセグメントを操作するためのメンバー関数

```
WaveField& Segment(int i, int j)
WaveField& Segment(int m)
```

戻り値　WaveField 型のカレントセグメントへの参照

説　明　第 1 の形式では，カレントセグメントを (i, j) に変更する．セグメントファイルにすでにセグメント (i, j) が存在する場合はそれをメモリに読み込む．存在しない場合はメモリ上にカレントセグメントを設定しそれをクリアする．

第 2 の形式では，カレントセグメントを (i = m%M_x, j = m/M_x) に変更する．その他の動作は第 1 の形式と同じである．

Note

- カレントセグメントのローカル原点はセグメントの位置に応じて正しく設定される．
- この関数実行以前のカレントセグメントの内容は，セグメントファイルにセーブされない．カレントセグメントをセーブするには SaveSeg() メンバー関数を用いる．
- カレントセグメントそのものは WaveField 型のオブジェクトであり，そのまま WaveField クラスのメンバー関数を用いることができる．
- 第 2 の形式は，下記のコード例のとおり，単一のループ変数による 1 重ループで全セグメントをスキャンすることを意図して用意されている．

Example 1 重ループによる全セグメントのスキャン

```
SegWaveField a(2, 3, 1024, 512);
int m;
for (m = 0; m < a.GetM(); m++)
{
    a.Segment(m);

    //-----------------
    //  カレントセグメントに対する何かの処理
    //-----------------

    a.SaveSeg();
}
```

```
void SaveSeg(void)
```

戻り値 なし

説　明 カレントセグメントをセグメントファイルにセーブする．

```
void DisposeSeg(void)
```

戻り値 なし

説　明 カレントセグメントのメモリを開放し，カレントセグメントを未定義にする．

Note

- セグメントファイルは削除されない．

```
bool Exist(int i, int j) const
bool Exist(int m) const
```

戻り値 判定結果

説　明 第 1 の形式では，セグメントファイル内にセグメント (i, j) がすでに存在するかどうか検査する．
第 2 の形式では，セグメントファイル内にセグメント (i = m%M_x, j = m/M_x) がすでに存在するかどうか検査する．

Note

- 第 2 の形式は，単一のループ変数による 1 重ループで全セグメントをスキャンする場合に使用することを意図して用意されている．

```
int GetCx(void) const
int GetCy(void) const
```

戻り値 カレントセグメントのインデックス

説　明 カレントセグメントの横方向と縦方向のインデックスを取得する．

Note

- セグメントのインデックス (i, j) は，$0 \leq i < M_x$，$0 \leq j < M_y$ である．

6.11.4 分割光波をファイルにセーブ/ロードするためのメンバー関数

以下は分割光波全体をファイルにセーブ/ロードするための関数である．カレントセグメントをセーブ/ロードするだけであれば，通常の WaveField クラスの SaveAsBmp() メンバー関数，LoadBmp() メンバー関数，SaveAsWf() メンバー関数，LoadWf() メンバー関数を用いることができる．

```
void SaveAsSegWf(const char* fname)
```

戻り値　なし

説　明　全セグメントをファイル名 fname の分割 WF 形式でセーブする．

Note

- この関数は WaveField クラスの同名関数をオーバーライドしている．
- a.WaveField::SaveAsSegWf(a.GetFileNmae(), i, j) として，WaveField クラスの SaveAsSegWf() メンバー関数と GetFileName() メンバー関数を用い，カレントセグメントのみをセグメントファイルの任意のセグメント (i, j) にセーブすることもできる．
- セグメントファイル自体も分割 WF 形式のファイルとなっている．

```
void LoadSegWf(const char* fname)
```

戻り値　なし

説　明　全セグメントの現在の内容を完全に破棄し，分割 WF 形式のファイル名 fname のファイルを読み込む．カレントセグメントは一時的に未定義状態になる．

Note

- この関数は WaveField クラスの同名関数をオーバーライドしている．
- a.WaveField::LoadSegWf(a.GetFileNmae(), i, j) として，WaveField クラスの LoadSegWf() メンバー関数と GetFileName() メンバー関数を用い，カレントセグメントの位置を変更せずにセグメントファイルの任意のセグメント (i, j) をカレントセグメントにロードすることもできる．

```
void SaveAsSegBmp(const char* fname, wfl::Mode mode, wfl::Gradation cs = wfl::GRAY)
const
void SaveAsSegBmp(const char* fname, wfl::Mode mode, BoundingBox bb, wfl::Gradation cs =
wfl::GRAY) const
```

