Revision : 1.5 *Date* : 2012/01/0301 : 33 : 49

User's guide for `SBProfile` class

Gary Bernstein

1. `SBProfile` concepts

The `SBProfile` class is a representation of a surface brightness distribution across a 2-dimensional image plane. The `SBProfile` software allows you to create real and Fourier-domain models of a wide variety of galaxy shapes, point-spread functions (PSFs), and their convolutions. The `C++` implementation consists of a base `SBProfile` class and several realizations of these classes:

- There are the "atomic" classes that represent specific analytic profiles: `SBGaussian, SBSersic, SBAiry, SBExponential, SBBox`, and `SBMoffat`.
- `SBLaguerre` represents an intensity pattern defined by a finite Gauss-Laguerre decomposition.
- `SBInterpolatedImage` represents a pattern defined by a grid of pixel values and a chosen interpolation scheme between pixel centers.
- `SBDistort` represents any affine transformation (sheared, magnified, rotated, and/or translated) of any other `SBProfile`.
- `SBAdd` represents the sum of any number of `SBProfile`s.
- `SBConvolve` represents the convolution of any number of `SBProfile`s.

A very broad range of behaviors can be modelled with this set. All `SBProfile` objects can return exact analytic expressions for the Fourier-domain values.[1] The use of `C++` virtual functions allows us to use exact analytic formulations of the real-space behavior of the models when they are available, and to otherwise produce real-space images sampled on regular grids via FFTs from Fourier domain.

The `SBProfile` code also includes an `SBParse` function which builds an `SBProfile` object from a string argument using an easy-to-understand syntax. This permits the user to write code that allows the characteristics of galaxies and/or PSFs to be set via command-line or parameter-file entries. The wrapper program `SBDraw.cpp` produces a FITS image of an `SBProfile` that is parsed from a command-line string. *Note that the* `SBParse` *function has been removed from the*

[1] Almost: the `SBInterpolatedImage` objects require some approximation to efficiently interpolate their Fourier-domain values.

GalSim repository in favour of the Python constructors. Please see https://github.com/GalSim-developers/GalSim/issues/190.

The principal method of the `SBProfile` class is `draw()` which produces a 2d `Image` of the intensity pattern $I(x, y)$ represented by the `SBProfile` instance. The image is sampled on a grid of spacing dx, which is stored in the image header under keyword `DX`. Each `SBProfile` is aware of the sampling dx and the total extent $N\,dx$ of the image that is necessary to render the object to some (not very rigorously defined) accuracy. One can therefore call `draw()` with no arguments and let each `SBProfile` choose its own image pitch and size. Or one can specify a value of dx and/or an image size to `draw` onto, and the `SBProfile` will insure that the FFTs are done on a grid with sufficient resolution and range to avoid aliasing or wrapping, then subsample the image onto your requested grid if necessary. Hence the code should be fairly robust at producing reliable rendering of the exact analytic results, although it is still possible for a `draw` request to fail (throw an `SBError` exception) if the required FFT would be too large.[2]

1.1. Conventions

A given `SBProfile` represents some distribution $I(x, y)$ of surface brightness across the plane. The flux of an `SBProfile` is defined as

$$f = \int dx\, dy\, I(x, y). \tag{1}$$

The units of f and x, y are arbitrary—the user is responsible for maintaining a consistent choice throughout the code. Note that when you use the `SBProfile.xValue()` method, or when you `draw` an image, the values returned are the *surface brightness* values $I(x, y)$ at the specified position, *not* the flux integrated over pixels.[3] An estimate of the object flux from the values I_{ij} returned in an image would hence be

$$f = \sum_{ij} I_{ij} (\mathtt{DX})^2. \tag{2}$$

The method `SBProfile.getFlux()` will return the exact analytic flux of the object. Constructors of the atomic derived classes will, by default, assume that an object with $f = 1$ is to be represented. The `setFlux()` method rescales the $I(x, y)$ function represented by an `SBProfile` to take the specified new value.[4].

