Source finding demo

Initialisation

imports:

In [1]:

```
%matplotlib ipympl
from matplotlib import pyplot as plt
import numpy as np
from astropy.io import fits
from numba import njit
```

Read data

Please *choose one* of the following:

images:

In [2]:

```
hdu = fits.open('data/CIG_335.fits')
data = hdu[0].data[3000:4000, 1500:2500] * 1.  # to make sure it's converted to fl
```

In [3]:

```
hdu = fits.open('data/hcg44_cube_R.fits')
data = hdu[0].data[69]*1.
```

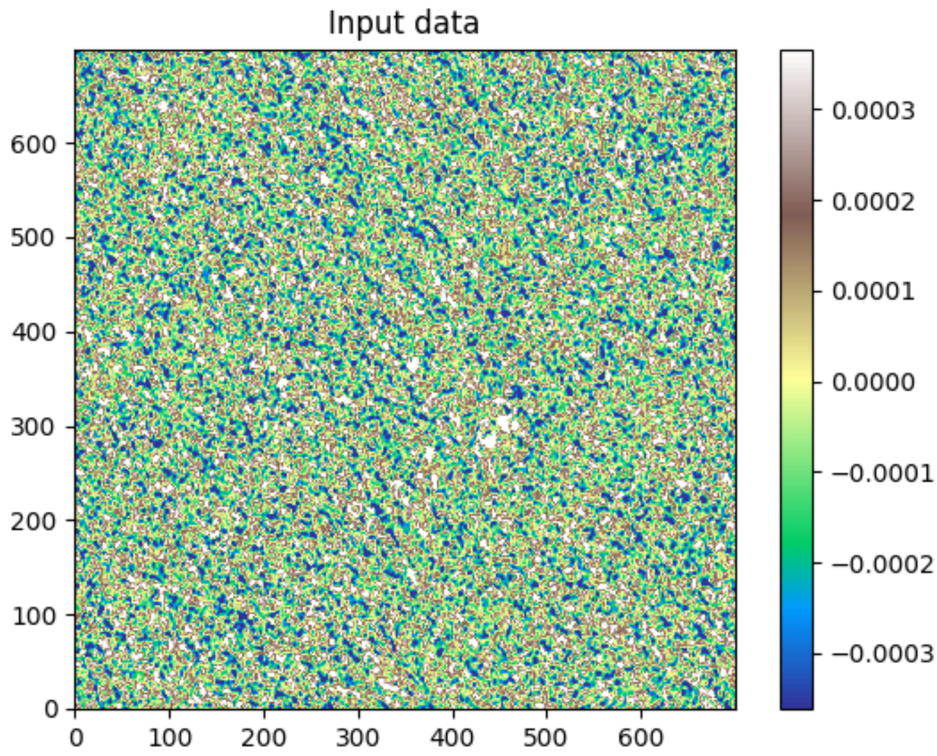
```
WARNING: VerifyWarning: Invalid 'BLANK' keyword in header.  The 'BLANK' keyword is
only applicable to integer data, and will be ignored in this HDU. [astropy.io.fit
s.hdu.image]
```

results:

In [4]:

```
# fig, ax = fm.select('data', nrows=1, ncols=3, sharex=True, sharey=True, figsize=
#fig, ax = fm.select('data')
plt.close('data')
fig = plt.figure('data')
ax = fig.subplots()

im = ax.imshow(data,
               interpolation='nearest', origin='lower', cmap='terrain',
               vmin=np.nanpercentile(data, 5), vmax=np.nanpercentile(data, 95),
              )
ax.set_title('Input data')
cb = fig.colorbar(im, ax=ax)
```

data



Find threshold

cumulative mass:

In [5]:
```
sorted_by_data = np.argsort(data.flatten())
sorted_data = data.flatten()[sorted_by_data]
```

Allow for statistical weights:

In [6]:
```
weights = np.ones_like(data)
sorted_weight = weights.flatten()[sorted_by_data]
```

In [7]:
```
cumulative_mass = np.nancumsum(sorted_weight)
sorted_weight /= cumulative_mass[-1]
cumulative_mass /= cumulative_mass[-1]
```

density peak (mode of the probability distribution):

Here we have a couple of free parameters:

In [8]:
```
delta = .5  # relative size of the smoothing kernel (delta <= 1)

# mass fractions to be tested:
m = np.linspace(delta**2,  # somewhat arbitrary :^(
                1/(1+delta),  # do not overshoot
                int(np.sqrt(data.size)) # a reasonable number of trials
               )
```

Find maximum:

