Practical 4. Localization / particle filter

In this practical, we will use Bayesian methods to estimate an animal's location, based on noisy observations.

The general idea is that we treat location as an unknown parameter. As per the usual Bayesian approach, we start with a prior distribution for location, and we update it in light of observations to get a posterior distribution. The location updates every timestep, according to a Markov chain, and the posterior is modified to account for this. The method is laid out in the questions of Practical 4 exercises (https://www.cl.cam.ac.uk/teaching/1920/DataSci/prac4sol.pdf), which you should attempt first, before coding. The guide here refers to $\delta^{(n)}$ and $\pi^{(n)}$, symbols that are defined in those exercises.

The technique we will use here is a *particle filter*, which is a Monte Carlo approximation to the exact method described in the example sheet. In Monte Carlo Bayesian analysis, instead of working with a prior distribution and a posterior distribution, we use a sample of parameter values from the prior distribution, and we update their weights using Bayes's rule. In the particle filter, each *particle* is a sampled location.

In [154]:
```
import numpy as np
import scipy.stats
import pandas
import matplotlib.pyplot as plt
import matplotlib.patches
import imageio
from IPython.display import clear_output
```

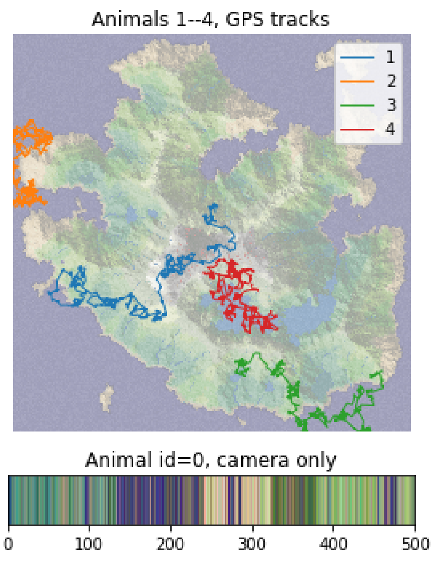
Preamble. The data

We have data from several animals which are wandering over a terrain. The animals are equipped with GPS and with cameras, but the GPS on animal 0 has stopped working. We would like to find out where this animal is.

Here is the terrain and the GPS+camera data. The camera records roughly the rgb values of the animal's current location, though there is some noise.

In [114]:
```
map_image = imageio.imread('../../notes/src/voronoi-map-goal-16000-shaded.png')
localization = pandas.read_csv('https://teachingfiles.blob.core.windows.net/datasets/localization.csv
localization.sort_values(['id','t'], inplace=True)

# Pull out observations for the animal we want to track
observations = localization.loc[localization.id==0, ['r','g','b']].values
```

In [119]:
```python
df = localization

fig,(ax,ax2) = plt.subplots(2,1, figsize=(4,5), gridspec_kw={'height_ratios':[4,.5]})
ax.imshow(map_image, alpha=.5)
w,h = map_image.shape[:2]
ax.set_xlim([0,w])
ax.set_ylim([0,h])

for i in range(1,5):
    ax.plot(df.loc[df.id==i,'x'].values, df.loc[df.id==i,'y'].values, lw=1, label=i)
ax.axis('off')
ax.legend()
ax.set_title('Animals 1--4, GPS tracks')

ax2.bar(np.arange(len(observations)), np.ones(len(observations)), color=observations, width=2)
ax2.set_xlim([0,len(observations)])
ax2.set_yticks([])
ax2.set_title('Animal id=0, camera only')

plt.tight_layout()
plt.show()
```



0. $\delta^{(0)}$: initial particles sampled from prior distribution

We'll start with a prior belief that the animal's location is uniformly distributed over the map.

The first step is to create a sample from the prior distribution, call it $\delta^{(0)}$, as per the practical exercise sheet. We'll store it as an $M \times 3$ array, one row per sample, with columns for x coordinate, y coordinate, and weight w. The prior distribution is uniform, so $w = 1/M$.

