1 Hyperspectral Instrument Data Resemblance Algorithm

HIDRA is a modular simulator built for Python 3+, capable of simulating stellar spectra as they would appear on a CCD detector after being processed by the optical setup, etc. The simulator is built in a modular way, so the users have the largest degree of freedom in choosing which parts of the simulator to utilise. The simulator was created as a part of my thesis work during my Master's degree in astronomy at Aarhus University. The thesis can be found in the Github repository that also contains the simulator and this document. This document is meant as a supplimenting document, as to help potential users get a more comprehensive guide to the inner workings of the simulator. First, a guide to running the simulator a single time is presented. Then, the simulator is used to find the transmission spectrum of an exoplanetary atmosphere, using appropriate input values. Both guides will in large parts be in a step-by-step fashion for most clarity. There will also be descriptions and docstrings of the individual functions used throughout the guide at the end of the document.

2 Breakdown of the inputs

For the simulator to work, the inputs must have the correct format and units. This section outlines the input parameters and they will be described using the following template:

variable name
Description of input
Format, with a brief explanation
[*Units of inputs*]
Notes: Further information about the input

in_spec
Input science spectrum
2D array, 2 columns. The first column contains the wavelength, the second contains the corresponding photon count of that wavelength, per square centimeter.
$[nm]$, $[counts\ s^{-1}\ nm^{-1}\ cm^{-2}]$
Notes: If the wavelength coverage is incomplete, i.e. does not cover to the specified wavelength span, values will be extrapolated using the interp function.

col_area
Effective collecting area of the telescope
Float, single value.
$[cm^2]$
Notes: This values will be multiplied onto each value of the *in_spec* array to get

the final photon count per wavelength.

img_size
Size of the CCD
Tuple, 2 integers.
[*pixel*], [*pixel*]
Notes: The size (in pixels) of the final image.

pl_scale
Plate-scale of the detector
Float, single value.
[$arcsec\ mm^{-1}$]
Notes:

pix_size
Pixel size of the detector
Float, single value.
[$mm\ pixel^{-1}$]
Notes: the physical size of each pixel of the detector.

bg_spec
Background spectrum
2D array, 2 columns. The first column contains the wavelength, the second contains the photon counts per wavelength, per square centimeter. Similar to the *in_spec* array.
[*nm*], [$counts\ s^{-1}\ nm^{-1}\ cm^{-2}\ arcsec^{-2}$]
Notes: Zodiacal light, and other background sources. If the wavelength coverage is incomplete, i.e. does not cover to the specified wavelength span, the missing values will be extrapolated. Currently, the inclusion of a background source is not working ideally - use at own risk.

exp
Exposure time
Integer.
[*s*]
Notes: Will be multiplied by the *step* input, to find the total amount of time steps to run the simulation over.

sub_pixel
Amount of sub-pixels per pixel
Integer.
[*sub-pixels*]
Notes: The full pixels will contain a number of sub-pixels equal to the square

of *sub_pixel* - ex. sub-pixel = 10 will lead to a pixel with a total of 100 sub-pixels.

wl_ran
Wavelength range
Tuple, 2 integers.
[*nm*]
Notes: Must contain the first and last wavelength to be included in the simulation. As the simulator can only simulate a spectral resolution of 1 nm, the endpoints must be whole integers.

eta_in
Total spectral throughput
2D array, 2 columns, the number of rows must be equal to the wavelength span. The first column contains the wavelength, the other contains the fractional throughput (values between 0 and 1).
[*nm*], [*decimal fraction*]
Notes: The input should be a combined spectral throughput of the entire system - spectrograph, mirrors, telescopes, fibers, anything that will affect the throughput. If the user has more than one array, all can be supllied by simply listing all .txt-files as the input in the *input_file.py* (e.g. *eta_in = '../sample_values/QE2.txt', '../sample_values/optics.txt'*, with QE2.txt and optics.txt being two different spectral througputs used as inputs) Strictly speaking, it can just be a single throughput - it should at least be the CCD quantum efficiency. If only a few values of the throughput is known, one can use the interp function to interpolate the missing values, prior to running the setup function.

slit
Size of the slit
Tuple, 1 string, 2 floats. The string is the unit type, the floats are the size of the slit.
[$'unit'$], [*arcsec or pixels*]
Notes: E.g. [$'pix'$, 2, 3.5]. The unit string can be either *ang* or *pix*, corresponding to angular units or pixels. The two following integers should be the size of the slit. The program will use the plate-scale and the pixel size to convert to internal units (pixels) if the slit is given in arcseconds.

psf: OBSOLETE, AS OF THE CURRENT VERSION!
Point Spread Function array
3D array. The entries must be floats. It must have x- and y-dimensions of *img_size*$\times$*sub_pix*. z-dimension is optional.
[*relative intensity, normalised*]
Notes: The user can opt to include several or all wavelengths in the *wl_ran*. If all colours are included, the *psf_col* input is not needed. Creating a simple model PSF can be done with the *psf_maker* function included in the simulator.

(OBSOLETE, as the PSF is handled within the *disperser* function by the astropy.AiryDisk2DKernel function)

disper
Dispersion arrays
2D array, 2 columns. The entries must be integers. The first column contains the x-dispersion per wavelength, the second contains the y-dispersion per wavelength. The array length must be equal to the wavelength span.
[*pixel*], [*pixel*]
Notes: Positions must be relative to the target position if no dispersion was present.

jitter
Spacecraft jitter
2D array, 2 columns. The first column contains the x-coordinate, the other contains the y-coordinate.
[*sub-pixel*], [*sub-pixel*]
Notes: Remember that the jitter should be measured in sub-pixels. Ideally this should in the future be handled by the script, but in the current version this does not work.
If the jitter is specified explicitly by the user, the input *step* is not used. The array length must correspond to the exposure duration.

step
Time steps per second
Integer
[*steps* s^{-1}]
Notes: Only needed if the *jitter* input is not explicitly specified by the user: then the default (generated) jitter will be used.

psf_col: OBSOLETE, AS OF THE CURRENT VERSION!
PSF wavelength list
Array. Must have the same length as z-dimension in the *psf*-input.
[*nm*]
Notes: Only needed if the supplied *psf* does not contain slices equal to the span of *wl_range*. The PSF values between these specified wavelengths will be interpolated, to achieve the correct number of slices.

in_CCD
CCD imperfections
2D array. It must be of the same size as *img_size*×*sub_pix*.
[*relative absorption sub-pixel*$^{-1}$]
Notes: Represents the sub-pixel-level imperfections of the CCD that arises dur-

ing manufacture.

3 Single simulation run

In order to run the simulator a single time, to examine a single observation of a stellar spectrum, a set of input values is needed. An example of such a .py-file is included in the Github repository. In order to simulate a single spectrum being acquired by a detector, the following steps are taken:

- Step 1: Necessary packages are imported. These include *numpy*, *scipy*, and *random*, aside from the file with all the functions and modules necessary (HIDRA.py). For convenience, matplotlib.pyplot can also be imported, so plotting results becomes easier.
- Step 2: Output file is specified and the *setup* function is called. The outputs of the function will be in the following order, and the function is called like so:

  ```
  spec_eff, spec_eff2, jitter, x_j, y_j, img_size,
  sub_pixel, pl_arc_mm, disper, mask, slitpos,
  background = HIDRA.setup(input_file.py)
  ```

  with *spec_eff*'s being spectra used by the disperser later, *jitter* is the jitter image used later, *x_j* and *y_j* are the corresponding jitter arrays for user convenience, *img_size* is the CCD size measured in sub-pixels, *pl_arc_mm* is the plate scale in $arcsec/mm$, *disper* is the dispersion array, *mask* is the slit mask to overlay the image later, *slitpos* is the position of the slit for user convenience, and lastly, background is the background light – which is currently not working.

  Most of these values will be needed as inputs for the *disperser* function later, but some are there only so the user can check certain values.

  - The *setup* function will load in every variable, and check lengths, shapes, etc. of the inputs, so everything should run smoothly.
  - It will also calculate the photon count per wavelength, by taking the input spectra, multiply it by the spectral throughput, the collecting area of the telescope, and then dividing by the number of steps per second.
  - If no jitter is specified, it is possible to generate a set of simplistic jitter arrays, using the *sinusoidal* function the following equation for both directions (x- and y):

$$x_i = A * \cos(f * i - \phi) \tag{1}$$

Where i is the index of the array, x is the position, A is the amplitude, and ϕ is the phase. The function will add several of these values together, with different amplitudes and frequencies – they should be given in lists with different values, so to make a sum of cosines (the code can only handle lists, not single values). A sample list could be [10, 7, 5, 3, 1, 0.1], or similar. You can also randomly generate these input values for the frequency and amplitudes, however you see fit – for example with the *random* package.

After the two jitter arrays have been generated, remember to also run the *jitter_im* function using the x- and y-arrays as inputs to generate a new jitter image, a 2D array with higher values at the centroid position. A quick note on the jitter image, for clarification: Consider an empty array of the same size as the PSF (however big or small it is, measured in sub-pixels). The jitter centroid will "move" around on this image, and every time it stops (every time-step) a value is added to the position (+1). That means, that where the jitter spends most of the time, the value is highest (ideally the center of the image).

- Step 3: The *disperser* function is called, with the respective inputs. Inputs are, in order: *wl_endpoints*, *jit_image*, *psf_ends*, *pos*, *image_size*, *dispersion*, *eff*, *mask_img* While most of the inputs are intuitive, some might not be. Here is a line of code that could be used to call the *disperser* function:

```
image1, image_wl1 = HIDRA.disperser(wl_endpoints=wl_ran,
jit_img=jitter, psf_ends=[15, 45], pos=slitpos,
image_size=img_size, dispersion=disper, eff=spec_eff,
mask_img=mask)
```

  The disperser function can take a few minutes to execute. It works in the following way:

  1. Arrays are created, but empty, to allocate space for them
  2. For-loop starts, and will run for the entire wavelength range
     (a) PSF is generated using the astropy.AiryDisk2DKernel function.
     (b) PSF is folded with the jitter image, using the scipy.signal.convolve2d function. The convolved image is essentially the Fourier transforms of the PSF and the jitter-image multiplied, and shifted back into real space.
     (c) Folded image is added (temporarily) to an empty image.
     (d) Slit mask is multiplied onto this temporary image.
     (e) The image is rolled - this means that the entries are shifted by an amount specified in the *disper* arrays. If there are decimal amounts of dispersion, this can also be handled. The function that handles the shift is the function numpy.roll. In this step, the photon count of the current wavelength is also multiplied.

(f) The temporary image is finally added onto the "real" image, and is then reset.

3. The for-loop can then go to the next wavelength, and do the same process, but this time using a new PSF, the next dispersion values and the color entry in the spectrum.
4. After all colors have been through the for-loop, the results are output as a 2D array with (size is equal to $sub\_pixel \times img\_size$).

- After the *disperser* function has been run, the 2D spectrum can be handled further – it is still measured in sub-pixels. If noise is to be included, it can be done at this time in the process. Afterwards, the pixels should be summed up to form full pixels, and the spectrum should be extracted – for simple cases it might be enough to simply collapse the 2D array in the y-direction. If all of these steps are desired, the convenience function *prep_func* can handle it at the same time. The noise is calculated with the noise2d function. It uses the following equation:

$$N_{i,\,j} = (\sqrt{counts_{i,\,j}} + RON) * \mathcal{N}(0,1) \tag{2}$$

with $N_{i,\,j}$ being the noise of entry i, j, $counts_{i,\,j}$ is the value of the i, j'th entry, RON is the Read-Out Noise of the CCD, and $\mathcal{N}(0,1)$ is a random number drawn from a normal distribution, between 0 and 1.

The noise and the output of *disperser* are added together, and they are then read out using the aptly named *read_out* function (which should only be used for simple cases). The function simply collapses a 2D array, column-wise. After this, the spectrum can be examined by simply plotting it. An example of this can be seen in figure **??**

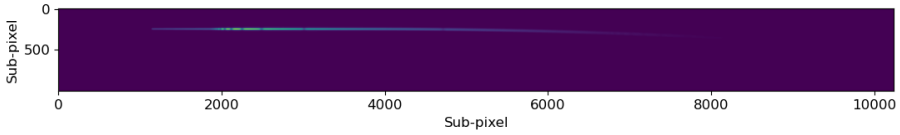

Figure 1: Result of a single simulation using the sample values, immediately after the *disperser* function. Note the large values of the axes, a result of the array being measured in sub-pixels.

4 Simulating transmission spectrum of transiting exoplanet

It is wholly possible to simulate the transmission spectrum arising from an exoplanet transiting it's host star. This can be done relatively simply by running

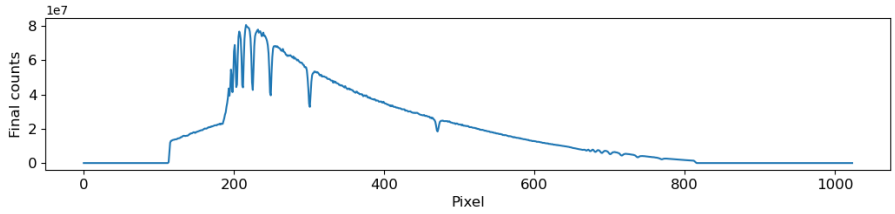

Figure 2: Result of a single simulation using the sample values, after being processed with the *prep_func* function.

the simulator twice, once for two different input spectra – one *without* the planet in transit, and one *with* the planet in transit. The code used to do this, might look similar to the following, with the two simulation already complete:

```
# Two simulations are complete, the result is image1 (no planet) and
image2 (yes planet), that have been processed by the prep_func as well

ro = image1 # ro = read-out of image Out of transit
ri = image2 # ri = --"-- In transit
no = HIDRA.noise(size=ro.shape, image=ro) # generate noise to add
ni = HIDRA.noise(size=ri.shape, image=ri)
ri = ri+ni
ro = ro+no
ro, ri, wave, delta = HIDRA.transmission_spec_func(image=ro, image2=ri,
sub_pixel=sub_pixel, wl_ran=inp.wl_ran, disper=disper, slitpos=slitpos,
img_size=img_size, move="y", noiseinp="n") #
plt.plot(wave, (ro-ri)/ro)
```

5 Selected docstrings

interp

```
Interpolates values between given input table values.
It is built upon the scipy.interp1d function.

Parameters
----------
x : array_like
 Input array of x values, eg. wavelength
 Ex: np.array([100, 217, 350])
y : array_like
 Input array of y values, eg. quantum efficiency, or mirror reflectance.
```

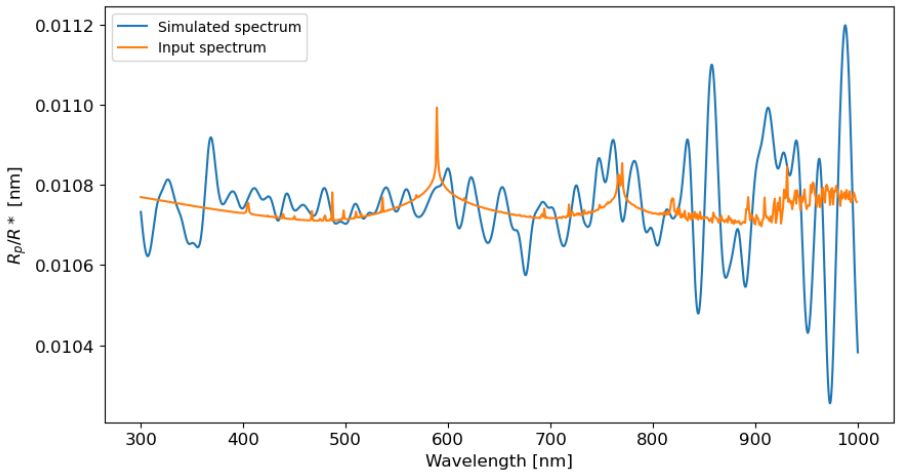

Figure 3: The resultant transmission spectrum. Evidently, this setup seems unlikely to detect the atmosphere of this planet. To truly tell though, you should repeat the simulations, using a different jitter each time, and use all observations in the data-processing.

```
 Ex: np.array([0.1, 0.7, 0.85])
wl_ran : tuple
 wavelength span. Entries must be integers
delta_lambda : float
 wavelength resolution, in nm.
kind : string
 type of interpolation. Valid options are 'linear', 'quadratic' and 'cubic'.
lowlim : float
 lower wavelength limit. Below this value, throughput will be set to 0
uplim : float
 upper wavelength limit. Above this value, thoughput will be set to 0

 Returns
 -------
interpolated : array_like
  Interpolated values between wl_start and wl_stop, with sharp cutoff
  beyond the specified limits.

 Notes
 -----
Check https://docs.scipy.org/doc/scipy/reference/generated/
scipy.interpolate.interp1d.html for more information about the
interpolator from scipy.
```

sinusoidal

```
Function to generate sinusoidal jitter

Parameters
----------
size : int
 Size of the desired output array.
frequency : list
 List with frequencies
amplitude : list
 List of amplitudes. Must be the same size as the frequency list
phase : float
 Phase of the sinusoidal.

Returns
-------
x : array
 Jitter of a sinusoidal nature, co-added j-times, with j being
 the number of entries in the frequency and amplitude lists.
```

noise2d

```
Function to generate noise of a 2D array.

Parameters
----------
x : array
 Input array. Could be a simulated spectrum
RON : float
 Read-out noise of the CCD, measured in photo-electrons. The default is 5.

Returns
-------
noise : array
Noise-array, same dimensions as the input array. The two can then be added
together for the "actual image"

The noise is calculated with the \hyperlink{noise2d}{noise2d} function.
It uses the following equation:

N_{i,j} = [sqrt(counts_{i,j})+RON]*\mathcal{N}(0,1)

with N_{i,j} being the noise of entry i,j, counts_{i,j} is the value of the
i,j'th entry, RON is the Read-Out Noise of the CCD, and \mathcal{N}(0,1) is
a random number drawn from a normal distribution, between 0 and 1.
```