3rd Year Computational Physics : Jessica Murphy

07/04/2024

Assignment 9

This python notebook investigates **chaos** through the Logistic equation which is used to model population dynamics that are competing for limited resources. The example used throughout this n otebook is inevstigating the population of insects and their competition for space and food. The equation is a discrete equation where the left hand side is an update of the values on the right hand side, and is as follows:

$$x_{n+1} = 4rx_n(1 - x_n) \quad \star$$

where

- x_n is the normalised population of insects in a given year,
- r is the population growth parameter, i.e. controlling the rate of reproduction,
- both of these are restricted to the range 0 to 1.

Essentially, the aim is to investigate aperiodic behavior and extreme sensitivity to inital conditions, which is a hallmark of a chaotic system.

Task One:

The objective of task one is to undertsand the constraints of the logistic equation, and the time it takes for different values of r to stabilise.

The population eventually dies out as n goes to infinity for a certain range of r values. What is this range?

To answer this question, physical conditions need to be imposed on this mathematical equation. Firstly, we need that $x(n)$ is a number between 0 and 1. Obviously if x(n) is at zero, there are no insects left and a negative number makes no sense. If $x(n) \geq 1$ then $x(n+1) \leq 0$ which again makes no snese in our system. If we choose $x(n) = 0.5$ (other values are valid too but we will use this one), then r is limited to the range $0 \leq r \leq 1$. And, r can be copnstrained further by considering the case where after a very long period of time the population stabilises to a constant numebr such that $x(n+1) = x(n) = C$, say. Then the logistic equation becomes;

$$C = 4rC(1 - C)$$

and this equation has quadratic solutions $C = 0$ or $C = 1 - \frac{1}{4r}$. So the condition that a positive and non-zero constant population is reached is that $r \geq 0.25$. This means that the range of r values where the population dies out is $0 \leq r \leq 0.25$ as n goees to infinity.

Next we are tasked to find the steady state values of x_n of the population for r = 0.3, 0.4, 0.5, 0.6.

How many years need to elapse in each before the result stabilises?

A breakdown of the steps fro teh code is given below in order to answer this question.

1. Import the relevant python modules necessary for plotting, calculations and data processing.
2. Define a function to calculate the logistic map, a sequence of values representing the state of a system over time.
   - X_0 is the initial state or strating population.
   - N is the number of iterations to perform.
3. Set the number of iterations.
4. For debugging and testing start with initial values of $r = 0.6, x_0 = 0.5$
5. Create a list of the different growth parameters.
6. Define a function to find the steady state solutions from the logistic function output, based on specified tolerance.
   - Iterate thorugh the population to find when change stabilise.
   - Create an if statement to keep the population iterations above the tolerance level.
7. Initialise an empty list to store the steady state values for different growt rates.
8. Output the analysis. Check the population settles down to 0.5833 after several years.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd # for data processing, data manipulation and analysis
plt.style.use('fivethirtyeight') # for background use style sheet from matplotlib
```

In [2]:
```
# Define the logistic map function
def logistic(r, X_0, N):
    # Initialize an array of zeros to store the population sizes for N generations
    X = np.zeros(N)
    X[0] = X_0  # Explicitly state the zeroth element

    # Use a for loop to iterate over N-1 generations to calculate population changes
    for i in range(N - 1):
        # Logistic map equation: X[n+1] = 4*r*X[n]*(1 - X[n])
        X[i + 1] = 4 * r * X[i] * (1 - X[i])

    # Return the array of population sizes over time
```

```
    return X

# Define the number of iterations to simulate
N = 100

# Test the function for r = 0.6 and x0 = 0.5
test_r = 0.6
test_x0 = 0.5
test_output = logistic(test_r, test_x0, N)

# Define r values to investigate
r_values = [0.3, 0.4, 0.5, 0.6]

# Function to find steady state from the logistic function output
# Adding a tolerance paaremeter to run the model until the difference in successive years is small
def find_steady_state(X, tolerance=1e-6):
    # Iterate through the population array to find when changes stabilise
    for i in range(1, len(X)):
        # Check if the change in population size falls below a tolerance level
        if abs(X[i] - X[i-1]) < tolerance:
            # If so, return the steady state value and the iteration number
            return X[i], i  # Returning steady state value and iteration
    return None, None  # In case no steady state is found

# Compute steady state values for given r values
# Initialise a list to hold steady state values for different growth rates
steady_states = []
for r in r_values:
    X = logistic(r, test_x0, N)
    steady_state, iterations = find_steady_state(X)
    steady_states.append((r, steady_state, iterations))

