Mir Standard Library

Yury V. Vishnevskiy

Version 1.0, 2022-08-16

Table of Contents

Introduction

Packages

UnitTest

Package with helper functions for doing unit tests. The functions raise exceptions of class `AssertError`, which is a subclass of `StandardError`.

`assert_eq(obj1, obj2)` → **`exception`** `or` **`nil`**

Raises exception if `obj1` is not equal to `obj2`. Uses `==` method.

```
import("UnitTest")
UnitTest::assert_eq(1, 2) // => AssertError exception
```

`assert_eq_eps(obj1, obj2, eps)` → **`exception`** `or` **`nil`**

Raises exception if objects (usually floating point numbers) are not equal within defined epsilon. Uses `eq_eps` method.

```
import("UnitTest")
UnitTest::assert_eq_eps(10.0/3.3  ,  3.03030303030303, Float::Epsilon)
```

`assert_neq(obj1, obj2)` → **`exception`** `or` **`nil`**

Raises exception if objects are equal. Uses `==` method.

`assert_true(obj)` → **`exception`** `or` **`nil`**

Raises exception if the object is not equal to true.

`assert_false(obj)` → **`exception`** `or` **`nil`**

Raises exception if the object is not equal to false.

`assert_nil(obj)` → **`exception`** `or` **`nil`**

Raises exception if the object is not equal to nil.

Physics

Package for physics.

ED

Package for electron diffraction.

`calc_s(lambda:, teta: nil, r: nil, l: nil) → float`

Calculation of s-value from electron wavelength `lambda` (required argument in Angstroms), diffraction angle `teta` (in degrees) or a pair `r` (distance in mm from the center of diffraction pattern to the point in the plane of detector) and `l` (distance in mm from diffraction point to the detector plane).

`u2lam(float) → float`

Calculates relativistic electron wavelength [Å] from accelerating voltage `u` [V]. See Eq. (4.3.1.33) on page 262 in [1].

`lam2u(float) → float`

Calculates accelerating voltage [V] from relativistic electron wavelength `lam` [Å]. See Eq. (4.3.1.33) on page 262 in [1].

SVector

Class of vectors with s-values. Subclassed from `FVector`.

Class object methods

`from_rv(rvec, lambda, l) → svec`

Create a vector of s-values from the vector of r-values. Input parameters are: the vector `rvec` of r-values in mm, the electron wavelength `lambda` (in Angstroms) and the distance from the diffraction point to the detector `l` (in mm).

Instance methods

`to_rv(lambda, l) → fvec`

Create a vector of r-values from the vector of s-values. Input parameters are floating point values of the electron wavelength `lambda` (in Angstroms) and the distance from the diffraction point to the detector `l` (in mm). Values in the output vector are in mm.

Pattern

Class of electron diffraction patterns. Defined in the `Physics::ED` package. Inherited from the `Media::Image` class. Implemented mostly in the `edpattern` module.

Instance methods

`average_profile(xcenter, ycenter, rvec, average: fvec, stdev: fvec, skewness: fvec, kurtosis: fvec, npix: ivec, weighting: bool, rwidth: float) → fvec`

Calculates average profile from radially symmetric images. Input parameters are integer coordinates of the center and a vector of r-values (distances from center in mm), in which profile values should be averaged. Others are optional named arguments. The `average` can be used if the averaged profile values should be on output in the user-defined vector. The `stdev`, `skewness`, `kurtosis` and `npix` can be defined if respective values of standard deviations, skewness, excess kurtosis and amounts of used pixels for each profile point are required. The `weighting` argument is used to turn on or off the weighting in the processing of pixels. The `rwidth` determines thickness (in fractions of pixels) of the processed profile rings on the diffraction pattern. It is advised to use odd numbers or values close to odd numbers for this parameter. The method takes into account current active mask and area of the image. This method returns a vector with averaged profile values.

```
import("Physics/ED/Pattern")
ptrn = Physics::ED::Pattern.new()
ptrn.import_tiff("ZnO_small8.tif")
ptrn.to_logri!()
rval = FVector.range(11.0, 0.1, 40.0)
vari = FVector.new(rval.size())
vprof = ptrn.average_profile(499, 495, rval, stdev: sd)
```

avprof_outliers_zscore(xc, yc, rvec, thr, average: vec, stdev: vec, weighting: bool, rwidth: float) → mask