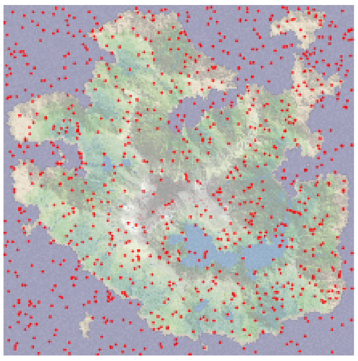
Here's also a handy function to visualize the particles, use `show_particles` .

In [120]:
```python
W,H = map_image.shape[:2]
M = 1000

# Prior belief δ_0, before any observation
δ0 = np.column_stack([np.random.uniform(0,W-1,size=M), np.random.uniform(0,H-1,size=M), np.ones(M)/M]
```

In [169]:
```python
def show_particles(particles, ax=None, s=1, c='red'):
    # Plot an array of particles, with size proportional to weight.
    # (Scale up the sizes by setting s larger.)
    if ax is None:
        fig,ax = plt.subplots(figsize=(2.5,2.5))
    ax.imshow(map_image, alpha=.5)
    w,h = map_image.shape[:2]
    ax.set_xlim([0,w])
    ax.set_ylim([0,h])
    w = particles[:,2]
    ax.scatter(particles[:,0],particles[:,1], s=w/np.sum(w)*s, color=c)
    ax.axis('off')
```

In [130]:
```python
fig,ax = plt.subplots(figsize=(4,4))
show_particles(δ0, s=400, ax=ax)
ax.set_title('$\delta^{(0)}$')
plt.show()
```

$\delta^{(0)}$



1. $\pi^{(0)}$: updated particle weights, given the first observation

Let the first observation be y_0. We want to update the weights of each particle, using Bayes's rule to condition on the probability of seeing y_0, as described in lecture notes section 6.1.1.

In [134]:
```python
y0 = observations[0]
print("First observation: rgb =", y0)
```

```
First observation: rgb = [0.27698774 0.4914071  0.42452084]
```

To apply Bayes's rule, we need a probability model for Y_0 given a particle's location. A reasonable guess is that Y_0 is a noisy version of the colour patch around the supposed location. Here's a handy utility to extract the average colour of a patch:

In [136]:
```python
def patch(im, xy, size=3):
    s = (size-1) / 2
    nx,ny = np.meshgrid(np.arange(-s,s+1), np.arange(-s,s+1))
    nx,ny = np.stack([nx,ny], axis=0).reshape((2,-1))
    neighbourhood = np.row_stack([nx,ny])
    w,h = im.shape[:2]
    neighbours = neighbourhood + np.array(xy).reshape(-1,1)
    neighbours = nx,ny = np.round(neighbours).astype(int)
    nx,ny = neighbours[:, (nx>=0) & (nx<w) & (ny>=0) & (ny<h)]
    patch = im[nx,ny,:3]
    return np.mean(patch, axis=0)/255
```

In [137]:
```python
loc = δ0[0,:2]
print("First particle is at", loc)

col = patch(map_image, loc, size=3)
print("Map terrain around this particle: rgb =", col)
```

```
First particle is at [569.46704103 171.22542076]
Map terrain around this particle: rgb = [0.25228758 0.25272331 0.43050109]
```

Task 1

Implement a `pr(y,loc)` function to compute the probability of observing `y` if the true location is `loc`.

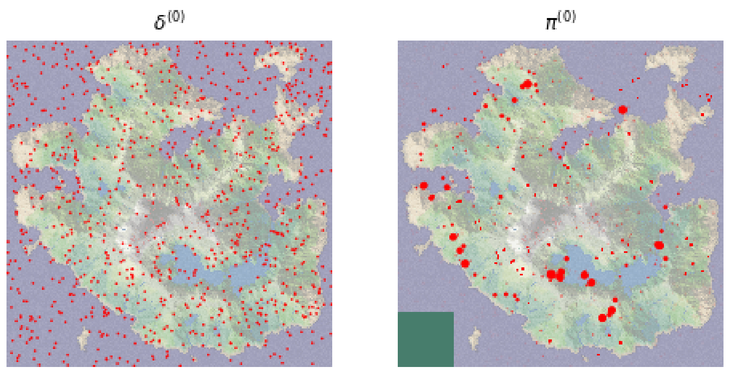
A reasonable probability model is that the observed rgb values in `y` are independent Gaussian random variables, with mean `patch(map_image, loc)`, and with a standard deviation that you should pick.

In [146]:
```
def pr(y, loc):
    # TODO: return a number

# Sanity check
y0 = observations[0]
loc = δ0[0,:2]
w = pr(y0, loc)

import numbers
assert isinstance(w, numbers.Number) and w>=0
```

Now we can use computational Bayes to update the weights of all the particles, and thereby obtain a sample of $\pi^{(0)}$.

In [147]:
```
y0 = observations[0]
w = np.array([pr(y0, (x,y)) for x,y,_ in δ0])
π0 = np.copy(δ0)
π0[:,2] = w / sum(w)
```

In [156]:
```
fig,(axδ,axπ) = plt.subplots(1,2, figsize=(8,4))
show_particles(δ0, ax=axδ, s=400)
show_particles(π0, ax=axπ, s=400)
axπ.add_patch(matplotlib.patches.Rectangle((0,0),100,100,color=y0))
axδ.set_title('$\delta^{(0)}$')
axπ.set_title('$\pi^{(0)}$')
plt.show()
```



2. $\delta^{(1)}$: wandering particles

The next step is to find the distribution of the animal's location, after a timestep. Formally, we want to find $\delta^{(1)}$, the distribution of location at time $t = 1$, conditional on the first observation y_0. This depends on our probability model for how the animal moves in each timestep.

The questions on the practical exercise sheet show how to find this exactly, via the transition matrix that describes the animal's movement. But the method there is impractical when the state space is large. Instead, we will use a Monte Carlo approximation.

To be precise: if we have a weighted set of particles representing a sample from $\pi^{(0)}$, then we can obtain a sample from $\delta^{(1)}$ by simply making each particle take a random step, generated from the distribution that we believe is a model for the animal's movement, and leaving the weights unchanged.

Task 2

Implement a `walk(loc)` function to simulate the animal's movement in one timestep.

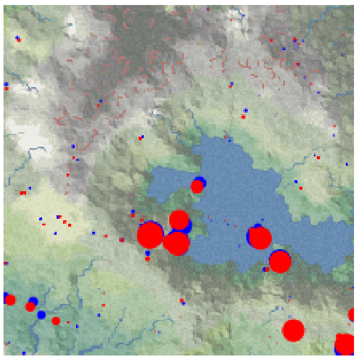
A reasonable probability model is that the animal chooses a direction uniformly in the range $[0, 2\pi)$, and then chooses a random distance, for example an Exponential random variable with mean 5. And then truncate the position to ensure it lies on the map — otherwise the `patch` function won't work.

In [166]:
```
def walk(loc):
    # TODO: return a new location

# Sanity check
loc = δ0[0,:2]
loc2 = walk(loc)
assert len(loc2)==2 and isinstance(loc2[0], numbers.Number) and isinstance(loc2[1], numbers.Number)
assert loc2[0]>=0 and loc2[0]<=W-1 and loc2[1]>=0 and loc2[1]<=H-1
```

Now we can apply this movement to all the particles, and thereby obtain a sample of $\delta^{(1)}$.

In [167]:
```
δ1 = np.copy(π0)
for i in range(len(δ1)):
    δ1[i,:2] = walk(δ1[i,:2])
```

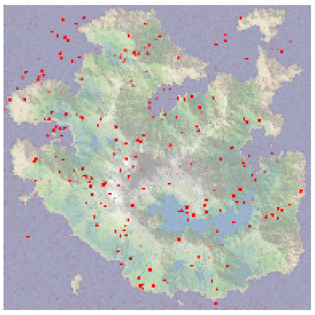
In [178]:
```
fig,ax = plt.subplots(figsize=(4,4))
show_particles(π0, ax=ax, s=4000, c='blue')
show_particles(δ1, ax=ax, s=4000, c='red')
ax.set_xlim([200,400])
ax.set_ylim([100,300])
plt.show()
```



3. Particle hygiene

Now we can apply these two update steps iteratively, updating the sample based on successive observations. Here's a simple animation.