# Show the last 10 values to observe stabilisation,
#and steady states for different growth rates
test_output[-10:], steady_states
```

Out[2]:
```
(array([0.58333333, 0.58333333, 0.58333333, 0.58333333, 0.58333333,
        0.58333333, 0.58333333, 0.58333333, 0.58333333, 0.58333333]),
 [(0.3, 0.1666701170799059, 45),
  (0.4, 0.3750003555893542, 13),
  (0.5, 0.5, 1),
  (0.6, 0.5833332143231952, 14)])
```

This code illustrates the amount of years it takes the different r values to reach their steady state solutions. A table is provided below to demonstrate this clearly.

r value	Steady state solution	Time elapsed (years)
0.3	0.1666701170799059	45
0.4	0.3750003555893542	13
0.5	0.5	1
0.6	0.5833332143231952	14

As expected, x_n settles down to $\approx$ 0.5833 after several years - specifically 14. These results show that the population of insects stabilises at different rates depending on the growth parameter r.

Task Two:

Task two aims to deepen understanding of the logistic map's complex dynamics, specifically how the system transitions between stability, periodicity, and chaos as the growth rate r changes. It further highlights the sensitivity to parameter values inherent to chaos theory, which is one of the two causes being investigated throughought. Task 3 involves investigating sensitivty to initial conditions.

The task is to compute x_n for $r = 0.8$ when $x_0 = 0.1$, and $r = 0.88$ when $x_0 = 0.1$. The steady points, and how many there are, is also recorded. Then, r is slowly increased to see how many steady points can be found. A plot of the steady solution values as a function of r is created. Any visble 'islands of stability' are located and analysed.

'Islands of stability' - areas where the system temporarily demonstrates regular, periodic behavior before becoming chaotic again as r grows.

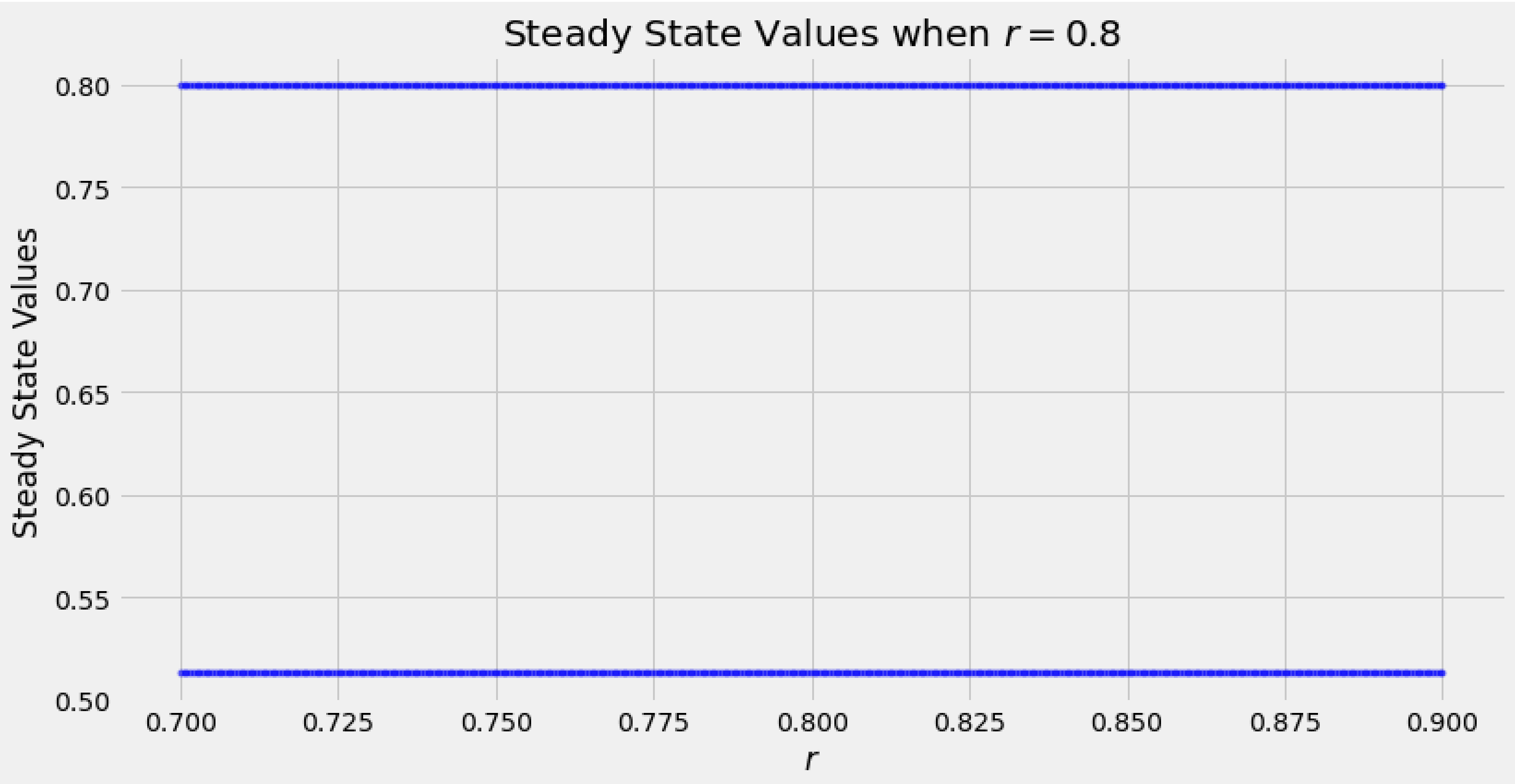
The breakdown of the steps followed for this task are given below:

1. Set the parameters; number of iterations and the number of iterations to ignore before considering the system to have reached a steady state.
2. Generate a linearly spaced array of r values in a suitable range for each r.
3. Initialise an empty list to store the steady state values for each r.
4. Create a for loop to iterate over each r value to compute the logistic equation and identify steady states for each.
5. Plot the steady state values as a function of r for better visualisation.
6. Use the Pandas DataFrame function to display the exact values of the stable solutions for x_n. Only the first row and first column are needed so the code is adjusted as appropriate. Then display this in a Pandas table.
7. Do this for inceasiung r values; $0.8, 0.88. 0.89, 0.899$ and follow the same steps for each.
8. Create a plot of the steady solution values as a function of r.

9. Zoom in on the chaotic region to discern any islands of stability.
10. Use same method as before to try count the solutions.

When $r = 0.8$

In [3]:
```
# Parameters
N1 = 10000  # Number of iterations, choosing a large number to allow the system to reach a steady state
transient = 9500  # Number of iterations to ignore before considering the system to have reached a steady state

# Generates a list of r values between 0.7 and 0.9 (inclusive) with fine resolution
r_values = np.linspace(0.7, 0.9, 400)

# Initialise an empty list to hold the steady state values for each r
steady_states1 = []

# Iterate over each r value to compute the logistic map and identify steady states
for r in r_values:
    # Generates the population series for the given value of r with initial population 0.1
    X = logistic(0.8, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states1.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states1[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)# Plots each steady-state value for the corresponding r value

# Set plot titles and labels
plt.title('Steady State Values when $r = 0.8$')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



Here for $r = 0.8$, there seems to be two steady points x_n at ≈ 0.52 and ≈ 0.80, when reading from the graph.

In [4]:
```
#pd.set_option('display.max_rows', None)  # None for all rows, or set a specific number like 100
pd.set_option('display.max_colwidth', None)  # None for no truncation, or set a specific width like 50

Task2_solutionA = pd.DataFrame({'First steady states': steady_states1})

Task2_solutionA.iloc[0]
```

Out[4]:
```
First steady states    [0.5130445095326298, 0.7994554904673701]
Name: 0, dtype: object
```

The exact values for x_n are 0.5130445095326298 and 0.7994554904673701.

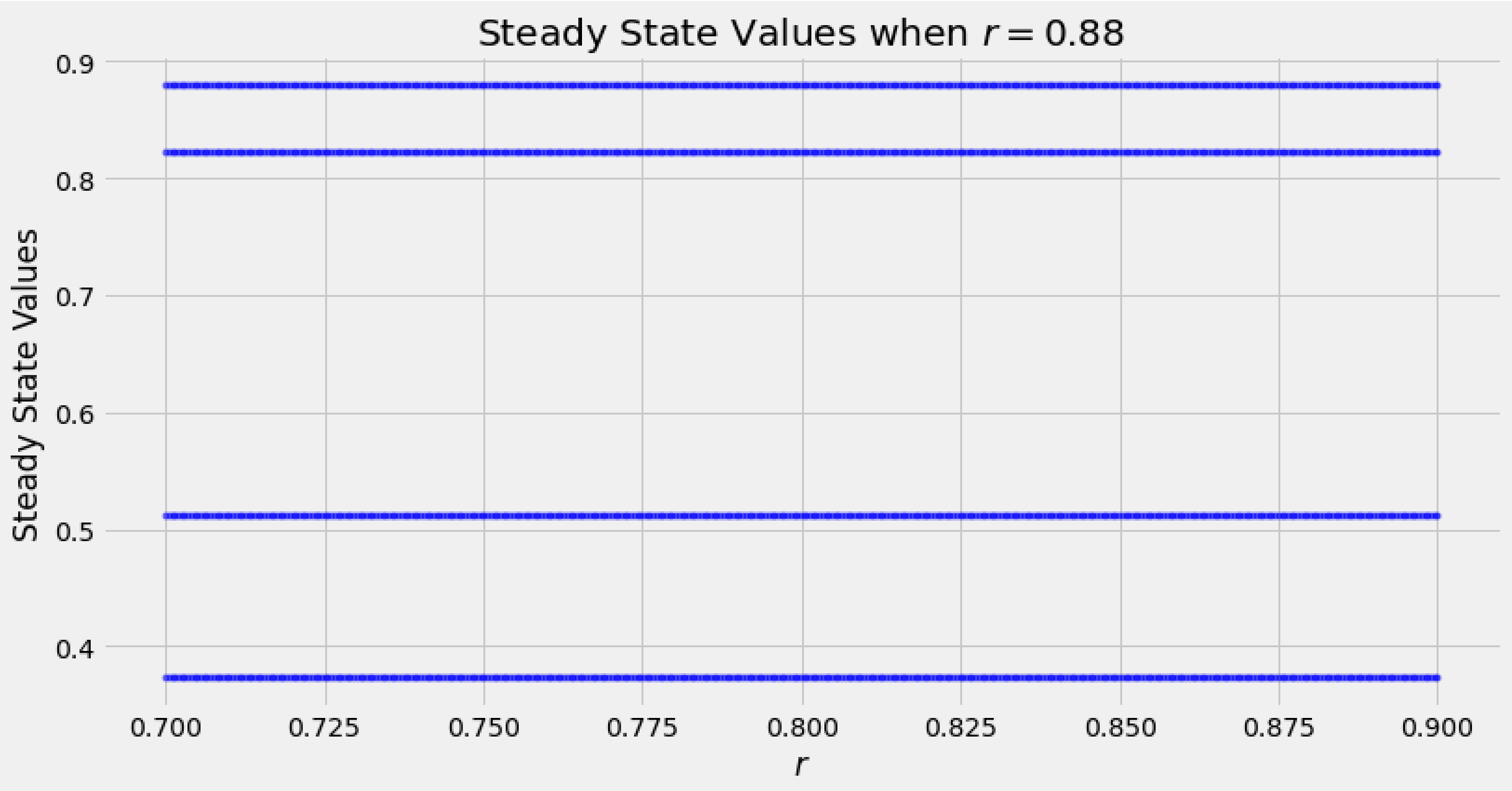
In [5]:
```
pd.set_option('display.max_colwidth', None)
pd.set_option('display.max_rows', None)

single_value = steady_states1[0]
solution_setA = pd.DataFrame({'Normalised population': [single_value]})

solution_setA
```