戻り値　なし

説　明　第 1 の形式では，ファイル名 fname の分割 BMP 形式ファイルとして全セグメントをセーブする．引数 mode と cs は，WaveField クラスの SaveAsBmp() メンバー関数のそれと同じであり，振幅像/位相像などのセーブ形式，カラースケール/グレースケールを指定する．第 2 の形式では，BoundingBox 型の bb で示された領域のみを分割 BMP 形式ファイルとして全セグメントをセーブする．

Note

- 分割 BMP 形式とは，ファイル名が “name[ii][jj].bmp” の形式で表される BMP ファイルである．ii と jj はそれぞれ 2 桁の数字であり，当該 BMP ファイルのセグメント位置を表す．
- 引数 fname のファイル名と拡張子は自動的に判別される．例えば，fname = "light.bmp"であった場合，生成される分割 BMP ファイルのファイル名は “light[ii][jj].bmp” となる．

- すべてのセグメントが保存されるわけではない．光波が設定されていないセグメントは BMP ファイルとして保存されない．

```
void SaveAsSegGrayBmp(const char* fname, Mode mode, int depth = 8, double gamma = 2.2)
const
void SaveAsSegGrayBmp(const char* fname, Mode mode, BoundingBox bb, int depth = 8,
double gamma = 2.2) const
```

戻り値 なし

説　明 第 1 の形式では，ファイル名 fname のグレイスケールの分割 BMP 形式ファイルとして全セグメントをセーブする．引数 mode は，WaveField クラスの SaveAsBmp() メンバー関数のそれと同じであり，振幅像/位相像などを指定する．また，depth はピクセル深度であり，gamma は画像のガンマ値である．第 2 の形式では，BoundingBox 型の bb で示された領域のみをグレイスケールの分割 BMP 形式ファイルとしてセーブする．

Note

- このメソッドは WaveField クラスの SaveAsGrayBmp() メンバー関数の SegWaveField クラス版である．
- depth は 1, 2, 8 のいずれかの値でなければならない．

```
void SaveAsCombinedBmp(const char* fname, wfl::Mode mode, wfl::Gradation cs = wfl::GRAY)
```

戻り値 なし

説　明 すべてのセグメントを統合し，ファイル名 fname の単一の BMP 形式ファイルとしてセーブする．引数 mode と cs は，WaveField クラスの SaveAsBmp() メンバー関数のそれと同じであり，振幅像/位相像などのセーブ形式，カラースケール/グレースケールを指定する．

Note

- この関数によって生成される BMP ファイルは一般に巨大なファイルとなるため，通常のアプリケーションでは表示できないことが多い．

```
void SaveAsCombinedGrayBmp(const char* fname, Mode mode, int depth = 8, double gamma =
2.2)
void SaveAsCombinedGrayBmp(const char* fname, Mode mode, BoundingBox bb, int depth = 8,
double gamma = 2.2
```

戻り値 なし

説　明 第 1 の形式では，すべてのセグメントを統合し，ファイル名 fname の単一のグレイスケール BMP 形式ファイルとしてセーブする．引数 mode は，WaveField クラスの SaveAsBmp() メンバー関数のそれと同じであり，振幅像/位相像などを指定する．また，depth はピクセル深度であり，gamma は画像のガンマ値である．第 2 の形式では，BoundingBox 型の bb で示された領域のみをセーブする．

Note

- このメソッドは WaveField クラスの SaveAsGrayBmp() メンバー関数の SegWaveField クラス版の一つである．
- depth は 1, 2, 8 のいずれかの値でなければならない．
- この関数によって生成される BMP ファイルは一般に巨大なファイルとなるため，通常のアプリケーションでは表示できないことが多い．

```
void LoadSegBmp(const char* fname, wfl::Mode mode)
```