In [9]:

```
x_top = np.interp((1+delta)*m, cumulative_mass, sorted_data)
x_mid = np.interp(m, cumulative_mass, sorted_data)
x_bot = np.interp((1-delta)*m, cumulative_mass, sorted_data)
rho_top = delta * m / (x_top - x_mid)
rho_bot = delta * m / (x_mid - x_bot)
peak = np.nanargmin((rho_top - rho_bot) ** 2)
data_mode = x_mid[peak]

m_background = 2 * m[peak]
m_signal = 1 - m_background
threshold_guess = np.interp(m_background, cumulative_mass, sorted_data)
```

purity:

In [10]:

```
m_above = 1 - cumulative_mass
m_symmetric_above = np.interp(2 * data_mode - sorted_data, sorted_data, cumulative_
left = np.where(sorted_data < data_mode)
m_symmetric_above[left] = m_background - cumulative_mass[left]
m_signal_above = m_above - m_symmetric_above
purity = m_signal_above / (m_above+1e-30)
purity[-1] = 1
```

results:

In [11]:

```
print(f'data_mode = {data_mode}')
print(f'threshold_guess = {threshold_guess}, m_signal = {m_signal}')
```

```
data_mode = -2.981189221007796e-06
threshold_guess = 0.000523295722409473, m_signal = 0.010014306151645225
```

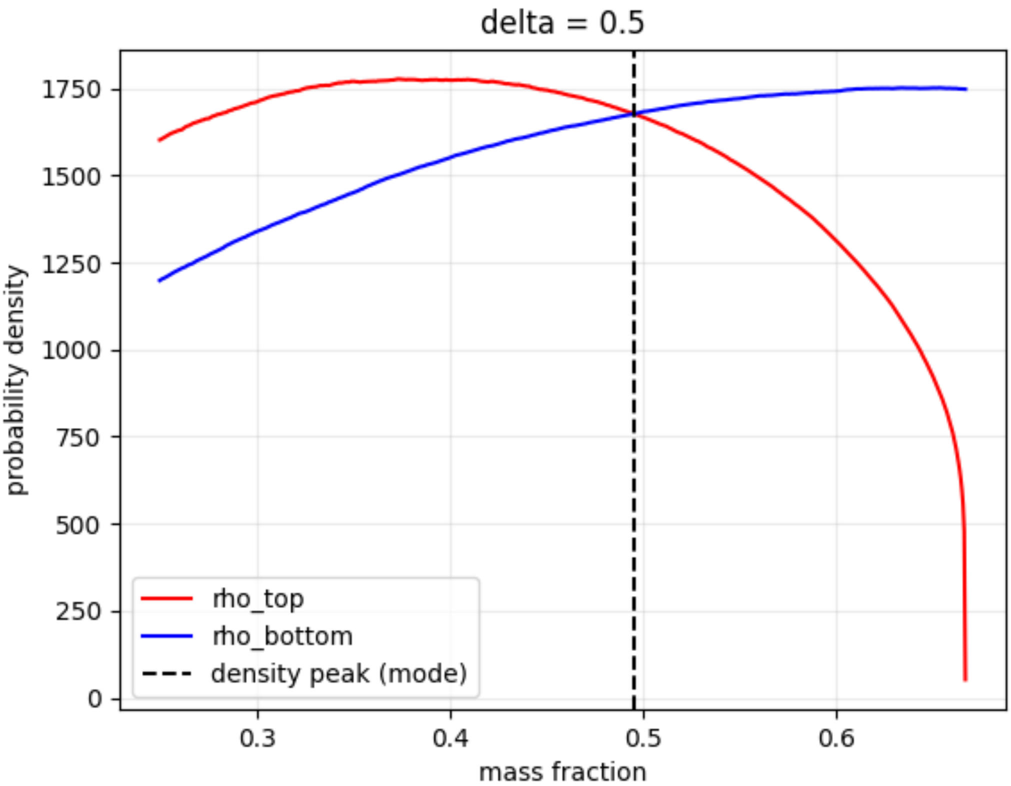
In [12]:

```
plt.close('density')
fig = plt.figure('density')
ax = fig.subplots()

ax.plot(m, rho_top, 'r-', label='rho_top')
ax.plot(m, rho_bot, 'b-', label='rho_bottom')
ax.axvline(m[peak], c='k', ls='--', label='density peak (mode)')

ax.grid(alpha=.25)
ax.legend()
ax.set_xlabel('mass fraction')
ax.set_ylabel('probability density')
ax.set_title(f'delta = {delta}')
```