Out[5]:

	Normalised population
0	[0.5130445095326298, 0.7994554904673701]

Now, when $r = 0.88$

In [6]:
```
# Initialise an empty list to hold the steady state values for each r
steady_states2 = []

for r in r_values:
    X = logistic(0.88, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states2.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states2[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Steady State Values when $r = 0.88$')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



Here for $r = 0.88$, there seems to be four steady points x_n at ≈ 0.38, ≈ 0.51, ≈ 0.82, and ≈ 0.88, when reading from the graph.

The exact steady states are given below.

In [7]:
```
Task2_solutionB = pd.DataFrame({'Steady states when r = 0.88': steady_states2})

Task2_solutionB.iloc[0]
```

Out[7]:
```
Steady states when r = 0.88    [0.37308439050627956, 0.512076361844488, 0.8233013467952679, 0.87948664484257955]
Name: 0, dtype: object
```

In [8]:
```
single_value = steady_states2[0]
solution_setB = pd.DataFrame({'Normalised population': [single_value]})

solution_setB
```

Out[8]:

	Normalised population
0	[0.37308439050627956, 0.512076361844488, 0.8233013467952679, 0.87948664484257955]

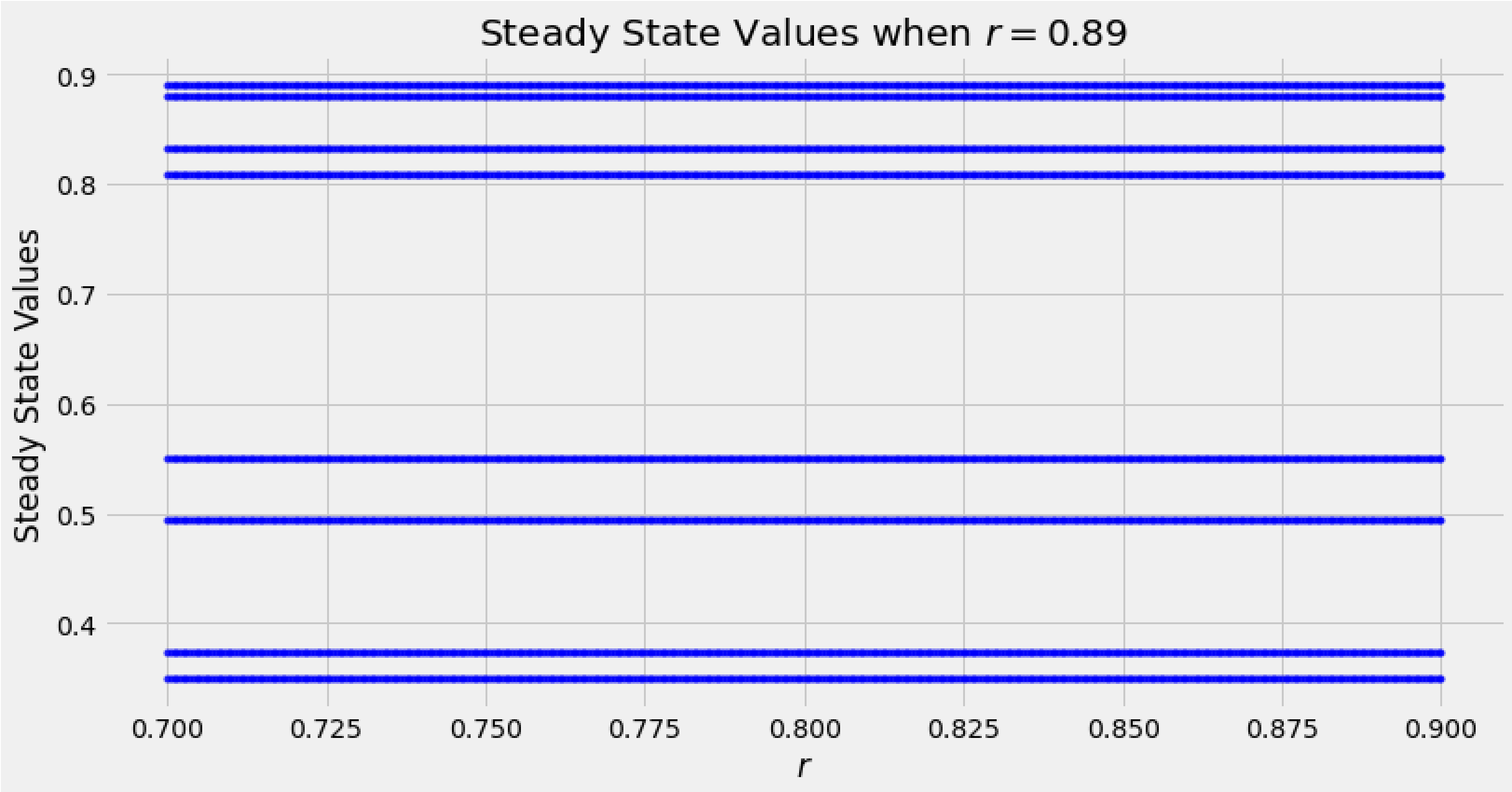
In [9]:
```
# Counting the number of elements in the first cell of the DataFrame
number_of_elements = len(Task2_solutionB.iloc[0, 0])

print(f"Number of elements in the first cell: {number_of_elements}")
```

```
Number of elements in the first cell: 4
```

When $r = 0.89$

In [10]:
```
# Initialise an empty list to hold the steady state values for each r
steady_states3 = []

for r in r_values:
    X = logistic(0.89, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states3.append(np.unique(X[transient:]))
```

```
# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states3[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Steady State Values when $r = 0.89$')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



Here for $r = 0.89$, there seems to be eight steady points x_n at ≈ 0.36, ≈ 0.38, ≈ 0.49, ≈ 0.56, ≈ 0.81, ≈ 0.84, ≈ 0.88, and ≈ 0.89 when reading from the graph.

An exact solution is given below.

In [11]:
```
Task2_solutionC = pd.DataFrame({'Steady states when r = 0.89': steady_states3})

Task2_solutionC.iloc[0]
```

Out[11]:
```
Steady states when r = 0.89    [0.34882408493538364, 0.34882408493538464, 0.3738136695381188, 0.37381366953812084, 0.494489958250335, 0.4944899582503389, 0.5508809653745559, 0.550880965374558, 0.8086392000275783, 0.8086392000275793, 0.8333141556162511, 0.8333141556162528, 0.8807836134106888, 0.8807836134106896, 0.8898919164061042, 0.8898919164061045]
Name: 0, dtype: object
```

In [12]:
```
single_value = steady_states3[0]
solution_setC = pd.DataFrame({'Normalised population': [single_value]})

solution_setC
```

Out[12]:

	Normalised population
0	[0.34882408493538364, 0.34882408493538464, 0.3738136695381188, 0.37381366953812084, 0.494489958250335, 0.4944899582503389, 0.5508809653745559, 0.550880965374558, 0.8086392000275783, 0.8086392000275793, 0.8333141556162511, 0.8333141556162528, 0.8807836134106888, 0.8807836134106896, 0.8898919164061042, 0.8898919164061045]

In [13]:
```
# Counting the number of elements in the first cell of the DataFrame
number_of_elements = len(Task2_solutionC.iloc[0, 0])

print(f"Number of elements in the first cell: {number_of_elements}")
```

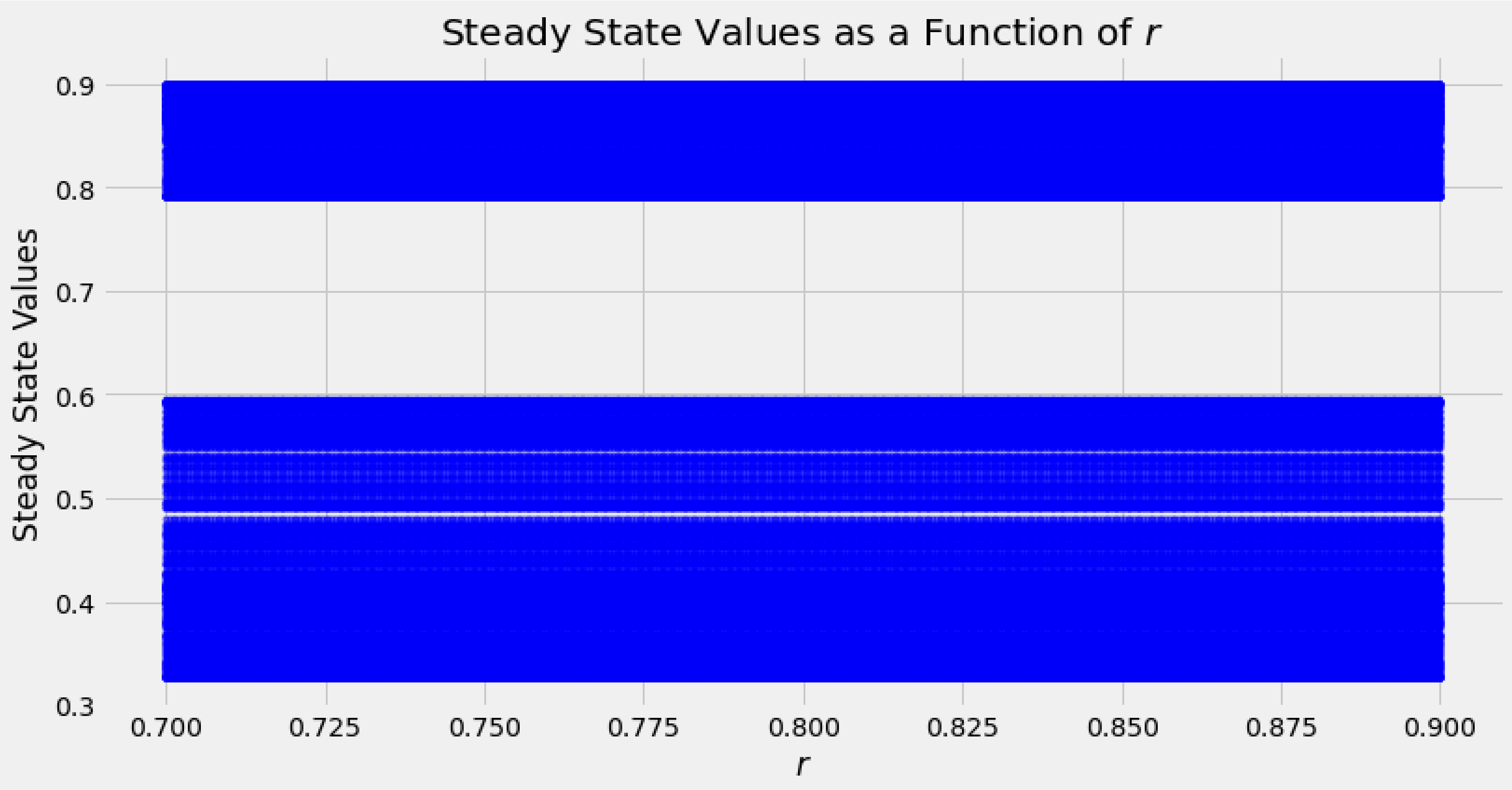
```
Number of elements in the first cell: 16
```

It should be noted that although there appears to be eight lines on the graph, seeming to have eight steady states, from the analytical Pandas solution there is actually 16 steady states. This highlights the importance of verifying answers using more than one method.

When $r = 0.899$

In [14]:
```
# Initialise an empty list to hold the steady state values for each r
steady_states4 = []

for r in r_values:
    X = logistic(0.899, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states4.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
```

```python
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states4[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Steady State Values as a Function of $r$')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



In [15]:
```python
Task2_solutionD = pd.DataFrame({'Steady states when r = 0.89': steady_states4})

Task2_solutionD.iloc[0]
```

Out[15]:
```
Steady states when r = 0.89    [0.326513221980316, 0.3267050642689482, 0.3267082340701375, 0.32672526953801184, 0.32672542038213
9, 0.3268001374829552, 0.32685362071005336, 0.3268939160775629, 0.3271968535736412, 0.32723019531648834, 0.3272732167667577, 0.3
273627401994539, 0.32766808703499106, 0.3277916829863875, 0.3283184511987697, 0.32838400721426736, 0.32871747921340483, 0.330205
00870788033, 0.3304717545690174, 0.3313679485121994, 0.33269877651047675, 0.3331282418135998, 0.3335879510596085, 0.333703802281
14893, 0.33536222799134807, 0.33570149821133677, 0.3360716780012212, 0.3363181478644799, 0.3370402708797139, 0.3383395969871051,
0.3384384303812414, 0.3389766337627187, 0.3389814523992602, 0.3395658891815094, 0.34020276461043464, 0.3407761150474233, 0.34144
73344484783, 0.3414821435446225, 0.34239260986105896, 0.3434108194668918, 0.344153893640815, 0.3449395469312867, 0.3474353858534
3015, 0.3482974438094472, 0.34878046671865587, 0.34979113677942186, 0.35173142737910384, 0.35189138951992327, 0.353639640602582
2, 0.3554967062709817, 0.35636972139709955, 0.3584418700389486, 0.35892371922162825, 0.35902347559372105, 0.36013552827652723,
0.3609774415563587, 0.3623785162404874, 0.3626921946022811, 0.3646461990314177, 0.3664848462302195, 0.3665001865824949, 0.366580
5935382638, 0.36710486228802236, 0.36724822680853486, 0.3681466507311414, 0.3682321681326191, 0.368554222623905, 0.372903230101
2782, 0.37541239986363384, 0.3757922544492381, 0.3762838152048863, 0.3776069242445, 0.37786313802359767, 0.3788346091712041, 0.3
797254622566616, 0.3797913863565593, 0.3801461921812776, 0.38014933771553394, 0.3809300112967301, 0.38128587465393815, 0.3817155
9135989397, 0.3828466271074453, 0.3837041257804081, 0.38504184975079, 0.3854881794155535, 0.3857350666879406, 0.38727552870395557
6, 0.3889687001469043, 0.3902913745581257, 0.3910308519797068, 0.391989048210539, 0.39372718219989883, 0.3939393543921715, 0.393
99720591079446, 0.394358135531448, 0.39732973948580447, 0.3981078834659311, 0.39879928738946974, 0.3988236240309778, 0.398954446
8412542, ...]
Name: 0, dtype: object
```

In [16]:
```python
# Counting the number of elements in the first cell of the DataFrame
number_of_elements = len(Task2_solutionD.iloc[0, 0])