戻り値　なし

説　明　ファイル名 fname の分割 BMP 形式ファイルをロードする．引数 mode，WaveField クラスの LoadBmp() メンバー関数のそれと同じであり，振幅像/位相像などの形式を指定する．

Note

- この関数は，読み込むセグメント数があらかじめわかっている場合に用いる．未知のセグメント数を読み込む場合は LoadSegBmpAuto() メンバー関数を用いる．
- この関数では，読み込むファイルのセグメント数は読み込み先オブジェクトのセグメント数と同じである仮定されている．
- サンプリング間隔，波長等は事前に読み込み先オブジェクトに設定されたものが有効である．
- 引数 fname のファイル名と拡張子は自動的に判別される．例えば，fname = "light.bmp"であった場合，ロードされる分割 BMP ファイルのファイル名は “light[ii][jj].bmp” となる．
- すべてのセグメントのファイルが存在しなくても良い．欠落したセグメントがあってもエラーにはならない
- 一つのセグメントも読み込まれない場合はエラーになる．

```
void LoadSegBmpAuto(const char* fname, wfl::Mode mode)
```

戻り値　なし

説　明　ファイル名 fname の分割 BMP 形式ファイルをロードする．引数 mode，WaveField クラスの LoadBmp() メンバー関数のそれと同じであり，振幅像/位相像などの形式を指定する．

Note

- この関数は，読み込むセグメント数が未知の場合に用いる．ファイルをスキャンし，存在するファイル中で最も大きなセグメント数が全光波のセグメント数とみなされる．
- サンプリング間隔，波長等は事前に読み込み先オブジェクトに設定されている値が維持される．
- 引数 fname のファイル名と拡張子は自動的に判別される．例えば，fname = "light.bmp"であった場合，ロードされる分割 BMP ファイルのファイル名は “light[ii][jj].bmp” となる．
- すべてのセグメントのファイルが存在しなくても良い．欠落したセグメントがあってもエラーにはならない
- 一つのセグメントも読み込まれない場合はエラーになる．

```
void LoadBmp(const char* fname, Mode mode, Complex backg = Complex(0, 0), double gamma =
1.0, ColorMode cm = GRAY_SCALE)
```

戻り値 なし

説 明 ファイル名 fname の単一の BMP 形式のファイルをフィールド全体の中央に読み込む．引数 mode は WaveField クラスの LoadBmp() メンバー関数のそれと同じであり，振幅像/位相像などの形式を指定する．backg は読み込んだ画像周辺の背景の複素値，gamma は画像のガンマ値であり，ColorMode 列挙型の cm で読み込むカラープレーンを指定する．

Note

- BMP 画像のピクセル数がフィールド全体のサンプリング数より大きい場合はエラーになる．
- BMP 画像を読み込むのに必要なサンプリング数が不明な場合は，LoadBmpAuto() メンバー関数を用いる．
- このメソッドは，WaveField クラスの LoadBmp() メンバー関数の SegWaveField クラス版である．
- BMP 画像のピクセルピッチは無視され，読み込み後のフィールドのサンプリング間隔と波長等は事前に読み込み先オブジェクトに設定されている値のままである．

```
void LoadBmpAuto(const char* fname, Mode mode, Complex backg = Complex(0, 0), double
gamma = 1.0, ColorMode cm = GRAY_SCALE)
```

戻り値 なし

説 明 ファイル名 fname の単一の BMP 形式のファイルをフィールド全体の中央に読み込む．引数は LoadBmp() メンバー関数と同じである．ただし，このメソッドでは，1 セグメントあたりのサンプリング数から画像を読み込むために必要なセグメント数が自動決定され，読み込み前のセグメント数から変更される．

Note

- BMP 画像のピクセルピッチは無視され，読み込み後のフィールドのサンプリング間隔と波長等は事前に読み込み先オブジェクトに設定されている値のままである．

```
void SaveAsCombinedWf(const char* fname)
```

戻り値 なし

説 明 全セグメントを統合し，ファイル名 fname の一つの WF 形式ファイルとしてセーブする．

Note

- 現在の実装では，この関数の実行には統合した WaveField オブジェクトと同じサイズのメモリが必要である．従って，用途はほぼデバッグ用に限定される (この関数を利用できるメモリサイズがあるなら，そもそも分割する必要がない)．