Fourier components of an `SBProfile` are defined by

$$\tilde{I}(k_x, k_y) = \int dx\, dy\, I(x, y) \exp(-i\mathbf{k} \cdot \mathbf{x}) \tag{3}$$

---

[2]As specified by the compiled-in parameter `MAXIMUM_FFT_SIZE`.

[3]Convolve your `SBProfile` with an `SBBox` to implement integration over pixels.

[4]There may be failures for objects that have zero flux.

such that $\tilde{I}(0,0) = f$. Convolutions are defined such that the Fourier transform of $A * B$ is $\tilde{A}\tilde{B}$. Note that this means PSFs should be defined with $f = 1$ to conserve flux.

2. `SBProfile` base class

The `SBProfile` base class contains pure virtual functions, so you cannot create an `SBProfile`. The drawing routines are, however, implemented in the base class. The methods of the class are:

2.1. Utilities

- `SBProfile* duplicate()` returns a pointer to a fresh copy of the object. The convention is that a duplicate should still be functional after the original is deleted.
- `bool isAxisymmetric()` returns `true` if the `SBProfile` is known to have rotational symmetry about $x = y = 0$. Many calculations can be simplified if this is true.
- `bool isAnalyticX(), isAnalyticK()` return `true` if the class can calculate values in real / Fourier space without resorting to FFT from the other domain. Note that at present all `SBProfile`s return `true` for `isAnalyticK()` and that the code depends upon this for drawing.
- `double maxK(), nyquistDx(), stepK()` give requirements on properly drawing the object. `maxK()` returns the value of $k_{\rm max}$ beyond which there is zero or negligible power in the object, *i.e.* the k value that an FFT must reach to avoid aliasing. `nyquistDx()` returns the real-space pixel size necessary for sampling without significant aliasing and defaults to $\pi/k_{\rm max}$ if not overridden. `stepK()` returns the resolution dk required in k space to render the object to desired precision. Since an FFT from a k-space image with pixel scale dk will yield a real-space rendition that assumes periodic boundary conditions at length $2\pi/dk$, this method specifies minimum acceptable real-space FFT size necessary to avoid "folding" the object.

2.2. Evaluation methods

Many of the evaluation methods take arguments as `Position<double>` classes. This template class is defined in `Bounds.h` and is simply two double-valued elements `x` and `y`. The constructor just takes the two arguments. You can for example request `xValue(Position<double>(12., -3.5))`. `SBProfile` values in Fourier space are returned as `DComplex`, which is a typedef in `Std.h` for `std::complex<double>`.

- `DComplex kValue(Position<double> p)` returns the value of the `SBProfile` transform at a specified position in k space.

- `double xValue(Position<double> p)` returns the value of the `SBProfile` at a specified position in real space. Some derived classes, *e.g.* `SBConvolve`, throw an exception for this method because real-space values are only obtainable via FFT, as indicated by a `false` return from `isAnalyticX()`.
- `Position<double> centroid()` returns the (x, y) centroid of the `SBProfile`.[5]
- `Position<double> centroid()` simply returns both coordinates of the centroid in a `Position` object.
- `double getFlux()` returns the object flux f.