print(f"Number of elements in the first cell: {number_of_elements}")
```

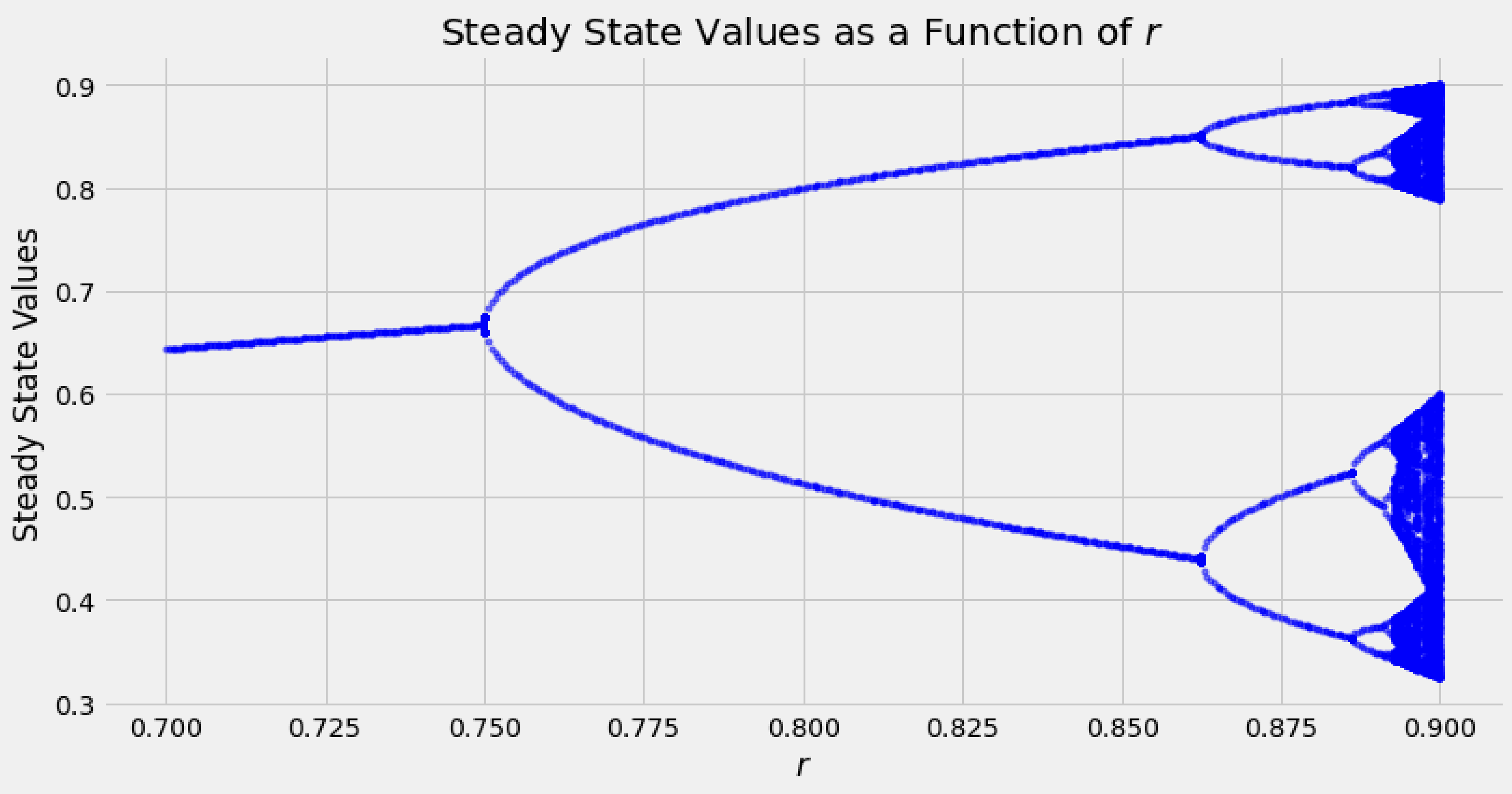
```
Number of elements in the first cell: 500
```

For $r = 0.899$, there are no clear steady points, and hence the solution has become chaotic.

Below a plot is made of the steady solution values as a function of r.

In [17]:
```python
# Define r values to investigate, with fine resolution in the chaotic region
r_values = np.linspace(0.7, 0.9, 400)
# Initialise an empty list to hold the steady state values for each r
steady_states5 = []

for r in r_values:
    X = logistic(r, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states5.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states5[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Steady State Values as a Function of $r$')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
```

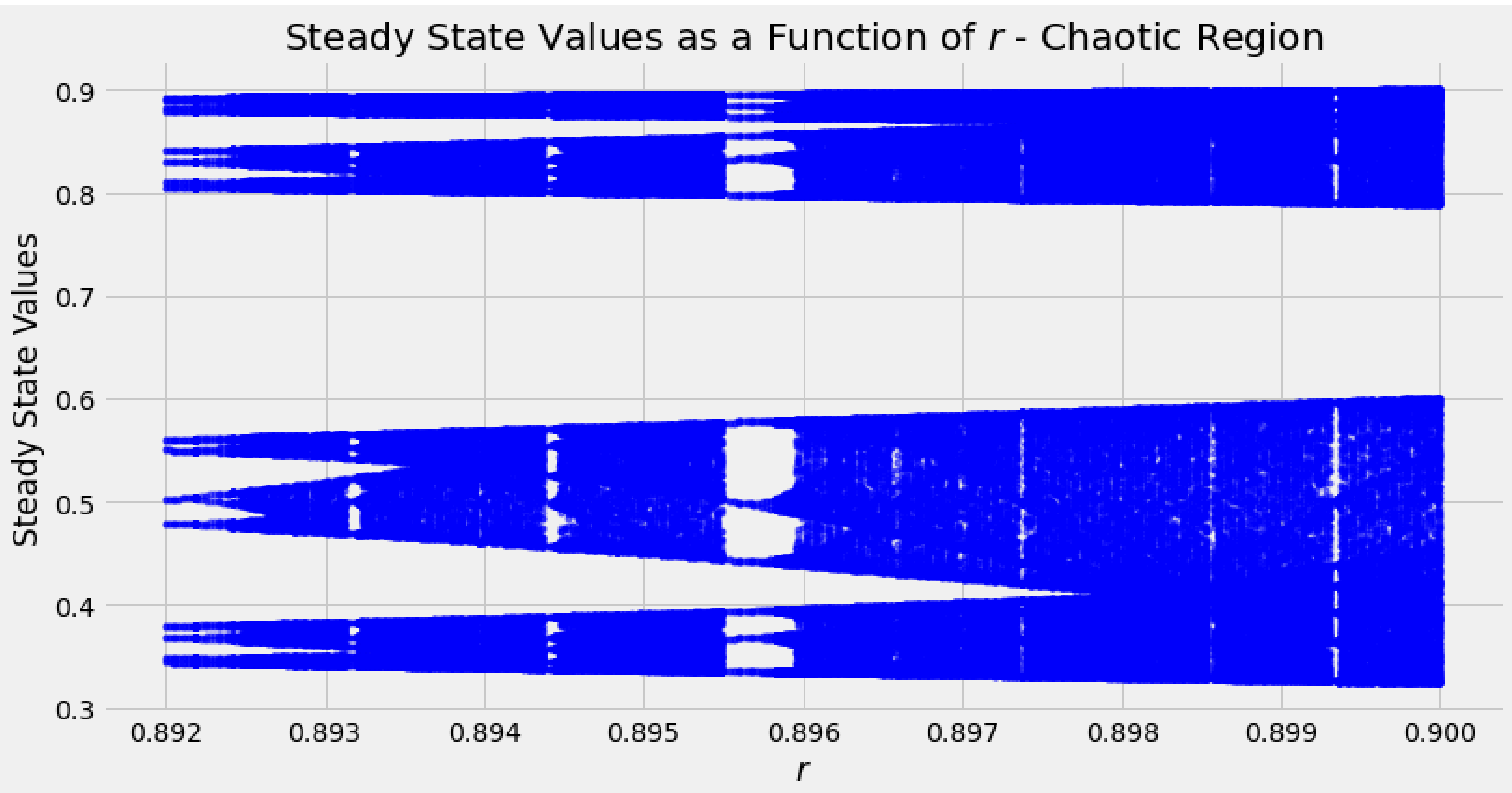
```
#plt.grid()
plt.show()
```



The plot illustrates the steady state values as a function of the growth parameter r in the range from 0.7 to 0.9. Each blue dot represents a steady state value that the population can achieve for a given r. As r increases within this range, we observe a transition from stability to chaos, marked by the increasing number of steady state values and then to regions where complex behavior is exhibited.

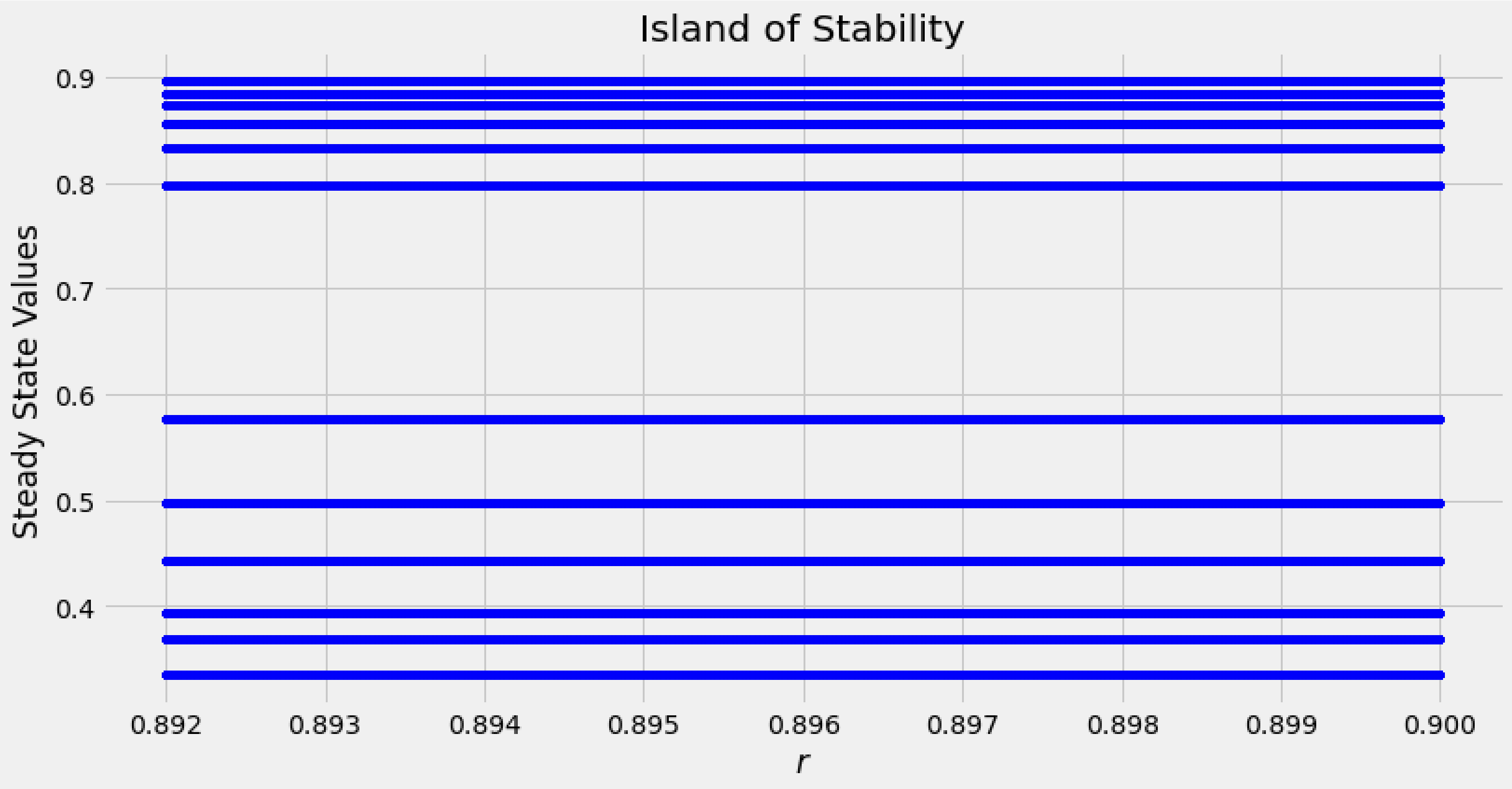
Zooming in the chaotic region:

In [18]:
```
# Define r values to investigate, with fine resolution in the chaotic region
r_values = np.linspace(0.892, 0.9, 400)
# Initialise an empty list to hold the steady state values for each r
steady_states6 = []

for r in r_values:
    X = logistic(r, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states6.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states6[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Steady State Values as a Function of $r$ - Chaotic Region')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



When zooming in to the chaotic region, there appears to be a few areas that have potential to be so-called 'islands of stability'. These are areas where the system temporarily exhibits regular behavior before becoming chaotic again as r increases further. However, the clearest 'island of stabilty' seems to be when r is in the range of about $0.8955 - 0.8958$. To investigate this further, another graph is plotted within this range.

In [19]:
```python
# Define r values to investigate, with fine resolution in the chaotic region
r_values1 = np.linspace(0.8956, 0.8958, 400)

# Initialise an empty list to hold the steady state values for each r
steady_states7 = []

for r in r_values:
    X = logistic(0.8957, 0.1, N1)
    # Only consider the values after the transient to identify steady states
    steady_states7.append(np.unique(X[transient:]))

# Plot the steady state values as a function of r
plt.figure(figsize=(12, 6))
for i, r in enumerate(r_values):
    for ss in steady_states7[i]:
        plt.plot(r, ss, 'b.', alpha=0.5)

plt.title('Island of Stability')
plt.xlabel('$r$')
plt.ylabel('Steady State Values')
#plt.grid()
plt.show()
```



As expected, there seems to be about 12 steady points at $\approx 0.34, 0.38, 0.39, 0.45, 0.58, 0.80, 0.84, 0.86, 0.87, 0.88, 0.89$.

In [20]:
```python
Task2_solutionE = pd.DataFrame({'Steady states in island of stability': steady_states7})

Task2_solutionE.iloc[0]
```

Out[20]:
```
Steady states in island of stability    [0.33477868750799894, 0.33477868750991346, 0.33477868751176776, 0.3347786875135639, 0.33
47786875153031, 0.33477868751698836, 0.3347786875186205, 0.3347786875202016, 0.3347786875217333, 0.33477868752321654, 0.33477868
75246532, 0.3347786875260451, 0.33477868752739304, 0.33477868752869866, 0.33477868752996315, 0.3347786875311889, 0.3347786875323
756, 0.33477868753352513, 0.3347786875346385, 0.3347786875357179, 0.3347786875367623, 0.3347786876017832, 0.3347786876028446, 0.
33477868760394097, 0.3347786876050723, 0.33477868760624013, 0.33477868760744606, 0.3347786876086913, 0.33477868760997653, 0.3347
78687611303, 0.3347786876126723, 0.33477868761408686, 0.3347786876155465, 0.33477868761705365, 0.3347786876186096, 0.33477868762
021623, 0.33477868762187507, 0.3347786876235868, 0.334778687625354, 0.3347786876271794, 0.3347786876290633, 0.33477868763100865,
0.3681522143514142, 0.3681522143675382, 0.3681522143831548, 0.36815221439828455, 0.3681522144129332, 0.36815221442712204, 0.3681
522144408729, 0.36815221445418844, 0.36815221446708696, 0.3681522144795786, 0.3681522144916782, 0.3681522145034057, 0.3681522145
1475344, 0.36815221452574864, 0.3681522145364028, 0.36815221454672425, 0.3681522145567206, 0.36815221456640007, 0.36815221457577
79, 0.36815221458486663, 0.3681522145936636, 0.368152215141281, 0.3681522151502266, 0.3681522151594561, 0.3681522151689851, 0.36
815221517882146, 0.3681522151889811, 0.3681522151994619, 0.3681522152102846, 0.3681522152214593, 0.36815221523299657, 0.36815221
52449048, 0.3681522152571992, 0.36815221526989567, 0.36815221528300174, 0.36815221529653097, 0.36815221531049946, 0.368152215324
91577, 0.36815221533980313, 0.3681522153551743, 0.3681522153710437, 0.36815221538742443, 0.394446563128311, 0.39444656313565185,
0.39444656314276194, 0.3944465631496505, 0.3944465631563192, 0.3944465631627793, 0.39444656316903975, 0.3944465631751023, 0.3944
4656318097476, 0.39444656318666205, 0.3944465631921706, 0.39444656319750976, 0.39444656320267635, 0.3944465632076823, 0.39444656
321253285, 0.394446563217232, ...]
Name: 0, dtype: object
```

In [21]:
```python
# Counting the number of elements in the first cell of the DataFrame
number_of_elements = len(Task2_solutionE.iloc[0, 0])

print(f"Number of elements in the first cell: {number_of_elements}")
```

```
Number of elements in the first cell: 500
```

This solution still appears to be chaotic but this may be due to the range of r values picked, i.e. the range could be too big so it includes chaotic solutions. A refined range may yield more stable results. This coupled with more evenly spaced samples may contribute an island of stability.

Task Three:

Task three delves into exploring the sensitivity to initial conditions, which is the second charachteristic of chaos being studied. Two initial conditions are given with a difference of 0.0001 and their divergence is looked into over a period of time.

1.

In [22]:
```
# Parameters for sensitivity analysis
r_new = 0.99
x0_1 = 0.1
x0_2 = 0.10001
N2 = 100

# Running the function for both initial conditions with a smaller N for illustration
solution_1_short = logistic(r_new, x0_1, 70)  # Shorter sequence for illustrative purposes
solution_2_short = logistic(r_new, x0_2, 70)  # Same here
Difference = np.abs(solution_2_short - solution_1_short)

# Displaying the first few values of each solution to illustrate
#solution_1_short[:50], solution_2_short[:50]

solution_set = pd.DataFrame({'x0 = 0.1': solution_1_short, 'x0 = 0.10001': solution_2_short, 'Difference': Difference})

solution_set
```

Out[22]:

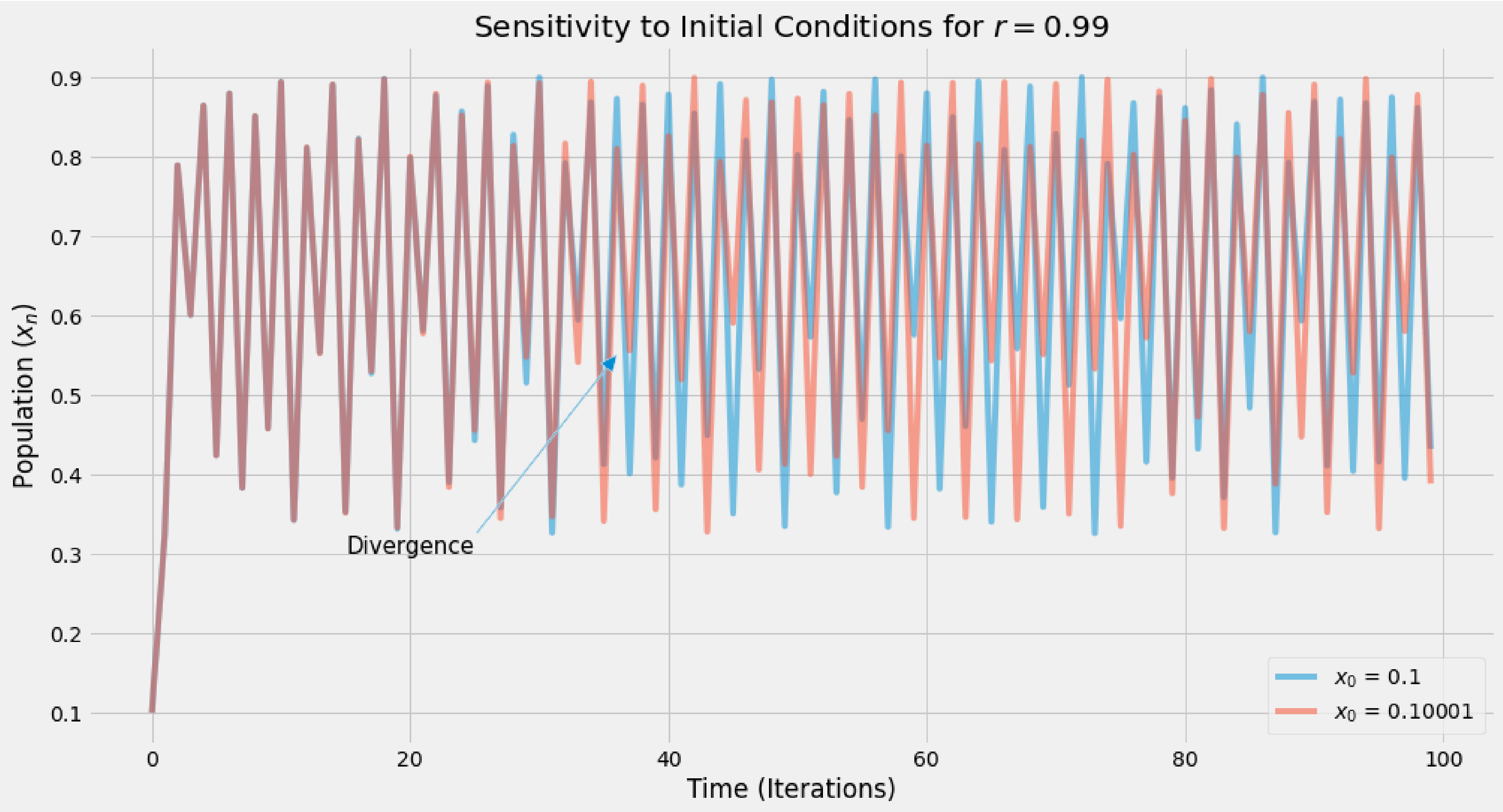
	x0 = 0.1	x0 = 0.10001	Difference
0	0.100000	0.100010	0.000010
1	0.356400	0.356432	0.000032
2	0.908341	0.908377	0.000036
3	0.329700	0.329584	0.000117
4	0.875152	0.874995	0.000157
5	0.432673	0.433140	0.000467
6	0.972050	0.972298	0.000248
7	0.107589	0.106661	0.000928
8	0.380214	0.377326	0.002887
9	0.933179	0.930407	0.002772
10	0.246929	0.256410	0.009481
11	0.736383	0.755030	0.018647
12	0.768728	0.732441	0.036287
13	0.704031	0.776046	0.072016
14	0.825151	0.688241	0.136910
15	0.571336	0.849678	0.278342
16	0.969848	0.505792	0.464056
17	0.115801	0.989867	0.874066
18	0.405468	0.039719	0.365748
19	0.954612	0.151042	0.803571
20	0.171578	0.507783	0.336205
21	0.562870	0.989760	0.426890
22	0.974347	0.040135	0.934213
23	0.098978	0.152555	0.053576
24	0.353159	0.511956	0.158797
25	0.904613	0.989434	0.084821
26	0.341701	0.041399	0.300301
27	0.890768	0.157155	0.733613
28	0.385310	0.524530	0.139221
29	0.937911	0.987617	0.049707
30	0.230608	0.048429	0.182179
31	0.702615	0.182491	0.520124
32	0.827431	0.590784	0.236647
33	0.565443	0.957362	0.391919
34	0.973040	0.161646	0.811395
35	0.103883	0.536644	0.432761
36	0.368642	0.984682	0.616041
37	0.921670	0.059728	0.861942
38	0.285890	0.222397	0.063493
39	0.808461	0.684828	0.123633
40	0.613213	0.854720	0.241508
41	0.939244	0.491727	0.447517
42	0.225976	0.989729	0.763753
43	0.692646	0.040256	0.652391
44	0.843034	0.152995	0.690040
45	0.524017	0.513166	0.010851
46	0.987716	0.989314	0.001598
47	0.048048	0.041866	0.006182
48	0.181127	0.158848	0.022278
49	0.587346	0.529117	0.058229
50	0.959788	0.986643	0.026855
51	0.152837	0.052188	0.100649

	x0 = 0.1	x0 = 0.10001	Difference
52	0.512733	0.195881	0.316852
53	0.989358	0.623746	0.365612
54	0.041694	0.929361	0.887667
55	0.158224	0.259972	0.101749
56	0.527428	0.761851	0.234423
57	0.987021	0.718479	0.268542
58	0.050730	0.800977	0.750247
59	0.190700	0.631275	0.440575
60	0.611160	0.921757	0.310597
61	0.941068	0.285598	0.655470
62	0.219618	0.807967	0.588349
63	0.678688	0.614420	0.064268
64	0.863560	0.938156	0.074596
65	0.466583	0.229756	0.236827
66	0.985578	0.700795	0.284783
67	0.056287	0.830339	0.774051
68	0.210352	0.557871	0.347519
69	0.657772	0.976738	0.318966

This table demonstrates that at the beginning, there is minimal difference between the two datasets, but as we progress towards about 70 iterations, there seems to be more of a substantial difference.

In [23]:

```
# Generating solutions for the two initial conditions
solution_1 = logistic(r, x0_1, N2)
solution_2 = logistic(r, x0_2, N2)

# Plotting the divergence of the solutions over time
plt.figure(figsize=(15, 8))
plt.plot(solution_1, label=f'$x_0$ = {x0_1}', alpha=0.5)
plt.plot(solution_2, label=f'$x_0$ = {x0_2}', alpha=0.5)
plt.xlabel('Time (Iterations)')
plt.ylabel('Population ($x_n$)')
plt.title('Sensitivity to Initial Conditions for $r = 0.99$')
plt.annotate('Divergence', xy=(36,0.55), xytext=(25,0.3), fontsize=15, arrowprops={'width':1,'headwidth':10, 'headlength':11}, hc
plt.legend()
#plt.grid()
plt.show()
```



From the graph, it is clear to see that the two solutions appear very similar with little to no disparity, and appear to be in phase. It is until they reach an iteration of about 35, larger discrepencies are starting to happen. Now they are beginning to show they are out of phase.

Task Four:

The aim of task four is to make an estimate of **Feigenbaum's** constant. The crucial requirement for this task to be somewhat successful is to have very accurate measurements of the critical points from the data measured in task two.

To do this accurately, a code was written that is designed to investigate period-doubling bifurcations in the logistic map. It aims to identify the values of the parameter r for which the period of the system's output doubles.

In [24]:

```
# Define a function to find the repeating period of a sequence
def find_period(X):
    max_lag = 50  # Maximum expected period to test
    min_lag = 1  # Minimum period to consider
    tolerance = 0.001  # Tolerance for considering values as equal
    # Loop through possible lag values to identify the period
    for lag in range(min_lag, max_lag):
        # Check if the sequence is approximately equal to itself when shifted by 'lag'
        if np.allclose(X[:-lag], X[lag:], atol=tolerance, rtol=0):
            # If a repeating pattern is found, return the lag as the period
            return lag
    return None  # No repeating period found within max_lag

# Parameters
r_start = 0.7 # Starting value of growth variable
r_end = 0.9# Ending value of growth variable
r_step = 0.0001 # Increment step for the parameter r
N = 10000 # Number of iterations to simulate
transient = 9000 # Number of initial iterations to discard (transient behavior)

last_period = None
# Initialise a list to store the points of period doubling
period_doublings = []

# Iterate over the range of r values
for r in np.arange(r_start, r_end, r_step):
    # Generate the sequence using the logistic function for the current value of r
    X = logistic(r, 0.1, N)
    # Focus on the steady-state behavior by discarding the transient part of the sequence
    steady_state = X[transient:]
    # Find the period of the steady-state sequence
    period = find_period(steady_state)
    # Crucial step: check for period doubling: if the current period is exactly twice the last observed period
    if period is not None and last_period is not None and period == 2 * last_period:
        # Record the r value and the period at which doubling occurred
        period_doublings.append((r, period))
    # Update the last observed period for comparison in the next iteration
    last_period = period

# Print the r values and corresponding periods where period doubling was observed
for doubling in period_doublings:
    print(f"Period doubling at r = {doubling[0]} with period = {doubling[1]}")
```

```
Period doubling at r = 0.7499999999999944 with period = 2
Period doubling at r = 0.8623999999999821 with period = 4
Period doubling at r = 0.8860999999999795 with period = 8
Period doubling at r = 0.8911999999999789 with period = 16
Period doubling at r = 0.8921999999999788 with period = 32
```

In [25]:

```
rn = 0.8860999999999795 # value for the nth bifurcation
rn_minus_1 = 0.8623999999999821   # value for the (n-1)th bifurcation
rn_minus_2 = 0.7499999999999944   # value for the (n-2)th bifurcation

# Calculating Feigenbaum's number
F = (rn_minus_1 - rn_minus_2) / (rn - rn_minus_1)
F
```

Out[25]:

```
4.742616033755274
```

The number that was caluclated for Feigenbaum's number is 4.742616033755274. However the 'true' value is $\sigma \approx 4.669201609102990$, which means that the answer calculated above is approximately 0.073414424652284 off the actual constant. This is relatively close to the answer in respect to various errors that may have came up due to the method of estimation. Error may have arised through the tolerance that was set. A lower tolernace may yield more accuarte results but may be computationally intensive.

This is a suitable value and therefore this notebook is concluded