Out[12]:

```
Text(0.5, 1.0, 'delta = 0.5')
```

density

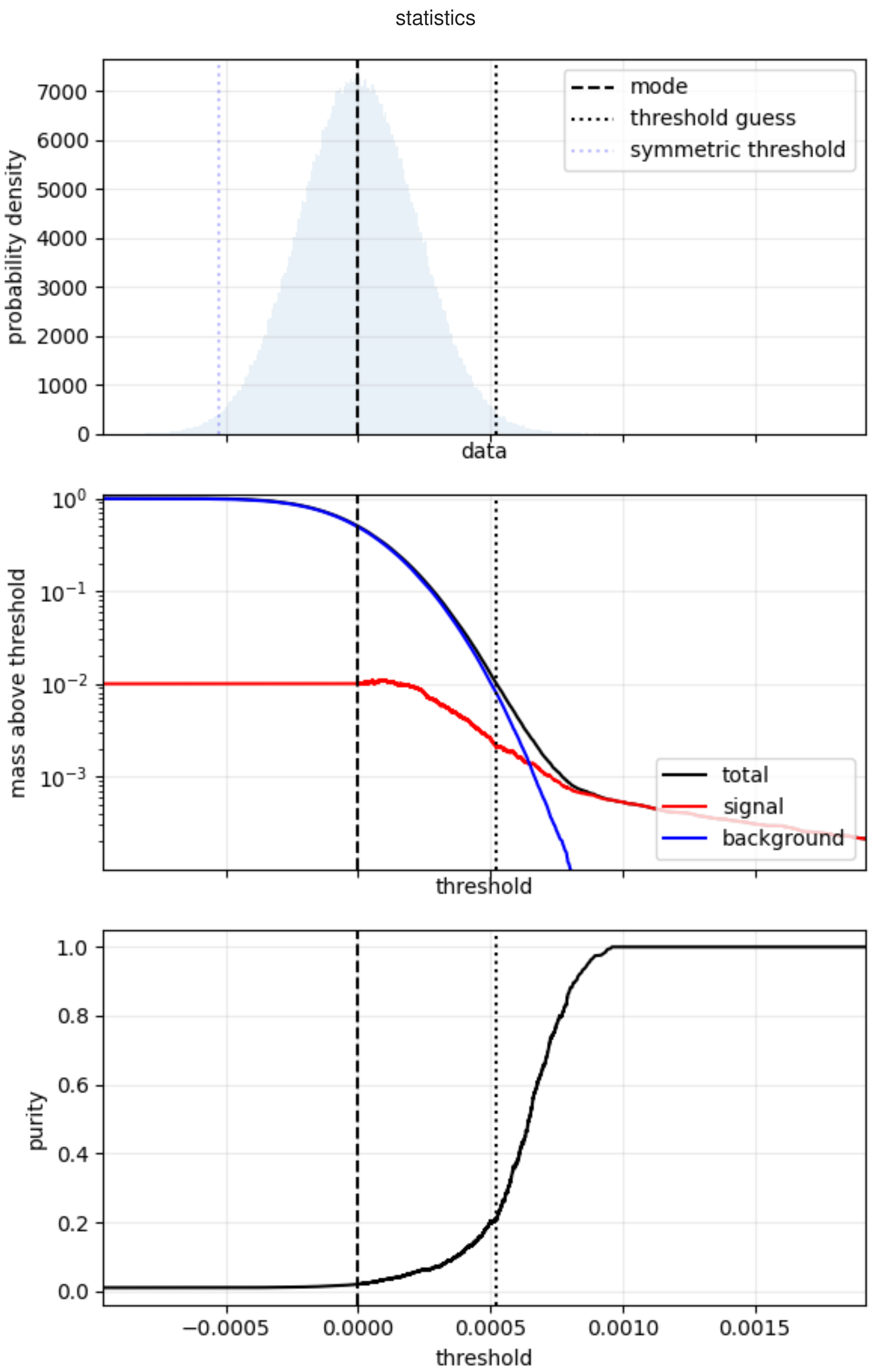

delta = 0.5
probability density
1750
1500
1250
1000
750
500
250
0
0.3
0.4
0.5
0.6
mass fraction
rho_top
rho_bottom
density peak (mode)