First, calculates averaged profile for the given center defined by the coordinates `xc` and `yc` and r-values in `rvec`. Second, this method finds outlier pixels using the z-score method with the threshold value `thr`. Optional named arguments can be provided if averaged values of profile and/or respective standard deviations are required. The arguments `weighting` and `rwidth` have the same meaning as in the `average_profile` method. This method returns mask with outliers.

```
ptrn = Physics::ED::Pattern.new()
ptrn.import_tiff("ZnO_small8.tif")
ptrn.to_logri!()
rval = FVector.range(11.0, 0.1, 40.0)
mask = ptrn.avprof_outliers_zscore(499, 495, rval, 1.0)
```

from_profile(xc, yc, r, v) → nil

Calculate diffraction pattern from profile values. `xc` and `yc` are coordinates (in pixels) of the center. `r` and `v` are vectors with distances (in mm) and respective values of profile. Note, active mask and area are taken into account.

```
vlen = 800
r = FVector.new(vlen)
v = FVector.new(vlen)
i = 0
while i < vlen
  r[i] = 0.1*i
  v[i] = Math::sin(r[i])
```

```ruby
  i += 1
end
ptrn = Physics::ED::Pattern.new()
ptrn.set_size(1500, 1000)
ptrn.set_resolution(508.0, 508.0)
ptrn.from_profile(750, 500, r, v)
ptrn.export_tiff("simulated_sin.tif")
```

`mask_rrange(xc, yc, rmin, rmax) → bitarr`

Creates mask on the basis of the minimal and maximal distances (in mm) from the center (`xc` and `yc` in pixels) of the diffraction pattern. Pixels outside of the [`rmin`, `rmax`] range are masked.

```ruby
ptrn.set_size(1500, 1000)
ptrn.set_resolution(508.0, 508.0)
ptrn.mask_rrange(750, 500, 10.0, 20.0)
```

`scancc_avprof_lsqre(cxmin, cxmax, cymin, cymax, rval, zthr: 0.0, skewf: 0.0, kurtf: 0.0, weighting: false, rwidth: 0.99, fast: true, verbose: false) → cxopt, cyopt`

Scan coordinates of the center by calculating averaged profile and testing least squared relative errors. Finds optimal coordinates for the center of the diffraction pattern by two-dimensional scanning of pixels in the ranges `cxmin` — `cxmax` and `cymin` — `cymax`. The profiles are calculated for the distances defined in the input fvector `rval` in mm. In calculations of profiles outliers can be detected using Z-score method if the respective threshold is defined with the `zthr` named parameter. If named arguments `skewf` and/or `kurtf` are defined, then the total tested functional value is calculated taking into account values of skewness and kurtosis for all profile values. Sums of squared skewness and kurtosis multiplied by respective factors `skewf` and `kurtf` are added to the sum of squared relative errors of profile values. Parameters `weighting`, `rwidth` and `fast` are passed to the methods `avprof_outliers_zscore` and `average_profile`. If the `verbose` argument is set to true, then information is printed on each iteration: currently tested coordinates of the center, total functional value, components of the total functional (sums of relative errors, skewnesses and kurtoses) and the number of detected outliers. This method returns list of two values, x and y coordinates of the found center.

`scancc_avprof_lsqrp(cxmin, cxmax, cymin, cymax, rval, zthr: 0.0, dzthr: 0.0, weighting: false, rwidth: 0.99, fast: true, verbose: false) → cxopt, cyopt`

Scan coordinates of the center by calculating averaged profile and testing least squares of residuals between model and real pixel values. Most of the parameters have the same meaning as in the `scancc_avprof_lsqre` method. With the parameter `dzthr` can be defined Z-score threshold value for detecting outliers of the difference image. These outlier pixels, as well as outliers possibly detected by profile averaging (see argument `zthr`), are not taken into account in calculations of functional values. The `verbose` argument can be used to turn on printing of information: currently tested coordinates of the center, total functional value, number of detected outliers in profile averaging, additional number of detected outliers for the delta image.

Media

Package for digital media.

Image

Class of images. Defined in the `Media` package. Implemented mostly in the `image` module.

Instance methods

Many methods take into account image mask and area. This means that only those pixels are processed which are not masked and which are the current in active area. If no mask and area are defined then all pixels are processed.

`import_bitarr(bitarr,width,height) → img`

Imports data from bit array and returns bilevel (1-bit) image of size width-x-height.

```
Media::Image.new().import_bitarr(BitArray.new(100), 20, 5)
              // => <Image: 20x5 gray1>
```

`import_fmat(fmat) → img`

Imports data from floating-point matrix and returns image of the same size as the matrix.

```
fmat = FMatrix.diagonal(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
img = Media::Image.new().import_fmat(fmat)
```

`inspect() → string`

Creates a string representing the image.

```
Media::Image.new().inspect() // => "<Image: 0x0>"
Media::Image.new().import_tiff("test.tif") // => "<Image: 125x125 gray16ui>"
```

`import_tiff(filename) → img`

Imports data from file in TIFF format. The argument `filename` should be given as string. Currently 8- and 16-bit grayscale compressed and uncompressed TIFFs are supported.

```
img = Media::Image.new().import_tiff("test.tif") // => <Image: 10x10 gray8ui>
```

`export_tiff(filename, args) → nil`

Export image data to a file in TIFF format. The argument `filename` should be given as string. Additional optional named arguments are:

- `compress`: 'options'
- `mask`: 'black' or 'white'

- `model`: 'gray16ui'
- `photo`: 'miniswhite' or 'minisblack'

Note, model options here are not related to data models used for images from this class but only define how data are stored in the created TIFF file.

```
img = Media::Image.new().import_tiff("test.tif")
                                  // => <Image: 10x10 gray8ui>
img.export_tiff("exported.tif")
img.export_tiff("img_masked_w.tif", mask:'white')
img.export_tiff("img_masked_b.tif", mask:'black')
img.export_tiff("img_photo_w.tif",  photo:'miniswhite')
img.export_tiff("img_photo_b.tif",  model:'gray16ui', photo:'minisblack')
```

`import_basimg(filename) → img`

`import_basimg(imgname,infname) → img`

Imports data from files in BAS IMG format. This format assumes existance of two files, one with pixel data and the other with information on image dimensions, etc. If one argument `filename` is given then the method tries to open files with names `filename.img` and `filename.inf`. Alternatively, both files can be defined explicitly by calling the method with two arguments.

```
// Open example.img and example.inf
Media::Image.new().import_basimg("example") // => <Image: 125x125 gray16ui>
```

`print_info() → nil`

Prints basic information about image.

```
img = Media::Image.new().import_tiff("test.tif")
img.print_info()
  // Dimensions: 3x3 pixels
  // Data model: gray16ui
  // Grayscale model: min-is-black
  // Bits per sample: 16
  // Samples per pixel: 1
  // Resolution: 72.000000 dpi (x), 72.000000 dpi (y)
  // => nil
```

`print_data() → nil`

Prints data in image.

```
img = Media::Image.new().import_tiff("10x10.tif")
img.print_data()
  //Sample #0:
  // 255 255 255 255 237 237 255 255 255 255
  // 255 255 237 220 203 203 220 237 255 255
  // 255 237 203 166 144 144 166 203 237 255
  // 255 220 166 115  86  86 115 166 220 255
  // 237 203 144  86  54  54  86 144 203 237
  // 237 203 144  86  54  54  86 144 203 237
  // 255 220 166 115  86  86 115 166 220 255
  // 255 237 203 166 144 144 166 203 237 255
  // 255 255 237 220 203 203 220 237 255 255
  // 255 255 255 255 237 237 255 255 255 255
  // => nil
```