6.11.5 分割光波の伝搬計算を行うメンバー関数

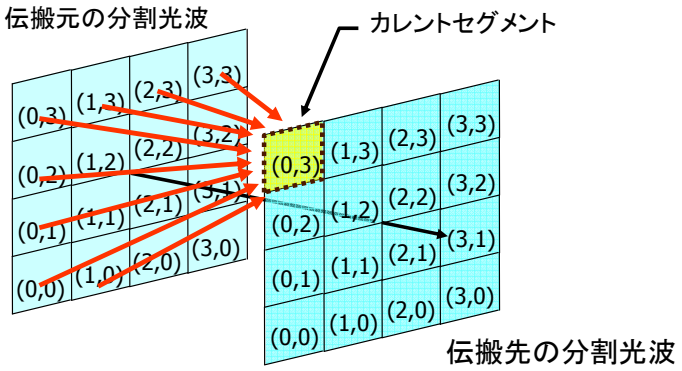


図 6.3　分割光波における伝搬計算

```
SegWaveField& ShiftedAsmProp(SegWaveField& source)
SegWaveField& ShiftedAsmProp(double dz)
```

戻り値　対象オブジェクトの参照

説　明　第 1 の形式では，図 6.3 に示すとおり，source の全セグメントの光波をシフテッド角スペクトル法でカレントセグメントへ伝搬計算する．全セグメントへ伝搬するためには，ループを用いてカレントセグメントを移動し，全セグメントをスキャンする必要がある．

第 2 の形式では, 全セグメントを z 軸方向に距離 dz の伝搬計算を行なう．関数内部で ShiftedAsmPropWhole() メンバー関数を呼び出して全セグメントをスキャンしているため，ループを用いる必要がなく，ソースが簡単になる．

Note

- この関数は内部的に WaveField クラスの ShiftedAsmPropAddEx() メンバー関数を呼び出している．
- source のセグメントサンプリング数と対象オブジェクトのセグメントサンプリング数が異なっていてもよい．
- source のサンプリング間隔や波長が対象オブジェクトのそれと異なっている場合はエラーになる．

```
SegWaveField& ShiftedAsmPropWhole(SegWaveField& source)
```

戻り値　対象オブジェクトの参照

説　明　source の全セグメントから対象オブジェクトの全セグメントへシフテッド角スペクトル法で伝搬する．内部で ShiftedAsmProp() メンバー関数を呼び出して全セグメントをスキャンしてるため，ループを用いる必要はない．

```
SegWaveField& ShiftedFresnelProp(SegWaveField& source)
SegWaveField& ShiftedFresnelProp(double dz)
```

戻り値　対象オブジェクトの参照

説　明　第 1 の形式では，図 6.3 に示すとおり，source の全セグメントの光波をシフテッドフレネル法でカレント

セグメントへ伝搬計算する．全セグメントへ伝搬するためには，ループを用いてカレントセグメントを移動し，全セグメントをスキャンする必要がある．

第 2 の形式では，全セグメントを z 軸方向に距離 dz の伝搬計算を行なう．関数内部で ShiftedFresnelPropWhole() メンバー関数を呼び出してカレントセグメントをスキャンしているため，ループを用いる必要がなく，ソースが簡単になる．

Note

- この関数は内部的に WaveField クラスの ShiftedFresnelPropAddEx() メンバー関数を呼び出している．
- source のセグメントサンプリング数と対象オブジェクトのセグメントサンプリング数が異なっていても良い．
- source のサンプリング間隔が対象オブジェクトのそれと異なっていても良い．
- source 波長が対象オブジェクトのそれと異なっている場合はエラーになる．

```
SegWaveField& ShiftedFresnelPropWhole(SegWaveField& source)
```

戻り値 対象オブジェクトの参照

説　明 source の全セグメントから対象オブジェクトの全セグメントへシフテッドフレネル法で伝搬する．内部で ShiftedFresnelProp() メンバー関数を呼び出して全セグメントをスキャンしているため，ループを用いる必要はない．

6.11.6 分割光波の演算を行うメンバー関数

```
void Add(const WaveField& wf)
void Add(SegWaveField& swf)
```

戻り値 なし

説　明 第 1 の形式では，ローカル座標を考慮して，単一のフィールド wf を対象フィールドに加算する．

第 2 の形式では，ローカル座標を考慮して，分割フィールド swf を対象フィールドに加算する．

Note

- この関数では補完は行われない．
- wf や swf のフィールの法線が $(0, 0, 1)$ でない場合や，フィールドの z 値が異なる (同一平面にない) 場合，フィールドのサンプリング間隔が異なる場合にはエラーになる．

```
void AddTo(WaveField& wf)
```

戻り値 なし

説　明 ローカル座標を考慮して，対象フィールドの全セグメントを単一のフィールド wf に加算する．

Note

- この関数では補完は行われない．

- wf や swf のフィールの法線が $(0,0,1)$ でない場合や，フィールドの z 値が異なる (同一平面にない) 場合，フィールドのサンプリング間隔が異なる場合にはエラーになる．

```
void Multiply(const WaveField& wf)
```

戻り値　なし

説　明　ローカル座標を考慮して，単一のフィールド wf を対象フィールドに乗算する．

Note

- この関数では補完は行われない．
- wf や swf のフィールの法線が $(0,0,1)$ でない場合や，フィールドの z 値が異なる (同一平面にない) 場合，フィールドのサンプリング間隔が異なる場合にはエラーになる．

```
void ResamplingCopy(const WaveField& wf, Interpol ip = wfl::CUBIC8)
```

戻り値　なし

説　明　ローカル座標を考慮して，単一のフィールド wf を対象フィールドの全セグメントに補完コピーする．補完法として Interpol 列挙型の ip を用いる．

Note

- wf や swf のフィールの法線が $(0,0,1)$ でない場合や，フィールドの z 値が異なる (同一平面にない) 場合，フィールドのサンプリング間隔が異なる場合にはエラーになる．

6.11.7　その他のメンバー関数

```
BoundingBox GetBoundingBox(int ii, int jj) const
BoundingBox GetBoundingBox(int m) const
```

戻り値　BoundingBox クラスで示されるそのセグメントの外接直方体

説　明　2 次元インデックス (ii, jj) または 1 次元インデックス m で指定されるセグメントの外接直方体を BoundingBox 型で返す．

6.12　ColorImage クラス

ColorImage クラスは，WaveField クラスのフィールドの強度像を，その波長のスペクトル画像とし，単一または複数波長のスペクトル画像を等色関数によって合成することによりカラー画像を生成するためのクラスである．ColorImage クラスの各ピクセルのカラーは CIE1931 XYZ 等色系で表現されており，容易に他の表色系に変換することができる．

ColorImage クラスでは，基本的に，そのコンストラクタでベースとなるスペクトル画像を設定する．次に，スペク

トル分布に従って，各波長の画像を AddSpectralImage() メンバー関数によって付け加えていく．すべてのスペクトル画像を付け加えたのちに，SaveAsBmpSRGB() メンバー関数によって sRGB 画像をセーブすることができる．

カラー画像の取り扱いをサポートするため，グローバル関数として，CIE1931 XYZ 表色系の等色関数 ColorMatchingCIE1931XYZ() 関数や，XYZ から sRGB に変換する ColorSRGB() 関数，XYZ カラーから xyY カラーを求める ColorXYY() 関数等も用意されている．

6.12.1 メンバー関数

```
ColorImage(int Nx, int Ny)
ColorImage(const WaveField& wf)
```

戻り値 なし

説　明 コンストラクタ．第 1 の形式ではピクセル数 Nx×Ny の画像を生成する．第 2 の形式では，WaveField 型のフィールド wf の強度分布をベース画像として画像を生成する．

Note

- wf の波長がベース画像の波長となる．
- CIE1931 等色関数を用いて CIE XYZ 画像を生成する．

```
void Clear(void)
```

戻り値 なし

説　明 現在のイメージをクリアする．

Note

- この関数は親クラスである PointArray クラスの同名の関数をオーバーライドしている．

```
int GetNx() const
int GetNy() const
```

戻り値 ピクセル数

説　明 それぞれ，水平 (x) 方向，垂直 (y) 方向のピクセル数を取得する．

```
Vector GetColorXYZ(int i, int j) const
```

戻り値 Vector 型で表された XYZ カラー

説　明 Vector 型で表された XYZ カラーとして整数座標 (i, j) のカラー値を取得する．

Note

- Vector クラスの x, y, z 要素が，XYZ カラーの (X, Y, Z) 値を表している．

```
Vector GetColorXYY(int i, int j) const
```

戻り値　Vector 型で表された xyY カラー

説　明　Vector 型で表された xyY カラーとして整数座標 (i, j) のカラー値を取得する．

Note

- Vector クラスの x, y 成分が xyY カラーの (x, y) 値を表し，z 要素が Y 値 (輝度値) を表している．

```
Color GetColorSRGB(int i, int j) const
```

戻り値　Color 型で表された sRGB カラー

説　明　Color 型で表された sRGB カラーとして，整数座標 (i, j) のカラー値を取得する．

Note

- 内部的には ColorSRGB() 関数を呼び出して XYZ ⇒ sRGB の変換を行っている．返還の詳細については ColorSRGB() 関数を参照．

```
double GetBrightness(int i, int j) const
```

戻り値　輝度値

説　明　整数座標 (i, j) のピクセルの輝度値を取得する．

Note

- XYZ 表色系の Y 値を取得している．

```
void AddSpectralImage(const WaveField& wf)
void operator+=(const WaveField& wf)
```

戻り値　なし

説　明　WaveField クラスのフィールド wf の強度像を，その波長の画像として付加する．この時，CIE1931 XYZ 等色関数 ColorMatchingCIE1931XYZ() 関数を用いてカラー変換を行っている．

Note

- ベースとなっている画像とサンプリング数 (ピクセル数) が異なる場合はエラーとなる．
- サンプリング間隔 (ピクセルピッチ) がベース画像と異なってもエラーにはならない．
- ベース画像および付加フィールドでは統一的な正規化が必要である．フィールド単位で個別に正規化された画像を付加すると正しいカラーにならない．
- 付加可能な波長範囲は ColorMatchingCIE1931XYZ() 関数を参照．

```
void SetWindow(WFL_RECT win)
void SetWindow(int left, int right, int bottom, int top)
```

戻り値 なし

説　明 第 1 の形式では，WFL_RECT 構造体 win で win.left, win.right, win.top, win.bottom として，ウィンドウの領域を設定する．RECT 構造体については，MSDN ライブラリを参照．
第 2 の形式では，直接これらを指定する．

Note

- ここでのウィンドウの考え方は，WaveField クラスのウィンドウ領域と基本的に同じものである．
- ウィンドウ領域は，以下の GetMaxY() メンバー関数，GetMaxXYZ() メンバー関数，GetMaxVecLength() メンバー関数等の計測関数，あるいは NormalizeWinY() メンバー関数，NormalizeWinXYZ() メンバー関数，NormalizeVec() メンバー関数等の正規化関数に影響する．

```
void NormalizeY(double y = 1.0)
```

戻り値 なし

説　明 XYZ カラーの Y 値の最大値が y となるように，画像全体を正規化する．デフォルトでは y = 1.0.

```
void NormalizeWinY(double y = 1.0)
```

戻り値 なし

説　明 ウィンドウ領域内で，XYZ カラーの Y 値の最大値が y となるように，画像全体を正規化する．デフォルトでは y = 1.0.

```
void NormalizeXYZ(double s = 1.0)
```

戻り値 なし

説　明 XYZ カラーの $(Y + X + Z)$ 値の最大値が s となるように，画像全体を正規化する．デフォルトでは s = 1.0.

```
void NormalizeWinXYZ(double s = 1.0)
```

戻り値 なし

説　明 ウィンドウ領域内で，XYZ カラーの $(Y + X + Z)$ 値の最大値が s となるように，画像全体を正規化する．デフォルトでは s = 1.0.

```
void NormalizeVec(double s = 1.0)
```

戻り値　なし

説　明　XYZ カラーの (X, Y, Z) ベクトルの長さの最大値が s となるように，画像全体を正規化する．デフォルトでは s = 1.0.

```
void NormalizeWinVec(double s = 1.0)
```

戻り値　なし

説　明　ウィンドウ領域内で，XYZ カラーの (X, Y, Z) ベクトルの長さの最大値が s となるように，画像全体を正規化する．デフォルトでは s = 1.0.

```
double GetMaxY(void)
```

戻り値　Y 値の最大値

説　明　ウィンドウ領域内で，XYZ カラーの Y 値の最大値を求める．

```
double GetMaxXYZ(void)
```

戻り値　$(Y + X + Z)$ 値の最大値

説　明　ウィンドウ領域内で，XYZ カラーの $(Y + X + Z)$ 値の最大値を求める．

```
double GetMaxVecLength(void)
```

戻り値　(X, Y, Z) ベクトルの長さの最大値が

説　明　ウィンドウ領域内で，XYZ カラーの Y 値の最大値を求める．

```
void SaveAsBmpSRGB(const char* fname)
```

戻り値　なし

説　明　ファイル名 fname の sRGB カラー画像として保存

Note

- 内部的には ColorSRGB() 関数を呼び出して XYZ ⇒ sRGB の変換を行っている．返還の詳細については ColorSRGB() 関数を参照.
- なお，ColorSRGB() 関数では，RGB カラー値のいずれかが 1.0 を超える場合，RGB ベクトルの長さ全体を縮

める形で 1.0 になるように変換している．これは，明度が高すぎる場合でも色味自体が変化しないようにするためである．

6.13 MaterialList クラス

このクラスは，Material クラスによって生成されたマテリアルのポインタを格納するためのクラスである．マテリアルは STL の vector<psl::Material*>として扱われる．この配列は可変長であるため，任意の位置に要素を挿入削除できる．ここでは，この配列のことをマテリアル表と呼ぶ．(柳谷)

6.13.1 メンバー関数

```
MaterialList(void)
MaterialList(const MaterialList& ml)
```

戻り値 なし

説　明 コンストラクタ．第 1 の形式では空のマテリアル表を生成する．第 2 の形式は，コピーコンストラクタである．

```
~MaterialList(void)
```

戻り値 なし

説　明 デストラクタ．MaterialList オブジェクトを明示的に削除する．対象オブジェクトで使用していたメモリは解放される．

```
void Clear(void)
```

戻り値 なし

説　明 全マテリアルへのポインタを削除し，空のマテリアル表にする．

```
unsigned int NumOfMaterial(void) const
```

戻り値 マテリアル数

説　明 マテリアル表に挿入されているマテリアル数を取得する．

```
MaterialType MaterialType(unsigned int n) const
```

戻り値 MaterialType 列挙型で与えられるマテリアルの種類

説　明　インデックス n のマテリアルの種類を取得する．

```
void InsertMaterial(const Material* matp)
```

戻り値　なし
説　明　Material クラスのポインタ matp からコピーを取得して，新しいマテリアルのエントリを挿入する．

```
const Material* operator[](insigne dint n) const
```

戻り値　Material クラスのポインタ
説　明　n 番目のマテリアルエントリへのポインタを取得する．

Note

- MaterialType() メンバー関数を用いて取得したポインタの MaterialType 列挙型を求め，それに対応するクラスにポインタをキャストして使用すること．

6.14　Material クラス

CG ソフトやモデルデータフォーマットに依存しないジェネリックなマテリアルクラス．CG ソフトによって異なった材質パラメータが必要な場合は，このクラスから派生する．(柳谷)

6.14.1　データメンバー

Material クラスには以下のデータメンバーが public アクセスとして定義されている．

ShadingModel	：	シェーディングモデル．
Color	：	基本色．Alpha が不透明度を与える．
DiffuseColor	：	拡散光色．RGB の拡散光反射率は 0.0〜1.0 の範囲あるものとして扱われる．
AmbientColor	：	環境光色．RGB の環境光反射率は 0.0〜1.0 の範囲あるものとして扱われる．
SpecularColor	：	鏡面光色．RGB の鏡面光反射率は 0.0〜1.0 の範囲あるものとして扱われる．
LuminescenceColor	：	自己発光色．RGB の自己発光率は 0.0〜1.0 の範囲あるものとして扱われる．
Shininess	：	光沢度．0 以上の値を持つ．ShadingModel が LAMBERT の場合は無視される．

この Material クラスから派生するクラスは，ShadingModel 列挙型で与えられる ShadingModel によって内容を変える場合がある．

6.14.2　メンバー関数

```
Material(psl::MaterialType mt, psl::ShadingModel sd = psl::LAMBERT)
```

戻り値　なし

説　明　コンストラクタ．MaterialType 列挙型 mt でマテリアルの種類を識別し，ShadingModel 列挙型 sd でシェーディングモデルを識別する．マテリアルの種類は，コンストラクタ以外で変更不可能である．

```
MaterialType MaterialType(void) const
```

戻り値　マテリアルの種類

説　明　マテリアルの種類を識別し，それを MaterialType 列挙型で取得する．

6.15　MqoMaterial クラス

このクラスは，メタセコイアの mqo 形式ファイルフォーマットのマテリアルを生成するクラスであり，Material クラスの派生クラスである．(柳谷)

6.15.1　データメンバー

MqoMaterial クラスには Material クラスのデータメンバーに加え，以下のデータメンバーが public アクセスとして定義されている．

Reflection	：	鏡面反射率．0.0∼1.0 の範囲にあるものとして扱われる．
VertexColorExist	：	頂点カラーの有無．
DoubleSide	：	両面描画の判定．
texNum	：	テクスチャ番号．

6.15.2　メンバー関数

```
MqoMaterial(psl::ShadingModel sd = psl::PHONG)
```

戻り値　なし

説　明　コンストラクタ．シェーディングモデルを ShadingModel 列挙型の sd で指定する．このときデータメンバーはテクスチャ番号を除いて，0 に設定される．テクスチャ番号は −1 となる．

```
MaterialType MaterialType(void) const
```

戻り値　マテリアルの種類

説　明　マテリアルの種類を識別し，それを MaterialType 列挙型で取得する．

参考文献

[1] J. W. Goodman: “Introduction to Fourier Optics, 2nd ed.”, chapter 3.10, McGraw-Hill (1996).

[2] K. Matsushima and T. Shimobaba: “Band-limited angular spectrum method for numerical simulation of free-space propagation in far and near fields”, Opt. Express, **17**, pp. 19662–19673 (2009).

[3] D. H. Bailey and P. N. Swarztrauber: “The fractional Fourier transform and applications”, SIAM Review, **33**, pp. 389–404 (1991).

[4] K. Matsushima: “Shifted angular spectrum method for off-axis numerical propagation”, Opt. Express, **18**, pp. 18453–18463 (2010).

[5] R. P. Muffoletto, J. M. Tyler and J. E. Tohline: “Shifted Fresnel diffraction for computational holography”, Opt. Express, **15**, pp. 5631–5640 (2007).

[6] K. Matsushima, H. Schimmel and F. Wyrowski: “Fast calculation method for optical diffraction on tilted planes by use of the angular spectrum of plane waves”, J. Opt. Soc. Am., **A20**, pp. 1755–1762 (2003).

[7] K. Matsushima: “Formulation of the rotational transformation of wave fields and their application to digital holography”, Appl. Opt., **47**, pp. D110–D116 (2008).

[8] T. M. Lehmann, C. Gönner and K. Spitzer: “Large-sized local interpolators”, Proceedings of IASTED Int. Conf., Computer Graphics and Imaging, pp. 156–161 (1999).

[9] K. Matsushima: “Computer-generated holograms for three-dimensional surface objects with shade and texture”, Appl. Opt., **44**, pp. 4607–4614 (2005).

[10] K. Matsushima and S. Nakahara: “Extremely high-definition full-parallax computer-generated hologram created by the polygon-based method”, Appl. Opt., **48**, pp. H54–H63 (2009).

[11] 近藤, 松島：“シルエット近似を用いた全方向視差 CGH の隠面消去”, 電子情報通信学会論文誌 D-II, **J87-D-II**, pp. 1487–1494 (2004).