In [13]:

```
L = max(data_mode-sorted_data[0], threshold_guess-data_mode)

plt.close('statistics')
fig = plt.figure('statistics', figsize=(6, 9))
fig.set_tight_layout(True)
ax = fig.subplots(nrows=3, sharex=True)

# density:

ax[0].hist(sorted_data, bins=np.linspace(sorted_data[0], data_mode + 5*L, int(np.s
ax[0].axvline(data_mode, c='k', ls='--', label='mode')
ax[0].axvline(threshold_guess, c='k', ls=':', label='threshold guess')
ax[0].axvline(2*data_mode-threshold_guess, c='b', ls=':', alpha=.25, label='symmet

ax[0].grid(alpha=.25)
ax[0].legend()
ax[0].set_xlabel('data')
ax[0].set_ylabel('probability density')

# mass:

ax[1].plot(sorted_data, m_above, 'k-', label='total')
ax[1].plot(sorted_data, m_signal_above, 'r-', label='signal')
ax[1].plot(sorted_data, m_symmetric_above, 'b-', label='background')
ax[1].axvline(data_mode, c='k', ls='--')
ax[1].axvline(threshold_guess, c='k', ls=':')

ax[1].grid(alpha=.25)
ax[1].legend(loc='lower right')
ax[1].set_xlabel('threshold')
#ax[1].set_xlim(sorted_data[0], data_mode + 5*L)
ax[1].set_ylabel('mass above threshold')
ax[1].set_yscale('log')
ax[1].set_ylim(1e-2*m_signal, 1.1)

# purity:

ax[2].plot(sorted_data, purity, 'k-')
ax[2].axvline(data_mode, c='k', ls='--')
ax[2].axvline(threshold_guess, c='k', ls=':')

ax[2].grid(alpha=.25)
#ax[2].legend()
ax[2].set_xlabel('threshold')
ax[2].set_ylabel('purity')

ax[0].set_xlim(sorted_data[0], data_mode + 2*L)
```

Out[13]:

```
(-0.0009640190401114523, 0.0019190945125598814)
```



Hierarchical Overdensity Tree (HOT)

routine definition:

In [14]:

```
@njit
def hot(data, threshold, weight):
    parent = -np.ones(data.size, dtype=np.int64)
    top = -np.ones(data.size, dtype=np.int64)
    area = np.zeros(data.size, dtype=np.int64)
    signal = np.zeros(data.size)

    for pixel in np.argsort(data.flatten())[::-1]:  # decreasing order
        row = pixel // data.shape[0]
        col = pixel % data.shape[0]
        if data[row, col] < threshold:
            break
        row0 = max(row-1, 0)
        row1 = min(row+2, data.shape[0])
        col0 = max(col-1, 0)
        col1 = min(col+2, data.shape[1])
        neighbour_parents = []
        for r in range(row0, row1):
            for c in range(col0, col1):
                p = parent[r*data.shape[0]+c]
                if (p > -1) and (p not in neighbour_parents):
                    neighbour_parents.append(top[p])
    #    if row==0:
    #        print(col, neighbour_parents)

        neighbour_parents = np.array(neighbour_parents)
        n_parents = neighbour_parents.size
        if n_parents == 0:
            selected_parent = pixel
        elif n_parents == 1:
            selected_parent = neighbour_parents[0]
        else:
            selected_parent = top[neighbour_parents[np.argmax(area[neighbour_paren
            for p in neighbour_parents:
                if p != selected_parent:
                    top[p] = selected_parent
                    #area[selected_parent] += area[p]
                    #signal[selected_parent] += signal[p]
    #        print(selected_parent)

        parent[pixel] = top[pixel] = selected_parent
        area[selected_parent] += 1
        signal[selected_parent] += weight[row, col]

    return parent, area[parent], signal[parent], top
```

normal and inverted images:

In [15]:

```
#signal_map = weights*purity.reshape(data.shape)*(data-data_mode)/(threshold_guess
signal_map = weights*((data-data_mode)/(threshold_guess-data_mode))**2
parent, area, signal, top = hot(data-data_mode, 0, signal_map)

avg_signal = (signal/(area+1e-30))
parent_list = np.unique(parent)
parent_area = area[parent_list]
parent_signal = signal[parent_list]
```

In [16]:

```
#signal_map = weights*(1-purity.reshape(data.shape))*(data_mode-data)/(threshold_g
signal_map_inv = weights*((data_mode-data)/(threshold_guess-data_mode))**2
parent_inv, area_inv, signal_inv, top_inv = hot(data_mode-data, 0, signal_map_inv)

avg_signal_inv = (signal_inv/(area_inv+1e-30))
parent_inv_list = np.unique(parent_inv)
parent_inv_area = area_inv[parent_inv_list]
parent_inv_signal = signal_inv[parent_inv_list]
```

selection criterion:

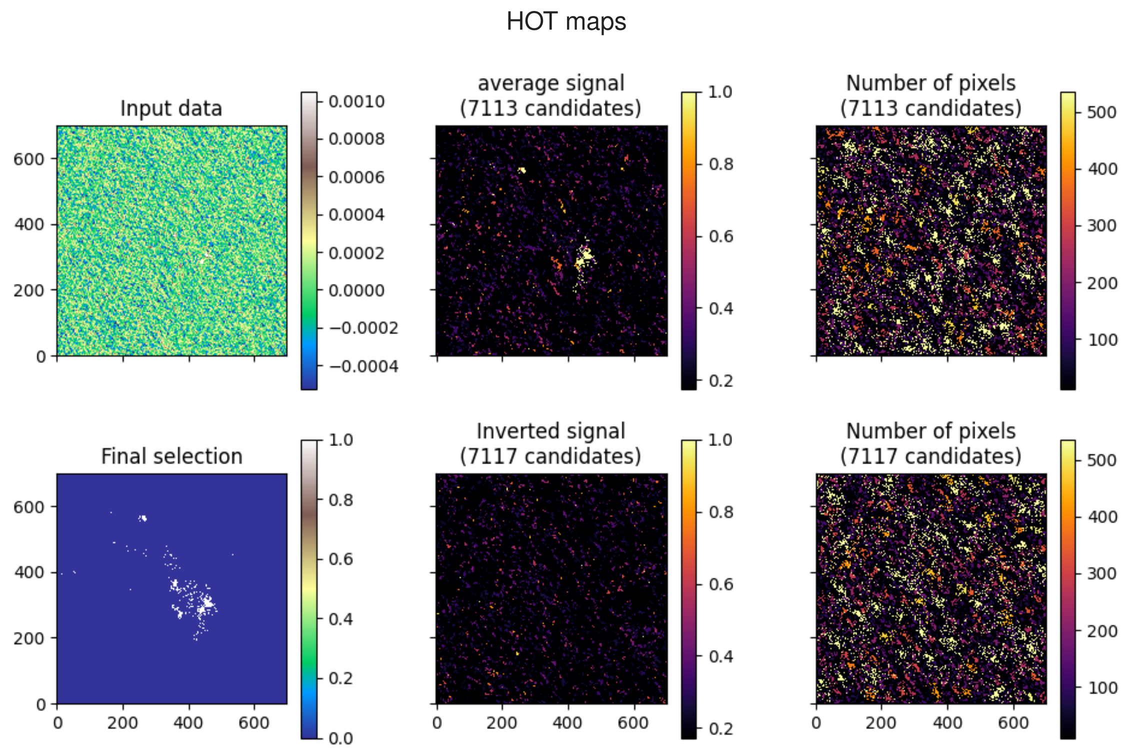
In [17]:

```
mu_bg = np.nanmean(signal_map_inv[data<data_mode])
size_bg_th = np.max(parent_inv_area[parent_inv_signal/(parent_inv_area+1e-30) > 1]
size_bg_mu = np.max(parent_inv_area[parent_inv_signal/(parent_inv_area+1e-30) > mu_

selection = avg_signal > mu_bg
selection &= area > size_bg_th
selection &= avg_signal > 1 + (mu_bg-1)*(area-size_bg_th)/(size_bg_mu-size_bg_th)
```

results:

In [24]:

```
plt.close('HOT maps')
fig = plt.figure('HOT maps', figsize=(9.5, 6))
fig.set_tight_layout(True)
ax = fig.subplots(nrows=2, ncols=3, sharex=True, sharey=True)

ax[0, 0].set_title('Input data')
im = ax[0, 0].imshow(
    data,
    interpolation='nearest', origin='lower', cmap='terrain',
#    vmin=-2, vmax=2,
    vmin=2*data_mode-threshold_guess, vmax=2*threshold_guess-data_mode,
    )
cb = fig.colorbar(im, ax=ax[0, 0])

ax[0, 1].set_title(f'average signal\n({len(np.unique(parent))} candidates)')
im = ax[0, 1].imshow(
    avg_signal.reshape(data.shape),
    interpolation='nearest', origin='lower', cmap='inferno',
    vmin=mu_bg, vmax=1,
    )
cb = fig.colorbar(im, ax=ax[0, 1])

ax[0, 2].set_title(f'Number of pixels\n({len(np.unique(parent))} candidates)')
im = ax[0, 2].imshow(
#    np.log10(area+1).reshape(data.shape),
    area.reshape(data.shape),
    interpolation='nearest', origin='lower', cmap='inferno',
    vmin=size_bg_th, vmax=size_bg_mu,
    )
cb = fig.colorbar(im, ax=ax[0, 2])


ax[1, 0].set_title('Final selection')
im = ax[1, 0].imshow(
#    signal_map_inv,
    selection[top].reshape(data.shape),
    interpolation='nearest', origin='lower', cmap='terrain',
    #vmin=-2, vmax=2,
    )
cb = fig.colorbar(im, ax=ax[1, 0])

ax[1, 1].set_title(f'Inverted signal\n({len(np.unique(parent_inv))} candidates)')
im = ax[1, 1].imshow(
    avg_signal_inv.reshape(data.shape),
    interpolation='nearest', origin='lower', cmap='inferno',
    vmin=mu_bg, vmax=1,
    )
cb = fig.colorbar(im, ax=ax[1, 1])

ax[1, 2].set_title(f'Number of pixels\n({len(np.unique(parent_inv))} candidates)')
im = ax[1, 2].imshow(
    area_inv.reshape(data.shape),
    #np.log10(area_inv+1).reshape(data.shape),
    interpolation='nearest', origin='lower', cmap='inferno',
    vmin=size_bg_th, vmax=size_bg_mu,
    )
cb = fig.colorbar(im, ax=ax[1, 2])
```



