Project 3 - Traffic Flow

December 16, 2021

Braden Fisher, Finn Bergquist, Aryeh Carmi

Section 1: Introduction

In this project, we model the density of traffic over time with the introduction of a perturbation. Our traffic model is produced by:

$$\frac{\partial\rho}{\partial t} + \frac{\partial f(\rho)}{\partial x} = 0$$

We use ρ to represent car density at a specific location and time, t for the time, $f(\rho)$ for flux (the number of cars passing through a certain position at a certain time), and x for position.

We observe that we cannot calculate $\frac{\partial\rho}{\partial t}$ directly with our equations available, but we work around this by finding $\frac{\partial f(\rho)}{\partial x}$ instead. This requires us to find the flux at each point (using various methods described below) and then differentiate with respect to space using some numerical method. The flux is defined as the number of cars passing by a specific point at a specific time, and is measured as a function of ρ at that location and time:

$$f(\rho) = \rho u_{max}(1 - \frac{\rho}{\rho_{max}})$$

u represents the car speed (we assume every car in a particular car density travels the same speed in the same direction), and u_{max} is the maximum speed at which a car can travel. We expect the car speed and density of cars to be inversely related, as cars (should) slow down when there are more cars and can speed up when there are fewer.

After calculating $\frac{\partial f(\rho)}{\partial x}$, we have $\frac{\partial\rho}{\partial t}$ at many points, and from this, we have coupled differential equations for density with respect to time, so we can integrate to calculate the densities at each position over time.

Our model's initial condition, which initially puts the perturbation of magnitude $\delta\rho$ at the center of the observed area ($0 \leq x \leq 1$) is:

$$\rho(x) = \overline{\rho} + \delta\rho e^{-\frac{(X - X_{center})^2}{\lambda^2}}$$

$\overline{\rho}$ represents the uniform initial density before the introduction of a perturbation, and $\delta\rho$ creates the perturbation. To put this into context, the perturbation represents a change in traffic density. As the cars continue to move, we expect this change in traffic density to move as well.

To model this, we will copy the process of integrating our derivatives at several different timesteps so that we have densities spread across the observed region over a period of time. With this information, we can also observe the characteristic speed, the speed at which the perturbation propagates:

$$v = u_{max}(1 - \frac{2\overline{\rho}}{\rho_{max}})$$

For some examples, we expect the perturbation to change shape as it propagates, meaning the characteristic speed of the perturbation can be different at the top and bottom of the wave. We call these eventual discontinuities shocks, and in our simulation, we will have two routines: one that captures the shocks and one that does not.

Section 2: Methods

We created one trafficFlow class to hold all of the data and methods we would need to interact with our model for this project. The major idea for this method is that we will maintain a 1-D array of all of the ρ values at each value of x between 0 and 1. We have made some choices for simplification, such that $\rho_{max} = u_{max} = 1$, meaning all densities and all speeds will be fractional values of ρ_{max} and u_{max}, respectively. Also, we will impose periodic boundary conditions, meaning moving past $x = 1$ has the effect of returning to $x = 0$ (so cars are essentially driving in circles in this model and never exit the observed region between $x = 0$ and $x = 1$). For x_{center}, we will use 0.5 because the observed region is always between 0 and 1, and we will use $\lambda = 0.1$ in the initialization equation.

Some of our methods are used to help initialize the class:

rho_of_x_init encodes the equation above to model the initial condition and is used by initialize_rho to initialize the ρ array.

f_of_rho gives us the flux f as a function of ρ.

As stated in the introduction, the project is split into two different overall implementations: one that does not account for shocks and one that does. Our methods can be organized into the two larger routines they help execute.

Routine 1: Method of lines (Does Not Capture Shocks)

Using the equation $\frac{\partial \rho}{\partial t} + \frac{\partial f(\rho)}{\partial x} = 0$, we isolated $\frac{\partial \rho}{\partial t}$, setting it equal to $\frac{-\partial \rho}{\partial t}$.

In our November 19th lecture, we defined a generic equation $\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x}$ to be used in our example application of the method of lines. We then defined a centered-differencing-based equation to get $\frac{\partial u}{\partial t}$ at each gridpoint i: $\frac{\partial u_i}{\partial t} = \frac{u_{i+1} - u_{i-1}}{2\Delta x}$.

In terms of our model, we substitute the $\frac{\partial f(\rho)}{\partial x}$ term for $-v\frac{\partial u}{\partial x}$, and $\frac{\partial \rho}{\partial t}$ for $\frac{\partial u}{\partial t}$. First, we initialize all ρ values using the initial condition equation above, and from there, we can use the $f(\rho)$ equation given to intialize all f values at the same x values at which we have ρ values. Then, we can use Python's odeint function to integrate the set of i coupled ODEs that we get after centered differencing of f with the following methods:

derivs_no_shocks: This method calculates the time derivative of ρ (using centered differencing for f) at every gridpoint.

runge_kutta_no_shocks: This uses odeint to solve the differential equations describing $\rho(t)$.

runge_kutta_no_shocks makes a call to odeint(), but we are aware that odeint's complexity may become more of an obstacle than a help, so we also have written our own Runge-Kutta stepper that implements the 4th-order scheme without calling odeint() if we need it in the future. These methods will allow for us to pass to them any value for a final time, as well as how many time values we want to collect values of ρ for. We will then plot ρ as a function of x for various values of t in order to capture how the perturbation moves and possibly changes shape.

Routine 2: High Resolution Shock Capturing (HRSC)

The major difference between HRSC and the basic method of lines approach is that HRSC will account for possible discontinuities in ρ (which evolve when a shock developes). If ρ is discontinuous, then we cannot compute derivatives such as $\frac{\partial f}{\partial x}$ directly as before. Instead, we treat each value of ρ as an average of all ρ values inside of the cell, where the recorded ρ value (we call it $\overline{\rho}$) is measured at the center of the cell. We rewrite our original $\frac{\partial \rho}{\partial t}$ equation in terms of integrals in order to obtain these averages:

$\frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial \rho}{\partial t} dx + \frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \frac{\partial}{\partial x} f(\rho) dx = 0$

This can be rearranged into what we have called "Equation Star" as in class:

$\frac{d\overline{\rho}_i}{dt} = -\frac{1}{\Delta x}(\overline{f}_{i+\frac{1}{2}} - \overline{f}_{i-\frac{1}{2}})$

Each $\overline{f}$ value refers to the flux across a boundary cell, where $i+\frac{1}{2}$ refers to the right cell boundary and $i-\frac{1}{2}$ refers to the left cell boundary. The following methods implement the suggested HRSC routine:

derivs_shocks: Calculates the time derivative of ρ at every grid point. In this case, we first have to carry out the reconstruction step. This is implemented in our reconstruct method, which uses minmod to approximate the values of ρ on either side of every boundary. Next, we solve the Riemann problem at each boundary, which is implemented in our get_all_fs_at_boundaries method, using shock speeds calculated by the calc_s function. If $s > 0$, $\overline{f}_{i+\frac{1}{2}}$ is the value of f calculated with the ρ value to the left of the boundary, otherwise it is calculated with the value of ρ to the right of the boundary. Lastly, we use equation star to get all derivatives $\frac{d\rho}{dt}$, then use odeint (or our own Runge-Kutta scheme) to integrate these again.

runge_kutta_shocks: Uses odeint to integrate derivatives found in derivs_shocks.

The following helper methods are employed along the way in both the Method of Lines and HRSC routines to impose periodic boundary conditions: add_one_to_index and subtract_one_from_index.

We calculated the characteristic speed, v, using the equation derived in class. This equation is: $v = u_{max}(1 - \frac{2\overline{\rho}}{\rho_{max}})$

We first use this equation to solve for v analytically (in calc_v_analytically) and then we solve for v numerically (in calc_v_numerically), by dividing the distance traveled by the perturbation (using the x-value at which there is the peak) by the time it took to travel.

We calculate the speed of the shock front in calc_s, using the equation derived in class. This equation is: $s = \frac{f(\rho_l)-f(\rho_r)}{\rho_l-\rho_r}$

calc_s_analytically (using the same equation from class) and calc_s_numerically (finding the distance between peaks and dividing it by the time taken to travel from one location to the next) are used after carrying out HRSC to compare our analytical result to our numerical one.

We track the positions of cars over time with update_car_positions. We begin with a set of initial car positions at $t = 0$, then iterate through each time step, calculating the speeds of the cars at that time, and updating the position of the cars using the equation $x_i^{n+1} = x_i^n + u_i^n \Delta t$, where n is the current time step, i is a certain car, x is the car's position, and u is the car's speed. Later on, we will plot x versus t for each car to analyze their trajectories.

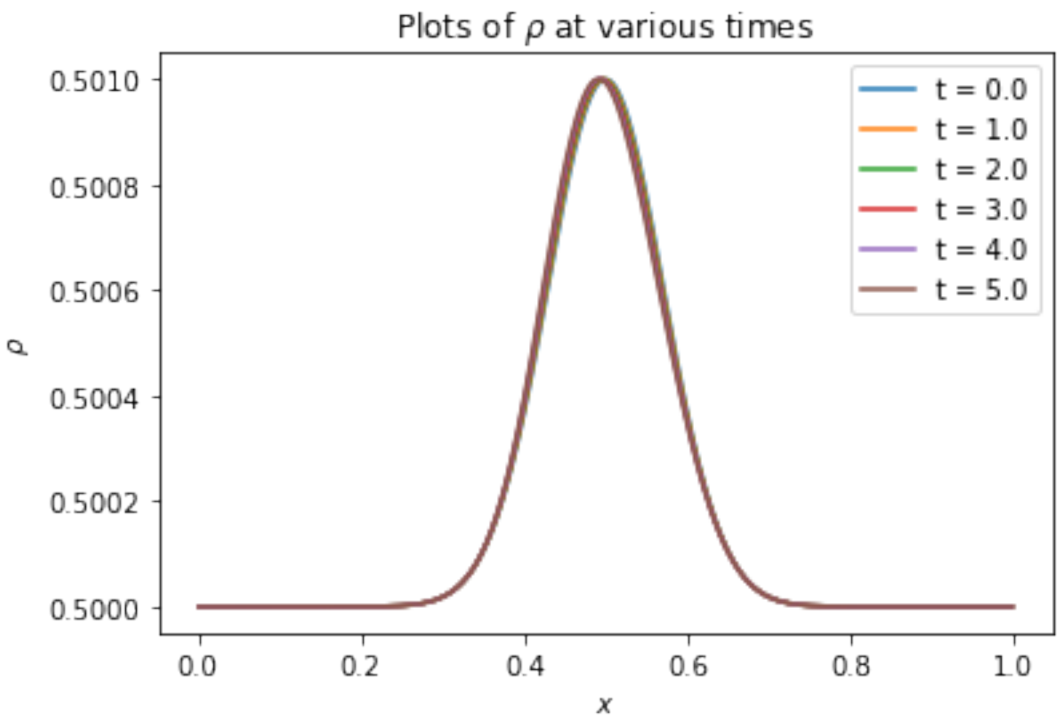
```
[10]: import numpy as np
import scipy.integrate as integrate
from matplotlib import rc
import matplotlib.pyplot as plt
import scipy.interpolate as interp
import warnings
warnings.filterwarnings('ignore')

class trafficFlow:
    def __init__(self, initial_rho_bar, delta_rho, num_rho_vals=499, rho_max=1,␣
 ↪u_max=1, x_center=0.5, lam=0.1, num_cars=10):
        self.rho_max = rho_max
        self.u_max = u_max
        self.x_center = x_center
        self.lam = lam
        self.initial_rho_bar = initial_rho_bar
        self.delta_rho = delta_rho #perturbation
        self.num_rho_vals = num_rho_vals
        self.rho_spacing = 1/(num_rho_vals + 1)
        self.x_vals = np.linspace(self.rho_spacing, 1 - self.rho_spacing,␣
 ↪num_rho_vals) #An array of the x-values at which we still store values of rho

        self.rhos = np.zeros((num_rho_vals))
        self.initialize_rho()

        self.fluxes = np.zeros((num_rho_vals))
        self.update_fluxes(self.rhos)

        #Index 0 corresponds to i = 1/2, index 1 corresponds to i = 3/2, and so␣
 ↪on
        self.rho_rights = np.zeros((self.num_rho_vals))
        self.rho_lefts = np.zeros((self.num_rho_vals))

        #Index 0 corresponds to i = 1/2, index 1 corresponds to i = 3/2, and so␣
 ↪on
        self.boundary_fs = np.zeros((self.num_rho_vals))

        self.num_cars = num_cars

    def rho_of_x_init(self, x):
        #Implements the initial equation given
        return self.initial_rho_bar + (self.delta_rho * np.exp(-((x - self.
 ↪x_center) ** 2) / self.lam ** 2))

    def initialize_rho(self):
        #Uses the equation above to initialize rho at every grid point
        for i in range(self.num_rho_vals):
            self.rhos[i] = self.rho_of_x_init(self.x_vals[i])

    def update_fluxes(self, rho_vect):
        #Updates the flux at every grid point by using the equation given (Not␣
↪used in HRSC)
        for i in range(self.num_rho_vals):
            self.fluxes[i] = rho_vect[i] * self.u_max * (1 - (rho_vect[i] /␣
↪self.rho_max))

    def f_of_rho(self, rho):
        #Returns the flux given a rho
        return rho * self.u_max * (1 - (rho / self.rho_max))

    def add_one_to_index(self, i, array):
        #Imposes periodic boundary conditions by checking if we are at the end␣
↪of the array
        if (i == len(array) - 1):
            return 0
        else:
            return i + 1

    def subtract_one_from_index(self, i, array):
        #Imposes periodic boundary conditions by checking if we are at the␣
↪beginning of the array
        if (i == 0):
            return len(array) - 1
        else:
            return i - 1

    def derivs_no_shocks(self, rho_vect, t):
        #Uses method of lines to set up vector of derivatives at every value of␣
↪x where we are storing a rho value
        derivs = np.zeros((self.num_rho_vals))
        self.update_fluxes(rho_vect)
        for i in range(self.num_rho_vals):
            right_index = self.add_one_to_index(i, rho_vect)
            left_index = self.subtract_one_from_index(i, rho_vect)
            derivs[i] = -( (self.fluxes[right_index] - self.fluxes[left_index])␣
↪/ (2 * self.rho_spacing)) #finite differencing
        return derivs

    def runge_kutta_no_shocks(self, t_final, n, rtol, atol):
        #A wrapper method that calls odeint() in order to solve the coupled␣
↪differential equations for rho(t) at each point
        t_vals = np.linspace(0, t_final + 1, n + 1, False)
        sol = integrate.odeint(self.derivs_no_shocks, self.rhos, t_vals,␣
↪rtol=rtol, atol=atol)
        return sol, t_vals

    def our_runge_kutta_no_shocks(self, rho_vect, t_final, n):
    #Our own implementation of a Runge-Kutta stepper to use when odeint() fails
        t_vals = np.linspace(0, t_final, n + 1, False)
        rho = np.array((n + 1) * [rho_vect]) #Initialize a matrix containing a␣
↪vector of rho values at each time step
        h = t_vals[1] - t_vals[0]
        for i in range(n):
            k1 = h * self.derivs_no_shocks(rho[i], t_vals[i])
            k2 = h * self.derivs_no_shocks(rho[i] + 0.5 * k1, t_vals[i] + 0.5 *␣
↪h)
            k3 = h * self.derivs_no_shocks(rho[i] + 0.5 * k2, t_vals[i] + 0.5 *␣
↪h)
            k4 = h * self.derivs_no_shocks(rho[i] + k3, t_vals[i] + h)
            rho[i+1] = rho[i] + (k1 + 2*(k2 + k3) + k4) / 6
        return rho, t_vals

    def plot_rho_at_times_no_shocks(self, t_final, n, our_version=False,␣
↪rtol=1e-5, atol=1e-5):
        #Plots the values of rho at the first 6 timesteps

        #Use odeint():
        if (not our_version):
            rho_vects_at_times, t = self.runge_kutta_no_shocks(t_final, n,␣
↪rtol=rtol, atol=atol)

        #Use our Runge-Kutta implementation:
        else:
            rho_vects_at_times, t = self.our_runge_kutta_no_shocks(self.rhos,␣
↪t_final, n)


        lab_0 = "t = " + str(t[0])
        plt.plot(self.x_vals, rho_vects_at_times[0], label=lab_0)

        lab_1 = "t = " + str(t[1])
        plt.plot(self.x_vals, rho_vects_at_times[1], label=lab_1)

        lab_2 = "t = " + str(t[2])
        plt.plot(self.x_vals, rho_vects_at_times[2], label=lab_2)

        lab_3 = "t = " + str(t[3])
        plt.plot(self.x_vals, rho_vects_at_times[3], label=lab_3)
        lab_4 = "t = " + str(t[4])
        plt.plot(self.x_vals, rho_vects_at_times[4], label=lab_4)

        lab_5 = "t = " + str(t[5])
        plt.plot(self.x_vals, rho_vects_at_times[5], label=lab_5)

        plt.xlabel(r"$x$")
        plt.ylabel(r"$\rho$")
        plt.title(r"Plots of $\rho$ at various times")
        plt.legend()

        return rho_vects_at_times

    def calc_v_analytically(self):
        #Calculates the characteristic speed using the equation from class
        return abs(self.u_max * (1 - ((2 * self.initial_rho_bar) / self.
↪rho_max)))

    def calc_v_numerically(self, rho_t0, rho_t1, t0, t1):
        #Calculates the characteristic speed by dividing the distance a␣
↪perturbation has traveled between two timestamps divided by the time taken␣
↪to do so
        #t0 is the beginning distribution of rho and t1 is the ending␣
↪distribution
        peak_x_0 = self.x_vals[np.argmax(rho_t0)]
        peak_x_1 = self.x_vals[np.argmax(rho_t1)]

        return abs((peak_x_1 - peak_x_0) / (t1 - t0))

    def minmod(self, a, b):
        #Implements minmod as we wrote it in class
        if (a * b < 0):
            return 0

        if (abs(a) < abs(b)):
            return a

        return b

    def reconstruct(self, rho_bar_vect):
        #Implements the reconstruction step that calculates the rho values to␣
↪the left and right of every boundary
        rho_lefts = np.zeros((self.num_rho_vals))
        rho_rights = np.zeros((self.num_rho_vals))
        for i in range(self.num_rho_vals):
            left_index = self.subtract_one_from_index(i, rho_bar_vect)
            right_index = self.add_one_to_index(i, rho_bar_vect)
            right_index_2 = self.add_one_to_index(right_index, rho_bar_vect)
            rho_lefts[i] = rho_bar_vect[i] + (1./2.) * self.
↪minmod(rho_bar_vect[right_index] - rho_bar_vect[i], rho_bar_vect[i] -␣
↪rho_bar_vect[left_index])
            rho_rights[i] = rho_bar_vect[right_index] - (1./2.) * self.
↪minmod(rho_bar_vect[right_index_2] - rho_bar_vect[right_index],␣
↪rho_bar_vect[right_index] - rho_bar_vect[i])
        return rho_lefts, rho_rights

    def calc_s(self, rho_left, rho_right):
        #calculates s at every cell boundary given the right and left rho␣
↪values at that boundary
        return (self.f_of_rho(rho_left) - self.f_of_rho(rho_right)) / (rho_left␣
↪- rho_right)

    def get_all_fs_at_boundaries(self, rho_lefts, rho_rights):
        #Uses the array of left and right rho values at every boundary to get f␣
↪at every boundary
        boundary_fs = np.zeros((self.num_rho_vals))
        for i in range(self.num_rho_vals):
            s = self.calc_s(rho_lefts[i], rho_rights[i])

            if (s >= 0):
                boundary_fs[i] = self.f_of_rho(rho_lefts[i])
            else:
                boundary_fs[i] = self.f_of_rho(rho_rights[i])

        return boundary_fs

    def eqn_star(self, f_plus, f_minus):
        #The equation from our notes that calculates the time derivative of rho
        return -1 * ((f_plus - f_minus) / self.rho_spacing)

    def derivs_shocks(self, rho_bar_vect, t, delta_t):
        #Uses method from class to set up vector of derivatives at every value␣
↪of x where we are storing a rho value
        derivs = np.zeros((self.num_rho_vals))
        rho_lefts, rho_rights = self.reconstruct(rho_bar_vect)
        boundary_fs = self.get_all_fs_at_boundaries(rho_lefts, rho_rights)
        for i in range(self.num_rho_vals):
            derivs[i] = self.eqn_star(boundary_fs[i], boundary_fs[i-1])
        return derivs

    def runge_kutta_shocks(self, t_final, n):
        #A wrapper method that calls odeint() in order to solve the coupled␣
↪differential equations for rho(t) at each point
        t_vals = np.linspace(0, t_final, n, False)
        delta_t = t_vals[1] - t_vals[0]
        sol = integrate.odeint(self.derivs_shocks, self.rhos, t_vals,␣
↪args=(delta_t,), rtol=1e-5, atol=1e-5)
        return sol, t_vals

    def our_runge_kutta_shocks(self, rho_vect, t_final, n):
        #Our owm implementation of a Runge-Kutta stepper to use when odeint()␣
↪fails
        t_vals = np.linspace(0, t_final, n, False)
        rho = np.array((n + 1) * [rho_vect]) #Initialize a matrix containing a␣
↪vector of rho values at each time step
        h = t_vals[1] - t_vals[0]
        for i in range(n):
            k1 = h * self.derivs_shocks(rho[i], t_vals[i], h)
            k2 = h * self.derivs_shocks(rho[i] + 0.5 * k1, t_vals[i] + 0.5 * h,␣
↪h)
            k3 = h * self.derivs_shocks(rho[i] + 0.5 * k2, t_vals[i] + 0.5 * h,␣
↪h)
            k4 = h * self.derivs_shocks(rho[i] + k3, t_vals[i] + h, h)
            rho[i+1] = rho[i] + (k1 + 2*(k2 + k3) + k4) / 6
        return rho, t_vals

    def plot_rho_at_times_shocks(self, t_final, n, our_version=False,␣
↪plot_cars=False):
        #Plots rho versus x at first 6 timesteps using HRSC
        #Use odeint():
        if (not our_version):
            rho_vects_at_times, t = self.runge_kutta_shocks(t_final, n)

        #Use our Runge-Kutta implementation:
        else:
            rho_vects_at_times, t = self.our_runge_kutta_shocks(self.rhos,␣
↪t_final, n)

        plt.figure()

        lab_0 = "t = " + str(t[0])
        plt.plot(self.x_vals, rho_vects_at_times[0], label=lab_0)

        lab_1 = "t = " + str(t[2])
        plt.plot(self.x_vals, rho_vects_at_times[2], label=lab_1)

        lab_2 = "t = " + str(t[4])
        plt.plot(self.x_vals, rho_vects_at_times[4], label=lab_2)

        lab_3 = "t = " + str(t[6])
        plt.plot(self.x_vals, rho_vects_at_times[6], label=lab_3)

        lab_4 = "t = " + str(t[8])
        plt.plot(self.x_vals, rho_vects_at_times[8], label=lab_4)

        lab_5 = "t = " + str(t[10])
        plt.plot(self.x_vals, rho_vects_at_times[10], label=lab_5)

        lab_6 = "t = " + str(t[12])
        plt.plot(self.x_vals, rho_vects_at_times[12], label=lab_6)

        lab_7 = "t = " + str(t[14])
        plt.plot(self.x_vals, rho_vects_at_times[14], label=lab_7)

        plt.xlabel(r"$x$")
        plt.ylabel(r"$\overline{\rho}$")
        plt.title(r"Plots of $\overline{\rho}$ at various times")
        plt.legend()

        if (plot_cars):
            self.plot_car_positions(rho_vects_at_times, t)

        return rho_vects_at_times

    def calc_s_numerically(self, rho_vect_t0, rho_vect_t1, t0, t1):
        #Calculates the speed of the shock front using the difference between␣
↪the maxima of the curves
        peak_x_0 = self.x_vals[np.argmax(rho_vect_t0)]
        peak_x_1 = self.x_vals[np.argmax(rho_vect_t1)]

        return abs((peak_x_1 - peak_x_0) / (t1 - t0))

    def calc_s_analytically(self, rho_vect):
        #Finds the location of the shock front and uses the equation from class␣
↪to calculate it analytically
        shock_index = np.argmax(rho_vect)
        rho_lefts, rho_rights = self.reconstruct(rho_vect)
        return abs((self.f_of_rho(rho_lefts[shock_index]) - self.
↪f_of_rho(rho_rights[shock_index])) / (rho_lefts[shock_index] -␣
↪rho_rights[shock_index]))

    def plot_car_positions(self, rho_vects_at_times, t_vect):
        #Goes through our timesteps and updates the cars' positions at each␣
↪timesteps by interpolating velocities to their
```

```
        #positions and adding the velocity times the change in time to their␣
↪old position
        car_pos_arrays_at_times = np.zeros((len(t_vect), self.num_cars))
        initial_car_positions = np.linspace(0, 1, self.num_cars, False)
        all_x_vals = np.concatenate((self.x_vals, np.array([0]), np.
↪array([1]))) #To be able to interpolate to anywhere in the grid

        for i, t in enumerate(t_vect):
            grid_velocities = self.u_max * (1 - (rho_vects_at_times[i] / self.
↪rho_max))

            grid_velocities = np.concatenate((grid_velocities, np.
↪array([grid_velocities[0]]), np.array([grid_velocities[-1]]))) #Assuming the␣
↪speeds at the ends of the grid are the same as the speeds one grid cell␣
↪inward
            u_of_x = interp.interp1d(all_x_vals, grid_velocities)

            if (i == 0):
                delta_t = t
                car_velocities = u_of_x(initial_car_positions)
                car_pos_arrays_at_times[i] = initial_car_positions +␣
↪car_velocities * delta_t

            else:
                delta_t = t_vect[i] - t_vect[i - 1]
                car_velocities = u_of_x(car_pos_arrays_at_times[i - 1])
                car_pos_arrays_at_times[i] = car_pos_arrays_at_times[i - 1] +␣
↪car_velocities * delta_t

            for j, pos in enumerate(car_pos_arrays_at_times[i]):
                  #new_pos = pos + car_velocities[i] * delta_t
                if (pos >= 1):
                    pos = pos - 1
                    car_pos_arrays_at_times[i][j] = pos


        plt.figure()
        for i in range(self.num_cars):
            car_positions = [t[i] for t in car_pos_arrays_at_times]
            car_positions = np.array(car_positions)
            car_positions[:-1][np.diff(car_positions) < 0] = np.nan #Don't plot␣
↪a straight vertical line where periodicity is imposed
            lab = "Car " + str(i + 1)
            plt.plot(t_vect, car_positions, label=lab)
        plt.legend()
        plt.xlabel(r"$t$")
        plt.ylabel(r"$x$")
        title_str = "Positions of " + str(self.num_cars) + " cars as a function␣
↪of time"
        plt.title(title_str)
```

Section 3: Results and Discussion

Now that we have a tested class of methods that allows us to interact with this model of traffic flow, we can begin to see the numerical results of our simulation when given different initial inputs as parameters. We will first test the method of lines portion of our implementation (not using HRSC to consider shocks yet) to see what happens when a shock begins to form. To get a sense of what this model looks like, we will first look at the results when the initial perturbation is small, such as $\delta\rho = 10^{-3}\rho_{max}$ (while keeping $\rho_{max} = 1$). We will also first use a value of $\overline{\rho} = \frac{\rho_{max}}{2}$ (so in this case, $\overline{\rho} = \frac{1}{2}$).

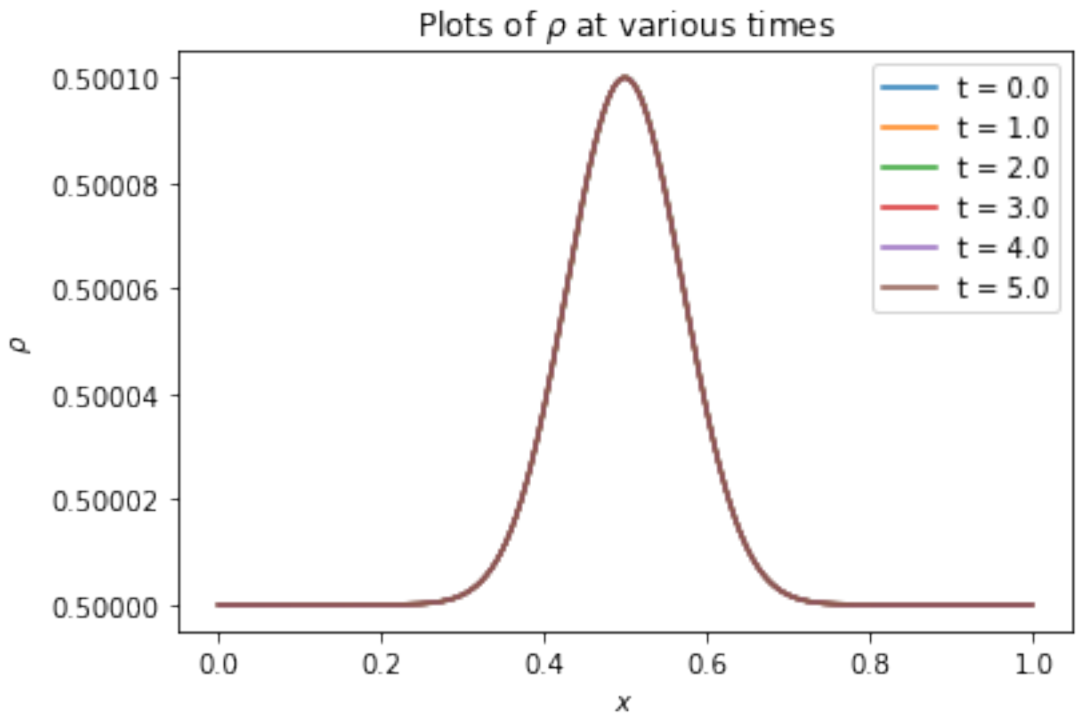
[2]:
```
traffic = trafficFlow(1./2., 1e-3)
rho_vects = traffic.plot_rho_at_times_no_shocks(10, 10)
```



The shape of these plots of the car density versus space at different times is what we would expect: the equation we are using to initialize $\rho(x)$ sets ρ to be higher near the center of the region considered, which it is here, and closer to the constant value $\overline{\rho}$ closer to the boundaries of the region, which we also see. The "bump" we see near the center of the graph is the perturbation in the density of the traffic flow.

By the equation for the characteristic speed outlined in the introduction, inserting $\overline{\rho} = \frac{\rho_{max}}{2}$ would lead us to expect that $v = 0$ analytically, which would mean that the perturbation would not be moving at all. However, the perturbation seems to be travelling over time ever so slightly (although going from $t = 0$ to $t = 4$ is a fairly large timescale anyway), but this may be due to our assumption that $\delta\rho$ is small enough that we only need the first-order term in the Taylor expansion in order to calculate the characteristic speed v analytically. It is possible that $\delta\rho = 10^{-3}$ is actually not small enough for that assumption to be true here, which would be why we would see some movement in the perturbation. Let's check to see if this is the case by examining the same situation with a smaller value for $\delta\rho$, such as 10^{-4}.

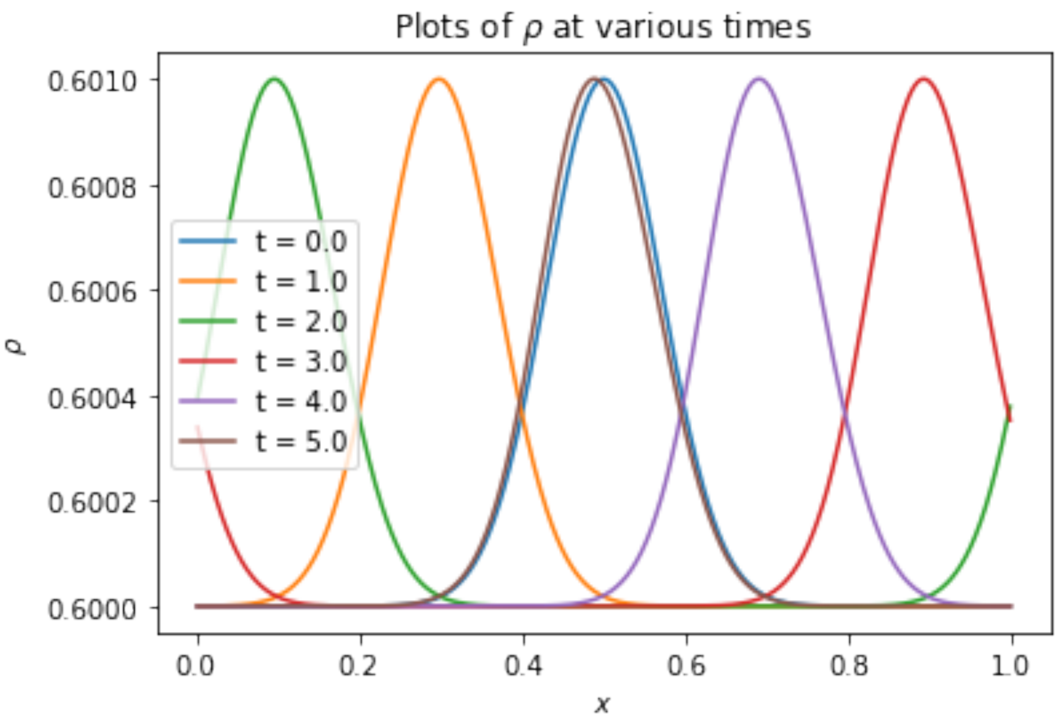
[3]:
```
traffic = trafficFlow(1./2., 1e-4)
rho_vects = traffic.plot_rho_at_times_no_shocks(10, 10)
```



Now, the graphs of $\rho(x)$ at the same time steps as above are overlapping more, suggesting that the perturbation is actually not moving as much, so the analytical prediction that $v = 0$ is more reflected in the numerical solution here. Now, to see a perturbation that is clearly moving, we will change $\overline{\rho}$ to something other than $\frac{\rho_{max}}{2}$ so that the characteristic speed v is nonzero and therefore the perturbation will move. We will choose $\overline{\rho} = 0.6\rho_{max}$. In order to see the perturbation moving without worrying about shocks, we will keep a small perturbation of $\delta\rho = 10^{-3}\rho_{max}$. As this example runs, we will also calculate the characteristic speed both analytically and numerically. In order to find v analytically, we will simply use $\overline{\rho} = 0.6\rho_{max}$ in the equation we used in class for v and have our method above return the result of this equation. Numerically, the characteristic speed refers to how fast the perturbation is travelling. So, we will calculate the distance (in terms of x) that the peak of the perturbation has travelled in one timestep, divided by the time between the two timesteps. This distance / time relationship will give the speed of the perturbation, which we will compare with the analytical result (for both, we will not consider direction and only consider magnitude for speed).

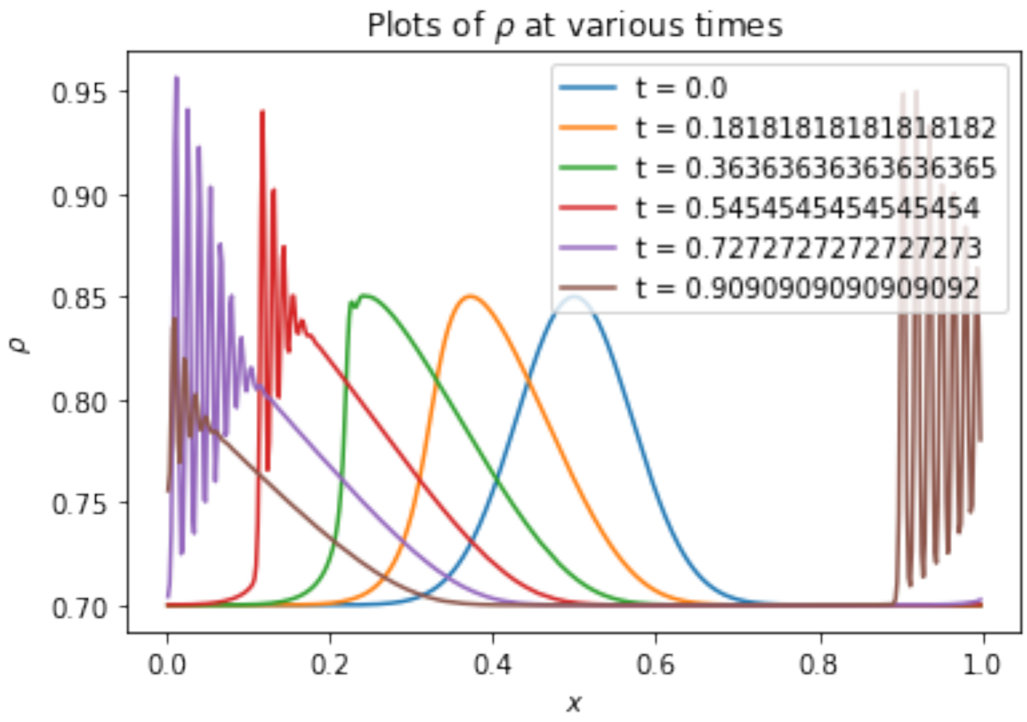
```
[4]: traffic = trafficFlow(0.6, 1e-3)
     rho_vects = traffic.plot_rho_at_times_no_shocks(5, 5, rtol=1e-8, atol=1e-8)

     print("The analytical characteristic speed is", traffic.calc_v_analytically())
     print("The numerical characteristic speed calculated between t = 0 and t = 1␣
      ↪is", traffic.calc_v_numerically(rho_vects[0], rho_vects[1], 0, 1))
```

```
The analytical characteristic speed is 0.19999999999999996
The numerical characteristic speed calculated between t = 0 and t = 1 is 0.202
```



As expected, with a value of $\overline{\rho}$ different than $\frac{\rho_{max}}{2}$, the perturbation is now moving to the left, as shown by the curves at increasing time steps with peaks further to the left (once the perturbation reaches $x = 0$, it comes back from $x = 1$ because of the periodic boundary conditions). Using the analytical equation for v with $\overline{\rho} = 0.6\rho_{max}$ gives a value of $v = 0.20$ (in units of the arbitrary length over the arbitrary time chosen so that $\rho max = u_{max} = 1$). Calculating the characteristic speed based on the locations of the peaks at different times gave a value of $v = 0.202$, which is very close to the analytical value we would expect. Even though the perturbation is moving, it is not visibly changing shape as it propagates. The curve looks the same at the final time it is plotted as it does at $t = 0$ with this smaller perturbation. For the perturbation to change shape, the peaks need to travel significantly faster than the bases, which happens when $\delta\rho$ is large enough that the nonlinear terms have a noticeable effect. So, to look at a perturbation that changes shape, we will look at a situation where $\overline{\rho} = 0.7\rho_{max}$ and $\delta\rho = 0.15\rho_{max}$, and make plots of $\rho(x)$ at various times again.

```
[5]: traffic = trafficFlow(0.7, 0.15)
     rho_vects = traffic.plot_rho_at_times_no_shocks(1, 10)
```

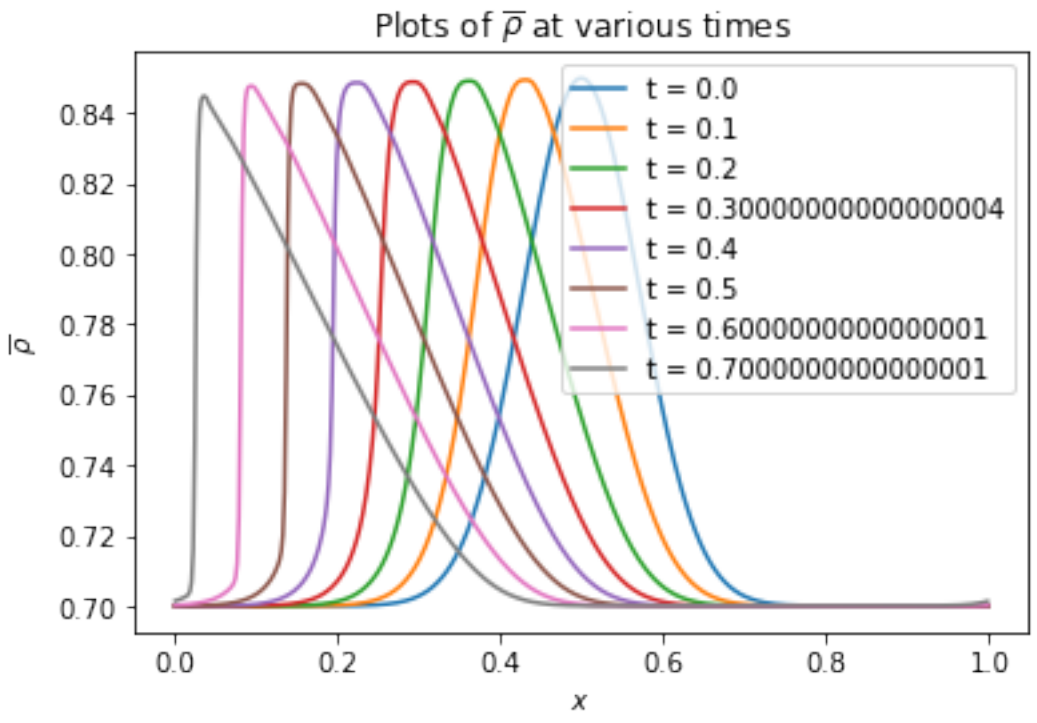


Now, the perturbation is clearly changing shape as it propagates, as the peak of the wave is traveling faster than the base, causing it to "tip over" like an ocean wave hitting the shore. This phenomenon happens because v, the speed at which the perturbation travels, has a linear dependence on ρ. Therefore, when the peak of the wave is at a sufficiently higher ρ than the base, which happens when $\delta\rho$ is not sufficiently small, the difference in characteristic speed between the peak and the base of the perturbation is large enough so that the peak travels noticeably faster than the base, as shown in the plots above.

It is also important to note that this is the first time we have seen a shock develop. A shock occurs when there is a discontinuity in the values of ρ once the peak has fully tipped over. At the later timesteps (note that we had to decrease the timescale and look between $t = 0.0$ and $t = 1.0$ in order for the code to still converge at all), the shock has formed, but the method of lines implementation is now quite unstable around the shock. Around the discontinuity, our current method is returning values of ρ that quickly oscillate around the shock. Ideally, we would like to have values of ρ that capture this shock in high resolution when plotted, so we need to switch from a simple method of lines implementation to a High-Resolution Shock Capturing (HRSC) scheme to examine situations such as this where the perturbation is large enough. So, let's look at the same scenario with our HRSC implementation:

```
[7]: traffic = trafficFlow(0.7, 0.15)
     rho_vects = traffic.plot_rho_at_times_shocks(1, 20)

     print("The analytical shock front speed measured at t = 0.7 is", traffic.
      ↪calc_s_analytically(rho_vects[14]))
     print("The numerical shock front speed calculated between t = 0.6 and t = 0.7␣
      ↪is", traffic.calc_s_numerically(rho_vects[12], rho_vects[14], 0.6, 0.7))
```

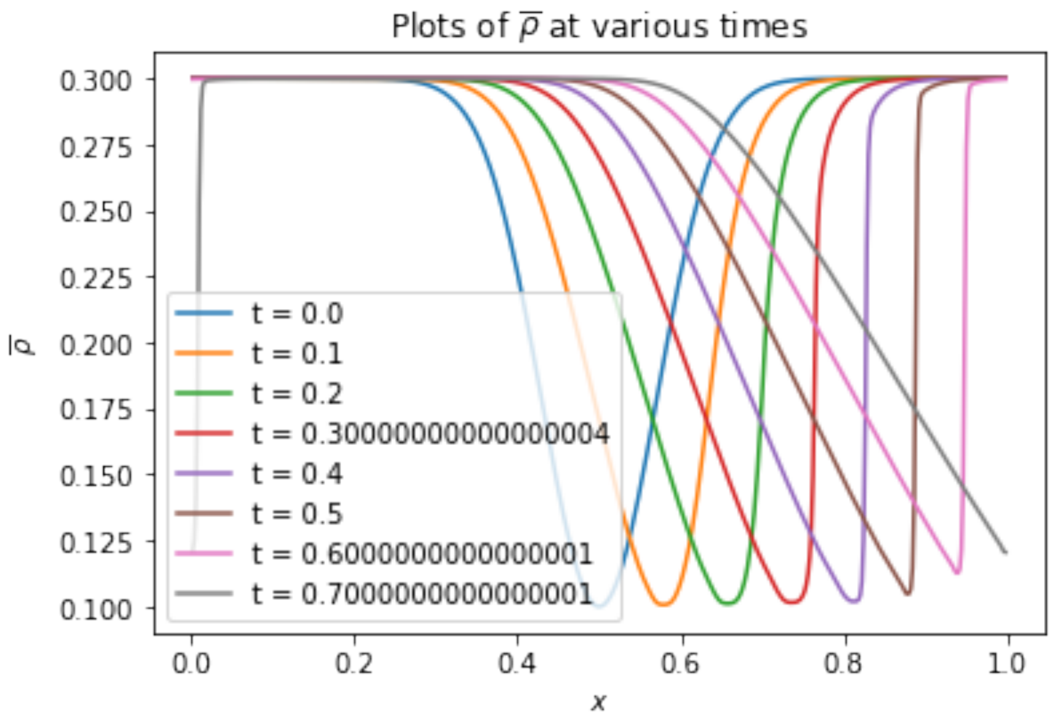
```
The analytical shock front speed measured at t = 0.7 is 0.6895468260752635
The numerical shock front speed calculated between t = 0.6 and t = 0.7 is
0.5800000000000001
```



There is a clear difference between the resolution of this wave and the resolution that resulted from the basic method of lines approach above. The shock forms as the discontinuity in ρ becomes sharper, which we can more clearly see now than when the code previously became unstable and caused large oscillations around the shock. The shock forms as the wave moves to the left (starting at the center of the grid) and the peak of the wave tips over, then continues to propagate (move to the left, then restart at the right of the grid with periodicity) after it is fully formed. In this simulation, we also calculated the shock speed s both analytically and numerically. For the analytical calculation, we used the equation $s = \frac{f(\rho_L)-f(\rho_R)}{\rho_L-\rho_R}$, using the curve at $t = 0.7$ so that enough time has passed for the shock to fully form and using the ρ_L and ρ_R values at the index of the curve's maximum value, as this is where the peak of the graph and therefore also the shock is located. We calculated s the same way we previously calculated v, which was by locating the peaks of the graph and measuring the distance it traveled between two time steps and dividing by the time passed between those two steps. However, in this case, we used curves at the final two time steps so that the shock had formed and we could safely say that the shock front (the discontinuity in ρ) was at the same x location as the maximum ρ value. As shown, the analytical and numerical values for s were 0.690 and 0.580, respectively.

As another test for our scheme, we will consider a new scenario where the perturbation goes in the opposite direction with a negative $\delta\rho$. We will use the values $\overline{\rho} = 0.3\rho_{max}$ and $\delta\rho = -0.2\rho_{max}$ as an example:

[8]:
```
traffic = trafficFlow(0.3, -0.2)
rho_vects = traffic.plot_rho_at_times_shocks(1, 20)
```



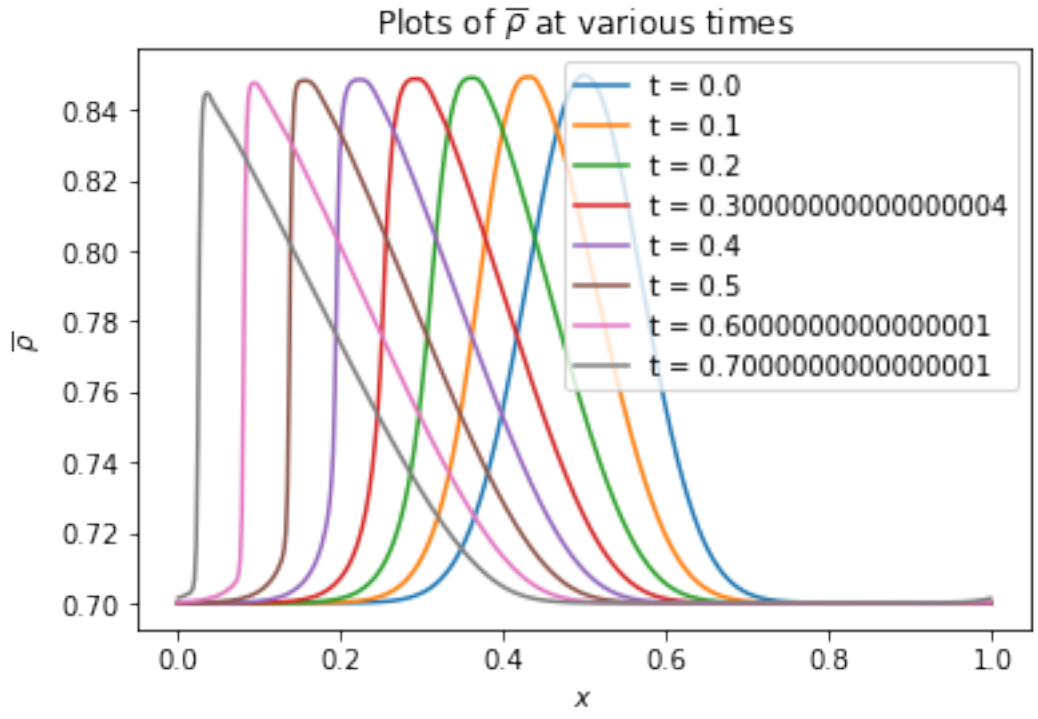
Now, the perturbation goes in the opposite direction as before, which makes sense with the change in sign for $\delta\rho$. Also, it travels to the right now, rather than the left, but the shock front still forms and then propagates to the right.

To understand why this occurs, we should tie our numerical simulation back to the physical situation of traffic flow we are attempting to model. In the above situations, when $\delta\rho$ is positive, there is an area on the road (initially the middle of the stretch of road) where the car density is higher than the constant density everywhere else (the perturbation). As time goes on, the cars to the left of the perturbation reach the area of higher density, and as they do so, must slow down, which causes the density to then increase to the left of the location of the previous perturbation. Meanwhile, where the car density was the highest initially, cars are able to speed up again because they are reaching an area of lesser density. So, the density gets higher to the left of the previous perturbation and lower at the location of the previous perturbation, which is why it moves to the left as shown with the waves above. However, as this pattern continues, cars must slow down quicker and quicker as they encounter the sudden change in car density. This sudden slowing down in turn causes even more of a sudden increase in car density even more to the left of the previous timestep, which is why the shock forms and then continues propagating to the left. The shock indicates an abrupt change in the density of the cars, which forces cars to slow down very quickly as they reach very slow or stopped traffic (which is why we encounter an area of traffic on a highway very quickly after seemingly driving along fairly quickly).

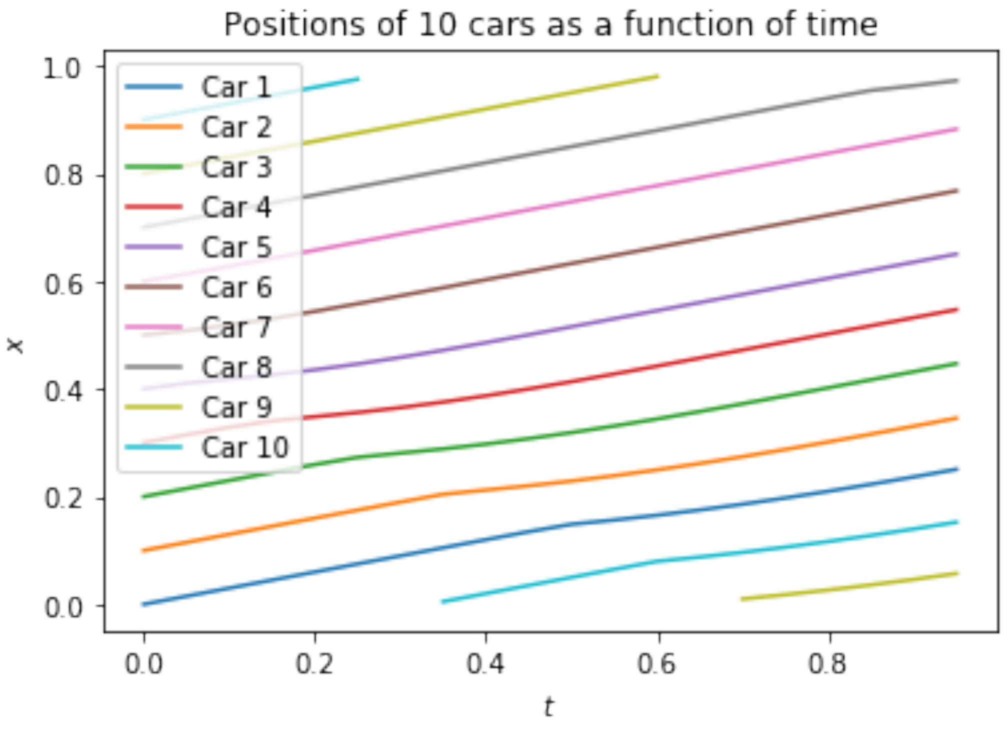
When the initial perturbation gets reversed, and the change in density is one such that the density at the middle of the road is less than the constant values at the other areas, the physical situation changes slightly, which is reflected in the graph above. Now, cars driving into the perturbation are allowed to speed up when they reach the left side of it because the car density decreases from the steady amount. On the right side of the perturbation, cars must slow down because they are going from the region of decreased car density to the region of constant (higher) car density. As more cars enter the location of the previous perturbation, the density there increases, which also forces the area of least density to move to the right. However, as this "valley" quickly moves to the right, it forces the frontier between the lowest density and the steady density to become even more of an abrupt change, which is why the shock forms. Again, the shock represents a very rapid change in the car density that forces cars to rapidly slow down. So, this example shows that even when there is an initial area of the road where the car density is lower, it is entirely possible that drivers will still reach an area where they feel as though they must slow down quite quickly to safely navigate rapidly changing car densities on the road.

To further see the effect this is having on the traffic being considered, we will track the positions of ten cars for the situation above where $\overline{\rho} = 0.7\rho_{max}$ and $\delta\rho = 0.15\rho_{max}$ using the method outlined in the Methods section above to update the position of each car by calculating their speeds at each time step based on the densities at their locations:

[11]:
```python
traffic = trafficFlow(0.7, 0.15)
rho_vects = traffic.plot_rho_at_times_shocks(1, 20, plot_cars=True)
```



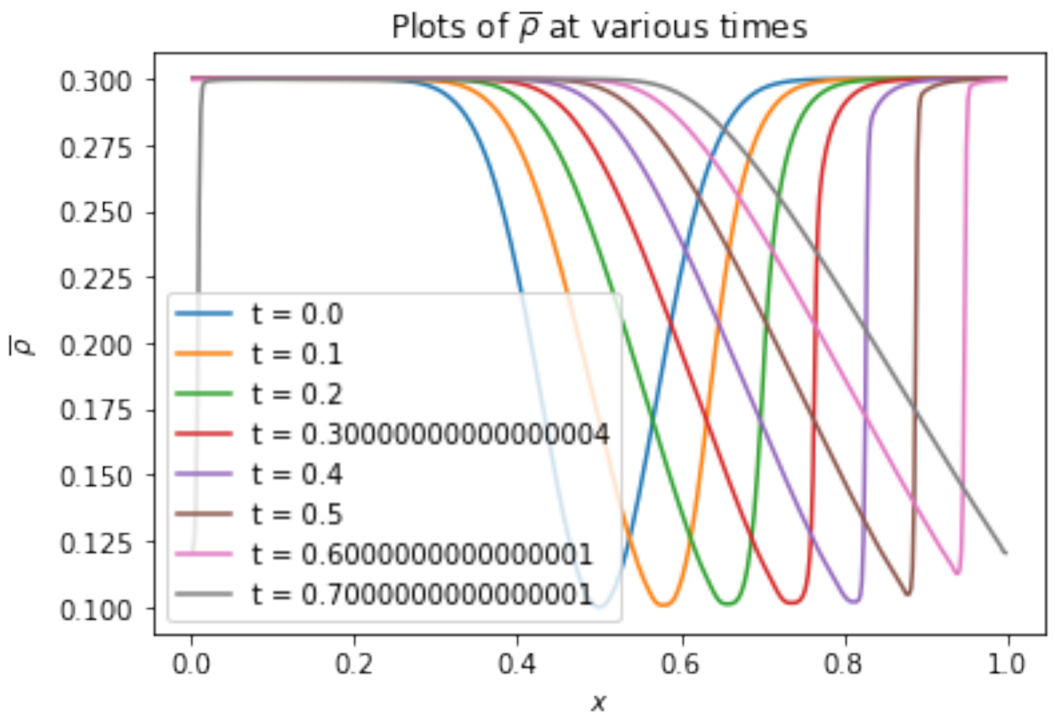
Plots of $\overline{\rho}$ at various times
t = 0.0
t = 0.1
t = 0.2
t = 0.30000000000000004
t = 0.4
t = 0.5
t = 0.6000000000000001
t = 0.7000000000000001
$\overline{\rho}$
x

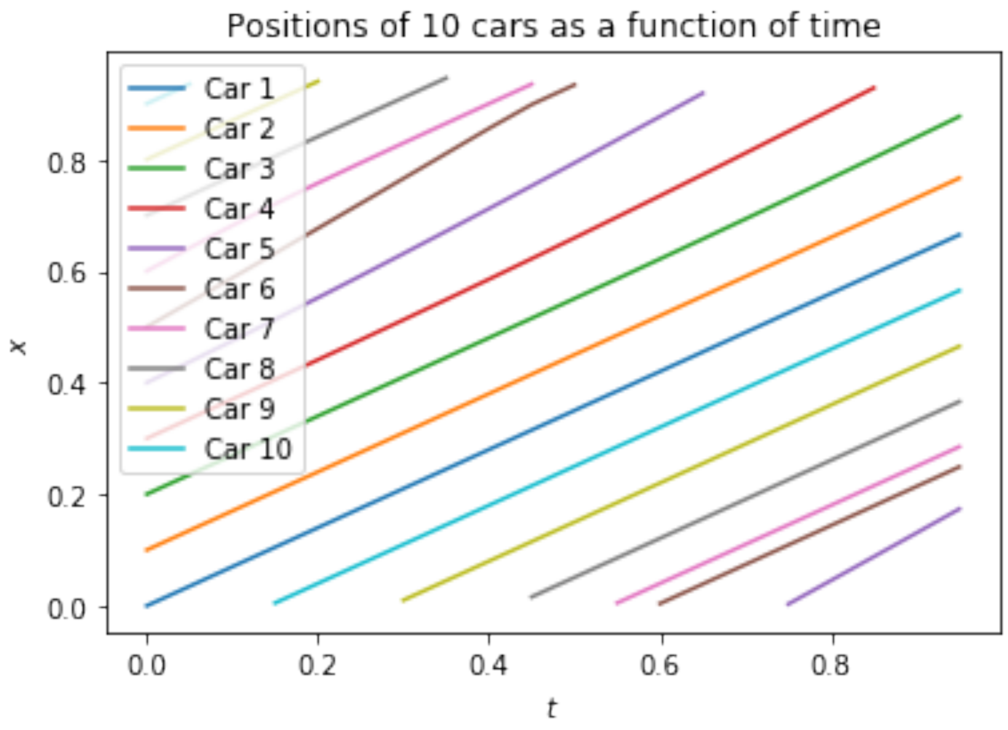


Positions of 10 cars as a function of time
Car 1
Car 2
Car 3
Car 4
Car 5
Car 6
Car 7
Car 8
Car 9
Car 10
x
t

This plot is a spacetime diagram showing the positions of ten cars over time with the situation created above, where the cars start at evenly spaced locations between $x = 0$ and $x = 1$. When a car reaches $x = 1$ (the top of the graph), its position resets at $x = 0$ with the same periodicity that has been imposed throughout the project. It is interesting to see that after some time, the cars at positions before $x = 0.5$ have their lines flatten. This makes sense, as the perturbation is still close to this location at this time, so cars in this area of higher density should be moving slower. Then, after this area, the lines become steeper again because the cars can move faster after they have moved past the perturbation in density. This area of flatter slope gradually moves downward in the graph, which makes sense because it shows that the perturbation, where the cars move slower, is moving to left, which agrees with the plot above. It becomes even more interesting after about $t = 0.2$, which is when the shock is really beginning to form. Once this is the case, the cars' speeds change increasingly abruptly around the location of the shock. There are areas where the lines quickly get flatter, which is when the cars are at the area of lowest density, then all of a sudden hit the shock and need to slow down extremely quickly to account for the abrupt increase in car density. Let's plot the car positions for the other scenario, with $\overline{\rho} = 0.3\rho_{max}$ and $\delta\rho = -0.2\rho_{max}$.

```
[12]: traffic = trafficFlow(0.3, -0.2)
      rho_vects = traffic.plot_rho_at_times_shocks(1, 20, plot_cars=True)
```





Now, the cars that are past $x = 0.5$ have their lines become steeper at some points, which makes sense because the perturbation is one with decreased density and the cars can move faster. Then, the lines flatten again when the cars reach the area of constant (higher) density again. After the shock has developed, this flattening becomes even more abrupt, as the cars must slow down even faster (this is most easily seen in car 6).

It is fascinating to be able to use a model such as this, governed by a hyperbolic partial differential equation, to numerically simulate such a common, everyday situation. Most people encounter sudden traffic jams, and may even wonder why they form so quickly after they are seemingly travelling at a constant speed for some time. This simulation shows and explains how perturbations in car densities (which can be caused by accidents, distractions, cars entering/exiting the highway, or other disruptions) can move through the stretch of road being considered, affecting cars at different times. It also shows how these perturbations can turn into shocks depending on how large they are, where there is a location of an abrupt increase in car density that propagates across the road and forces cars to change speed very rapidly. These types of shocks are a common reason for accidents in areas of dense traffic, suggesting that drivers should pay close attention to the traffic around them to try to maintain a constant density by leaving enough space between them and other cars. Although the model used in this project is a simple one to model a complex situation like traffic flow, it demonstrates phenomena that are seen in real life and shows why potentially harmful accidents can happen when traffic is quite dense.

```
[ ]:
```