`entropy() → float`

Calculates and returns information entropy for the image. This method takes into account mask and area of the image. The image should contain data of models 'gray8ui' or 'gray16ui'.

```
img = Media::Image.new().import_tiff("10x10.tif")
img.entropy()                                  // => 3.17
```

`statistics() → list of values`

Calculates statistics for the image. The list of values includes average pixel value, standard deviation, skewness, excess kurtosis, minimal value, two coordinates of the pixel with the minimal value, maximal value, two coordinates of the pixel with the maximal value, total number of samples (pixels used in calculation of statistics). This method takes into account mask and area of image.

```
tif = Media::Image.new().import_tiff("dot10x10.tif")
tif.statistics()
// => 199.92,61.50300641415669,-0.9682298330044166,-
0.301812444417843,54.0,4,4,255.0,0,0,100
```

`average() → list of values`

Calculates average value of pixel for the image. The list of values includes the calculated average pixel value and number of pixels used in calculation. This method takes into account mask and area of the image. u/

`stdev() -→ list of values`

Calculates standard deviation of pixel values for the image. The output list includes the calculated value of standard deviation and number of pixels used in calculation. This method takes into account mask and area of the image.

`min() → list of values`

`max() → list of values`

These methods determine minimal and maximal pixel values for an image. The output list contains several values: minimal or maximal pixel values, coordinates of the corresponding pixel with minimal or maximal value and total number of considered pixels. This method takes into account mask and area of the image.

`to_powri(drange:, exp:) → img`

`to_powri!(drange:, exp:) → img`

Converts image from gray16ui to gray64fp data model calculating relative intensities of pixels using the general formula

$$I = drange \times \left(\frac{J}{G}\right)^{exp}$$

where *drange* is the dynamic range (default value is `1.0e5`), *exp* is the exponent (must be defined explicitly), *G* is the maximal grayscale value (65535 for 16 bit images), *J* is the grayscale value of the pixel (min-is-white grayscale model is assumed). Note, for TIFF files produced by Amersham Typhoon software *drange* is typically 10^5 and *exp* must be 1.0. GEL files by Amersham Typhoon software are also in fact in TIFF format but coded so that *exp* is equal to 2.0. The method with `!` does calculation in place, changes the model of the original image and returns the original image. The other method returns a new image as the result of the conversion and leaves unmodified the original image.

```
img = Media::Image.new().import_tiff("image.gel")
img2 = img.to_powri(drange: 1.0e6, exp: 2.0)
img = Media::Image.new().import_tiff("image.tif")
img.to_powri!(exp: 1.0)
```

`from_powri(drange:, exp:, model:) → img`

`from_powri!(drange:, exp:, model:) → img`

These methods do the opposite to the to_powri() and to_powri!(), namely convert gray64fp images to gray8ui or gray16ui data models (depending on the required `model:` named argument). The named arguments `drange:` and `exp:` correspond to those in to_powri() and to_powri!() methods, i.e. the used here formula is

$$J = G \times \left(\frac{I}{drange}\right)^{\frac{1}{exp}}$$

The default value for the named argument `drange:` is 10^5. Note, the calculation is done only for pixels in area and not masked pixels. Pixels witn negative and zero values are converted to pixels with $J = 0$.

```
img = Media::Image.new().import_tiff("image.gel")
img2 = img.to_powri(drange: 1.0e6, exp: 2.0)
img3 = img2.from_powri(drange: 1.0e6, exp: 2.0, model: 'gray16ui')
```

`to_fujiri(psize:, sens:, drange:) → img`

`to_fujiri!(psize:, sens:, drange:) → img`

Converts image from gray16ui to gray64fp data model calculating relative intensities of pixels using the general Fuji formula [2, 3]:

$$I = \left(\frac{psize}{100}\right)^2 \times \frac{4000}{sens} \times 10^{drange\left(\frac{J}{G} - 0.5\right)}$$

where *psize* is the size of pixels in µm (typical values are 10.0, 25.0, 50.0, 100.0 and 200.0), *sens* is the sensitivity parameter (typical values are 1000.0, 4000.0 and 10000.0), *drange* is the dynamic range parameter (typical values are 4.0 and 5.0 corresponding to the dynamic ranges 10^4 and 10^5, respectively), *G* is the maximal grayscale value (65535 for 16 bit images), *J* is the grayscale value of the pixel (min-is-white grayscale model is assumed). Note, these parameters are defined as operational settings for the particular IP-scanner. In this method you should use the same values of parameters as they were defined at the scanning. Note, Amersham Typhoon software can assume the formula $I = 10^{5\frac{J}{G}}$ irrespective of settings. This corresponds to the general formula above with parameters *psize*=100.0, *sens*=12.649111 and *drange*=5.0. The method with `!` does calculation in place, changes the model of the original image and returns the original image. The other method returns a new image as the result of the conversion and leaves unmodified the original image.

```
img = Media::Image.new().import_basimg("image")
img2 = img.to_fujiri(psize: 50.0, sens: 4000.0, drange: 5.0)
img.to_fujiri!(psize: 50.0, sens: 4000.0, drange: 5.0)
```

`from_fujiri(psize:, sens:, drange:, model:) → img`

`from_fujiri!(psize:, sens:, drange:, model:) → img`

These methods do the opposite to the to_fujiri() and to_fujiri!(), namely convert gray64fp images to gray8ui or gray16ui data models (depending on the required `model:` named argument). The named arguments `psize:`, `sens:` and `drange:` correspond to those in to_fujiri() and to_fujiri!() methods, i.e. the used here formula is

$$J = G \times \left(\frac{\log_{10}\left(\frac{I}{\left(\frac{psize}{100}\right)^2 \times \frac{4000}{sens}} \right)}{drange} + 0.5 \right)$$

Note, the calculation is done only for pixels in area and not masked pixels. Pixels witn negative and zero values are converted to pixels with $J = 0$.

```
img = Media::Image.new().import_basimg("image")
img2 = img.to_fujiri(psize: 100.0, sens: 4000.0, drange: 5.0)
img3 = img2.from_fujiri(psize: 100.0, sens: 4000.0, drange: 5.0, model: 'gray16ui')
```

`to_logri(base:) → img`

`to_logri!(base:) → img`

Converts image from gray8ui or gray16ui to gray64fp data model calculating relative intensities of pixels using the formula

$$I = \log_{base}\frac{G}{G-J}$$

where G is the maximal grayscale value (255 and 65535 for 8 and 16 bit images, respectively), J is the grayscale value of the pixel. By default base is equal to e so natural logarithm is used. In special cases when $J = G$ the relative intensity is calculated as $I = \log_{base}(G+1)$. The formulae above are for the min-is-white grayscale model. In the case of min-is-black model the pixel values are automatically recalculated as $J = G - J$. The method with ! does calculation in place, changes the model of the original image and returns the original image. The other method returns a new image as the result of the conversion and leaves unmodified the original image.

```
img = Media::Image.new().import_tiff("image.tif")
img2 = img.to_logri()
img.to_logri!(base: 10.0)
```

`from_logri(base:, model:) → img`

`from_logri!(base:, model:) → img`

These methods do the opposite to the to_logri() and to_logri!(), namely convert gray64fp images to gray8ui or gray16ui data models (depending on the required model: named argument) using exponentiation. The named argument base: corresponds to that in to_logri() and to_logri!() methods, i.e. the used here formula is

$$J = G - \frac{G}{base^I}$$

The default value for the named argument base: is e. Note, the calculation is done only for pixels in area and not masked pixels. Pixels witn negative and zero values are converted to pixels with $J = 0$.

```
img = Media::Image.new().import_tiff("image.tif")
img2 = img.to_logri()
img3 = img2.from_logri(model: 'gray16ui')
```

`scale_data(sf) → img`

`scale_data!(sf) → img`

Multiply pixel values by the factor sf (must be a floating point number). The data model of the image must be gray64fp. This method takes into account image mask and area. The method scale_data returns a new image, the method scale_data! modifies the image data in place and returns it.

```
img = Media::Image.new()
img.set_size(10, 5)
img.set_data(5.0)
img.scale_data!(0.1)
img2 = img.scale_data(10.0)
```

`normalize_minmax(min, max) → img`

`normalize_minmax!(min, max) → img`

Applies a linear transformation (rescaling, min-max normalization) so that minimal and maximal pixel values become `min` and `max`. This method takes into account image mask and area. The version of the method with the `!` mark modifies the original image and returns it. The other method returns a new normalized image while the original image remains unchanged.

```
img = Media::Image.new().import_tiff("image.tif").to_logri!()
img2 = img.normalize_minmax(0.0, 1.0)
```

`abs() → img`

`abs!() → img`

Converts image data to absolute values. Expects gray64fp data model. These methods take into account image mask and area. The version of the method with the `!` mark modifies the original image and returns it. The other method returns a new image while the original image remains unchanged.

```
img = Media::Image.new()
img.set_size(10, 5)
img.set_data(5.0)
img2 = img.abs()
```

`mask_outliers_zscore(fac) → bitarr`

Detects outlier pixels in the image using Z-score method. The argument `fac` must be a floating point number, which is used as a factor for the calculated standard deviation of pixel values. The created mask is returned as a bit array. Note, the mask is not applied to the image automatically. This method takes into account current image mask and area.

```
img = Media::Image.new().import_tiff("my.tif") // => <Image: 125x125 gray16ui>
img.to_logri!()                                // => <Image: 125x125 gray64fp>
img.mask = img.mask_outliers_zscore(3.0)       // => <Bit array: size=15625>
```

`active_pixels() → int`

Returns number of pixels, which are in the current area and not masked.

```
img = Media::Image.new()
img.set_size(10, 5)
img.active_pixels() // => 50
```

`width() → int`

`height() → int`

`size() → width,height`

Return image width or height, or return a list of the two values, respectively.

`set_size(width, height) → nil`

Sets width and height (in pixels) for image. This method destroys previous image data.

```
img = Media::Image.new()
img.set_size(100, 50)
```

`set_resolution(xres, yres) → nil`

Sets resolution (in dpi) along x- and y-directions for image.

```
img = Media::Image.new()
img.set_resolution(299.9, 300.1)
```

`resolution() → xres, yres`

Returns a list of two floating point values, corresponding to the image resolution (in dpi) in x and y directions, respectively.

`set_data(value) → img`

Set image pixel values equal to `value`. Masked and not in area pixels are not set. The type of `value` must correspond to the image data model. If the model is not yet defined then it is initialized automatically.

```
img = Media::Image.new()  // => <Image: 0x0>
img.set_size(10, 5)
img.set_data(5.5)         // => <Image: 10x5 gray64fp>
img2 = Media::Image.new()
img2.set_size(10, 10)
img2.set_data(111)        // => <Image: 10x10 gray8ui>
img3 = Media::Image.new()
img3.set_size(10, 10)
img3.set_data(12345)        // => <Image: 10x10 gray16ui>
```

`set_random_gauss(generator, mu: 0.0, sigma: 1.0) → img`

`set_random_gauss!(generator, mu: 0.0, sigma: 1.0) → img`

Set pixel values to random floating point numbers from Gaussian distribution with parameters `mu` and `sigma`, respectively. The first argument must be a generator of random numbers. The other two are optional named arguments for mean value and standard deviation of the distribution with default values 0.0 and 1.0, respectively. The returned image has 'gray64fp' data model. These methods take into account image mask and area, i.e. only pixels in area (if defined) and not masked pixels are set. The version of the method with the `!` mark modifies the original image and returns it. The other method returns a new image while the original image remains unchanged.

```
generator = Math::Random.new()
img = Media::Image.new()
img.set_size(100, 100)
img.set_random_gauss!(generator, mu: 15.0, sigma: 10.0)
```

`img1 + float → img`

`img1 + img2 → img`

`img1 - float → img`

`img1 - img2 → img`

`img1 * img2 → img`

`img1 * float → img`

Arithmetic operations with pixel values. A new image as a product of an operation is returned. Only pixels in area (if defined) and not masked pixels are processed.

```
img = Media::Image.new()
img.set_size(10, 5)
img.set_data(-1.0)
img2 = img + 3.0
img3 = img * img2
img4 = 0.1*(img3 - img)
```

`convert(model) → img`

`convert!(model) → img`

Convert image from current data model to the model defined by input argument `model` as a symbol. Current implementation allows conversion between models 'gray16ui' and 'gray64fp'. If no model was defined for the image before, then the model is just set and pixel values are initialized to be zero. The method `convert` returns a new image. In case of `convert!` the original image is converted and returned.

```
img = Media::Image.new()
img.set_size(10, 10)
img.set_data(0.0)
img2 = img.convert('gray16ui')
img.convert!('gray16ui')
```

`equalize() → img`

`equalize!() → img`

Performs histogram equalization. The method with exclamation mark modifies data of the original image and returns it. The other method returns a new image.

```
orig_hilbert = Media::Image.new().import_fmat(FMatrix.hilbert(100,100)).convert!
('gray16ui')
equalized_hilbert = orig_hilbert.equalize()
equalized_hilbert.export_tiff("hilbert16_equal.tif")
```

`img1 == img2 → true` or `false`

`img1 != img2 → true` or `false`

Testing for (non-)equality of two images. Images are equal if they have the same widths, heights, resolutions in both directions, data models (including biths per sample and samples per pixel) and values of pixels. Comparison of pixels is done taking into accound mask and area of the first image. In case of gray64fp data model the comparison is done taking into account machine accuracy (see Float::Epsilon). If `img2` is not an image then the objects are not equal.

```
tif1 = Media::Image.new().import_tiff("gray8.tif")
tif2 = Media::Image.new().import_tiff("gray8.tif")
tif1 != tif2 // => false
tif1 == tif2 // => true
tif1 == 1 // => false
```

`sumsqr() → float`

Calculates and returns sum of squared pixel values. This method takes into account mask and area of the image. Implemented for the 'gray64fp' model.

```
img = Media::Image.new().import_fmat(FMatrix.hilbert(10,10))
img.sumsqr() // => 3.18810710572248
```

Functions

`basimg2tiff(imgf, tiff: nil) → nil`

Convert image in BAS IMG format to an image in TIFF format. The input image is defined by normal argument `imgf`, which must be the name of image file without extension. Two files are expected to exist with this name and extensions '.img' and '.inf'. The name of output can be created automatically by adding the '.tif' extesion to the input `imgf` argument or can be defined explicitly using named argument `tiff`.

```
Media::Image::basimg2tiff("../../myfile", tiff: "converted.tif")
Media::Image::basimg2tiff("pattern")
```

References

[1] A. W. Ross, M. Fink, R. Hilderbrandt, J. Wang, and V. H. J. Smith, “Complex scattering factors for the diffraction of electrons by gases,” in *International Tables for Crystallography Volume C: Mathematical, physical and chemical tables*, Third edition., E. Prince, Ed. Dordrecht/Boston/London: Kluwer Academic Publishers, 2004, pp. 262–391.

[2] N. Vogt, R. Rudert, A. N. Rykov, N. M. Karasev, I. F. Shishkov, and J. Vogt, “Use of imaging plates (IPs) in the gas-phase electron diffraction (GED) experiments on the EG-100 M apparatus. The tetrachloromethane molecule as a test object,” *Struct. Chem.*, vol. 22, no. 2, pp. 287–291, 2011, doi: 10.1007/s11224-010-9707-6.

[3] R. J. F. Berger, M. Hoffmann, S. A. Hayes, and N. W. Mitzel, “An Improved Gas Electron Diffractometer – The Instrument, Data Collection, Reduction and Structure Refinement Procedures,” *Z. Naturforsch. B Chem. Sci.*, vol. 64b, no. 11/12, pp. 1259–1268, 2009.