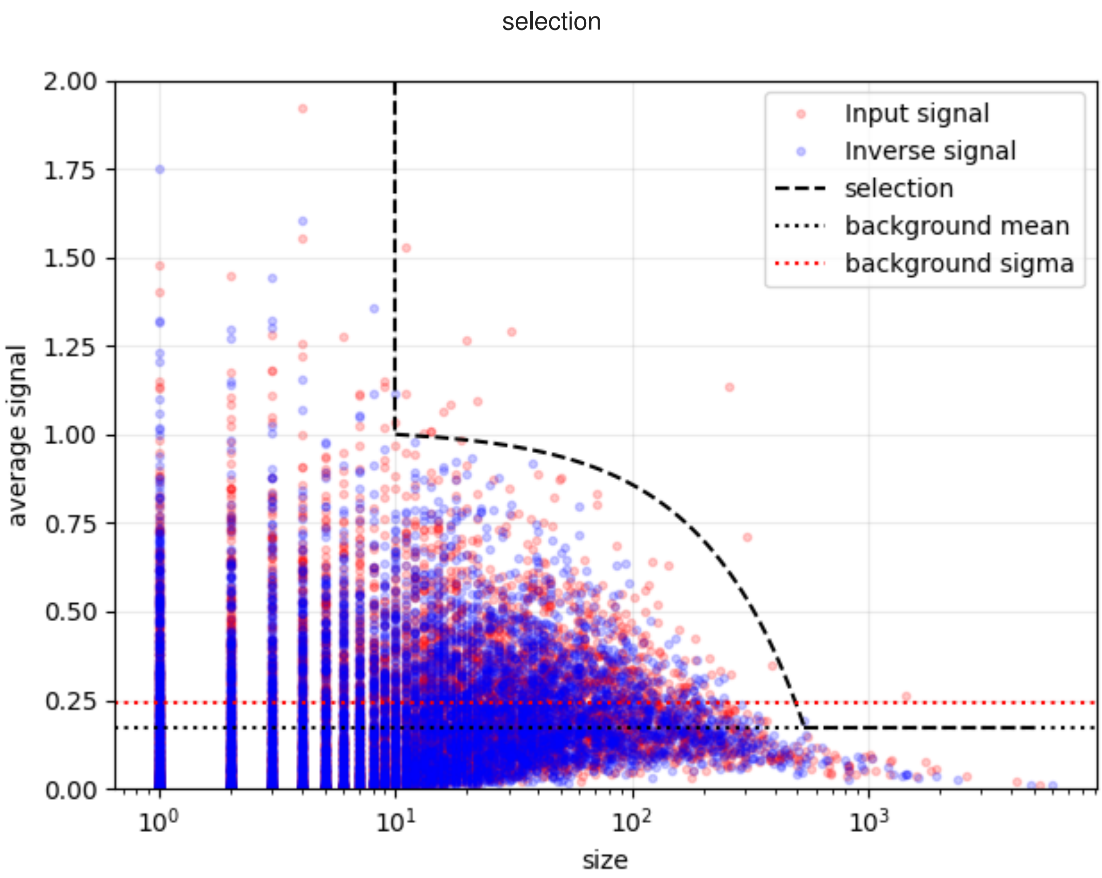
In [25]:

```
plt.close('selection')
fig = plt.figure('selection')
fig.set_tight_layout(True)
ax = fig.subplots()

ax.plot(parent_area, parent_signal/parent_area, 'r.', alpha=0.2, label='Input sign
ax.plot(parent_inv_area, parent_inv_signal/parent_inv_area, 'b.', alpha=0.2, label

aa = np.linspace(size_bg_th, size_bg_mu, 101)
ss = 1 + (mu_bg-1)*(aa-size_bg_th)/(size_bg_mu-size_bg_th)

ax.plot([size_bg_th, size_bg_th], [np.nanmax(parent_signal/parent_area), 1], 'k--'
ax.plot(aa, ss, 'k--', label='selection')
ax.plot([size_bg_mu, np.max(parent_area)], [mu_bg, mu_bg], 'k--')

ax.axhline(mu_bg, c='k', ls=':', label='background mean')
ax.axhline(np.nanstd(signal_map_inv[data<data_mode]), c='r', ls=':', label='backgr
#aa = np.logspace(0, 3)
#ss = np.nanstd(signal_map_inv[data<data_mode]) * np.power(aa, .5)
#ax.plot(aa, mu+ss/aa, 'k:')
#ax.plot(aa, mu-ss/aa, 'k:')

ax.grid(alpha=.25)
ax.legend(loc='upper right')
ax.set_xlabel('size')
ax.set_xscale('log')
ax.set_ylabel('average signal')
#ax.set_yscale('log')
#ax.set_ylim(0.1*mu_bg, np.nanmax(parent_signal/parent_area))
ax.set_ylim(0, 2)
```

```
/tmp/ipykernel_117628/1993769162.py:6: RuntimeWarning: invalid value encountered i
n true_divide
  ax.plot(parent_area, parent_signal/parent_area, 'r.', alpha=0.2, label='Input si
gnal')
/tmp/ipykernel_117628/1993769162.py:12: RuntimeWarning: invalid value encountered
in true_divide
  ax.plot([size_bg_th, size_bg_th], [np.nanmax(parent_signal/parent_area), 1], 'k
--')
```

