Decentralized, Deterministic Robot Swarm Control using Blob Methods for PDEs

July 10, 2020

Matt Haberland[1*], **Katy Craig**[2], **Karthik Elamvazhuthi**[3], **Olga Turanova**[4]

1. BioResource and Agricultural Engineering Department, Cal Poly
2. Department of Mathematics, University of California, Santa Barbara
3. Department of Mathematics, University of California, Los Angeles
4. Department of Mathematics, Michigan State University

∗ Corresponding Author, mhaberla@calpoly.edu

1 Introduction

As the technology to create swarms of inexpensive robots matures, the need for swarm control algorithms arises. One control problem of particular interest is that of coverage: how should individual robots behave for the swarm to achieve a prescribed density in each portion of a domain? There is at least one obvious solution: a central server determines a desired position of each robot, and each robot moves to the assigned location. But when the number of robots is very large or when the communication capability of robots is limited, a decentralized controller would be preferred. Here, we investigate an approach based on a numerical solution method for partial differential equations.

We begin by expressing the density of robots mathematically as the equilibrium solution $\bar{\rho}(x)$ of a diffusive PDE

$$\frac{\partial \rho}{\partial t} - \Delta \left(\frac{\rho}{\bar{\rho}} \right)^m = 0 \tag{1}$$

over domain $\Omega \subset \mathbb{R}^d$ with no-flux boundary condition and $m \geq 1$. For example, if our domain is the unit square in the plane and we want our robots to be distributed uniformly, $\bar{\rho}(x) = 1, x \in \Omega$.

In blob methods for PDEs [1], the initial condition of such a PDE is expressed as a sum of discrete Dirac masses, and the PDE reduces to a system of ordinary differential equations that govern the movement of the Dirac masses over time. Here, we use the ordinary differential equations governing the Dirac mass movements to prescribe the velocities of simulated robots, and this control law guides the swarm to achieve the desired distribution $\bar{\rho}(x)$. After presenting an implementation of the control law and swarm simulation, we describe the computational experiments in progress and discuss improvements to the computational efficiency.

1.1 The Mollifier

In many swarm coverage applications, a robot's "mass", or influence, is not considered to be concentrated at a single point, but is instead considered to be distributed axisymmetrically according to some *mollifier* function $\phi(x)$, such as the Gaussian $\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$.

[ ]:
```
# %% Install LaTeX to improve the appearance of plots.
# Thank you for your patience : )
%%capture
!apt install texlive-fonts-recommended texlive-fonts-extra cm-super dvipng
```

[ ]:
```
# %% Imports and Setup

import numpy as np

import sympy
from sympy import pi, sqrt, exp

import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
matplotlib.rcParams['text.latex.preamble'] = [r'\usepackage{amsmath}']
matplotlib.rcParams['figure.figsize'] = [8, 6]
matplotlib.rcParams.update({'font.size': 16})

def caption(text, ll=70, loc=0, inc=-0.04):
  '''Adds a caption below the current matplotlib figure'''
  fig = plt.gcf()
  words = text.split(" ")
  cap = "Figure {0}. ".format(caption.count)
  for word in words:
    cap += word + " "  # inefficient but simple
    # break lines at end of word after line length `ll` characters
    if len(cap) > ll:
      fig.text(.5, loc, cap.strip(), ha='center')
      loc += inc
      cap = ""
  fig.text(.5, loc, cap.strip(), ha='center')
  caption.count += 1
caption.count = 1  # number figures sequentially

sympy.init_printing()  # pretty-print SymPy math

# Generate Gaussian mollifier (symbolic and numeric)
x = sympy.symbols('x')
```

```
x_n = np.linspace(-3, 3, 300)  # _n is for "NumPy" (to distinguish from SymPy)
phi = 1/sqrt(2*pi)*exp(-x**2/2)  # Gaussian
phi_n = sympy.lambdify((x,), phi, modules="numpy")

# Plot the Gaussian mollifier
plt.xlabel(r'$x$')
plt.ylabel(r'$\phi(x)$')
plt.plot(x_n, phi_n(x_n))
caption("The Gaussian mollifier $\phi(x)$ represents a robot's influence: "
        "greatest near the robot and diminishing with distance.")
```

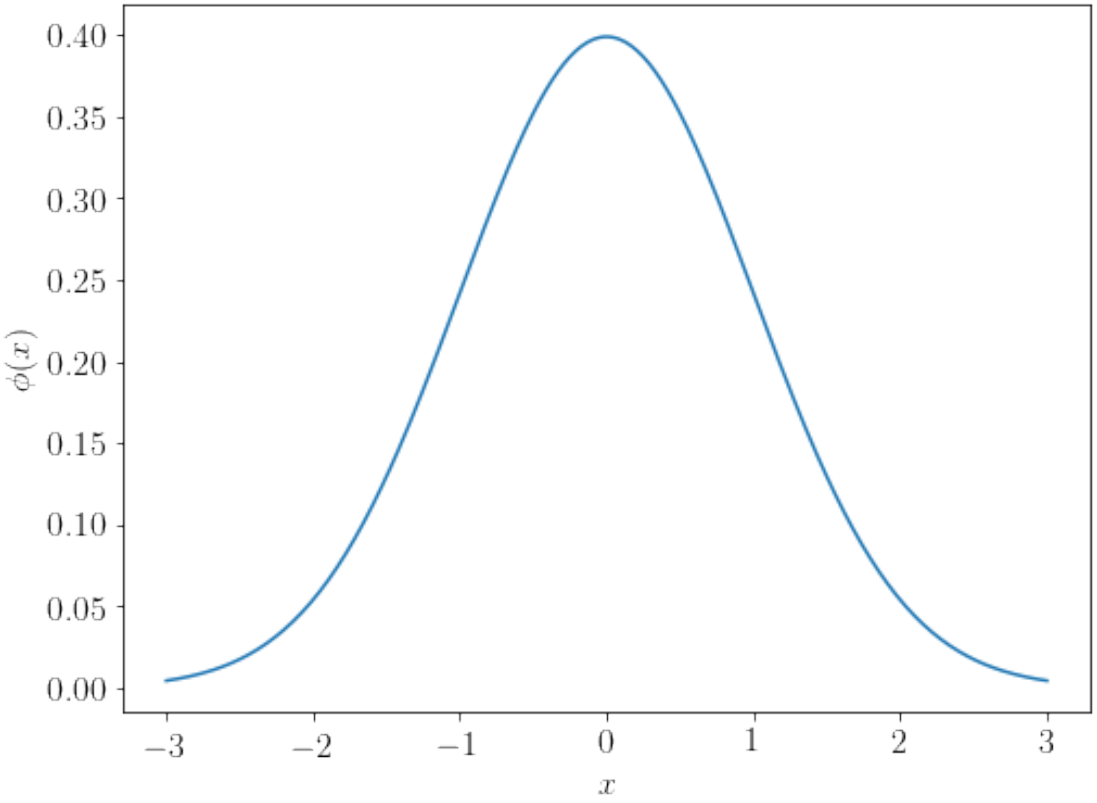

Figure 1. The Gaussian mollifier $\phi(x)$ represents a robot's influence: greatest near the robot and diminishing with distance.

1.2 The Robot Blob Function

If the i^{th} robot in a swarm is located at position x_i and a robot's influence has a radius parameter ϵ that characterizes how quickly the influences fades as the distance from the robot increases, then associated with each robot is a *robot blob function* $\phi_\epsilon(x, x_i) = \frac{1}{\epsilon^d}\phi\left(\frac{x-x_i}{\epsilon}\right)$.

[ ]:
```
# Generate robot blob function (symbolic and numeric)
xi, eps = sympy.symbols('x_i, \\epsilon')
```

```
d = 1  # in code we'll stick with one spatial dimension
phi_i = 1/eps**d * phi.subs(x, (x-xi)/eps)
phi_i_n = sympy.lambdify((x, xi, eps), phi_i, modules="numpy")

# Plot example robot blob functions
plt.xlabel(r'$x$')
plt.ylabel(r'$\phi_\epsilon(x, x_i)$')
plt.plot(x_n, phi_i_n(x_n, -1, 2),
         x_n, phi_i_n(x_n, 0, 1),
         x_n, phi_i_n(x_n, 1, 0.5))
plt.legend((r'$\phi_{\epsilon=2}(x, x_i=-1)$',
            r'$\phi_{\epsilon=1}(x, x_i=0)$',
            r'$\phi_{\epsilon=0.5}(x, x_i=1)$'))
caption("Three robot blob functions $\phi_{\epsilon}(x, x_i)$.")
```

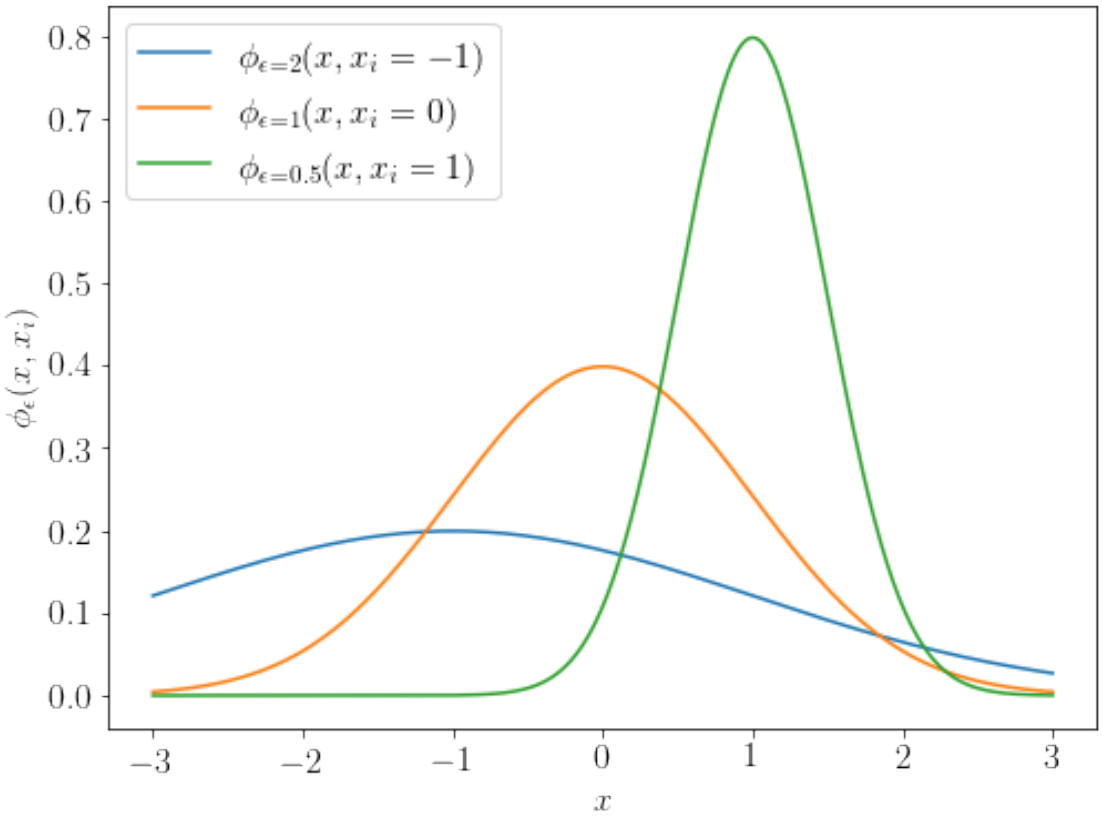

Figure 2. Three robot blob functions $\phi_\epsilon(x, x_i)$.

1.3 The Swarm Blob Function

We define the *swarm blob function* as a weighted sum of the influences of all the robots,

$$\rho_\epsilon(x) = \sum_i \phi_\epsilon(x, x_i) m_i, \tag{2}$$

where m_i is the i^{th} robot's *mass*, which quantifies the strength of the robot's influence. The swarm blob function allows us to unambiguously quantify at every point in a domain the "density" of a finite number of discrete robots.

[ ]:
```
# parameter xp allows us to use a different array math backend;
#we'll try CuPy later
def rho_e_n(x_n, xi_n, eps_n=None, xp=np):
  '''Swarm blob function'''
  xi_n = xp.atleast_2d(xi_n).reshape(-1, 1)  # ensure column
  n_r = np.max(xi_n.shape)  # number of robots in swarm
  mi = 1/n_r       # each robot has an equal fraction of the total mass, 1

  # For a physical robotic swarm, radius parameter would depend on the
  # capabilities of each robot and their application.
  # For theoretical study, we assume that radius parameter scales with the
  # number of robots as follows.
  if eps_n is None:
    eps_n = 2/n_r**0.95

  return xp.sum(phi_i_n(x_n, xi_n, eps_n)*mi, axis = 0)

# Plot example swarm blob functions
xi_n = np.array([-1, 0, 1])
eps_n = 0.4    # eps = 0.4 is chosen for effective visualization
n = xi_n.size
mi = 1/n
plt.xlabel(r'$x$')
plt.ylabel(r'$\rho_\epsilon(x)$')
plt.plot(x_n, mi*phi_i_n(x_n, xi_n[0], eps_n), '--',
         x_n, mi*phi_i_n(x_n, xi_n[1], eps_n), '--',
         x_n, mi*phi_i_n(x_n, xi_n[2], eps_n), '--',
         x_n, rho_e_n(x_n, xi_n, eps_n), '-')
plt.legend((r'$m_i\phi_{\epsilon}(x, x_i=-1)$',
            r'$m_i\phi_{\epsilon}(x, x_i=0)$',
            r'$m_i\phi_{\epsilon}(x, x_i=1)$',
            r'$\rho_\epsilon(x)$'),
           bbox_to_anchor=(1.04,1), loc="upper left")
caption("A swarm blob function $\\rho_{\epsilon}$: the (weighted) sum "
        "of the robot blob functions.")
```

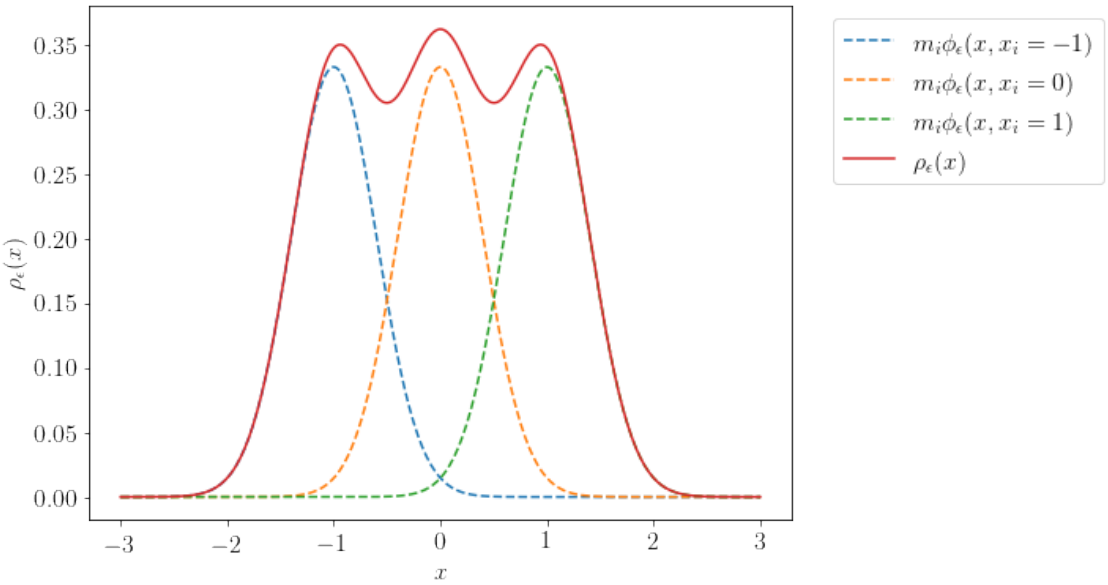

Figure 3. A swarm blob function ρ_ϵ: the (weighted) sum of the robot blob functions.

1.4 The Control Law

In terms of these functions, the control law derived from (1) with $m = 2$ can be written

$$v_i = -a(x_i)\left(\rho_\epsilon(x_i)\nabla a(x_i) + a(x_i)\nabla\rho_\epsilon(x_i) + \sum_j m_j \nabla\phi_\epsilon(x_i, x_j)a(x_j)\right), \tag{3}$$

where $a(x) = \frac{1}{\bar{\rho}(x)}$ and v_i is the instantaneous velocity of the i^{th} robot.

2 Simulation

2.1 Initial Distribution

Suppose that $n_r = 10$ robots in a one-dimensional world are initially clustered about the origin.

[ ]:
```
# Generate initial configuration of 10 robots and plot swarm blob function
np.random.seed(4)
n_r = 10
xi_0 = np.random.rand(n_r)-0.5

plt.plot(xi_0, np.zeros_like(xi_0), '.',
         x_n, rho_e_n(x_n, xi_0), '-')
plt.xlabel(r'$x$')
plt.ylabel(r'$\rho_\epsilon(x)$')
plt.legend(("Initial Robot Position", "Swarm Blob Function"))
caption("Example robot positions and corresponding swarm blob function.")
```

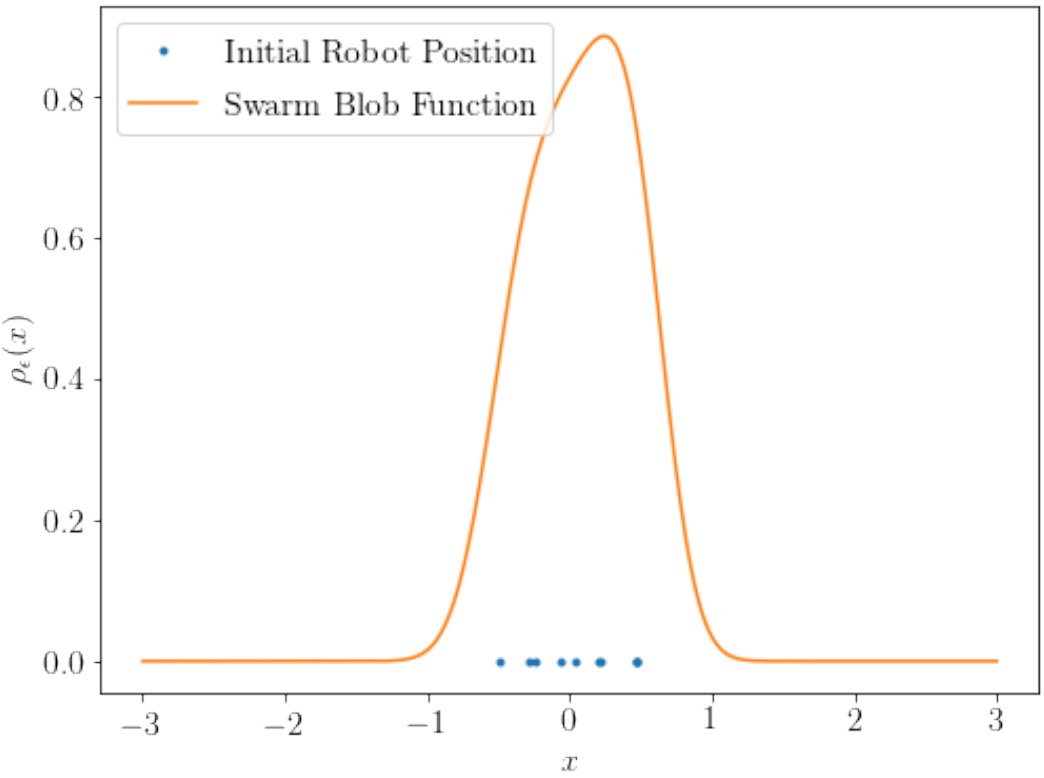

Figure 4. Example robot positions and corresponding swarm blob function.

2.2 Target Distribution

We wish for the robots to distribute themselves according to some *target distribution* $\bar{\rho}(x)'$ within the domain $\Omega = [-2, 2]$ according to the control law given by (3). To do this, we express the condition that the robots be distributed uniformly by choosing the target distribution $\bar{\rho}'(x)$ to be a constant, e.g. $\bar{\rho}'(x) = \frac{1}{4}$. We can enforce the no-flux boundary condition by multiplying our desired distribution by a boxcar function $B_\Omega(x)$, which has value one within the domain and zero outside the domain. The equilibrium solution to (1) is therefore

$$\bar{\rho}(x) = \bar{\rho}'(x) B_\Omega(x). \tag{4}$$

In our one dimensional case, $B_\Omega(x) = H(x+2) - H(x-2)$, where $H(x)$ is the Heaviside step function.

[ ]:
```
def rho(x):
  '''Example target distribution'''
  return 1/4

def H(x):
  '''Heaviside step function'''
  y = np.zeros_like(x)
  y[x >= 0] = 1
```

```
  return y

# generate boxcar function and use it to bound target distribution
boxcar = lambda x_n: H(x_n + 2) - H(x_n - 2)
rho_n = lambda x_n: rho(x_n) * boxcar(x_n)

plt.plot(x_n, rho_n(x_n),'C2')
plt.xlabel(r'$x$')
plt.ylabel(r'$\bar \rho(x)$')
caption("An example target distribution: uniform within the domain and "
        "zero outside.")
```

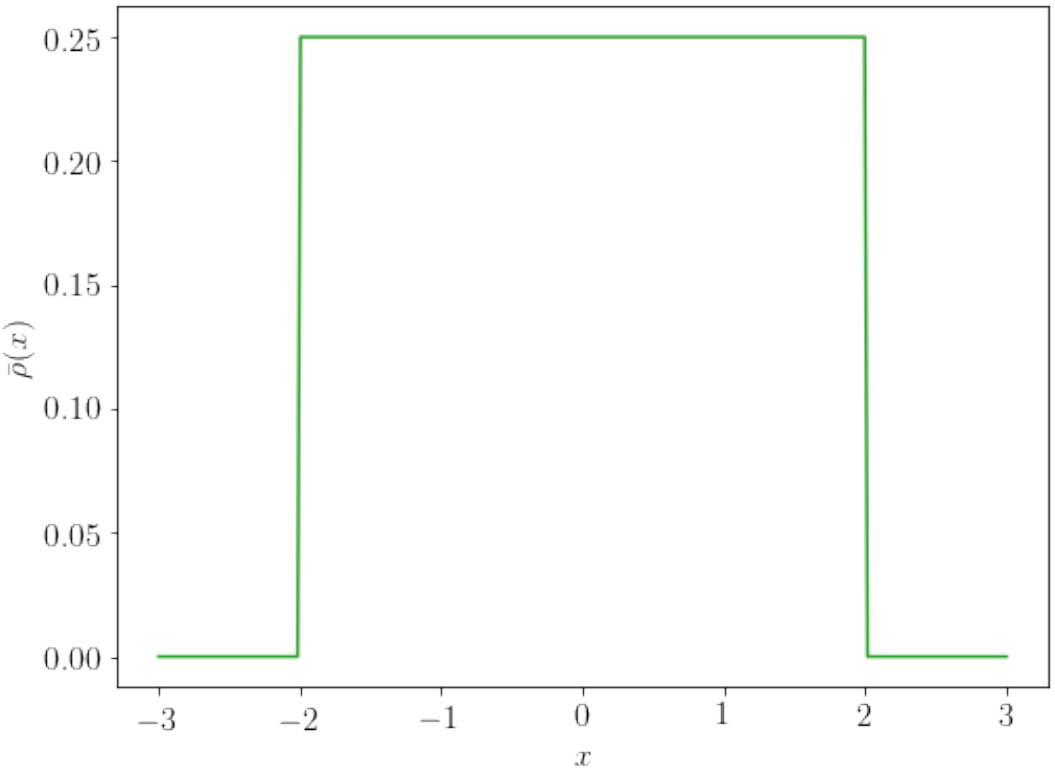

Figure 5. An example target distribution: uniform within the domain and zero outside.

Note that although we have chosen $\bar{\rho}'(x) = \frac{1}{4}$ to be normalized such that $\int_\Omega \bar{\rho}'(x) = 1$, this is not strictly required. A target distribution of any constant value simply means that the target density of robots is the same at every point within the domain. The density of the swarm is specified in relative terms (e.g. $\bar{\rho}(x_1) = 2$ and $\bar{\rho}(x_2) = 4$ means that the swarm's influence is twice as dense at x_2 than at x_1) rather than as an absolute value (e.g. number of robots per meter).

For simulation, we need to make two adjustments.

1. Our control law (3) involves the gradient $\nabla \rho_\epsilon(x)$, so we replace the discontinuous Heaviside step function with a smoothed approximation: the logistic function $\tilde{H} = \frac{1}{2} + \frac{1}{2}\tanh(kx)$, where k is some large constant.

2. Our control law (3) involves a term $a(x) = \frac{1}{\bar{\rho}(x)}$, so to avoid division by zero in simulation, we add a small positive constant δ.

Thus, in place of the boxcar function above, we use an approximation that is smooth and positive everywhere:

$$\tilde{B}_\Omega(x) = \tilde{H}(x+2) - \tilde{H}(x-2) + \delta. \tag{5}$$

[ ]:
```
from sympy import tanh, Rational

bounds = (-2, 2)
rho = sympy.Rational(1, 4)

def H(x):
  '''Smoothed Heaviside step function'''
  k = 10  # the larger the constant, the closer the approximation
  return 1/2 + 1/2 * sympy.tanh(k*x)

delta = 0.001
boxcar = H(x - bounds[0]) - H(x - bounds[1]) + delta
rho = rho*boxcar
rho_n = sympy.lambdify(x, rho, modules="numpy")

plt.plot(x_n, rho_n(x_n),'-C2')
plt.xlabel(r'$x$')
plt.ylabel(r'$\bar \rho (x)$')
caption("An example target distribution, uniform within the domain and zero "
        "outside, modified to be smooth and positive everywhere.")
```

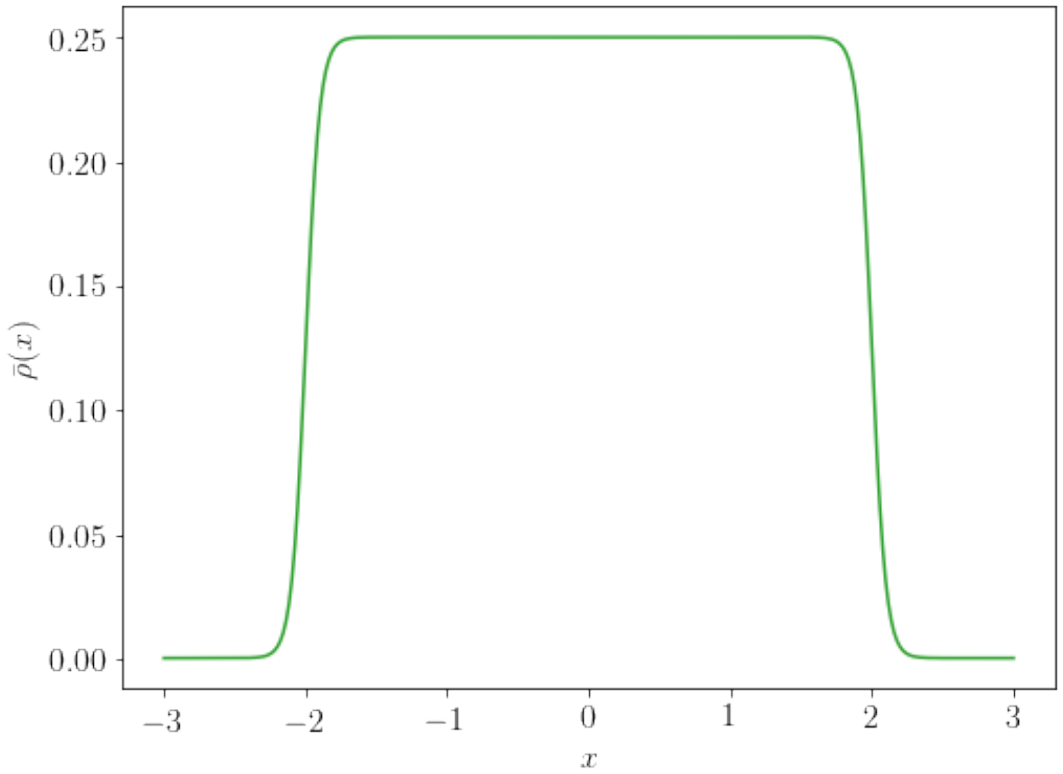

Figure 6. An example target distribution, uniform within the domain and zero outside, modified to be smooth and positive everywhere.

2.3 Control Law Implementation

We define $a(x) = \frac{1}{\bar{\rho}(x)}$, take derivatives symbolically, and create numerical functions.

[ ]:
```
from IPython.display import display

# Use SymPy to take derivates and generate numerical functions. This way,
# we can test different target distributions simply by channging `rho` above.
a = 1/rho
da = sympy.diff(a, x)
drho = sympy.diff(rho, x)
dphi_i = sympy.diff(phi_i, x)
# As an example, display a derivative taken by SymPy
display(dphi_i)

a_n = sympy.lambdify(x, a, modules = "numpy")
da_n = sympy.lambdify(x, da, modules = "numpy")
rho_n = sympy.lambdify(x, rho, modules = "numpy")
drho_n = sympy.lambdify(x, drho, modules = "numpy")
phi_i_n = sympy.lambdify((x, xi, eps), phi_i, modules = "numpy")
```

```
dphi_i_n = sympy.lambdify((x, xi, eps), dphi_i, modules = "numpy")

def get_robot_mass_and_radius(xi_n, eps_n):
  '''Computer robot mass and, if not already specified, radius parameter'''
  # see comments in rho_e_n
  n_r = np.max(xi_n.shape)
  mi = 1/n_r
  eps_n = 2/n_r**0.95
  return mi, eps_n

def drho_e_n(x_n, xi_n, eps_n=None, xp=np):
  '''Gradient (derivative, in this case) of the swarm blob function'''
  xi_n = xp.atleast_2d(xi_n).reshape(-1, 1)  # ensure column
  mi, eps_n = get_robot_mass_and_radius(xi_n, eps_n)
  return xp.sum(dphi_i_n(x_n, xi_n, eps_n)*mi, axis = 0)
```

$$-\frac{\sqrt{2}e^{-\frac{(x-x_i)^2}{2\epsilon^2}}}{4\sqrt{\pi}\epsilon^3}(2x-2x_i)$$

Using these, we can define a function that calculates the velocity of each robot given the position of each robot.

[ ]:
```
def v(xi_n, eps_n=None, xp=np):
  '''Velocity of each robot according to (3)'''
  xi_n = xp.atleast_2d(xi_n)
  mi, eps_n = get_robot_mass_and_radius(xi_n, eps_n)
  term1 = rho_e_n(xi_n, xi_n, xp=xp) * da_n(xi_n)
  term2 = a_n(xi_n) * drho_e_n(xi_n, xi_n, xp=xp)
  term3 = xp.sum(mi * dphi_i_n(xi_n, xi_n.T, eps_n) *  a_n(xi_n.T), axis = 0)
  return (-a_n(xi_n)*(term1 + term2 + term3)).flatten()
```

2.4 ODE Solution

Finally, we can integrate the velocities of the robots over time to get their final positions.

[ ]:
```
from scipy.integrate import solve_ivp

t = np.linspace(0, 2, 100)
# solve_ivp will pass in t (time) and y (robot positions), so our function
# needs to accept both of them
dydt = lambda t, y: v(y)
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, t_eval=t, method='BDF')

def plot_final_blob_function(sol):
  '''Plot the swarm blob function against the target distribution'''
  xi_n_f = sol.y[:, -1]
  plt.plot(xi_n_f, np.zeros_like(xi_n_f), 'oC3',
```

```
          x_n, rho_e_n(x_n, xi_n_f), '-C1',
          x_n, rho_n(x_n), '-C2')
  plt.xlabel(r'$x$')
  plt.ylabel(r'$\rho(x)$')
  plt.legend(("Final Robot Position",
              "Swarm Blob Function $\\rho_{\\epsilon}(x)$",
              "Target Distribution $\\bar \\rho(x)$"),
             bbox_to_anchor=(1.04,1), loc="upper left")

plot_final_blob_function(sol)
caption("Comparison of $\\rho_{\\epsilon}(x)$, the steady state swarm blob "
        "function achieved by our controller, against $\\bar \\rho(x)$, the "
        "target distribution.")
```

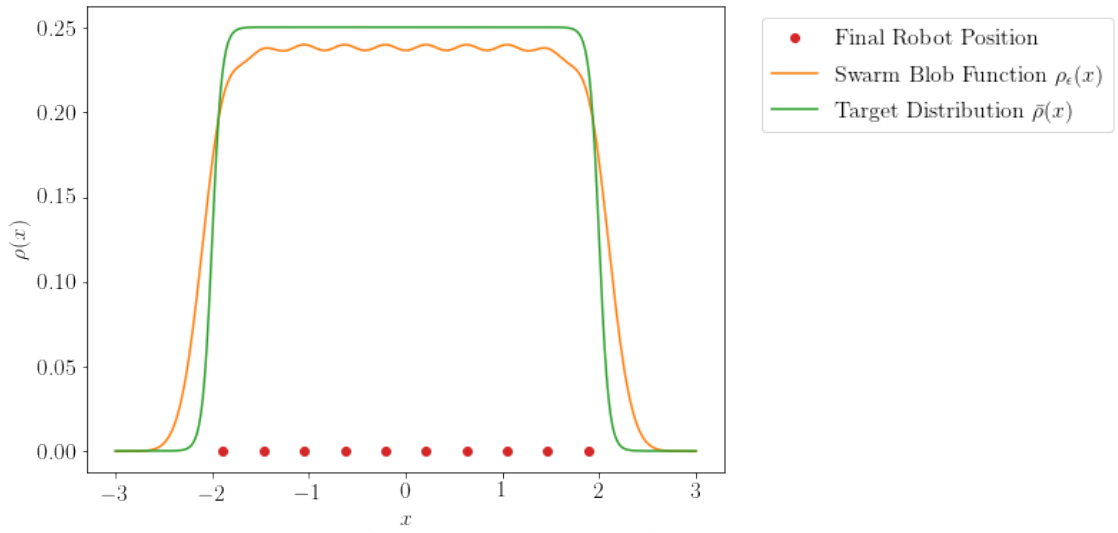

Figure 7. Comparison of $\rho_\epsilon(x)$, the steady state swarm blob function achieved by our controller, against $\bar{\rho}(x)$, the target distribution.

Note that robot masses have been chosen such that the total swarm mass, the integral under the swarm blob function, is equal to that under the target distribution. However, because the number of robots is small and their influence radius parameter ϵ is large, the robot blob functions are rather diffuse and some of this mass 'leaks' beyond the edges of the domain. Consequently, the height of the swarm blob function does not reach the targeted value of 0.25 within the domain. As the swarm size gets larger and ϵ gets smaller, this error tends to be reduced.

Note also that the robots rapidly approach their final positions and then settle into a steady state.

[ ]:
```
def plot_traces(sol):
  '''Plot robot trajectories through time'''
  plt.plot(sol.y.T, sol.t, '-C4',
           sol.y[:, 0], np.ones_like(sol.y[:, 0])*sol.t[0], 'oC0',
           sol.y[:, -1], np.ones_like(sol.y[:, -1])*sol.t[-1], 'oC3')
```

```
  plt.xlabel(r'$x_i(t)$')
  plt.ylabel(r'$t$')
  plt.xlim(-3, 3)
  plt.gca().invert_yaxis()

plot_traces(sol)
caption("Trajectories $x_i(t)$ of robots guided by our controller toward a "
        "uniform distribution.")
```

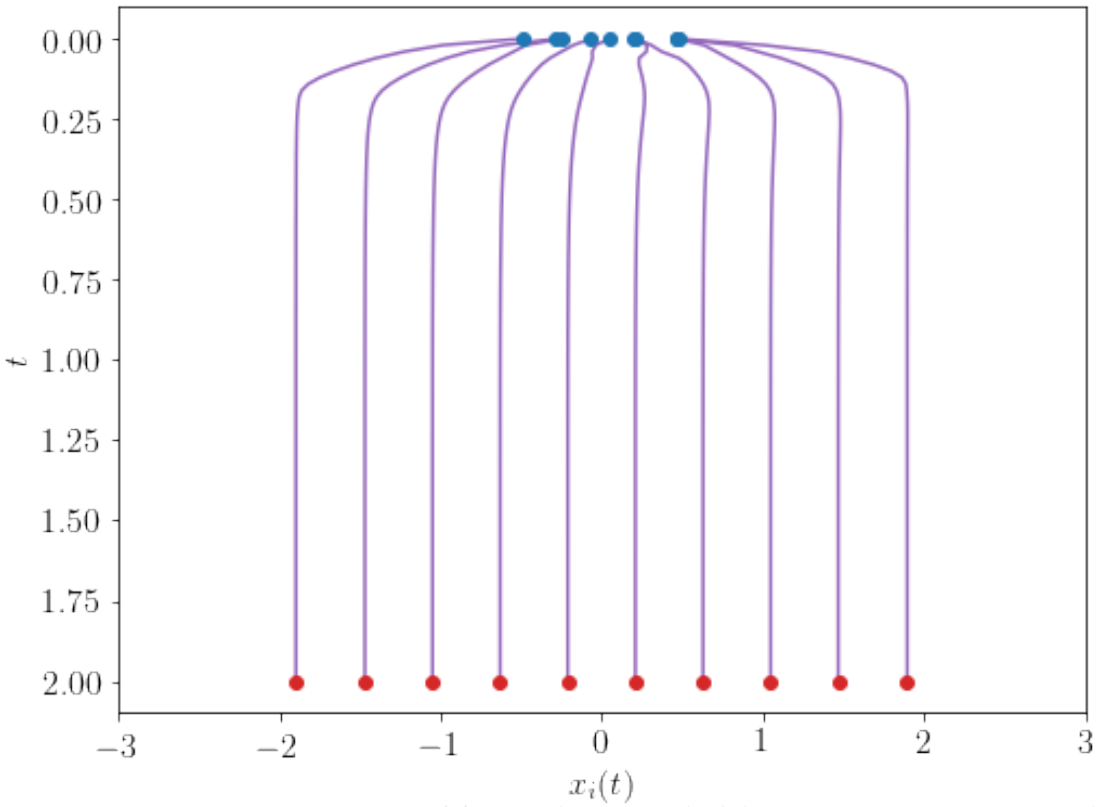

Figure 8. Trajectories $x_i(t)$ of robots guided by our controller toward a uniform distribution.

3 Experiments

There is a family of control laws similar to (3) that can be developed based on (1) using, for example, different values of m and different mollifier functions $\phi(x)$. Using computational experiments, we wish to compare these different controllers in terms of their speed and accuracy, and for each controller we seek to understand the effects of swarm parameters, such as the number of robots n and the size of each robots' influence ϵ, on these metrics of swarm performance.

3.1 Quantifying Accuracy

One way of quantifying the accuracy of the controller is by comparing the swarm blob function ρ_ϵ against the target distribution $\bar{\rho}$ using the error metric from [2]:

$$e_1(t) = \int_\Omega |\rho_\epsilon(x,t) - \bar{\rho}(x,t)|\, dx. \tag{6}$$

[ ]:
```
from scipy.integrate import quad

def e1_distance(xi_n):
  '''Absolute area between the swarm blob function and target distribution'''
  rho_e1 = lambda x_n: rho_e_n(x_n, xi_n)
  error_int = lambda x_n: np.abs(rho_n(x_n) - rho_e1(x_n))
  return quad(error_int, -3, 3)[0]

# Generate a visualization of the error metric
xi_n_f = sol.y[:, -1]
e1f = e1_distance(xi_n_f)
plt.plot(xi_n_f, np.zeros_like(xi_n_f), 'oC3',
         x_n, rho_e_n(x_n, xi_n_f), '-C1',
         x_n, rho_n(x_n), '-C2')
plt.fill_between(x_n, rho_e_n(x_n, xi_n_f), rho_n(x_n),
                 color='C4', alpha=0.5)
plt.xlabel(r'$x$')
plt.ylabel(r'$\rho(x)$')
plt.legend(("Final Robot Position", "Swarm Blob Function", "Target␣
 ↪Distribution"),
           bbox_to_anchor=(1.04,1), loc="upper left")
plt.annotate("$e_1 = {0:.3f}$".format(e1f),
             xy=(0, 0.245), xycoords='data',
             xytext=(0.5, 0.2), textcoords='data',
             arrowprops=dict(arrowstyle="-",
                             connectionstyle="arc3"),
             )
caption("Visualization of the error metric $e_1$: the absolute area between "
        "the swarm blob function and target distribution.")
```

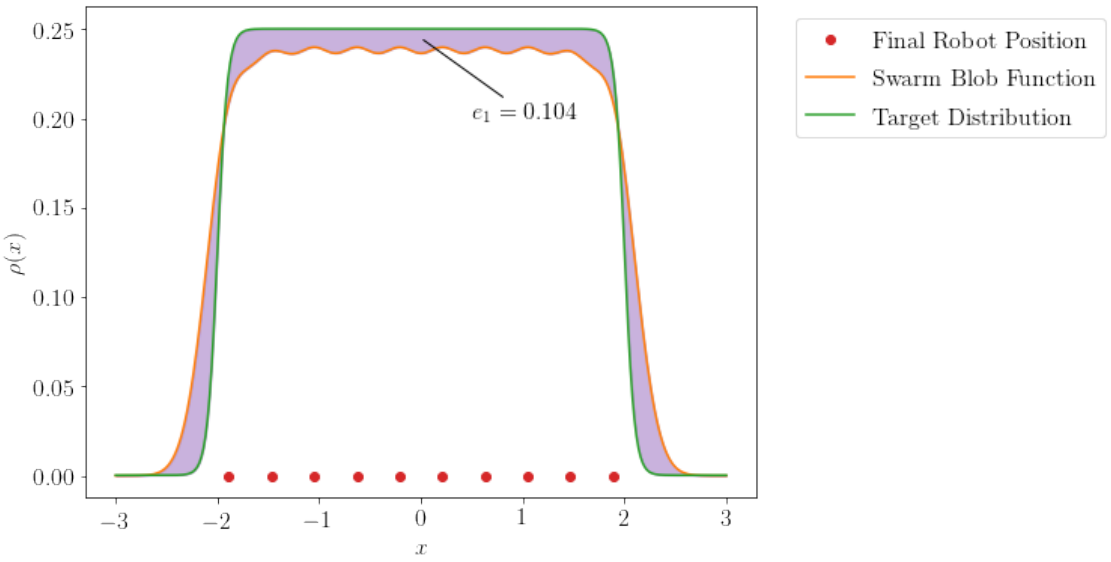

Figure 9. Visualization of the error metric e_1: the absolute area between the swarm blob function and target distribution.

Alternatively, we can use the Wasserstein ("Earth-Movers") distance, which we denote e_2, to compare the discrete robot positions against the desired density distribution.

Below, we plot the value of these two metrics as the swarm configuration evolves over time according to the control law.

[ ]:
```
from scipy.stats import wasserstein_distance
from scipy.integrate import trapz

def e2_distance(xi_n):
  '''The Wasserstein Distance error metric'''
  return wasserstein_distance(x_n, xi_n, rho_n(x_n), 1/n_r*np.ones(n_r))

# a faster approximation of e1 which we will prefer below
def e1_distance(xi_n):
  '''Absolute area between the swarm blob function and target distribution'''
  rho_e1 = rho_e_n(x_n, xi_n)
  return trapz(np.abs(rho_e1 - rho_n(x_n)), x_n)

# Compute the values of the error metric at each time point
n_r, n_t = sol.y.shape
e1 = np.empty(n_t)
e2 = np.empty(n_t)
for i in range(n_t):
  e1[i] = e1_distance(sol.y[:, i])
  e2[i] = e2_distance(sol.y[:, i])

# Plot the error metric over time
```

```
plt.plot(t, e1, t, e2)
plt.xlabel('$t$')
plt.ylabel('$e(t)$')
plt.legend(('$e_1(t)$', '$e_2(t)$'))
caption("Error metrics $e_1$ and $e_2$ through time for the example above.")
```

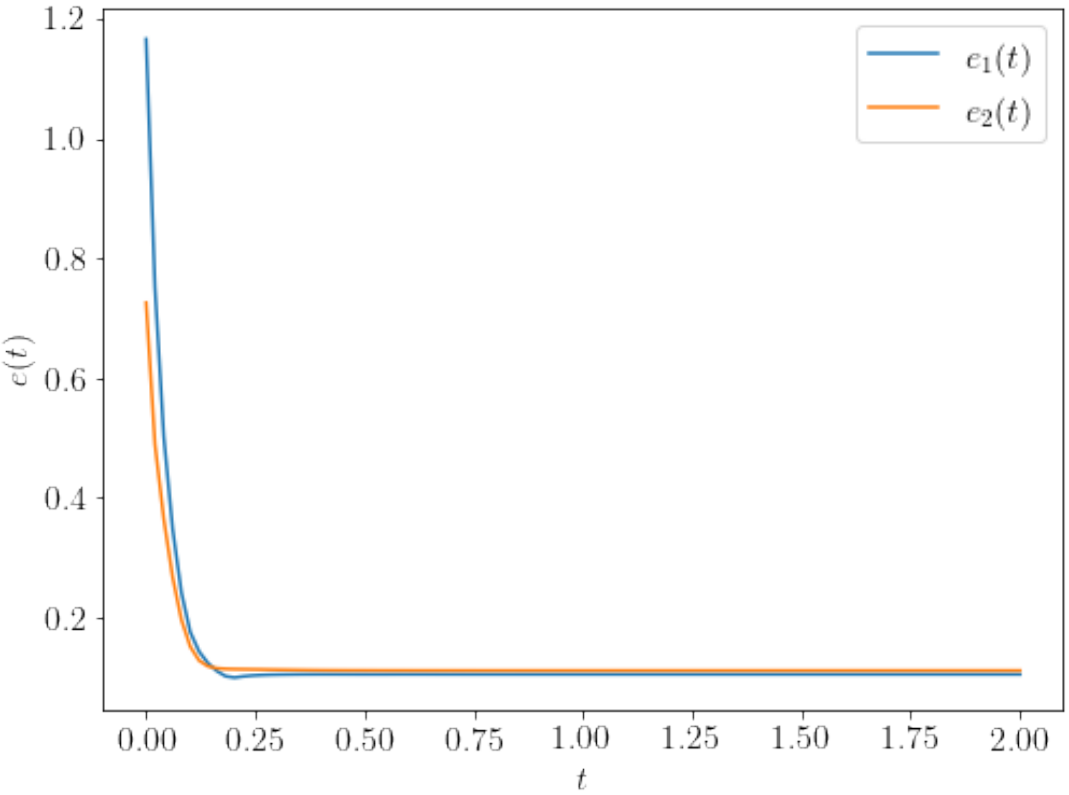

Figure 10. Error metrics e_1 and e_2 through time for the example above.

3.2 Quantifying Speed

Since the controller drives the system towards a steady state, we can quantify the speed of the controller using a conventional definition of settling time: the time after which the value of the error metric remains within 2% if its final value. Defining e_0 as the value of the error metric at the beginning of the simulation and e_∞ as the value of the error metric at the end of the simulation, the value of the error metric at which the system is said to have settled is

$$e_{\text{settle}} = e_{ss} + 0.02\,(e_0 - e_{ss})\,, \tag{7}$$

and the settling time

$$t_{\text{settle}} = \max_t t \tag{8}$$

$$\text{s.t. } e(t) = e_{\text{settle}}. \tag{9}$$

[ ]:
```
from scipy.interpolate import interp1d

# Compute the settling time
e0 = e1[0]         # initial value of error metric
e_ss = e1[-1]      # final value of error metric
settle = 0.02      # convention for steady state: within 2% of final value
e_settle = e_ss + settle * (e0 - e_ss)  # error value 2% above final value
# Find time when e_settle is reached.  This code assumes that the error crosses
# the 2% threshold exactly once and stays below the threshold thereafter, which
# is true in practice
t_of_e = interp1d(e1, t)
t_settle = float(t_of_e(e_settle))

# Depict the settling time graphically
plt.plot(t[t<1.625], e1[t<1.625], '-',
         t[t<0.25], np.ones_like(t[t<0.25])*e0, '--',
         t, np.ones_like(t)*e_settle, '--',
         t_settle, e_settle, 'o')

plt.annotate("$e_0$", xy=(0.275, e0-0.0075))

plt.annotate("", xy=(1.5, e_settle+0.1), xycoords='data',
             xytext=(1.5, e_settle), textcoords='data',
             arrowprops=dict(arrowstyle="<-"))
plt.annotate("", xy=(1.5, e_ss-0.1), xycoords='data',
             xytext=(1.5, e_ss), textcoords='data',
             arrowprops=dict(arrowstyle="<-"))
plt.annotate("$2\\% (e_0-e_{ss})$", xy=(1.5, 0.25))

plt.annotate("", xy=(1.25, 0), xycoords='data',
             xytext=(1.25, e_ss), textcoords='data',
             arrowprops=dict(arrowstyle="<->"))
plt.annotate("$e_{ss}$", xy=(1.275, (0+e_ss)/2-0.0075))

plt.annotate("", xy=(1.75, 0), xycoords='data',
             xytext=(1.75, e_settle), textcoords='data',
             arrowprops=dict(arrowstyle="<->"))
plt.annotate("$e_\\text{settle}$", xy=(1.775, (0+e_settle)/2-0.0075))

plt.annotate("$(t_\\text{settle}, e_\\text{settle})$", xy=(t_settle, e_settle),␣
 ↪xycoords='data',
             xytext=(0.35, 0.25), textcoords='data',
             arrowprops=dict(arrowstyle="-"))

plt.xlim(-0.05, 2)
plt.ylim(0, 1.2)
plt.xlabel('$t$')
```

```
plt.ylabel('$e_1(t)$')
plt.legend(('Error Metric $e_1(t)$',))
caption("Depiction of the 2\% settling time of the error metric.")
```

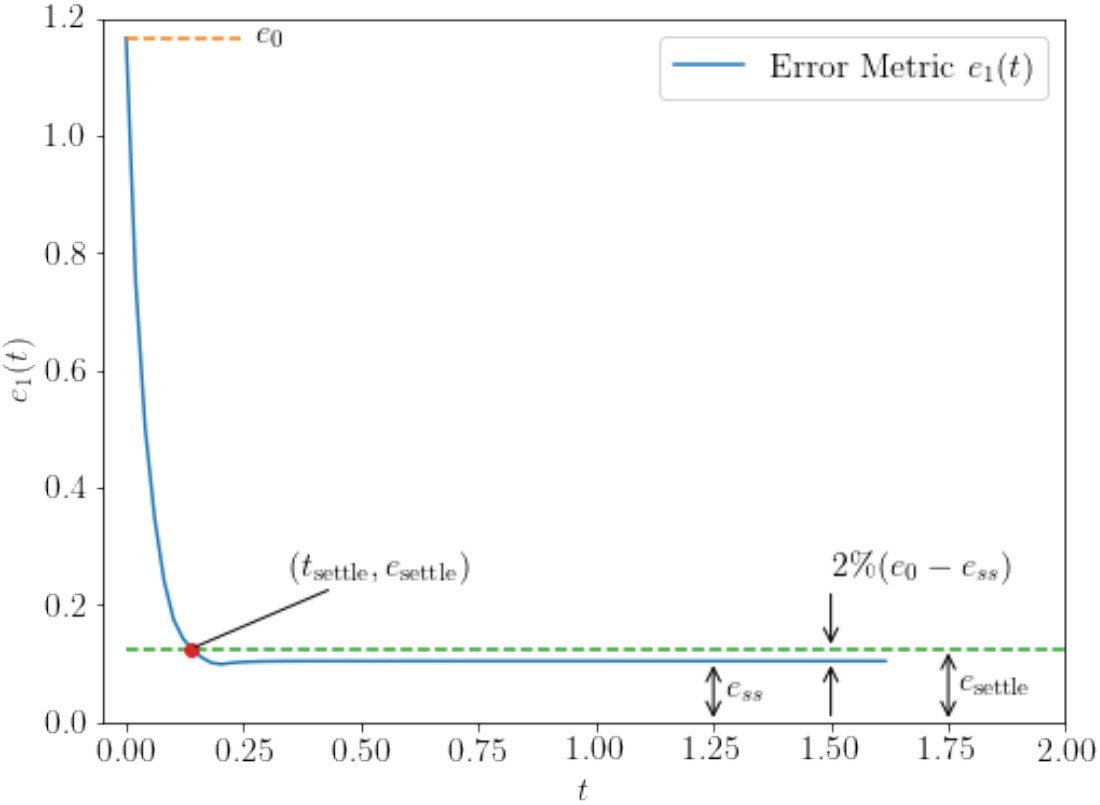

Figure 11. Depiction of the 2% settling time of the error metric.

3.3 Optimal Swarm Configuration

Note that it may not be possible, given swarm parameters such as the number of robots and each robot's radius of influence, to reduce the value of an error metric to zero. To better interpret whether the achieved value of an error metric is acceptable, we use numerical optimization to find the minimum possible value of the error metric.

[ ]:
```
from scipy.optimize import minimize

# Compute the optimal swarm configuration that minimizes error metric e1
res = minimize(e1_distance, xi_n_f, bounds=[bounds]*len(xi_n_f))
assert(res.success == True)
e1_min = res.fun

# Plot the swarm blob function achieved by the controller against the optimal
# swarm blob function
```

```
plt.plot(xi_n_f, np.zeros_like(xi_n_f), 'oC3',
         res.x, np.zeros_like(res.x), '.C6',
         x_n, rho_e_n(x_n, xi_n_f), '-C1',
         x_n, rho_e_n(x_n, res.x), '-C6',
         x_n, rho_n(x_n), '-C2')
plt.xlabel(r'$x$')
plt.ylabel(r'$\rho(x)$')
plt.legend(("Final Robot Position", "Optimal Robot Position",
            "Swarm Blob Function", "Optimal Blob Function",
            "Target Distribution"),
           bbox_to_anchor=(1.04,1), loc="upper left")
caption("Comparison of the steady state swarm coverage achieved by our "
         "controller and the optimal robot placements that minimize $e_1$.")
```

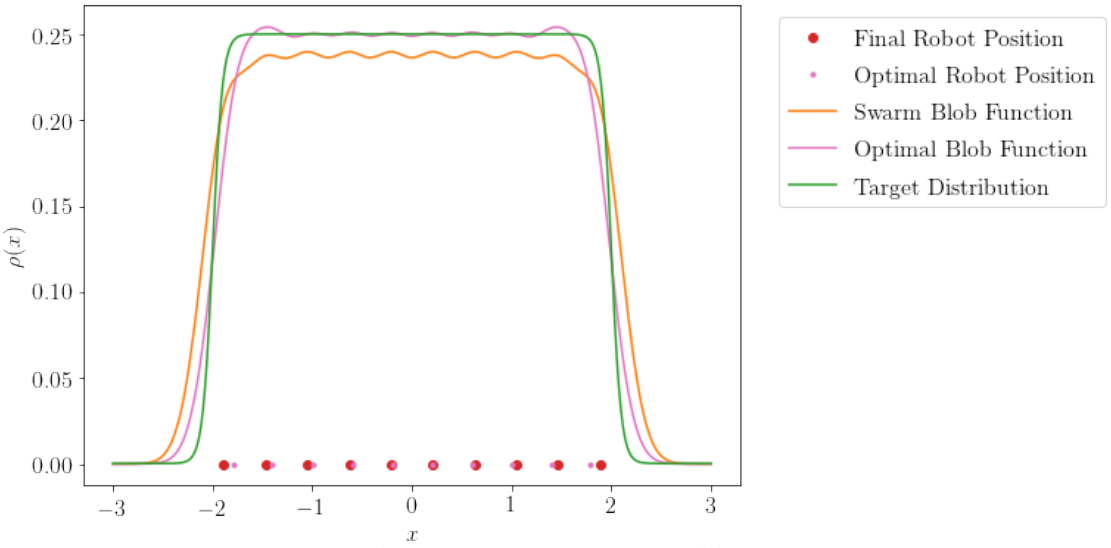

Figure 12. Comparison of the steady state swarm coverage achieved by our controller and the optimal robot placements that minimize e_1.

3.4 Simulation Termination

For a large swarm, the error metric is not very sensitive to position changes of individual robots, so it is not a reliable way to detect when the swarm distribution has stopped evolving and the simulation should end. Instead, we stop the simulation when the L^1-norm of the robot velocities have fallen below a threshold, which is manually chosen to maintain accuracy without requiring unnecessarily long simulation.

[ ]:
```
# stop the simulation when all velocities fall below this threshold
v_end = 0.01  # works well in practice
def event(t, yi):
  '''Terminal event function for integration'''
  return np.sum(np.abs(dydt(t, yi))) - v_end
```

```
event.terminal = True
event.direction = -1

# Simulate until event function is triggered
t = np.linspace(0, 20, 500)
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                t_eval=t, events=event, method='BDF')
tf = sol.t[-1]
xf = sol.y[:, -1]
e1f = trapz(np.abs(rho_n(x_n) - rho_e_n(x_n, xf)), x_n)
print("The swarm stopped evolving at t = {0:.3f}".format(tf))

# Simulate for much longer
t = np.linspace(0, tf*101, 500)
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                t_eval=t, method='BDF')  # no event detection
t_inf = sol.t[-1]
x_inf = sol.y[:, -1]
e_inf = trapz(np.abs(rho_n(x_n) - rho_e_n(x_n, sol.y[:, -1])), x_n)

# Compare the two results
max_error_change = (e1f-e_inf)/e_inf*100
max_robot_displacement = float(np.max(np.abs(x_inf - xf))/np.diff(bounds)*100)

print("Over the next {0:.3f} seconds, the value of the error metric "
      "changes by {1:.3f}%".format(t_inf-tf, max_error_change))
print("and the maximum displacement of any robot "
      "is {0:.3f}%.".format(max_robot_displacement))
```

```
The swarm stopped evolving at t = 1.082
Over the next 108.216 seconds, the value of the error metric changes by -0.000%
and the maximum displacement of any robot is 0.008%.
```

3.5 Independent Variables

We are currently exploring the effects of the following independent variables on the accuracy and speed of our control laws.

3.5.1 Target Distribution

The example above is for a uniform target distribution, but the control law can also guide the swarm to achieve a prescribed non-uniform coverage. How do the properties of the target distribution, such as smoothness, affect controller performance?

[ ]:
```
def derive_functions(rho, bounds, backend="numpy"):
  '''Derive the controller for a given target distribution'''

  # same as before, but collected in a function
```

```
  boxcar = H(x - bounds[0]) - H(x - bounds[1]) + 0.001
  rho = rho*boxcar

  a = 1/rho
  da = sympy.diff(a, x)
  drho = sympy.diff(rho, x)
  dphi_i = sympy.diff(phi_i, x)

  a_n = sympy.lambdify(x, a, modules=backend)
  da_n = sympy.lambdify(x, da, modules=backend)
  rho_n = sympy.lambdify(x, rho, modules=backend)
  drho_n = sympy.lambdify(x, drho, modules=backend)
  phi_i_n = sympy.lambdify((x, xi, eps), phi_i, modules=backend)
  dphi_i_n = sympy.lambdify((x, xi, eps), dphi_i, modules=backend)

  return a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n

rho = (x + 2)/8  # a different target distribution: the ramp function
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds)
t = np.linspace(0, 2, 100)
dydt = lambda t, y: v(y)
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                t_eval=t, events=event, method='BDF')

plot_final_blob_function(sol)
caption("Steady state swarm coverage (blob function) for a ramp target "
        "distribution.")
```

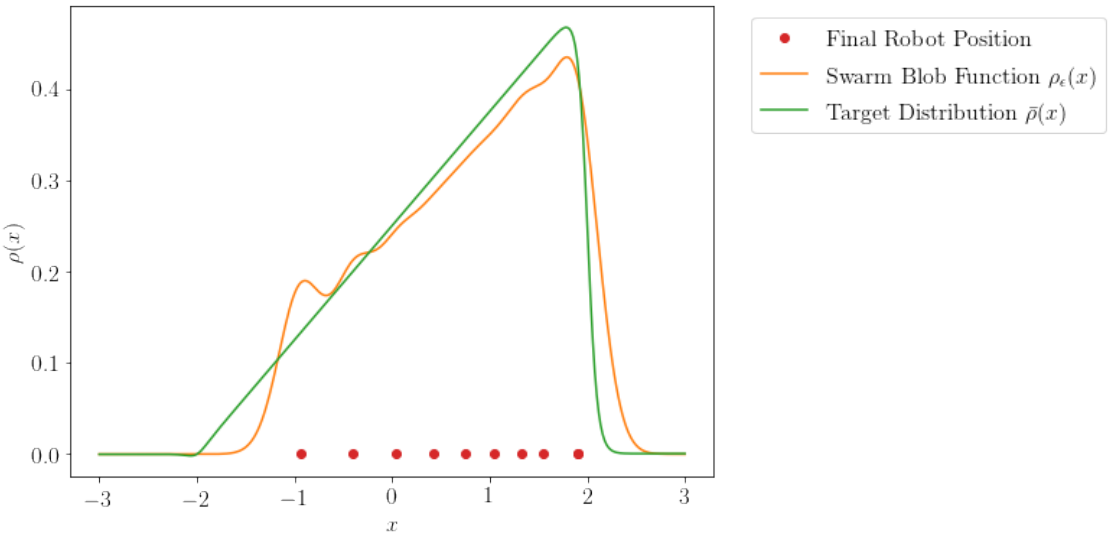

Figure 13. Steady state swarm coverage (blob function) for a ramp target distribution.

3.5.2 Initial Swarm Configuration

The example above begins with all robots clustered about the origin. Intuitively the initial configuration will affect the settling time, but how much? Does it affect the final configuration?

[ ]:
```
# back to the uniform target distribution
rho = Rational(1,4)
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds)

xi_0_alt = 1.5 + np.random.rand(10)*0.1  # a different initial configuration
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0_alt,
                t_eval=t, events=event, method='BDF')
plot_traces(sol)
caption("Trajectories of robots driven by our controller toward a uniform "
        "distribution.")
```

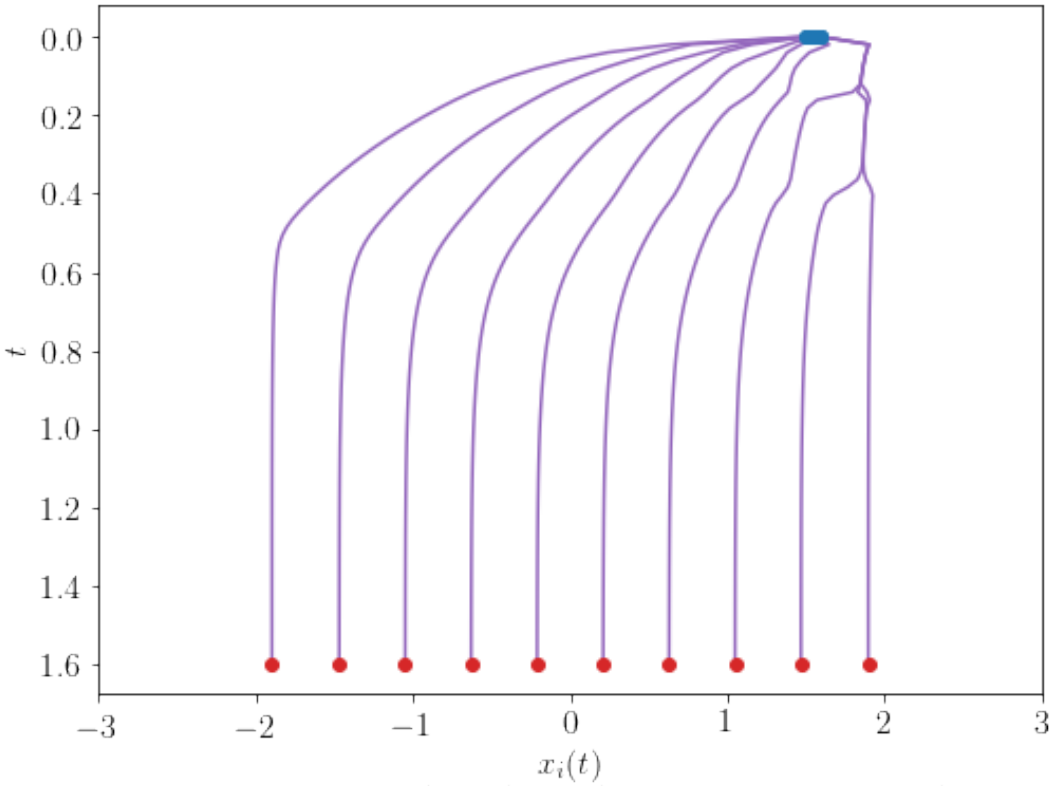

Figure 14. Trajectories of robots driven by our controller toward a uniform distribution.

3.5.3 Robot Speed Limits

In the example above, there were no limitations on a robot's speed. Without a speed limit, measuring the settling time is not very meaningful, as any controller can be trivially modified to achieve

any desired settling time: simply scale all robot velocities by a constant factor. However, these controllers tend to work even when a speed limit is imposed on the robots. Which performs best with this limitation?

[ ]:
```
def dydt(t, y):
  '''Calculate robot velocities subject to a speed limit'''
  v_max = 1  # speed limit
  v_prescribed = v(y)
  v_actual = v_prescribed
  v_actual[v_actual > v_max] = v_max
  v_actual[v_actual < v_max] = -v_max
  return v_actual

sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                t_eval=t, events=event, method='RK45')

plot_traces(sol)
caption("Trajectories of speed-limited robots governed by our controller.")
```

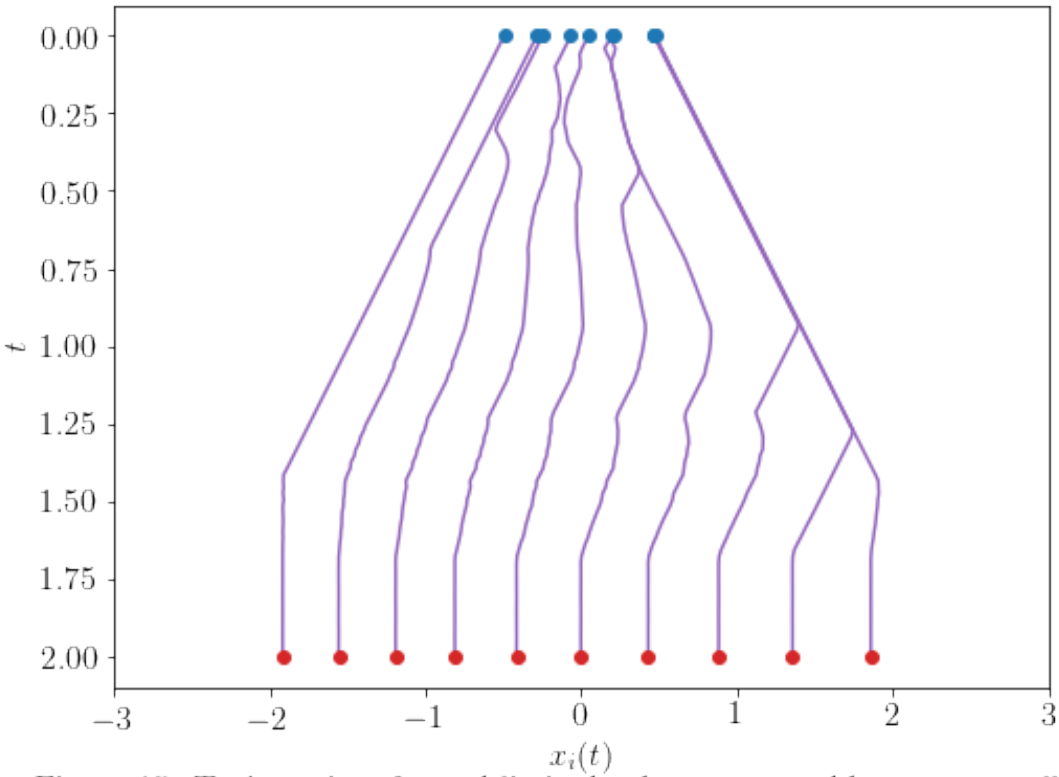

Figure 15. Trajectories of speed-limited robots governed by our controller.

3.5.4 Mollifier Function ϕ and Influence Radius ϵ

Previously, we used the Gaussian function as the mollifier ϕ to describe how the robot's influence diminishes as a function of distance. The Gaussian function is always nonzero, that is, it has *noncompact support*. Real robots, however, will have limited sensing range and influence, so we are interested in the use of mollifiers with *compact support*. If robots are not able to sense the influence of all others in the swarm, will the swarm still converge to the desired distribution? Do some mollifiers, such as those used as kernels in nonparametric statistical estimation, perform better than others?

[ ]:
```
# Gaussian mollifier
phi_gaussian = phi
phi_i_gaussian = 1/eps**d*phi_gaussian.subs(x, (x-xi)/eps)
phi_i_n_gaussian = sympy.lambdify((x, xi, eps), phi_i_gaussian, modules="numpy")

# A Triweight mollifier
phi_triweight = sympy.Piecewise(
    (0, x < -2),
    (0, x > 2),
    (Rational(35/64)*(1-(x/2)**2)**3, True )
)
phi_i_triweight = 1/eps**d*phi_triweight.subs(x, (x-xi)/eps)
phi_i_n_triweight = sympy.lambdify((x, xi, eps), phi_i_triweight,␣
 ↪modules="numpy")

# Plot the two mollifiers
plt.xlabel(r'$x$')
plt.ylabel(r'$\phi_\epsilon$')
plt.plot(x_n, phi_i_n_gaussian(x_n, 0, 1),
         x_n, phi_i_n_triweight(x_n, 0, 1))
plt.legend((r'Gaussian Mollifier (noncompact)',
            r'Triweight Mollifier (compact)'),
           bbox_to_anchor=(1.04,1), loc="upper left")
caption("Comparison of Gaussian and triweight mollifier functions.")
```

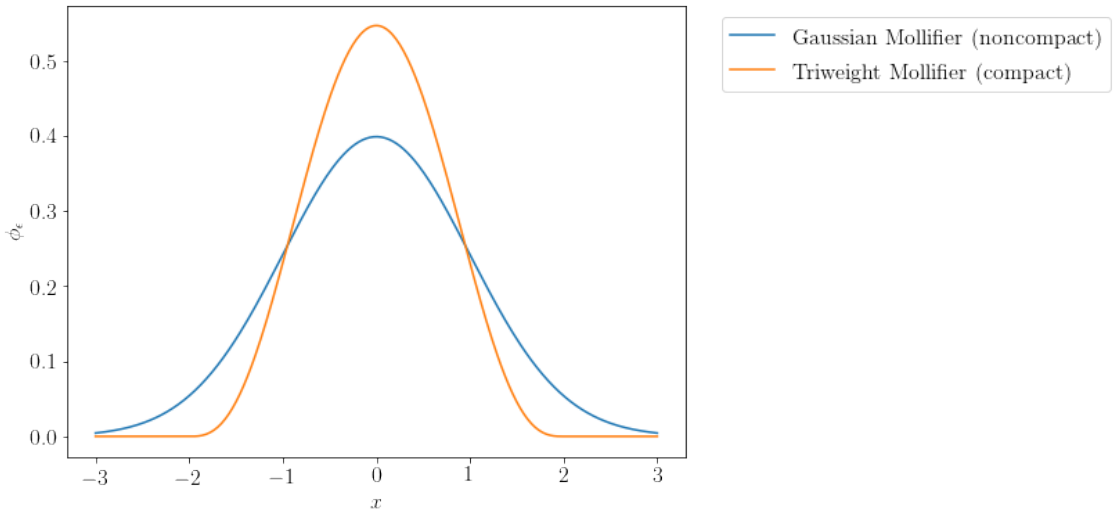

Figure 16. Comparison of Gaussian and triweight mollifier functions.

[ ]:
```
# Use the triweight mollifier to develop the controller
phi_i = 1/eps**d*phi_triweight.subs(x, (x-xi)/eps)
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds)

# Simulate the system
dydt = lambda t, y: v(y)
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                t_eval=t, events=event, method='BDF')
plot_final_blob_function(sol)
caption("Steady state swarm coverage (blob function) achieved by our "
        "controller using a triweight mollifier.")
```

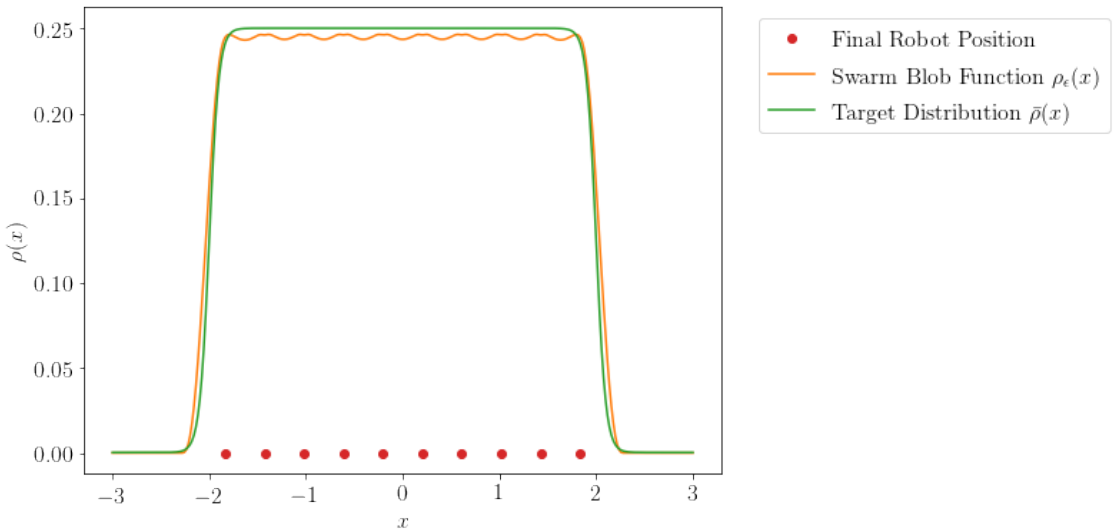

Figure 17. Steady state swarm coverage (blob function) achieved by our controller using a triweight mollifier.

3.5.5 Swarm Size

As the number of robots increases, the ability of the swarm to achieve the desired distribution should improve, but at what rate? How many robots are needed to achieve a desired accuracy?

[ ]:
```
# Back to Gaussian mollifier
phi_i = 1/eps**d*phi_gaussian.subs(x, (x-xi)/eps)
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds)

# Simulate the swarm for range of swarm sizes
np.random.seed(4)
n_rs = np.array([10, 20, 40, 80]) # np.arange(10, 45, 5)
e2s = []
for n_r in n_rs:
  e = np.random.rand(n_r)-0.5
  xi_0 = 0 + e
  sol = solve_ivp(dydt, (t[0], t[-1]), xi_0,
                  t_eval=t, events=event, method='BDF')
  xi_n_f = sol.y[:, -1]
  if n_r in {10, 20, 40}:
    plt.plot(x_n, rho_e_n(x_n, xi_n_f))
  e2s.append(e2_distance(xi_n_f))

# Plot the swarm blob function for several swarm sizes
plt.plot(x_n, rho_e_n(x_n, xi_n_f), '--')
plt.xlabel(r'$x$')
plt.ylabel(r'$\rho(x)$')
plt.legend(("Swarm Blob Function ($n_r = 10$)",
            "Swarm Blob Function ($n_r = 20$)",
            "Swarm Blob Function ($n_r = 40$)",
            "Target Distribution"),
          bbox_to_anchor=(1.04,1), loc="upper left")
caption("Swarm coverage (blob function) achieved by our controller for "
        "several swarm sizes.")
```

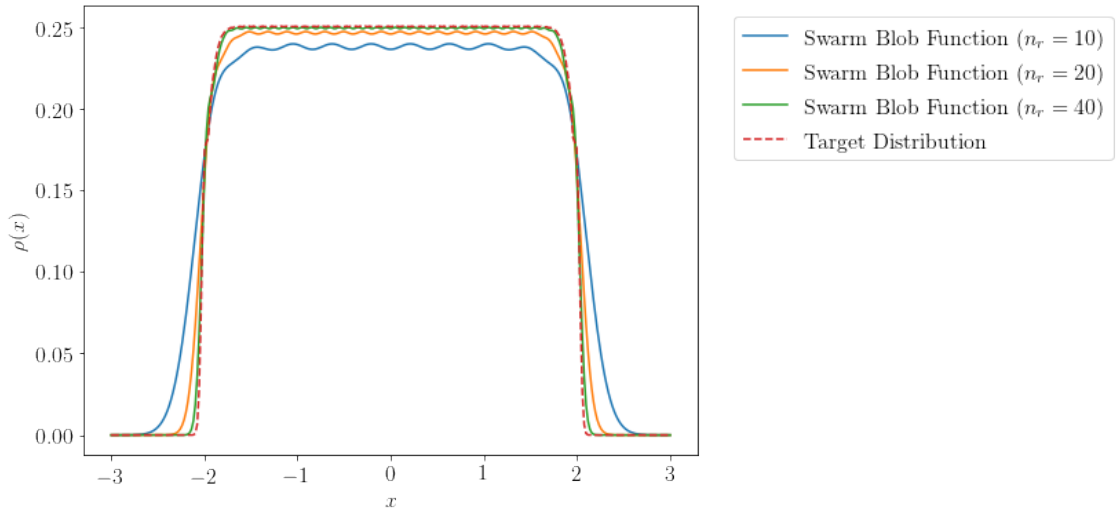

Figure 18. Swarm coverage (blob function) achieved by our controller for several swarm sizes.

[ ]:
```
# Plot the convergence of error metric e2 as swarm size increases
n_rs = np.array(n_rs)
plt.loglog(n_rs, e2s, '.-',
           n_rs, np.sqrt(0.1)/np.sqrt(n_rs), '--',
           n_rs, 1/n_rs, '--',
           n_rs, 10/n_rs**2, '--')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Final Value of Error Metric $e_2$")
plt.legend(("Experimental Data",  r"$e_2 \propto n_r^{-1/2}$", r"$e_2 \propto␣
 ↪n_r^{-1}$", r"$e_2 \propto n_r^{-2}$"))
caption("Convergence of error metric $e_2$ as swarm size $n_r$ increases. "
        "The steady state value of the error metric appears to decrease as "
        "the inverse of the swarm size.")
```

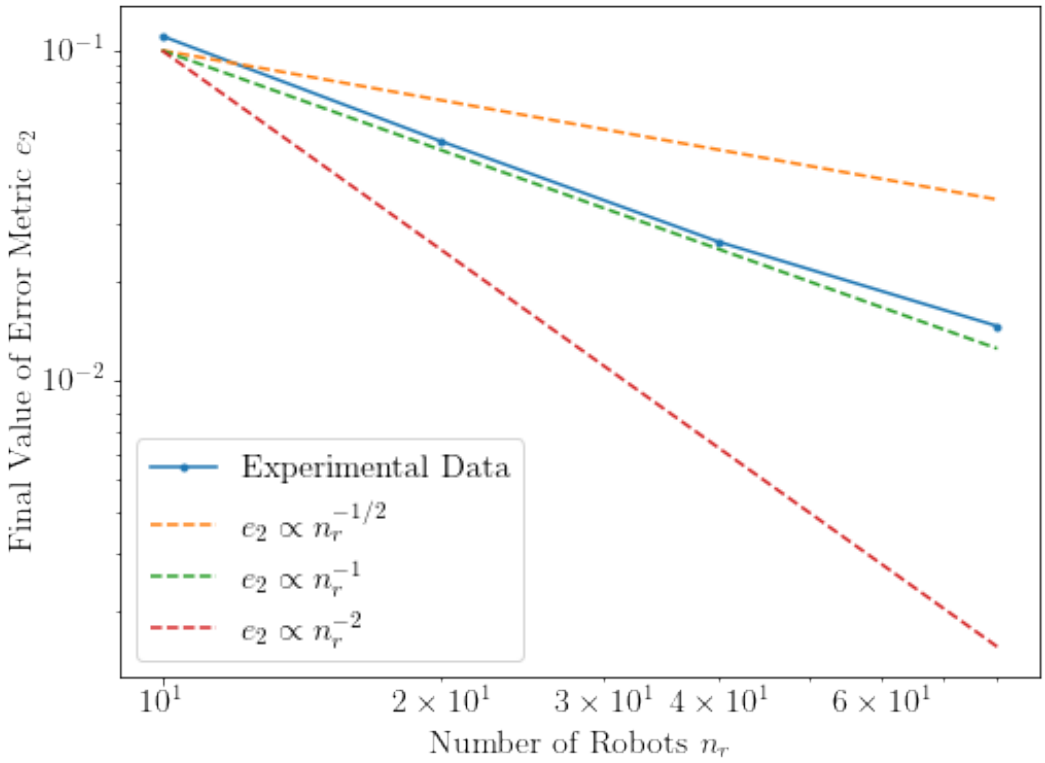

Figure 19. Convergence of error metric e_2 as swarm size n_r increases. The steady state value of the error metric appears to decrease as the inverse of the swarm size.

Numerical results will guide analytical study of the mathematical properties of these controllers.

4 Computational Discussion

4.1 Computation Complexity of Simulation

We must simulate large swarms to investigate the effect of swarm size on controller speed and accuracy, but without care, this becomes very computationally expensive. Based on the form of (3), we would expect the computational complexity to be at least $O(n_r^2)$: in order to calculate the velocity v_i for each of n_r robots i, we must sum contributions from n_r robots j.

[ ]:
```
from timeit import timeit

# Time velocity computation for a range of swarm sizes
n_rs = 10*2**np.arange(12)
v_times = []
for n_r in n_rs:
  t_0 = 0
  xi_0 = np.random.rand(n_r)-0.5
```

```
  def timethis():
    dydt(t_0, xi_0)
  time = timeit(timethis, number=1)
  v_times.append(time)

# Plot the results
plt.loglog(n_rs, v_times, '.-', n_rs, 1e-7*n_rs**2, '--')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Velocity Calculation Time $T_v$")
plt.legend(("Measured Velocity Calculation Time $T_v$",  r"$T_v \propto
 ↪O(n_r^2)$"))
caption("Computation time $T_v$ of robot velocities as a function of swarm size "
        "$n_r$. As expected, $T_v$ appears to increase with the square of the "
        "number of robots.")
```

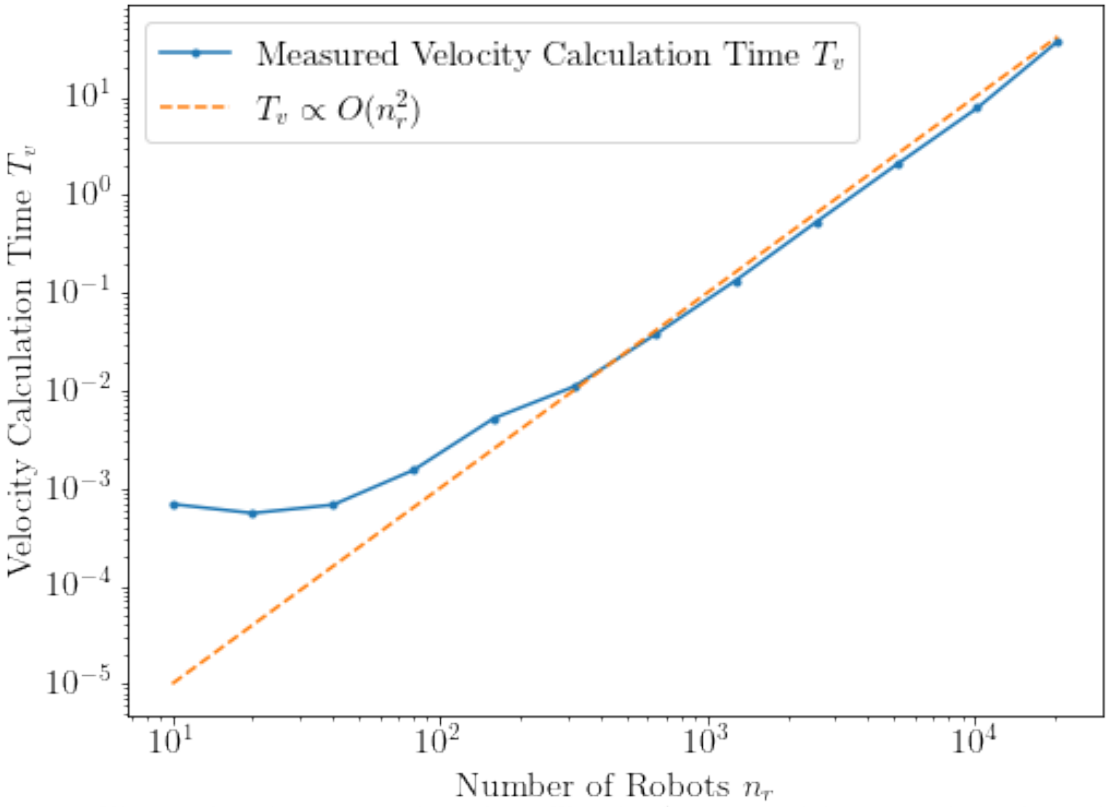

Figure 20. Computation time T_v of robot velocities as a function of swarm size n_r. As expected, T_v appears to increase with the square of the number of robots.

However, total simulation time increases even faster than that.

[ ]:
```
# Measure simulation execution time for a range of swarm sizes
# Track execution time, swarm velocity computations, and Jacobian calculations
exec_times = []
v_count = []
J_count = []

t = np.linspace(0, 2, 100)

def dydt(t, y):
  dydt.count += 1
  return v(y)
dydt.count = 0

n_rs = (10*2**np.arange(0, 5.5, 0.5)).astype(int)
for n_r in n_rs:
  xi_0 = np.random.rand(n_r)-0.5

  def timethis():
    sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, events=event, method='BDF')
    J_count.append(sol.njev)  # forgive me for including these in the timed␣
 ↪function

  time = timeit(timethis, number=1)
  exec_times.append(time)
  v_count.append(dydt.count)

# Plot the rise in execution time against the rise in swarm size
plt.loglog(n_rs, exec_times, '.-', n_rs, exec_times[0]/100*n_rs**2, '--')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Simulation Execution Time $T$")
plt.legend(("Measured Simulation Execution Times $T$",
            r"$T \propto O(n_r^2)$"))
caption("Simulation execution time $T$ as a function of "
        "swarm size $n_r$. For large $n_r$, $T$ appears to increase faster "
        "than the square of the swarm size.")
```

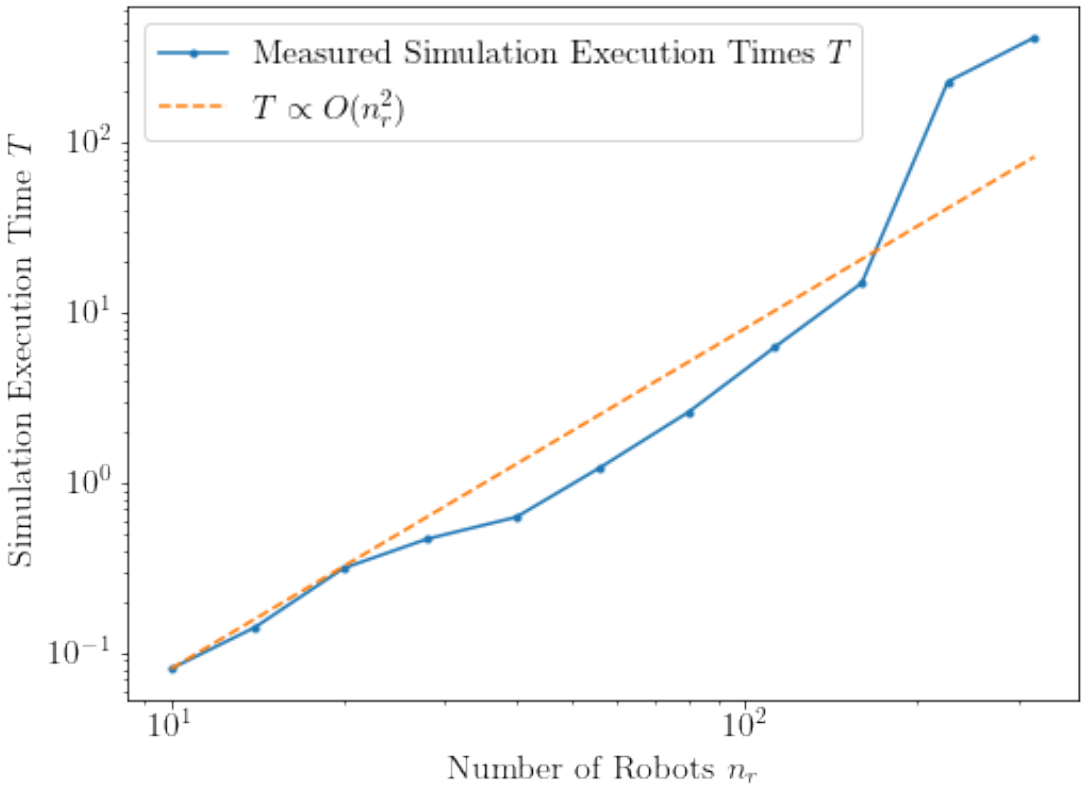

Figure 21. Simulation execution time T as a function of swarm size n_r. For large n_r, T appears to increase faster than the square of the swarm size.

The increase in slope at the end is not noise; rather, the execution time T is increasing faster than $O(n_r^2)$ for n_r sufficiently large. This additional computational complexity comes from a few factors, the most significant of which is that the number of velocity function executions required also increases with the swarm size.

[ ]:
```
plt.loglog(n_rs, v_count, '.-',
           n_rs, v_count[0]/n_rs[0]*n_rs, '--',
           n_rs, v_count[0]/n_rs[0]**2*n_rs**2, '--',)
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Velocity Function Evaluations $n_v$")
plt.legend(("Measured Velocity Function Evaluations $n_v$",
            r"$n_v \propto O(n_r)$",
            r"$n_v \propto O(n_r^2)$"))
caption("The number of swarm velocity computations $n_v$ increases with "
        "swarm size $n_r$.")
```

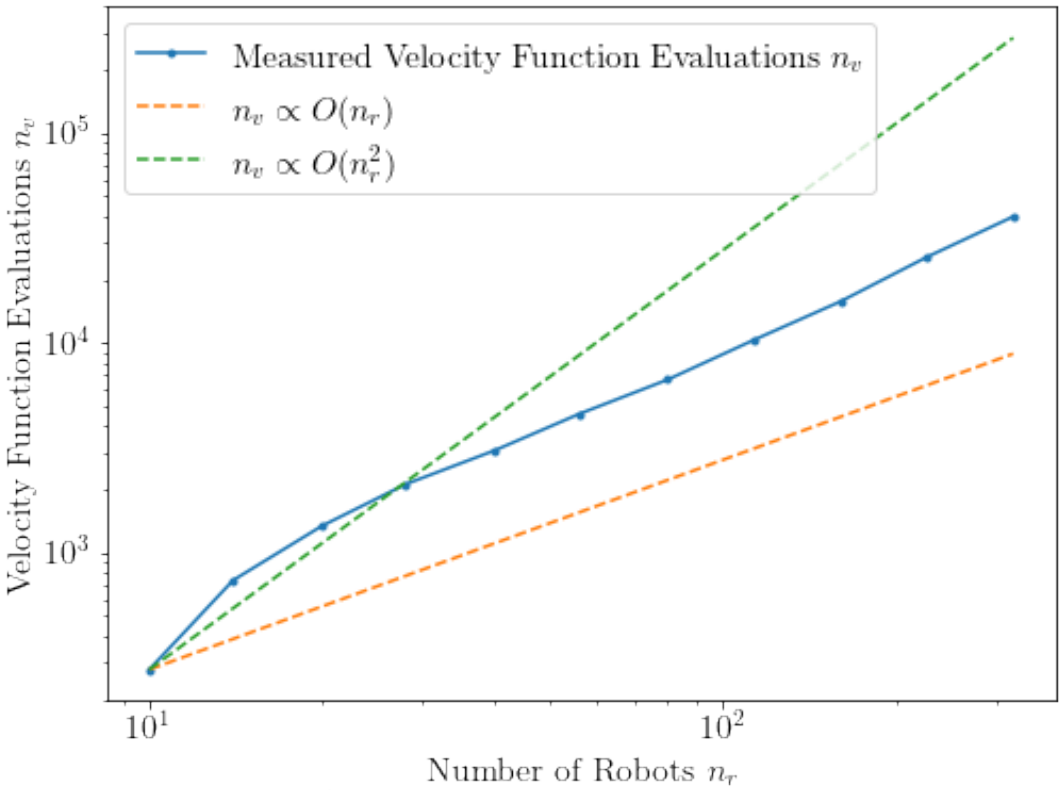

Figure 22. The number of swarm velocity computations n_v increases with swarm size n_r.

To see why, we need to address the choice of ODE solver. In most simulations above, the choice has been `'BDF'` because the ODEs are known to be stiff. It is quite clear that an implicit solver is the appropriate choice when we compare the performance of all the solvers.

[ ]:
```
# Time simulation with several ODE solvers
n_rs3 = 10*2**np.arange(3)

explicit_solvers = ['RK23', 'RK45', 'DOP853']
explicit_colors = ['limegreen', 'forestgreen', 'darkgreen']
implicit_solvers = ['Radau', 'BDF', 'LSODA']
implicit_colors = ['lightblue', 'skyblue', 'steelblue']
solvers = explicit_solvers + implicit_solvers
colors = explicit_colors + implicit_colors
solver_times = {}
legend = []

for solver, color in zip(solvers, colors):
  solver_times[solver] = []
  for n_r in n_rs3:
    np.random.seed(0)
```

```
    xi_0 = np.random.rand(n_r)-0.5
    def timethis():
      sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, events=event, method=solver)
    time = timeit(timethis, number=1)
    solver_times[solver].append(time)
  plt.loglog(n_rs3, solver_times[solver], '.-', color=color)
  legend.append(solver)

# Plot the results
legend.append('$T \propto O(n_r^2)$')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Simulation Execution Time $T$")
plt.legend(legend, bbox_to_anchor=(1.04,1), loc="upper left")
caption("Simulation execution time $T$ as a function of "
        "swarm size $n_r$ for several ODE solvers. For our controller, the "
        "implicit methods (e.g. Radau, Backwards Differentiation Formula) are "
        "faster than the explicit methods (e.g. Runge-Kutta, Dormand-Prince).")
```

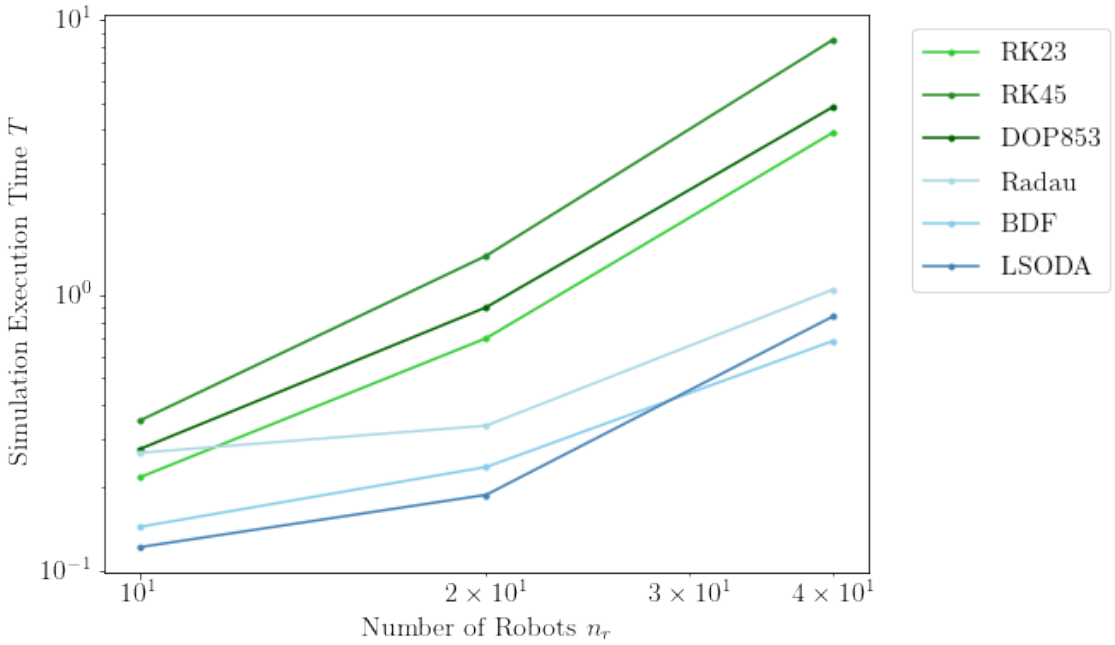

Figure 23. Simulation execution time T as a function of swarm size n_r for several ODE solvers. For our controller, the implicit methods (e.g. Radau, Backwards Differentiation Formula) are faster than the explicit methods (e.g. Runge-Kutta, Dormand-Prince).

The explicit solvers (green) are much slower than the implicit solvers(blue), and the gap appears to grow as the swarm size increases. We will continue our discussion assuming use of the BDF solver, as that has typically been the fastest in practice, but the discussion below applies to all of the implicit solvers.

Because BDF is an implicit solver, it needs to solve an equation $f\left(x(t_k), x(t_{k+1})\right) = 0$ for $x(t_{k+1})$

at each time step t_k, and to do so it must evaluate the Jacobian of f, that is, the matrix of partial derivatives $\frac{\partial f_i}{\partial x_j}$. As implemented above, `solve_ivp` does this using finite differences, which involves evaluating the robot velocities $O(n_r)$ times. As discussed above, evaluating the robot velocities is itself an $O(n_r^2)$ operation according to (3). Hence, the computational complexity is at least $O(n_r^3)$.

Fortunately, the number of Jacobian evaluations seems to rise relatively slowly with the swarm size.

[ ]:
```
plt.plot(n_rs, J_count, '.-',
         n_rs, 2*np.sqrt(n_rs), '--',
         n_rs, 0.7*n_rs, '--')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Number of Jacobian Evaluations $n_J$")
plt.legend(("Measured Number of Jacobian Evaluations $n_J$",
            r"$n_J \propto O(\sqrt{n_r})$",
            r"$n_J \propto O(n_r)$"))
caption("Number of Jacobian evaluations during integration as a function of "
        "swarm size. Fortunately, the number of Jacobian evaluations $n_J$ "
        "does not appear to increase with swarm size very rapidly.")
```

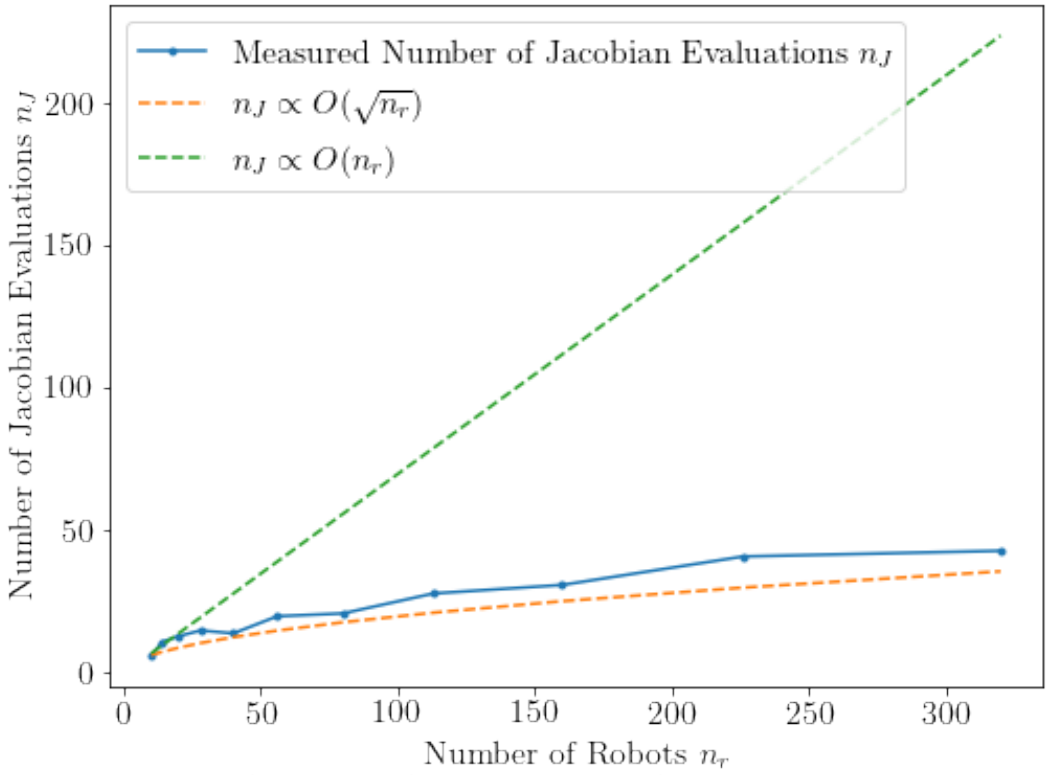

Figure 24. Number of Jacobian evaluations during integration as a function of swarm size. Fortunately, the number of Jacobian evaluations $n_J$does not appear to increase with swarm size very rapidly.

4.2 Simulation Speed Improvements

Now that we understand why the execution takes so long, we can begin to address it.

4.2.1 NumExpr

Using NumExpr instead of NumPy as the array math backend provides some relief when multiple CPU cores are available.

[ ]:
```
# Rederive controller and use NumExpr as the backend for computation
n_rs = 10*2**np.arange(12)
phi = 1/sqrt(2*np.pi)*exp(-x**2/2)  # numexpr doesn't define pi
phi_i = 1/eps**d*phi.subs(x, (x-xi)/eps)
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds,␣
 ↪backend="numexpr")

# Measure velocity calculation time for a range of swarm sizes
v_times_numexpr = []
for n_r in n_rs:
  t_0 = 0
  xi_0 = np.random.rand(n_r)-0.5
  def timethis():
    dydt(t_0, xi_0)
  time = timeit(timethis, number=1)
  v_times_numexpr.append(time)

# Plot the results
plt.semilogx(n_rs, np.divide(v_times, v_times_numexpr), '.-')
plt.ylim(0, 2)
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("NumExpr Speedup")
caption("Ratio of velocity calculation times using NumPy vs NumExpr. NumExpr "
        "tends to provide a modest but noticeable speedup for large swarm "
        "sizes, especially on platforms with several available CPU cores.")

# Figure saved with notebook was generated using a Colab Pro instance.
# For larger swarms, the observed speedup is typically between 1.25 and 2.5x.
```

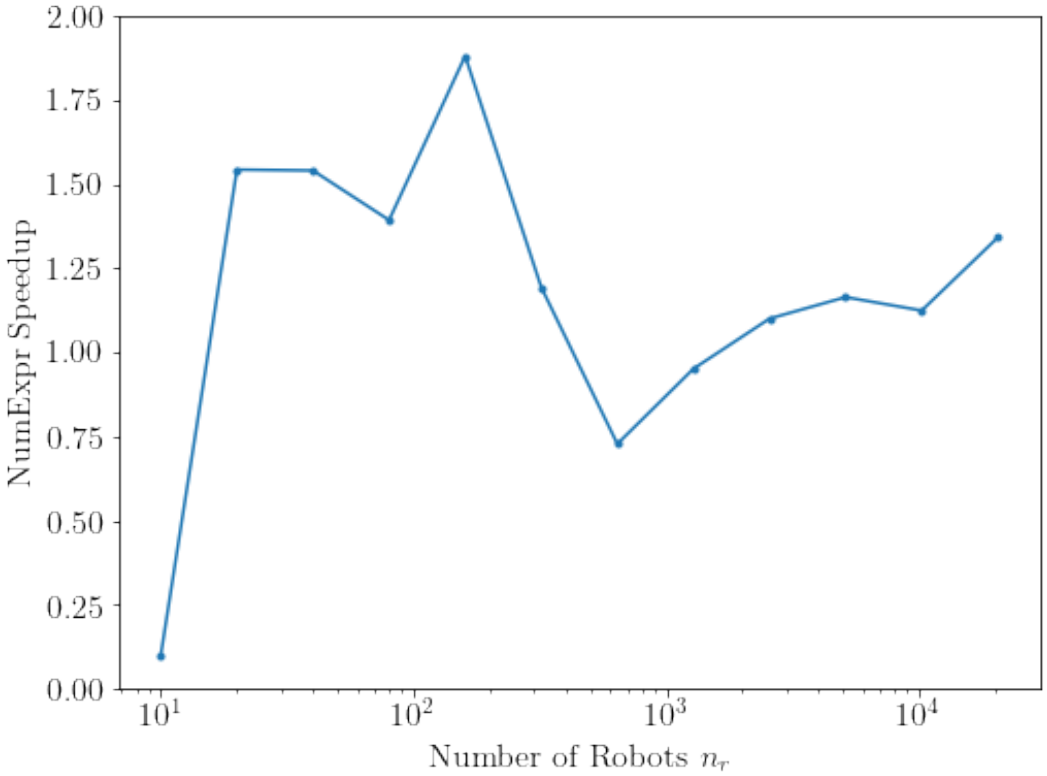

Figure 25. Ratio of velocity calculation times using NumPy vs NumExpr. NumExpr tends to provide a modest but noticeable speedup for large swarm sizes, especially on platforms with several available CPU cores.

4.2.2 CuPy

Alternatively, use of a GPU to perform the velocity calculations (via CuPy) is very promising.

[ ]:
```
import cupy as cp

# Rederive controller and specify no backend for computation; this allows
# CuPy to be used
a_n, da_n, rho_n, drho_n, phi_i_n, dphi_i_n = derive_functions(rho, bounds,␣
 ↪backend=None)

# Measure velocity computation time for a range of swarm sizes
n_rs = 10*2**np.arange(12)
v_times_cupy = []
for n_r in n_rs:
  t_0 = 0
  xi_0 = np.random.rand(n_r)-0.5
  def timethis():
    # convert position array to a CuPy array and calculate velocities
```

```
    speed = v(cp.array(xi_0), xp=cp)
    # retrieve velocities as a NumPy array
    speed_numpy = cp.asnumpy(speed)
  time = timeit(timethis, number=1)
  v_times_cupy.append(time)

# Plot the results
plt.semilogx(n_rs, np.divide(v_times, v_times_cupy), '.-')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("CuPy Speedup")
caption("Ratio of swarm velocity computation times using NumPy (CPU) vs CuPy "
        "(GPU). CuPy dramatically improves the speed of the swarm velocity "
        "computation.")

# Figure saved with notebook was generated using a Colab Pro instance w/
# Tesla P100 GPU.
```

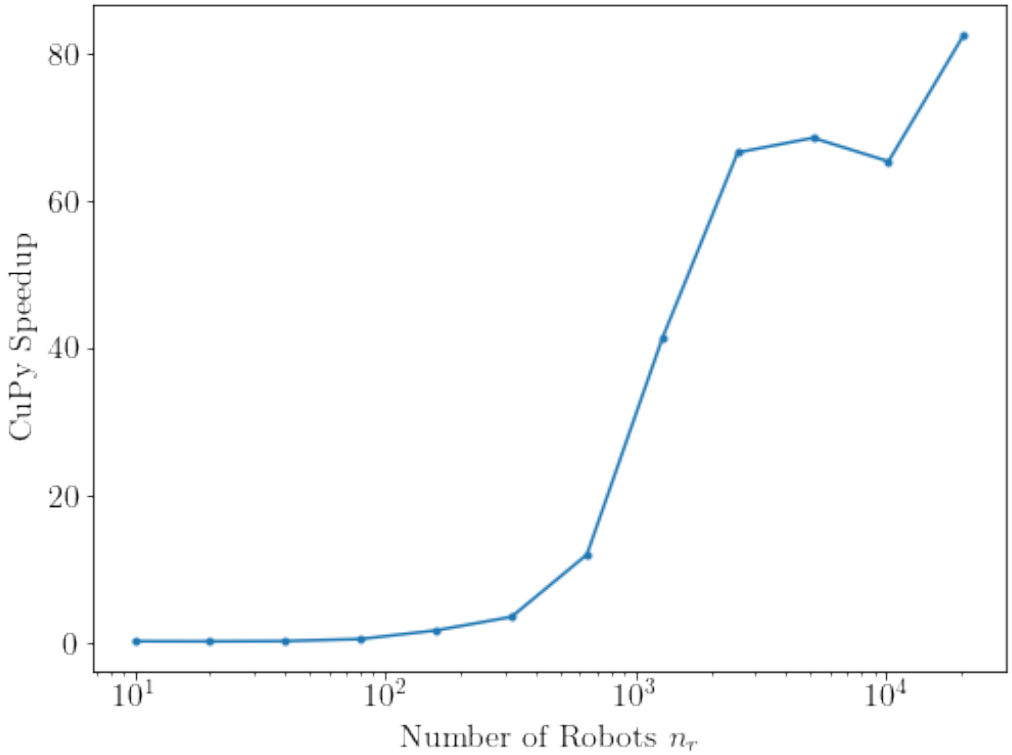

Figure 26. Ratio of swarm velocity computation times using NumPy (CPU) vs CuPy (GPU). CuPy dramatically improves the speed of the swarm velocity computation.

The speedup levels off as all the cores of the GPU become engaged, but even a constant speedup this large is sufficient to substantially increase the size of swarm that can be simulated in reasonable time.

Unfortunately, the speedup in velocity calculation is not immediately reflected in the integration times.

[ ]:
```
# Time simulation using CuPy backend for a range of swarm sizes
exec_times_cupy = []
t = np.linspace(0, 2, 100)

def dydt(t, y):
  return cp.asnumpy(v(cp.array(y)))

n_rs = n_rs = (10*2**np.arange(0, 5.5, 0.5)).astype(int)
for n_r in n_rs:
  xi_0 = np.random.rand(n_r)-0.5

  def timethis():
    sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, events=event, method='BDF')

  time = timeit(timethis, number=1)
  exec_times_cupy.append(time)

# Compare against simulation with NumPy backend
plt.loglog(n_rs, exec_times, '.-', n_rs, exec_times_cupy, '.-')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Simulation Execution Time $T$")
plt.legend(("Simulation Execution Time (NumPy)", "Simulation Execution Time␣
 ↪(CuPy)"))
caption("Simulation execution times using NumPy (CPU) vs CuPy "
        "(GPU). The maximum speedup (for $n_r={0}$ robots) is ${1:.2f}\\times$."
        "".format(n_rs[-1], exec_times[-1]/exec_times_cupy[-1]))

# Figure saved with notebook was generated using a Colab Pro instance w/
# Tesla P100 GPU.
```

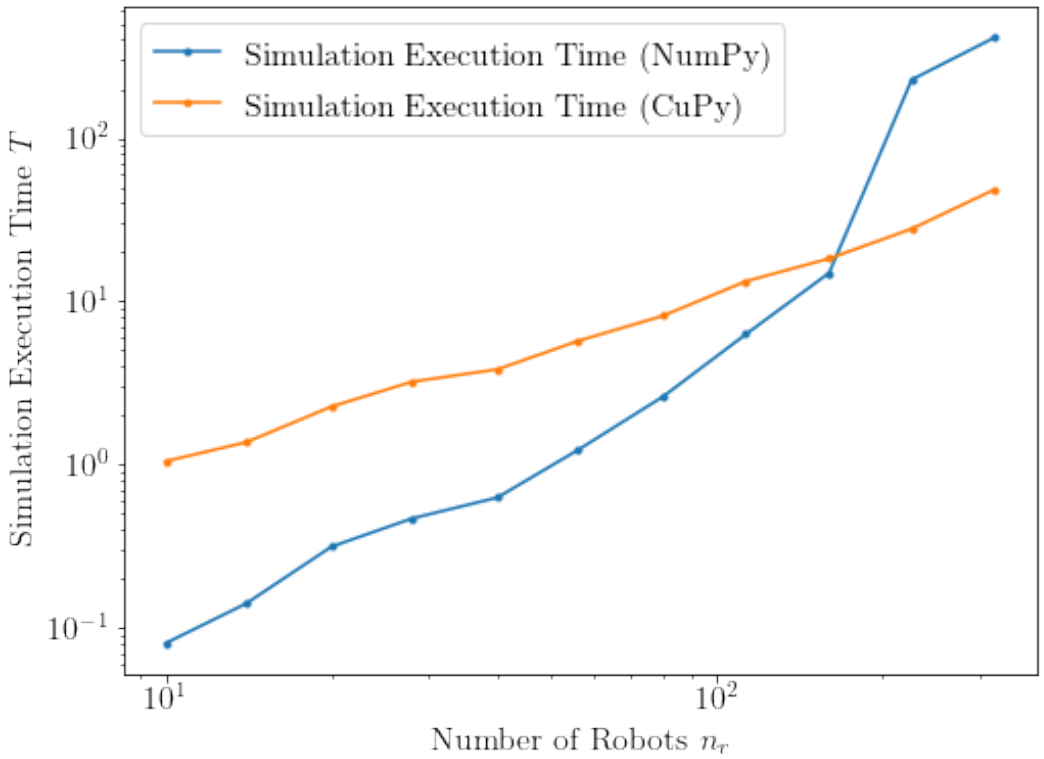

Figure 27. Simulation execution times using NumPy (CPU) vs CuPy (GPU). The maximum speedup (for $n_r = 320$ robots) is 8.53×.

While the GPU certainly improves performance for large swarms, the speedup (up to 8.5×) is not quite as impressive as we would hope based on Figure 26. The reason is that every time `dydt` is called, the array containing the positions of the robots must by converted to a CuPy array and sent to the GPU. Upon completion, the evaluated velocities have to be returned to the CPU as a NumPy array in order for the integrator `solve_ivp` to solve for robot positions at the next time step. The overhead of this data conversion and transmission dramatically slows the integration. (We haven't found an integration suite in the SciPy ecosystem that keeps most or all computation on the GPU, so we are considering writing our own.)

In the meantime, we can substantially reduce the number of calls to `dydt`, and thus the overhead of communication with the GPU, using the `vectorized` option of `solve_ivp`. As mentioned above, each time the Jacobian is needed, `solve_ivp` calls `dydt` n times, each time with a different 1D array of n robot positions. By activating the `vectorized` option, `solve_ivp` will call `dydt` only once with a 2D array containing n columns of n robot positions.

[ ]:
```
# modifications to several previously defined functions are needed to enable
# vectorized calculation of the Jacobian, but the effort is worth it!

def dydt(t, y):
  # when vectorized=False, y is 1D (row)
  # when vectorized=True, y is 2D (column or full matrix)
  transpose = False
```

```
  if y.ndim > 1:
    y = y.T
    transpose = True

  y = cp.array(y)
  # calculations are done with 3D array; last dimension is singleton when done
  res = v(y)[:,:,0]
  dydt.count += 1

  # cache L1 norm of velocities for event function to avoid re-calculation
  if y.shape[0] == 1:
    dydt.v_mags[t] = np.sum(np.abs(res))

  # need to convert back to original shape
  if transpose:
    res = res.T
  else:
    res = res.ravel()

  return cp.asnumpy(res)
dydt.v_mags = {}
dydt.count = 0

v_end = 0.01
def event(t, yi, dydt=dydt):
  # calculate L1 norm of velocities only if not already cached
  if t not in dydt.v_mags:
    dydt.v_mags[t] = np.sum(np.abs(dydt(t, yi)))
  return dydt.v_mags[t] - v_end

def get_robot_mass_and_radius(xi_n):
  n_r = np.max(xi_n.shape)
  mi = 1/n_r
  eps_n = 2/n_r**0.95
  return mi, eps_n

def rho_e_n(x_n, xi_n):
  mi, eps_n = get_robot_mass_and_radius(xi_n)
  return np.sum(phi_i_n(x_n, xi_n, eps_n)*mi, axis = -1, keepdims=True)

def drho_e_n(x_n, xi_n):
  mi, eps_n = get_robot_mass_and_radius(xi_n)
  return np.sum(dphi_i_n(x_n, xi_n, eps_n)*mi, axis = -1, keepdims=True)

def v(xi_n):
  xi_n = np.atleast_2d(xi_n)
  m, n = xi_n.shape
```

```
  xi_n1 = xi_n.reshape(m, n, 1)
  xi_n2 = xi_n.reshape(m, 1, n)
  mi, eps_n = get_robot_mass_and_radius(xi_n)
  term1 = rho_e_n(xi_n1, xi_n2) * da_n(xi_n1)
  term2 = a_n(xi_n1) * drho_e_n(xi_n1, xi_n2)
  term3 = np.sum(mi * dphi_i_n(xi_n1, xi_n2, eps_n) *  a_n(xi_n2),
                 axis = -1, keepdims=True)
  return (-a_n(xi_n1)*(term1 + term2 + term3))

# Time simulation using CuPy backend for a few swarm sizes
exec_times_cupy2 = []
t = np.linspace(0, 2, 100)

n_rs = (10*2**np.arange(0, 5.5, 0.5)).astype(int)
for n_r in n_rs:
  xi_0 = np.random.rand(n_r)-0.5

  def timethis():
    sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, events=event,
                    vectorized=True, method='BDF')

  time = timeit(timethis, number=1)
  exec_times_cupy2.append(time)

# Compare the results against simulation with NumPy
plt.loglog(n_rs, exec_times, '.-',
           n_rs, exec_times_cupy2, '.-',
           n_rs, exec_times[0]/100*n_rs**2, '--')
plt.xlabel("Number of Robots $n_r$")
plt.ylabel("Simulation Execution Time $T$")
plt.legend(("Simulation Execution Time (NumPy)",
            "Simulation Execution Time (CuPy)",
            r"$T \propto O(n_r^2)$"))
caption("Simulation execution times using NumPy (CPU) vs CuPy "
        "(GPU) with vectorized Jacobian calculations. "
        "The maximum speedup with vectorized Jacobian calculations "
        "(for $n_r={0}$ robots) is ${1:.2f}\\times$."
        "".format(n_rs[-1], exec_times[-1]/exec_times_cupy2[-1]))

# Figure saved with notebook was generated using a Colab Pro instance w/
# Tesla P100 GPU. CuPy speedup is also impressive with the free version of
# Colab.
```

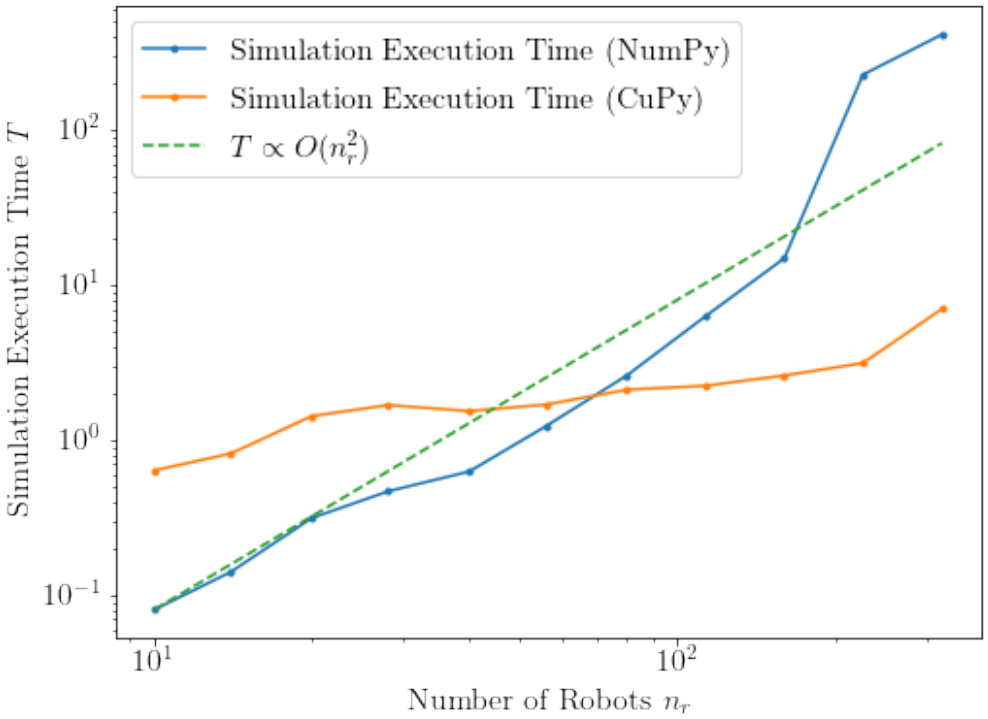

Figure 28. Simulation execution times using NumPy (CPU) vs CuPy (GPU) with vectorized Jacobian calculations. The maximum speedup with vectorized Jacobian calculations (for $n_r = 320$ robots) is 59.01×.

Use of a GPU with CuPy enables simulation of much larger swarms than was previously feasible.

[ ]:
```
# Simulate with large swarm
xi_0 = np.random.rand(640)-0.5
sol = solve_ivp(dydt, (t[0], t[-1]), xi_0, t_eval=t, events=event,␣
 ↪vectorized=True, method='BDF')

# This needs to be adjusted slightly after vectorization
def plot_final_blob_function(sol):
  x_n_col = x_n.reshape(-1, 1)
  xi_n_f = sol.y[:, -1].reshape(1, -1)
  plt.plot(xi_n_f.ravel(), np.zeros_like(xi_n_f).ravel(), 'oC3',
           x_n_col, rho_e_n(x_n_col, xi_n_f).ravel(), '-C1',
           x_n_col, rho_n(x_n_col).ravel(), '-C2')
  plt.xlabel(r'$x$')
  plt.ylabel(r'$\rho(x)$')
  plt.legend(("Final Robot Position",
              "Swarm Blob Function",
              "Target Distribution"),
              bbox_to_anchor=(1.04,1), loc="upper left")
```

```
plot_final_blob_function(sol)
caption("The steady state coverage (blob function) of a large swarm "
        "($n_r=640$ robots) closely matches the target distribution.")
```

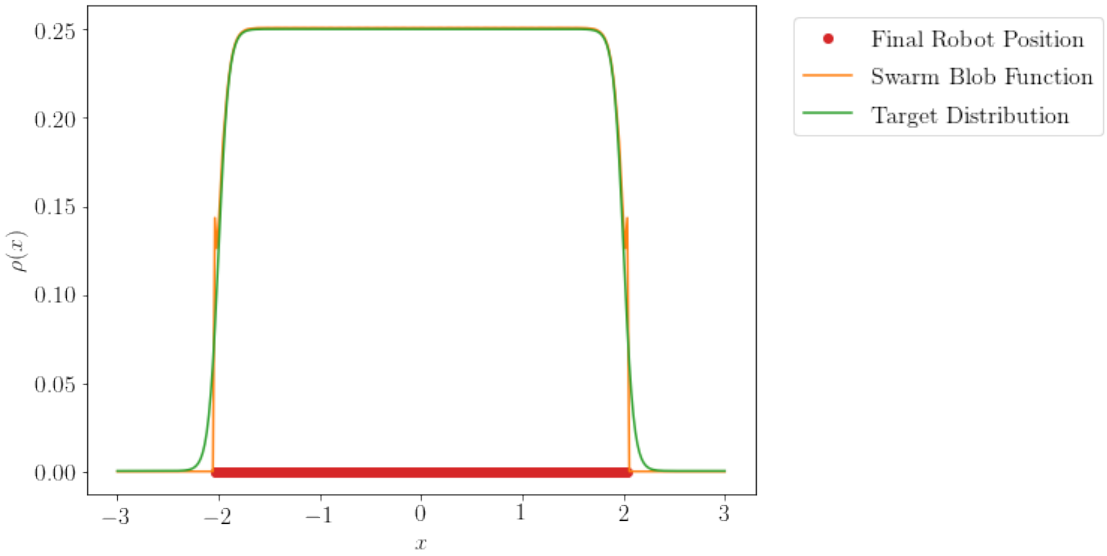

Figure 29. The steady state coverage (blob function) of a large swarm ($n_r = 640$ robots) closely matches the target distribution.

4.2.3 JIT Compilation

There are a few other performance improvement ideas that we have not yet implemented. All stem from the fact that within (3), the quantity $e^{\frac{-(x_i-x_j)^2}{2\epsilon^2}}$ appears several times. First, in the course of evaluating (3), this term is evaluated repeatedly with identical inputs; this redundancy should be eliminated. Second, we can exploit the symmetry of the pairwise interactions between robots, $e^{\frac{-(x_i-x_j)^2}{2\epsilon^2}} = e^{\frac{-(x_j-x_i)^2}{2\epsilon^2}}$, to reduce the number of calculations by a factor of approximately two. Finally, and most importantly, sufficiently distant robots have negligible effect on one another: $e^{\frac{-(x_i-x_j)^2}{2\epsilon^2}} \approx 0$ when $(x_i - x_j)^2 \gg \epsilon^2$. By partitioning the space, we can in $O(1)$ time determine which $O(1)$ interactions must be computed, and therefore we can exploit this sparsity of interactions to reduce the computational complexity of evaluating (3) to $O(n)$. These ideas do not seem easy to implement in a vectorized way, but use of a Python compiler such as Numba or Pythan could make them feasible.

References

[1] Carrillo, José Antonio, Katy Craig, and Francesco S. Patacchini. "A blob method for diffusion." *Calculus of Variations and Partial Differential Equations* 58.2 (2019): 53.

[2] Anderson, Brendon G., et al. "Quantifying Robotic Swarm Coverage." *International Conference on Informatics in Control, Automation and Robotics*. Springer, Cham, 2018.