Rediscovering `SEDNA`

We track the orbit of `Sedna` at the times of observations and select the observations where we expect `Sedna` to be within frame. Then we compute the shifted `RA/DEC` adjustments and stack the data and look in the area where we expect `Sedna` to be (in respect to the earliest frame).

Load Data

In [1]:
```
import ephem
import numpy as np
import astropy.units as u
import pandas as pd
from astropy.coordinates import SkyCoord
from astropy.coordinates import match_coordinates_sky
from tqdm import tqdm
import matplotlib.pyplot as plt
%matplotlib inline
from astroquery.mpc import MPC
import os
from tqdm import tqdm
from astropy.io import fits
from astropy.time import TimeDelta, Time
import torch
import numpy as np
from astropy.io import fits
import matplotlib.pyplot as plt
from astropy.visualization import (ImageNormalize, ZScaleInterval, LogStretch, MinMaxInterval, AsinhStretch)
from astropy.wcs import WCS
from scipy.ndimage import shift
from astropy.coordinates import SkyCoord
```

Read the frames that have been aligned to the earliest frame in the dataset.

In [2]:
```
directory = os.fsencode('aligned')
files = []
for sub1 in os.listdir(directory):
    sub1_name = os.fsdecode(sub1)
    if '.fits'in sub1_name:
        files.append('aligned'+"/"+sub1_name)
```

Obtain the maximum Magnitude Zero point from the files

In [3]:
```
MAGZPL_LIST = []
FWHM_list = []
for i in tqdm(range(len(files))):
    # Open the FITS file
    filename = files[i]
    hdul = fits.open(filename)
    # Access the primary HDU (Header Data Unit)
    primary_hdu = hdul[0]
    # Get the data and header
    header = primary_hdu.header
    MAGZPL_LIST.append(header['MAGZP'])
    FWHM_list.append(header['SEEING'])
    hdul.close()
MAGZPL = np.max(MAGZPL_LIST)
print(MAGZPL)
```

```
100%|██████████████████████████████████████████████████| 253/253 [00:00<00:00, 1210.91it/s]
26.340047
```

Load the data and apply the following preprocessing steps:

- Open data
- Remove pixels that are 1/2 of the max saturated pixels (replace with Nan)
- Subtract the median of the sky off (non Nan values) calculated from the entire sq deg frame
- scale the flux based on magnitude as the highest magnitude zero point.

In [4]:
```
total_data = {}
filenames = {}
for i in tqdm(range(len(files))):
    # Open the FITS file
    filename = files[i]
    hdul = fits.open(filename)
    # Access the primary HDU (Header Data Unit)
    primary_hdu = hdul[0]
    # Get the data and header
    data = primary_hdu.data
    header = primary_hdu.header
    # ====== make sure to keep only images that are NOT all NANs
    time = header['SHUTOPEN']
    time = Time(time.replace('T',' '), format='iso', scale='utc')
    # ========== step 1: remove saturated pixels ==========
    saturate = header['SATURATE']
    data = np.where(data > saturate/2, np.nan, data)
    # =========  step 2: subtract the median (sky subtraction) =======
    median = np.nanmedian(data)
    data = data - median
    # =========  step 3: scale flux =============
    MAGZP = header['MAGZP']
    scale = 10**(0.4*(MAGZPL - MAGZP))
    data = data * scale
    header = primary_hdu.header
    total_data[time] = torch.tensor(data.astype(np.float32))
    filenames[time] = filename
    hdul.close()

dates = list(total_data.keys())
```

```
100%|██████████████████████████████████████████████████| 253/253 [00:28<00:00,  8.73it/s]
```

Simulate `Sedna` Orbit

We then simulate Sedna's orbit.

In [5]:
```
result = MPC.query_object('asteroid', name='sedna')[0]
```

In [6]:
```
## Establish the observer at Palomar Mountain
Observer = ephem.Observer()

# Palomar
Observer.lat = "33.3563"
Observer.lon = "-116.8650"
Observer.elevation = 1872
Observer.epoch= ephem.J2000

# Create an EllipticalBody for Sedna
obj = ephem.EllipticalBody()

# Set the orbital elements
obj.name = "Sedna"
obj._a = float(result["semimajor_axis"])  # Semi-major axis (AU)
obj._e = float(result["eccentricity"])    # Eccentricity
obj._inc =  float(result["inclination"])  # Inclination (degrees)
obj._om = float(result["argument_of_perihelion"])  # Argument of perihelion (degrees) = varpi - Omega
obj._Om = float(result['ascending_node'])  # Longitude of ascending node (degrees)
obj._M = float(result['mean_anomaly'])    # Mean anomaly (degrees)
obj._epoch = ephem.J2000  # Epoch (e.g., '2000')
obj._epoch_M = result['epoch'].replace("-","/")  # Epoch (e.g., '2000')
```

```
# Compute position for a specific date and time
Observer.date = '2021/11/05 06:38:51.072'
obj.compute(Observer)
print(np.degrees(obj.g_ra), np.degrees(obj.g_dec))

c = SkyCoord(ra=np.degrees(obj.a_ra)*u.degree, dec=np.degrees(obj.a_dec)*u.degree)
c.to_string('hmsdms')
```

```
59.29711253259594 8.150705163885599
```

Out[6]: '03h56m00.36838108s +08d05m12.58001173s'

In [7]:
```
sedna_orbit = pd.DataFrame([{"a": obj._a, "e": obj._e, "inc": np.degrees(obj._inc), "varpi": np.degrees(obj._om + obj._Om),
                             "Omega": np.degrees(obj._Om), "M": np.degrees(obj._M)}])
sedna_orbit
```

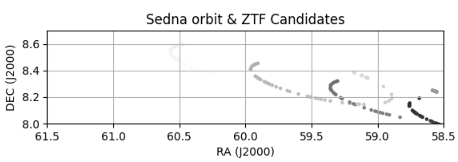
Out[7]:

	a	e	inc	varpi	Omega	M
0	552.160583	0.861808	11.92774	455.268753	144.394699	358.594452

In [8]:
```
# Lists to store RA and Dec values
ra_list = []
dec_list = []
lat_list = []
lon_list = []
dstr = []
dist = []
jd_dates = []
# Compute RA and Dec for each date
for date in dates:
    jd_dates.append(date.jd)
    Observer.date = date.to_datetime()
    obj.compute(Observer)
    ra_list.append(obj.a_ra)
    dec_list.append(obj.a_dec)
    # print(obj.a_ra, obj.a_dec)
    lat_list.append(obj.hlat)
    lon_list.append(obj.hlon)
    dist.append(obj.earth_distance)
    dstr.append(str(ephem.Date(date.to_datetime())))


# Convert RA and Dec to degrees
ra_degrees = np.array([ephem.degrees(ra) * 180 / np.pi for ra in ra_list])
dec_degrees = np.array([ephem.degrees(dec) * 180 / np.pi for dec in dec_list])



earliest = 2458803.500000 # nov 16th 2019 00:00:00
latest = 2460264.500000 # nov 16th 2023 00:00:00
```

In [9]:
```
import gc
fig, ax = plt.subplots()
ax.scatter(ra_degrees, dec_degrees, s=5, c=jd_dates, cmap='gray', )
ax.grid(True)
ax.set_aspect("equal")
ax.set_xlabel("RA (J2000)")
ax.set_ylabel("DEC (J2000)")
ax.set_title("Sedna orbit & ZTF Candidates")

ax.set_xlim(61.5,58.5)
ax.set_ylim(8.0,8.7)
del fig, ax
gc.collect()
```

Out[9]: 15



Shift Stack

We use effectively the same shift-stack GPU algorithm except instead of working in pixel velocity, I am working in absolute shifts in pixel space. This is because the orbits already tell us the expected `RA/DEC` positions and thus easier to manipulate

In [10]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from time import time

class Shift_Stack(nn.Module):
    def __init__(self, frames):
        super(Shift_Stack, self).__init__()
        self.frames = self.frame_subtraction(frames).cuda()[:,None,:,:]
        self.B = self.frames.shape[0] # predefine batch sizes (number of frames)
        self.grid = F.affine_grid(
            torch.eye(2, 3).unsqueeze(0).repeat(self.B, 1, 1),  # Identity matrix for rotation and scaling
            size=self.frames.size(),  # Output size
            align_corners=False
        ).cuda()

    # @torch.jit.script
    def frame_subtraction(self, frame):
        """
        first frame subtractions
        """
        frame = frame - frame[0,:,:]
        non_nan_values = frame[~frame.isnan()]
        self.adjustment = abs(torch.min(non_nan_values))
        self.eps = 1e-3
        frame += abs(self.adjustment ) +self.eps
        non_nan_values = frame[~frame.isnan()]
        assert torch.min(non_nan_values) >= 0.0
        return frame

    def shift_images(self, images, shifts):
        """
        Shifts a batch of images by specified (x, y) values for each image.

        images (torch.Tensor): A batch of images of shape (B, C, H, W),
                               where B is batch size, C is number of channels,
                               H is height, and W is width.
        shifts (torch.Tensor): A tensor of shape (B, 2), where each row
                               represents the (B, x, y) shift for the corresponding image.
        """
        B, C, H, W = images.shape
        shifts_x, shifts_y = shifts[:, 0], shifts[:, 1]
        # Create a grid for affine transformations
        start = time()
        grid = self.grid.clone()
        # Adjust the grid by the shifts
        grid[..., 0] += 2 * shifts_x.view(-1, 1, 1) / W  # Normalize shift in x direction
        grid[..., 1] += 2 * shifts_y.view(-1, 1, 1) / H  # Normalize shift in y direction
        start = time()
        # Perform the grid sampling to shift the images
        shifted_images = F.grid_sample(
            images, grid, mode='bilinear', padding_mode='zeros', align_corners=False
        ).cuda()
        shifted_images[shifted_images == 0] = float('nan')
        shifted_images -= abs(self.adjustment +self.eps)
```

```
        return shifted_images

    def forward(self, dxdy):
        """
        Preprocessing handing before calling shift_images
        """
        dx, dy = dxdy[:,0], dxdy[:,1]
        shifts_x = dx # pixel shift in x
        shifts_y = dy # pixel shift in y
        shifts = torch.stack([shifts_x, shifts_y] , axis = 1)
        frames = self.frames
        batched_shift = self.shift_images(frames, shifts)
        batched_shift = batched_shift[:,0,:,: ] # meant for future multi-filters... [curr. index zero for 1 filter]
        stacked = torch.nanmedian(batched_shift, axis = 0)
        return stacked
```

Sort the `dates` the `ra` and `dec` lists together to prepare for stacking and to calculate the relative shifts needed.

In [11]:
```
def sort_lists(list1, list2, list3):
  """
  Sorts list1 and updates list2 and list3 to maintain the same order.
  """

  # Create a list of tuples, pairing elements from all three lists
  combined = list(zip(list1, list2, list3))

  # Sort the list of tuples based on the first element of each tuple
  combined.sort(key=lambda x: x[0])

  # Unzip the sorted tuples back into separate lists
  sorted_list1, sorted_list2, sorted_list3 = zip(*combined)

  return list(sorted_list1), list(sorted_list2), list(sorted_list3)


sorted_keys, ra_reorder, dec_reorder = sort_lists(list(total_data.keys())[:], ra_degrees[:], dec_degrees[:])
```

Filter for frames that *actually* contain `SEDNA` by checking that the `RA/DEC` is within frame

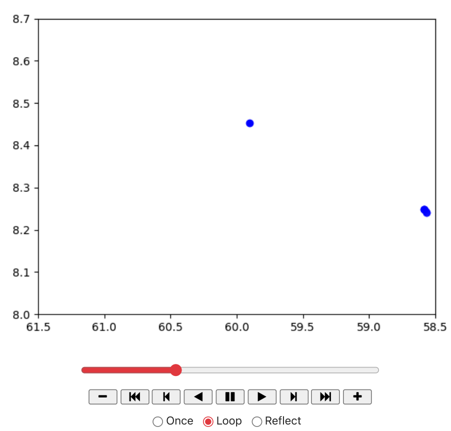
In [12]:
```
for i in range(len(ra_reorder)):
    if ra_reorder[i] > 58.8 and ra_reorder[i]<59.40:
        break
print(i)
cut = i
sorted_keys, ra_reorder, dec_reorder = sorted_keys[cut:], ra_reorder[cut:], dec_reorder[cut:]
earliest_frame = sorted_keys[0]
```

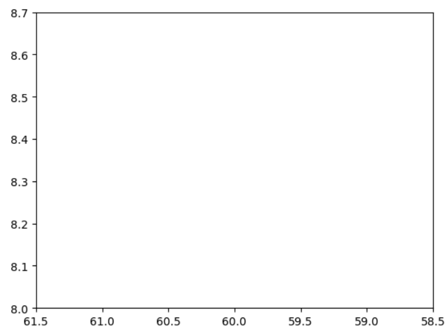
119

Visualise the expected orbit path of `SEDNA` at the times of observation.

In [13]:
```
from astropy.io import fits
import matplotlib.pyplot as plt
from astropy.visualization import (ImageNormalize, ZScaleInterval, LogStretch, MinMaxInterval, AsinhStretch)
import numpy as np
import matplotlib.pyplot as plt
from celluloid import Camera
from IPython.display import HTML

norm = ImageNormalize(stretch=AsinhStretch(a=0.0001), interval=ZScaleInterval())

%matplotlib inline
# Set up the figure and axes
fig, ax = plt.subplots()

# Initialize Celluloid Camera
camera = Camera(fig)


data = total_data[earliest_frame]
count = 0
# Animate the sine wave with a changing phase
for i in range(10,dec_degrees.shape[0]-cut):
    for k in range(10):
        ax.scatter(ra_reorder[i-k], dec_reorder[i-k], alpha=1-0.1*k, color='blue')
    ax.set_xlim(61.5,58.5)
    ax.set_ylim(8.0,8.7)
    camera.snap()  # Capture the frame


# Create the animation
animation = camera.animate(interval=100)  # 100ms between frames

# Display the animation
HTML(animation.to_jshtml())  # Render as JS HTML
```

Out[13]:





Load up the sorted dates to order the frames

In [14]:
```
frames = []
frame_times = []
adjusts = []

max_frames = 150
for i, key in enumerate(sorted_keys):
    if i < max_frames:
        frames.append(total_data[key])
        frame_times.append(key.to_value('jd'))
frames = torch.stack(frames)
frame_times = torch.tensor(frame_times)
print(frames.shape, frame_times.shape)
```

```
torch.Size([134, 3080, 3072]) torch.Size([134])
```

Calculate the shift and stack needed for each frame in respect to the EARLIEST frame

In [15]:
```
# convert arcsecond / pixel ==> degree / pixel
degrees_per_pixel = header['PIXSCALE']/ 3600

dx = [0]
dy = [0]
for i, key in enumerate(sorted_keys):
    if i < max_frames and i > 0:
        dx.append((ra_reorder[0] - ra_reorder[i])/ degrees_per_pixel)
        dy.append((dec_reorder[0] - dec_reorder[i])/ degrees_per_pixel)
shifts = torch.zeros((frames.shape[0], 2)).cuda()
shifts[:, 0] = torch.tensor(dx)
shifts[:, 1] = torch.tensor(dy)
print(shifts.shape)
```

```
torch.Size([134, 2])
```

Load the frames into memory

In [16]:
```
stacking_job = Shift_Stack(frames)
```

Execute the job.

In [17]:
```
from time import time
start = time()
result = stacking_job(shifts)
print(time()-start, "runtime [s]")
```

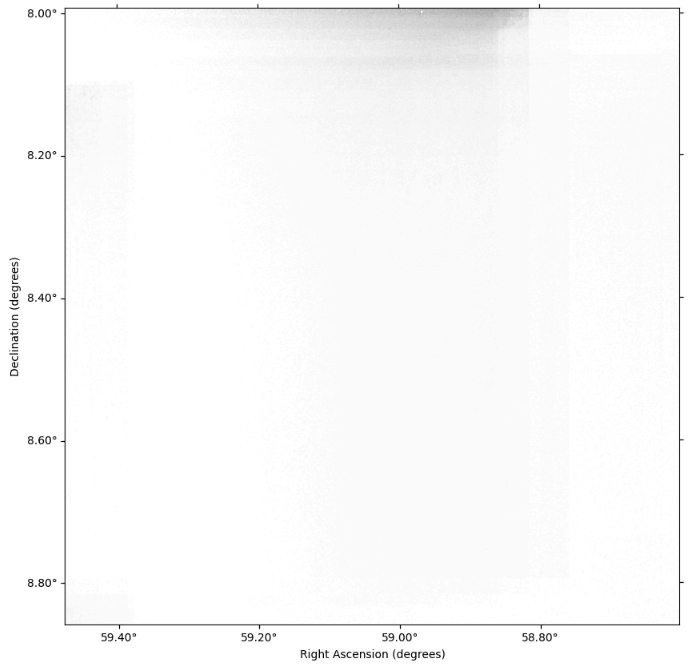
```
0.28650975227355957 runtime [s]
```

Result

Below is a visualisation of the full 1sq degree frame added together

In [18]:
```
import matplotlib.pyplot as plt
from astropy.wcs import WCS
from astropy.io import fits

ind = 0
print(ra_reorder[ind], dec_reorder[ind])
# Load the FITS image
hdu = fits.open(filenames[sorted_keys[ind]])[0]
wcs = WCS(hdu.header)

# Create a figure and axes with the WCS
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(projection=wcs)

res = result.values.cpu().numpy()
# x, y = 2000, 500
# res[x:x+10, y:y+10] = 1e10
# Display the image
ax.imshow(res, origin='lower', cmap='gray_r', vmax = 100, vmin=-0)

# Set RA and Dec labels in degrees
ax.coords[0].set_major_formatter('d.dd')  # RA in degrees with 2 decimal places
ax.coords[1].set_major_formatter('d.dd')  # Dec in degrees with 2 decimal places
ax.coords[0].set_axislabel('Right Ascension (degrees)')
ax.coords[1].set_axislabel('Declination (degrees)')
# Set the x-axis limits using pixel coordinates
# Show the plot

plt.show()
```

```
59.30358301894467 8.318980375576833
```

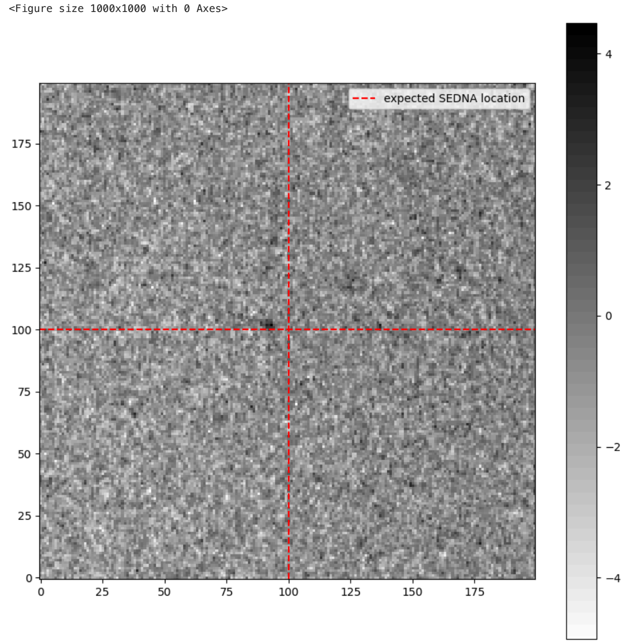


Rooming on To `SEDNA` 's expected location

We inspect the frame indexed to where Sedna should be.

In [19]:
```
import matplotlib.pyplot as plt
from astropy.wcs import WCS
from astropy.io import fits

ind = 0
print('integration expects SEDNA to be at: ',ra_reorder[ind], dec_reorder[ind])
print("time", sorted_keys[ind])
# Load the FITS image
hdu = fits.open(filenames[sorted_keys[ind]])[0]
wcs = WCS(hdu.header)
pixel_coords = wcs.all_world2pix([[ra_reorder[ind], dec_reorder[ind]]], 0)[0]

# Create a figure and axes with the WCS
fig = plt.figure(figsize=(10, 10))


x, y = int(pixel_coords[1]), int(pixel_coords[0])
res = result.values.cpu().numpy()[ x-100:x+100, y-100:y+100]

# Create the plot
plt.figure(figsize=(10,10))
plt.imshow(res, origin='lower', cmap='gray_r')
# Draw vertical lines
plt.axvline(100, color='red', linestyle='--')
plt.colorbar()
# Draw horizontal lines
plt.axhline(100, color='red', linestyle='--', label="expected SEDNA location")
plt.legend()
plt.show()
```

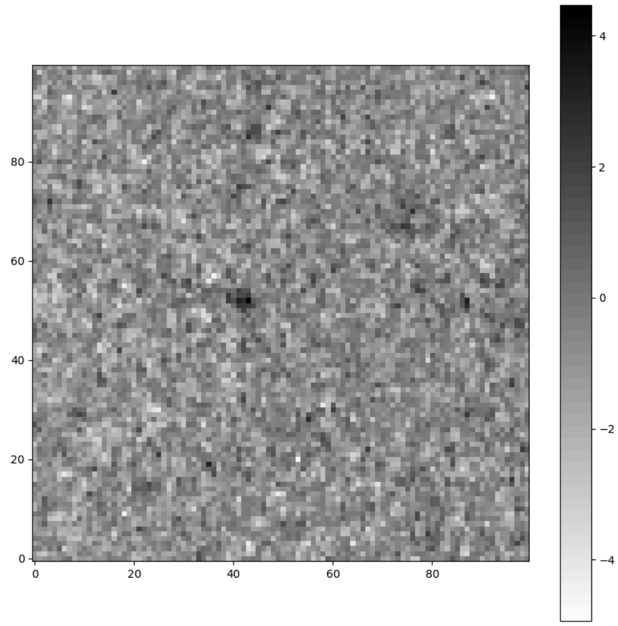
```
integration expects SEDNA to be at:  59.30358301894467 8.318980375576833
time 2021-08-08 11:09:40.607
<Figure size 1000x1000 with 0 Axes>
```



In [20]:
```
import matplotlib.pyplot as plt
from astropy.wcs import WCS
from astropy.io import fits
```

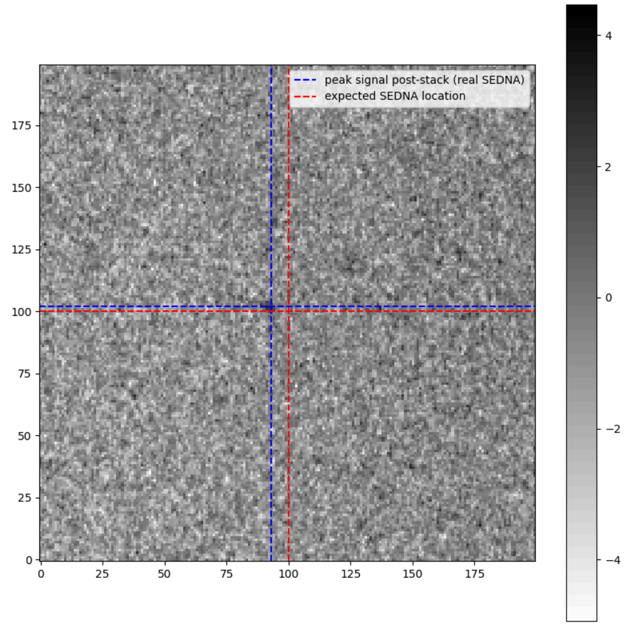
```
ind = 0
print('integration expects SEDNA to be at: ',ra_reorder[ind], dec_reorder[ind])
# Load the FITS image
hdu = fits.open(filenames[sorted_keys[ind]])[0]
wcs = WCS(hdu.header)
pixel_coords = wcs.all_world2pix([[ra_reorder[ind], dec_reorder[ind]]], 0)[0]

# Create a figure and axes with the WCS
fig = plt.figure(figsize=(10, 10))

x, y = int(pixel_coords[1]), int(pixel_coords[0])
res = result.values.cpu().numpy()[ x-50:x+50, y-50:y+50]

# Create the plot
plt.figure(figsize= (10,10))
plt.imshow(res, origin='lower', cmap='gray_r')
# Draw vertical lines
plt.colorbar()
plt.show()
```

```
integration expects SEDNA to be at:  59.30358301894467 8.318980375576833
<Figure size 1000x1000 with 0 Axes>
```



In [24]:
```
import matplotlib.pyplot as plt
from astropy.wcs import WCS
from astropy.io import fits

ind = 0
print('integration expects SEDNA to be at: ',ra_reorder[ind], dec_reorder[ind])
# Load the FITS image
hdu = fits.open(filenames[sorted_keys[ind]])[0]
wcs = WCS(hdu.header)
pixel_coords = wcs.all_world2pix([[ra_reorder[ind], dec_reorder[ind]]], 0)[0]

# Create a figure and axes with the WCS
fig = plt.figure(figsize=(10, 10))


x, y = int(pixel_coords[1]), int(pixel_coords[0])
res = result.values.cpu().numpy()[ x-100:x+100, y-100:y+100]
# Convert to a NumPy array for easier manipulation

# Get the index of the maximum value
max_index = np.unravel_index(np.argmax(res), res.shape)
# Create the plot
plt.figure(figsize=(10,10))
plt.imshow(res, origin='lower', cmap='gray_r')
# Draw vertical lines
plt.axvline(100, color='red', linestyle='--')
plt.axhline(max_index[0], color='blue', linestyle='--')
plt.axvline(max_index[1], color='blue', linestyle='--', label="peak signal post-stack (real SEDNA)")

# Draw horizontal lines
plt.axhline(100, color='red', linestyle='--', label="expected SEDNA location")
plt.colorbar()
plt.legend()
plt.show()
```

```
integration expects SEDNA to be at:  59.30358301894467 8.318980375576833
<Figure size 1000x1000 with 0 Axes>
```

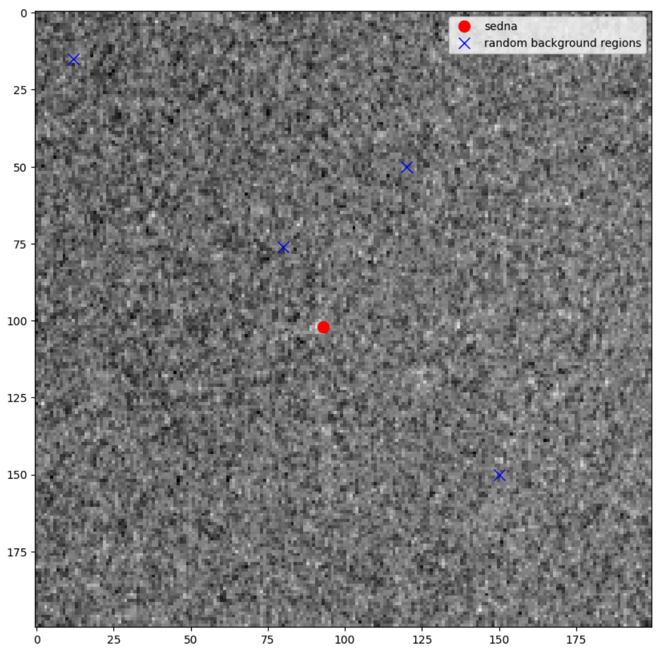


Calculate `SEDNA`'s SNR

In [22]:
```
import numpy as np
from astropy.io import fits
from astropy.stats import sigma_clipped_stats
from astropy.stats import sigma_clipped_stats
from photutils.aperture import CircularAperture, CircularAnnulus, aperture_photometry
from photutils.background import Background2D, MedianBackground
```

In [25]:
```
import numpy as np
from astropy.io import fits
from astropy.wcs import WCS
import matplotlib.pyplot as plt

def calculate_snr(image_data, star_x, star_y, fwhm, background_regions):
  """
  Calculates the SNR of a star in an image.
  """

  # Calculate aperture radius
  aperture_radius = 6.0/5.0 * fwhm

  # Extract star aperture
  y, x = np.ogrid[-aperture_radius:aperture_radius+1, -aperture_radius:aperture_radius+1]
  dist_from_center = np.sqrt(x**2 + y**2)
  # cut out everything that is not in the star
  star_aperture = dist_from_center <= aperture_radius
  star_flux = np.sum(image_data[int(star_y-aperture_radius):int(star_y+aperture_radius+1),
                               int(star_x-aperture_radius):int(star_x+aperture_radius+1)][star_aperture])

  # Calculate background flux and RMS
  background_values = []
  for bg_x, bg_y in background_regions:
    bg_aperture = image_data[int(bg_y-aperture_radius):int(bg_y+aperture_radius+1),
                             int(bg_x-aperture_radius):int(bg_x+aperture_radius+1)][star_aperture]
    background_values.extend(bg_aperture.flatten())
  background_mean = np.median(background_values)
  background_rms = np.std(background_values)

  # Calculate SNR
  signal = (star_flux - background_mean)
  noise = np.sqrt(background_rms**2+signal**2)
  snr = signal / noise

  return snr

image_data = res

# Example values (replace with your actual data)
star_x = max_index[1]
star_y = max_index[0]
fwhm = np.mean(FWHM_list)
background_regions = [(12, 15), (120, 50), (80, 76), (150, 150)]  # Example background regions

# Calculate SNR
snr = calculate_snr(image_data, star_x, star_y, fwhm, background_regions)
print(f"SNR of the star: {snr:.2f}?")

plt.figure(figsize=(10,10))
plt.imshow(image_data, cmap='gray')
plt.plot(star_x, star_y, 'ro', markersize=10, label="sedna")  # Plot star position
count = 0
for bg_x, bg_y in background_regions:
    if count ==0:
        plt.plot(bg_x, bg_y, 'bx', markersize=10, label="random background regions")  # Plot background region centers
        count +=1
    else:
        plt.plot(bg_x, bg_y, 'bx', markersize=10)  # Plot background region centers
plt.legend()
plt.show()
```

```
SNR of the star: 1.00?
```


sedna
random background regions
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175