Out[25]:

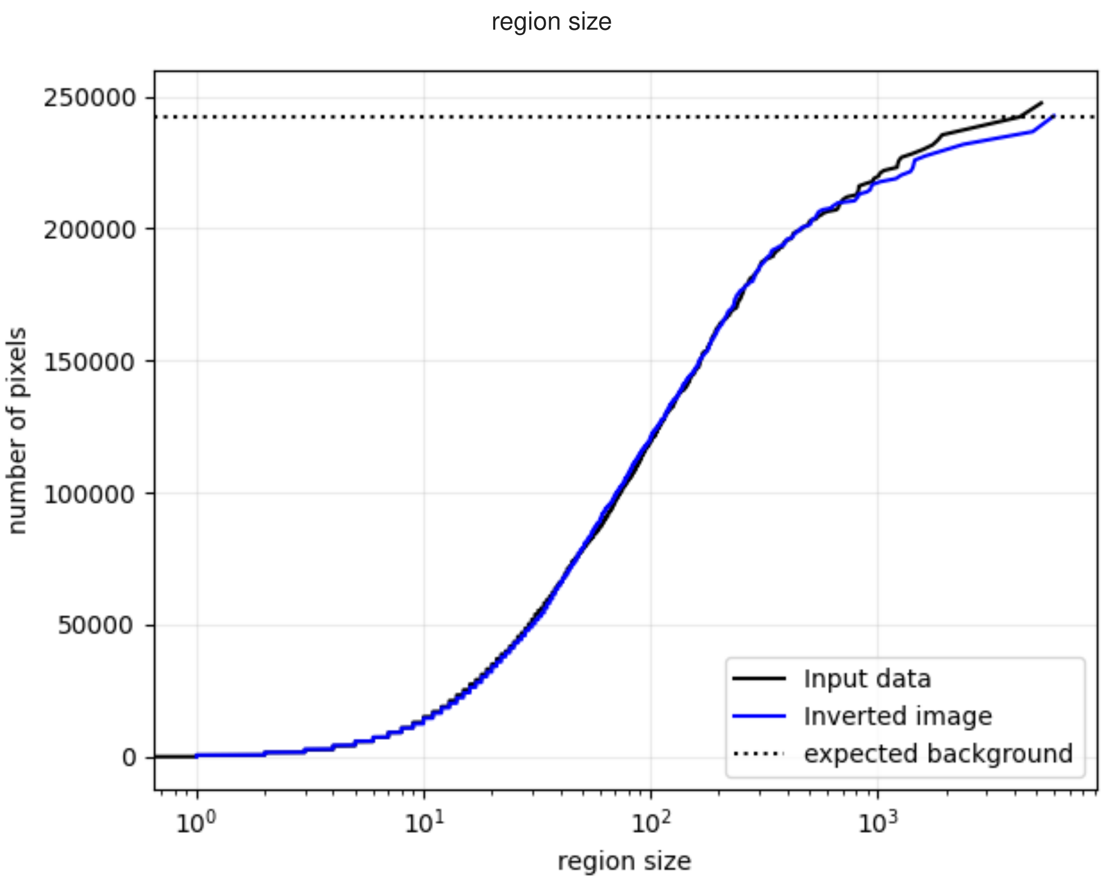
```
(0.0, 2.0)
```



In [20]:

```
plt.close('region size')
fig = plt.figure('region size')
fig.set_tight_layout(True)
ax = fig.subplots()

sorted_area = np.sort(parent_area)
sorted_area_inv = np.sort(parent_inv_area)
ax.plot(sorted_area, np.cumsum(sorted_area), 'k-', label='Input data')
ax.plot(sorted_area_inv, np.cumsum(sorted_area_inv), 'b-', label='Inverted image')
ax.axhline((m_background/2)*data.size, c='k', ls=':', label='expected background')

ax.grid(alpha=.25)
ax.legend()
ax.set_xlabel('region size')
ax.set_xscale('log')
ax.set_ylabel('number of pixels')
#ax.set_yscale('log')
```