In [184]:
```
num_particles = 1000
W,H = map_image.shape[:2]

particles = np.column_stack([np.random.uniform(0,W-1,size=num_particles),
                             np.random.uniform(0,H-1,size=M),
                             np.ones(num_particles)/num_particles])

for obs in observations[:50]:
    # Compute π, the posterior after observing y
    w = np.array([pr(obs, (px,py)) for px,py,_ in particles])
    particles[:,2] = w / sum(w)
    # Compute δ, the locations after a movement step
    for i in range(num_particles):
        particles[i,:2] = walk(particles[i,:2])

    # Plot the current particles
    fig,ax = plt.subplots(figsize=(3.5,3.5))
    show_particles(particles, ax, s=20)
    plt.show()
    clear_output(wait=True)
```



When you run this, you will likely find that the output is completely useless! The problem is numerical instability. We're only using 1000 samples, and many of these samples get assigned a weight that is almost zero, so we end up with a tiny sample.

Task 3a

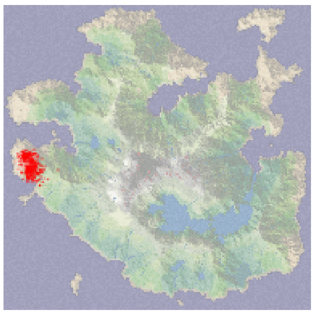
Plot a histogram of particle weights, after 0, 1, 5, 50, and 100 timesteps.

Task 3b

Implement a function `prune_spawn(particles)` . This should delete the lowest-weighted 20% of the particles. Then it should take the highest-weighted 20% of the particles, and split them in two. In other words it should create a duplicate at the same location, and give both the original and the duplicate half the weight. The two versions will diverge in the future, as they take different steps.

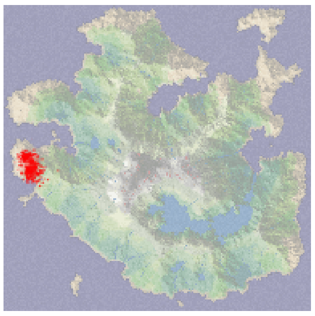
Apply this function every iteration, and show an animation of the result.

After the final observation, you should see something like this:



In [185]:
```
def prune_spawn(particles):
    # TODO: prune and spawn particles
```

In [187]:
```python
num_particles = 1000
W,H = map_image.shape[:2]

particles = np.column_stack([np.random.uniform(0,W-1,size=num_particles),
                             np.random.uniform(0,H-1,size=M),
                             np.ones(num_particles)/num_particles])

for obs in observations:
    # Compute π, the posterior after observing y
    w = np.array([pr(obs, (px,py)) for px,py,_ in particles])
    particles[:,2] = w / sum(w)
    # Compute δ, the locations after a movement step
    for i in range(num_particles):
        particles[i,:2] = walk(particles[i,:2])
    # Prune/spawn
    prune_spawn(particles)
    # Plot the current particles
    fig,ax = plt.subplots(figsize=(3.5,3.5))
    show_particles(particles, ax, s=20)
    plt.show()
    clear_output(wait=True)
```



4. Learn from the data

Task 4

The `localization` dataset contains GPS traces for four other animals, along with their camera readings. Use this data to fit probability models for movement and for observation. (We should use data, not just invent them out of thin air.)

5. Report

Task 5

Compute a probability weight for each pixel on the image, by smoothing your particle filter. It's up to you to find a way to smooth it. Submit your weights, in the form of a png image with the alpha channel reflecting your weights. (I shall normalize them to sum to one.)

I know the true location of the animal. I shall score your answer by the weight you assign to its true pixel coordinates.