2.3. Transformations

- `void setFlux(double flux)` sets a new flux value for the object. Will fail if the object had zero flux before since rescaling cannot be done.
- `void setCentroid(Position<double> p)` will reset the object centroid to the specified coordinates. Most of the atomic classes are defined to be centered at the origin, and will throw an exception for this method. The `shift()` method is the preferred means to implement a translation.
- `SBProfile* rotate(const double theta)` returns a pointer to a *new* `SBProfile` which is a version of the original rotated by angle θ.
- `SBProfile* shift(double dx, double dy)` returns a pointer to a *new* `SBProfile` which is a version of the original shifted by (dx, dy) in the real-space plane. Note that this is implemented by adding appropriate phases to the Fourier components. So it's intended only to move objects by a small fraction of their size. If you command a large shift, then when you `draw()` the object you will find it wrapped around the edges of the image.
- `SBProfile* shear(double e1, double e2)` returns a pointer to a *new* `SBProfile` which is a sheared version of the original. The shear matrix is taken to have unit determinant so that the flux is unaltered. A circular object will be transformed into an elliptical one with major/minor axes such that $(a^2 - b^2)/(a^2 + b^2) = \sqrt{e_1^2 + e_2^2}$, and position angle β such that $e_2/e_1 = \tan 2\beta$.
- `SBProfile* distort(const Ellipse e)` returns a pointer to a *new* `SBProfile` which is a translated, magnified, and/or sheared version of the orginal. The `Ellipse` class is defined in `Shear.h`. If there is magnification in the transformation, then the flux will be changed, since we consider surface brightness to be the quantity conserved in the transformation.

[5]Will fail if flux is zero. Currently will throw an exception for `SBLaguerre` because I have been too lazy to code the calculation.

2.4. Drawing methods

All of the drawing classes produce 2d sampled renderings of the `SBProfile`, using the class defined in `Image.h`. Note that `Image`s are objects resident in memory; they can very easily be saved as FITS-format images using the class in `FITSImage.h`. For all of the drawing routines, the pixel scale and size of the image can either be specified or the code will estimate a value that is appropriate for accurate unaliased rendering.

- `Image draw(double dx, int wmult)` returns an image rendering of the `SBProfile`. Taking the default values of $dx = 0$ and `wmult` $= 1$ will allow the code to choose the pixel scale and image size that appropriately capture the object scale and detail, and avoid aliasing or folding from Fourier space. A non-zero specified dx will force construction of an image at the requested scale. The overall size of the image can be increased by some integral factor `wmult` if desired, to gain better k-space resolution and fidelity and reduce any effects of wrapping. If the `SBProfile` has an means of direct estimation of the real-space values (`isAnalyticX()` returns `true`), then this will be used to draw, otherwise the rendering will be done via FFT from k-space values.

- `double draw(Image img, double dx, int wmult)` functions the same way except that the rendering is done in the input `Image` object. If the input `Image` has non-null dimensions, these will be used to draw the image. If the input image is too small to draw via FFT methods without folding, the FFT will be done in a larger workspace image and the smaller region copied into the input `Image`. The `dx` and `wmult` parameters function as with `draw`. Note that the `DX` keyword of the input image is ignored.

- `plainDraw()` and `fourierDraw()` have the same arguments as `draw()`, but they force the rendering to be done by real-space calculation or by FFT from Fourier space, respectively. If `plainDraw()` is called on an object that cannot execute `xValue()`, an exception will result.

- `void drawK(Image Re, Image Im, double dk, int wmult)` draws the real and imaginary parts of the Fourier transform of the `SBProfile` into the two input images. As with real-space drawing, the default $dk = 0$ instruct the class to select its own pixel scale, and the dimensions are taken from the input image (if any—and they should match), otherwise selected automatically and potentially scaled by `wmult`.

- `fourierDrawK()` can force the drawing of an `SBProfile` in Fourier space to be done via FFT from real space. This may not be possible and is probably not useful. `plainDrawK()` draws directly in k space and is the default behavior of `drawK()`.

3. The derived classes

3.1. Atomic SBProfiles

The atomic classes are the "building blocks" of the SBProfile facility. With the exception of SBLaguerre[6] and SBInterpolatedImage, the atomic classes represent circular objects (or boxes) centered at the origin. The transformations and addition classes are used to compose more complex shapes from these. Unless otherwise specified, the flux arguments for all atomic classes will default to unity, and also any parameters specifying the object size will default to 1 as well. Some of the atomic classes have additional parameters, as noted below.

- SBGaussian(double flux, double sigma): $I \propto \exp(-r^2/2\sigma^2)$.
- SBExponential(double flux, double r0: $I \propto \exp(-r/r_0)$.
- SBSersic(double n, double flux, double re): $I \propto \exp[-(r/r_0)^{1/n}]$. Note that the input size parameter to the Sersic is the "effective radius" r_e that encloses half of the light, not the r_0 that appears in the formula. The conversion is done automatically by the class, and depends on n. The Fourier transform for arbitrary n must be calculated numerically. These are done and then cached by the class in case another SBSersic of the same n is created later. The code currently limits $0.5 \leq n \leq 4$. Note that Sersic profiles with $n = 1/2$ and $n = 1$ are the Gaussian and exponential profiles, respectively. The SBGaussian and SBExponential classes use the analytic k-space formulae for these profiles, while SBSersic does not; also the sizes are specified by half-light radius in SBSersic, but by the more conventional σ and r_0 for the specialized classes.
- SBAiry(double D, double obs, double flux) represents the diffraction pattern from an annular pupil with central obscuration of diameter $\epsilon =$ obs times the outer diameter. obs defaults to 0. The size of the Airy pattern is specified by D which is D/λ. The real-space formula is
$$I \propto \left[J_1(\pi r D/\lambda) - \epsilon J_1(\epsilon \pi r D/\lambda)\right]^2 . \quad (4)$$
- SBBox(double xw, double yw, double flux) represents a rectangular box of dimensions $x_w \times y_w$, centered at the origin. By default yw is set to zero, signalling that it should be set equal to xw to make a square. xw itself defaults to unity. The surface brightness inside the box is $I = f/(x_w y_w)$.
- SBMoffat(double beta, double truncationFWHM, double flux, double re): $I \propto [1 + (r/r_D)^2]^{-\beta}$. The Moffat profile is truncated to zero beyond a radius of truncationFWHM times the FWHM of the profile. Note that neither the FWHM or r_D is specified in the constructor:

[6] which I'm not going to document yet out of laziness

the half-light radius r_e is given instead. The methods `setFWHM()` or `setRd()` can be used to rescale the Moffat profile to the desired FWHM or r_D.

- `SBInterpolatedImage(int Npix, double dx, const Interpolant2d& i, int Nimages)` is a brightness pattern specified by values on a square grid of pixel size `dx` and pixel indices ranging from $-N_{\rm pix}/2$ to $N_{\rm pix}/2-1$ (for even values of `Npix`—for odd `Npix`, the input data are assumed symmetric about the origin). Values between pixel centers are defined by the instance `i` of the `Interpolant2d` base class—the `Interpolant.h` file defines all of the commonly used interpolation schemes, such as `Linear, Cubic,` and `Lanczos`. One can have `Nimages`> 1 to allow the `SBInterpolatedImage` to produce brightness as a weighted combination of several pixel arrays. `SBInterpolatedImage.kValue()` returns the Fourier coefficient of the continuous interpolated image, so the choice of interpolant influences the values returned and the $k_{\rm max}$ of the class. `SBInterpolatedImage` is explained more fully in §5.

3.2. Transformation classes

The `SBProfile` daughter classes described here represent modifications to other `SBProfile`s. They take other `SBProfile`s as constructor or method arguments. The convention is that fresh copies of the input `SBProfile`s are made and stored by the transformation classes, so that the originals can be deleted. Also, any changes to the input `SBProfile`s are not propagated into their transformed versions after the transformation is defined.

- `SBDistort(const SBProfile& sbin, double mA, double mB, double mC, double mD, Position<dou` `x0` represents an affine transformation of the input profile `sbin`. The intensity is defined as $I(x,y) = I_{\rm in}(x',y')$, where

$$\left(\begin{array}{c} x \\ y \end{array} \right) = \left(\begin{array}{cc} A & B \\ C & D \end{array} \right) \left(\begin{array}{c} x' \\ y' \end{array} \right) + \left(\begin{array}{c} x_0 \\ y_0 \end{array} \right). \tag{5}$$

  You can also construct an `SBDistort` using the `Ellipse` class to represent affine transformations without rotation.

- `SBAdd, SBConvolve` are the sum and convolution, respectively, of an arbitrary number of input `SBProfile`s. Each class has constructors specifying 0, 1, 2, or a list of input `SBProfile`s. Each also has an `add(const SBProfile& rhs)` method which will add another term to the sum/convolution. A multiplicative scaling factor can optionally be specified when `add`ing a new `SBProfile` to a `SBAdd`.

4. Parser syntax

Note that the `SBParse` *function has been removed from the GalSim repository in favour of the Python constructors. Please see https://github.com/GalSim-developers/GalSim/issues/190.*

The `SBParse` function (declared in `SBParse.h`) takes a string argument and returns a pointer to an `SBProfile` described by the string. Remember to delete the `SBProfile` when you are done with it. Parsing errors will throw an exception of class `SBError`. The input string has a very simple syntax composed of the following elements:

- Words identifying an atomic `SBProfile` type, followed by whitespace-separated arguments. All primitives are set to unit flux (see below for flux modification.) Optional arguments and their defaults are listed in brackets in this list of implemented atomic types:
  - `gauss` $[\sigma = 1]$
  - `exp` $[r_e = 1]$ — note use of r_e as size parameter here.
  - `sersic` n $[r_e = 1]$
  - `box` $[x_w = 1]$ $[y_w = x_w]$
  - `airy` D/λ ϵ —note no default for the obscuration ϵ, it must be specified.
  - `moffat` β *truncationFWHM* $[r_e = 1]$
  - `laguerre` *filename* —the named file contains the coefficients for the Gauss-Laguerre expansion in a standard form.
  - `pixel`—*not yet implemented in the parser.*
- *Modifiers* which alter the characteristics of the `SBProfile` described immediately to their left. Each modifier is a single letter followed by one or more whitespace-separated parameters. Modifiers are applied in left-to-right order.
  - `D` m will dilate (magnify) the `SBProfile` by linear factor m. Note the flux also increases by m^2 since surface brightness is conserved.
  - `T` x y will translate (shift) the `SBProfile` by the vector (x, y) on the image plane.
  - `R` θ will rotate the object by angle θ (rotation is measured from x through y axis).
  - `F` f will set the flux of the object to f (will fail if object had zero flux initially).
  - `S` e_1 e_2 will shear the `SBProfile` with a unit-determinant transformation.
- Binary operators + and $*$ will sum the `SBProfile`s on either side.
- Parentheses () alter the order of operations.

The precedence of operations, from highest to lowest, is: parentheses; modifiers; convolution ($*$); then addition ($+$). Whitespace is required between all keywords, operators, and arguments, except for parentheses.

Some examples:

```
sersic 1. 3. S 0.8 0. F 0.75 + sersic 4. S 0.2 0. F 0.25
```

would mimic a spiral galaxy that has an $n = 1$ disk with $r_e = 3$, highly flattened ($e = 0.8$), holding 75% of the flux, added to a bulge component with $n = 4$, $r_e = 1$, slightly flattened $e = 0.2$, with 25% of the flux. The total flux is 1 and both components are centered at the origin and aligned with the x axis.

```
(exp 3. S 0.8 0. F 0.75 + sersic 4. S 0.2 0. F 0.25) * airy 2 0.3 * box
```

would represent the same galaxy convolved with an airy pattern with obscuration 0.3 and $D/\lambda = 2$, then convolved with a unit square *i.e.* to represent the pixel square.

4.1. Command-line program

The `SBDraw.cpp` program parses a string given on its command line, draws the `SBProfile` using the default dx and image size—optionally overridden by command-line arguments—and writes the image to a specified FITS file. The SB string should be enclosed in quotes so that it is taken as a single command-line argument. Running the program with no arguments will yield a help message giving the order of arguments.

5. `SBInterpolatedImage`

To use the `SBInterpolatedImage` class, include `SBInterpolatedImage.h` and compile/link with `SBInterpolatedImage.cpp`.

5.1. Definitions

An instance of `SBInterpolatedImage` with even-valued array size $N_{\rm pix}$, pixel scale Δ, $N_{\rm images}$ image planes, and an interpolant with 2d kernel $K(x, y)$ defines a surface brightness pattern

$$I(x,y) = \left(\sum_{i,j=-N_{\rm pix}/2}^{N_{\rm pix}/2-1} \delta(x - i\Delta)\delta(y - j\Delta) \sum_{k=0}^{N_{\rm images}-1} w_k a_{ijk} \right) * K(x/\Delta, y/\Delta). \quad (6)$$

For odd-valued $N_{\rm pix}$, the spatial indices run from $-(N_{\rm pix}-1)/2$ to $+(N_{\rm pix}-1)/2$. Here the $*$ represents a convolution. The interpolant K is usually defined to be unity at the origin and to have $K(m,n)=0$ for non-zero integer values m and n. The pixel values a_{ijk} are initialized to zero and the weights w_k are initialized to unity. These are accessed via the methods

- `void setPixel(double value, int ix, int iy, int iz=0)` where $\{i,j,k\}=\{$`ix`, `iy`, `iz`$\}$.
- `double getPixel(int ix, int iy, int iz=0)`.
- `void setWeights(const DVector& wts)` sets the weights from the vector.
- `DVector getWeights()`
- `void setFlux(double flux=1.)` will rescale the weight vector to produce the specified total flux.

The `SBInterpolatedImage::kValue()` method returns the Fourier transform $\tilde{I}(kx,ky)$ of $I(x,y)$ in (6). This is *not* just the discrete Fourier transform of the input pixel grid. I will not describe the mathematics of this transform here, but it is important to realize that the k-space values are obtained by first zero-padding the pixel grid, then doing an FFT to a k-space grid, and interpolating in k space to the (k_x,k_y) specified in the call to `kValue`. Therefore one needs to choose an interpolant for k space as well as the interpolant in x space that defined the original brightness pattern. The formally correct interpolant to use in k space is `SincInterpolant`, however this is very slow to use because its kernel covers the entire array, requiring $\approx (4N_{\rm pix})^2$ kernel evaluations and summations for *each* call to `kValue()`. Therefore the default k-space interpolant is a 3rd-order `Lanczos` filter (described below), which produces a worst-case fractional error of $\approx 0.6\%$ in the `kValue` using only a 6×6 kernel. The worst-case error can be reduced by a factor 2–3, for example, by using a 5th-order Lanczos filter instead (10×10 kernel). These methods of `SBInterpolatedImage` allow you to change or view the interpolants specified for x and k space:

- `void setXInterpolant(const Interpolant2d& interp)`
- `const Interpolant2d& getXInterpolant()`
- `void setKInterpolant(const Interpolant2d& interp)`
- `const Interpolant2d& getKInterpolant()`

Here is an example of how one might construct an `SBInterpolatedImage` defined on a 32×32 grid with pixel scale $\Delta=0.5$, with Lanczos3 interpolation in the x-space and the k-space sinc interpolation approximated by a Lanczos5 interpolant (see following section for details on interpolants):

```
double tolerance=0.001;
```

```
Lanczos lan3(3, true, tolerance);
InterpolantXY lan3_2d(lan3);
Lanczos lan5(5, true, tolerance);
InterpolantXY lan5_2d(lan5);

SBInterpolatedImage sbp(32, 0.5, lan3_2d);
sbp.setPixel(value, -32, -32); // repeat to fill array...
sbp.setKInterpolant(lan5_2d);
```

5.2. Interpolants

The header `Interpolant.h` (in the subdirectory `utilities2`) defines the base class for interpolants and several derived classes. The abstract base class `Interpolant` defines a one-dimensional interpolation kernel, and the abstract base class `Interpolant2d` is for two dimensions. The only current implementation of `Interpolant2d` is `InterpolantXY`, which is defined as the separable product of a 1d interpolant in each the x and y directions. As in the example above, you construct the 2d `InterpolantXY` by handing it a reference to the 1d `Interpolant` you want it to use.

The interpolation functions assume that they will be operating on data given at integer x values, *i.e.* a pixel scale of unity. The `SBInterpolatedImage` classes do the scaling to general pixel scales `dx`. We often need Fourier transform of the interpolation kernel, which is available via the `Interpolant::uval()` method. Each `Interpolant` returns its kernel's extent in real and Fourier space via the `xrange()` and `urange()` methods. Formally, it cannot have finite kernels in both domains. However most of the `Interpolant` constructors allow you to specify a `tolerance` that gives the value below which kernel elements may be considered negligible, and dropped. In other words the `tolerance` describes the fractional accuracy with which the interpolant approximates its exact mathematical definition. It does *not* specify the accuracy with which the kernel interpolates a given function.

The current implementations of `Interpolant` available are:

- `Nearest(double tol)` is nearest-neighbor interpolation, *i.e.* the boxcar function:

$$K(x) = \begin{cases} 1 & |x| < 0.5 \\ 0.5 & |x| = 0.5 \\ 0 & |x| > 0.5 \end{cases} \tag{7}$$

  Use of `Nearest` is usually ill-advised even though it has the smallest footprint, as it introduces high-frequency components if used as x-space interpolant by the `SBInterpolatedImage`, and performs very poorly as a k-space interpolant.

- `Linear(double tol)` is linear interpolation, with range ±1:

$$K(x) = \begin{cases} 1 - |x| & |x| < 1 \\ 0 & |x| \geq 1. \end{cases} \tag{8}$$

  `Linear` is also a fairly poor choice since it rings to high frequencies as well.

- `Cubic(double tol)` is the next polynomial interpolation, with range ±2:

$$K(x) = \begin{cases} 1 - \frac{5}{2}|x^2| + \frac{3}{2}|x^3| & |x| < 1 \\ 2 - 4|x| + \frac{5}{2}|x^2| - \frac{1}{2}|x^3| & 1 \leq |x| < 2 \\ 0 & |x| \geq 2. \end{cases} \tag{9}$$

  `Cubic` is a good choice for a 4-point interpolant, better than 2nd-order `Lanczos` in some respects even though they differ from each other by < 0.02.

- `SincInterpolant(double tol)` is mathematically perfect for band-limited data and hence introduces no spurious frequency content beyond $k_{\max} = \pi/\Delta$ for input data at pixel scale Δ. It is, however, formally infinite in extent and very large even when truncated by a modest `tol`.

$$K(x) = \frac{\sin(\pi x)}{\pi x} \tag{10}$$

  Be careful with `SincInterpolant`. It will give you exact results in `SBInterpolatedImage::kValue()` if used as k-space interpolant, but probably be intolerably slow for anything requiring more than a few calls to `draw`. And as an x-space interpolant, it leads to very large extent for the interpolated real space `xValue()` results. [The long name for this interpolant is to avoid confusion with the `sinc` function that is defined in interpolation code and often by other codes.]

- `Lanzos(int n, bool fluxConserve=false, double tol=1e-3)` is an approximation to the band-limiting sinc filter with range $\pm n$ pixels:

$$K(x) = \begin{cases} \frac{\sin \pi x}{\pi x} \frac{\sin n\pi x}{n\pi x} & |x| < n \\ 0 & |x| \geq n \end{cases} \tag{11}$$

  The Lanczos filter is a good compromise between kernel size and accuracy. It has the defect that $\sum_j K(x+j) \neq 1$ for non-integral x, in other words it does not conserve the brightness of a uniform background, which is a major drawback for many astronomical images. This can be remedied to improve interpolation with little degradation of the band-limiting properties, simply by dividing $K(x)$ by this sum. Set `fluxConserve=true` on construction in order to specify this behavior.

5.3. Notes and caveats

The `SBInterpolatedImage` class should allow very accurate manipulation of finite sampled brightness patterns. However there are limitations to any discrete representation and also some behavior of the class to be aware of:

- The size and resolution of images produced when drawing using `SBInterpolatedImage` will depend on choice of x-space intepolation kernel. The real-space footprint of the image will grow with the size of interpolation kernel. Conversely, more compact kernels are less band-limited and have a higher $k_{\rm max}$ which means that images drawn by Fourier methods with the class will need longer FFTs, unless the high-frequency response of the interpolation kernel is rolled off by convolution with some other `SBProfile`.
- The `Nearest`, `SincInterpolant`, and `Linear` interpolants have very long tails in real or Fourier space, so if you use them you may end up with infeasibly large FFTs to perform if you are not careful.
- The class caches Fourier transforms of the padded images. Changing a pixel value (via `setPixel`) invalidates the cache and triggers new Fourier transforms next time an evaluation is done. I expect that the vast majority of uses will set all the pixel values initially and never change them, but the code is safe under changes.
- Likewise the weighted sum over the $N_{\rm images}$ image planes is cached, with the cache invalidated each time `setWeights()` is called. This should speed up most applications.
- If you just have a single image plane, the weight will default to 1. and can be ignored, and you can omit the 3rd index for `setPixel()` and `getPixel()`, so you can completely ignore the multi-plane capability of the class.
- The `Interpolant2d` instances used in both x and k space are stored as references. Therefore you should not delete them until after all the `SBInterpolatedImage`s that use them. It is safe to use a single `Interpolant` instance for many `SBInterpolatedImage` instances, however the code is not thread-safe.
- Keep in mind that changing weights also changes the centroid and flux of the pattern produced by `SBInterpolatedImage`.

5.4. Wrappers

I expect there to be several wrappers built to fill `SBInterpolatedImage` arrays from several data formats, for example an `SBFits` class should appear soon. One wrapper that already exists, thanks to Daniel Grun, is the `PSFExModel` class (using `PSFEx.h` and `PSFEx.cpp`) which constructs a `SBInterpolatedImage` object by reading PSF descriptions output by Emmanuel Bertin's

`PSFEx` code.[7] These are pixellated PSF models which vary as polynomial functions of position in an image. Each polynomial term becomes one `SBInterpolatedImage` image plane, and a call to `PSFExModel::fieldPosition(double x, double y)` will calculate the polynomial terms for position (`x,y`) and apply the appropriate weights to the image planes. The x-space interpolation is assumed to use 3rd-order Lanczos filtering, and one can choose to promote the k-space interpolant from the default 3rd-order Lanczos to 5th order for improved accuracy. The method `PSFExModel::sb()` returns a pointer to the `SBInterpolatedImage` that then represents the PSF.

6. Implementing a new atomic class

Will write this later.

7. Installation

The `SBProfile` codes rely heavily on the FFTW package available at `fftw.org`. The top lines of the makefile should be edited to insure that the include files and libraries for FFTW are in the appropriate paths.

As currently written the routines also require Mike Jarvis's Template Matrix-Vector (TMV) routines, hosted at `http://code.google.com/p/tmv-cpp/`. This package can in turn make use of several kinds of highly optimized libraries for linear algebra. It takes a while to install and compile, but is overkill for the basic `SBProfile` tasks, since they only use TMV to multiply 2×2 matrices. So I could probably write the dependence on TMV out of the package if requested.

Once the required libraries are installed and placed into the appropriate makefile locations, you should be able to simply say "`make SBDraw`" and build all of the subroutines and the `SBDraw` driver. I have successfully compiled the code with the gcc version 4.2.1 that is included with the Apple XCode package, and also the Intel `C++` compiler `icpc` version 12.0.4.

[7] See `www.astromatic.net`.