Out[20]: Text(0, 0.5, 'number of pixels')



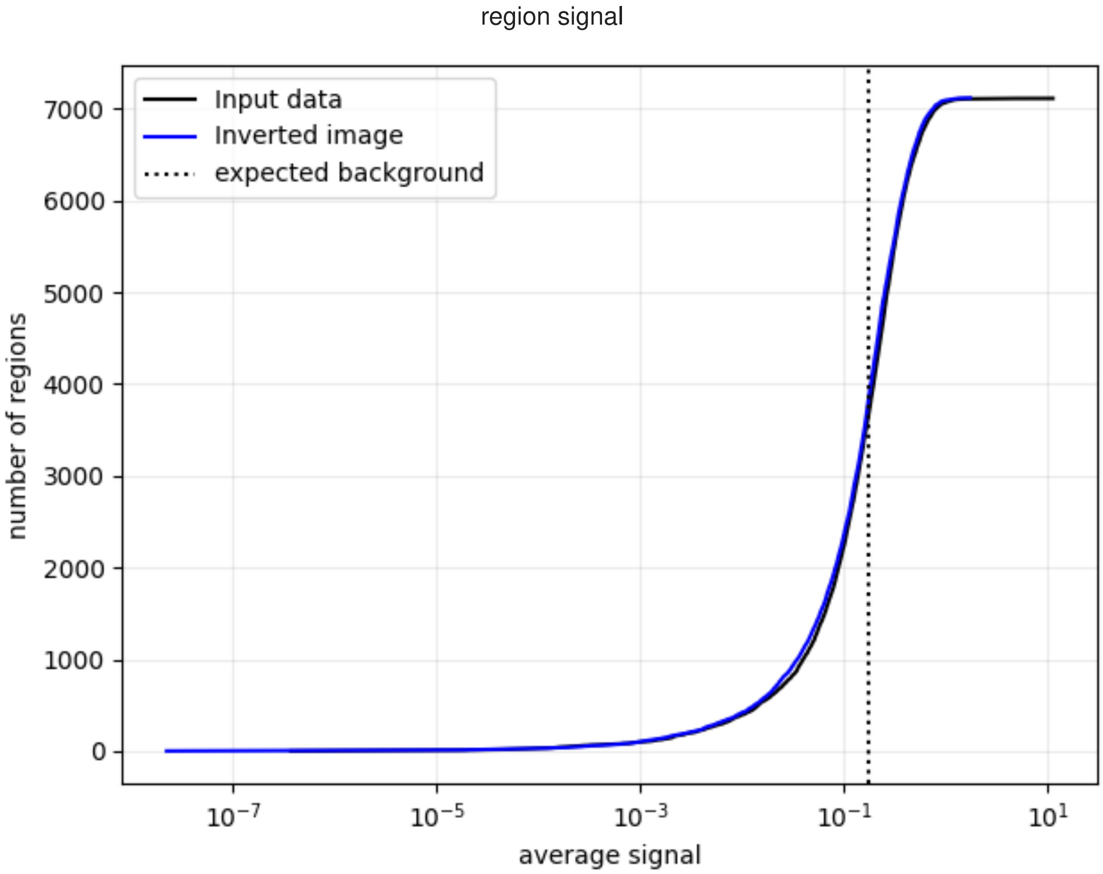
In [21]:

```
plt.close('region signal')
fig = plt.figure('region signal')
fig.set_tight_layout(True)
ax = fig.subplots()

sorted_avg = np.sort(parent_signal/parent_area)
sorted_avg_inv = np.sort(parent_inv_signal/parent_inv_area)
ax.plot(sorted_avg, np.arange(sorted_avg.size)+1, 'k-', label='Input data')
ax.plot(sorted_avg_inv, np.arange(sorted_avg_inv.size)+1, 'b-', label='Inverted im
ax.axvline(mu_bg, c='k', ls=':', label='expected background')

ax.grid(alpha=.25)
ax.legend()
ax.set_xlabel('average signal')
ax.set_xscale('log')
ax.set_ylabel('number of regions')
#ax.set_yscale('log')
```

```
/tmp/ipykernel_117628/1134602932.py:6: RuntimeWarning: invalid value encountered i
n true_divide
  sorted_avg = np.sort(parent_signal/parent_area)
```

Out[21]:

```
Text(0, 0.5, 'number of regions')
```


region signal
Input data
Inverted image
expected background
number of regions
average signal
7000
6000
5000
4000
3000
2000
1000
0
10^{-7}
10^{-5}
10^{-3}
10^{-1}
10^{1}

In [ ]: