Aim – To implement Genetic algorithm.

Theory –

The use of genetic algorithms and genetic programming in control engineering has started to expand in the last few years. This is mainly for two reasons: the physical cost of implementing a known control algorithm and the difficulty to find such an algorithm for complex plants [56]. Broadly, evolutionary algorithms for control can be classified as either "pure" evolutionary or hybrid architectures. This chapter reviews some of the most successful applications of genetic algorithms and genetic programming in control engineering and outlines some general principles behind such applications. Demonstration of the power of these techniques is given, by describing how a hybrid genetic controller can be applied to the control of complex (even chaotic) dynamic systems. In particular, the detumbling and attitude control problems of a satellite are considered.

The Stability of Genetic Algorithm Based Controllers

Control theory has experienced a great deal of advancement over the past several decades. The control of linear time-invariant systems with known dynamics has been thoroughly explored. When faced with uncertainties in the system's dynamics, techniques such as H-infinity optimal control have proven effective and robust [1]. A variety of methods have been developed to control systems with nonlinear dynamics. These nonlinear control methods work well if the system is linearizeable; however, many linear approximations to nonlinear systems yield inconclusive models which do not provide a means to control the system [2]. The control of unknown systems is currently a popular research topic, employing techniques such as system identification, neural networks, and fuzzy logic. These methods have been extended to adaptive techniques for the control of time-varying systems. Still, a single method sufficiently robust to apply to such a broad class of systems has yet to be developed.

In recent years, researchers have become increasingly interested in the use of Genetic Algorithms as a means to control various classes of systems. Genetic Algorithms are robust search techniques based on the principles of evolution. Extensive research has been performed exploiting the robust properties of Genetic Algorithms and demonstrating their capabilities across a broad range of problems. These evolutionary methods have gained recognition as general problem solving techniques in many application, including function optimization, image processing, classification and machine learning, training of neural networks, and system control. This study focuses on the use of Genetic Algorithms in the control of unknown systems, and in particular, the stability of those systems.

Genetic Operators for On-Line Adaptive Controls

Stability Analysis of Genetic Algorithm Controllers

This study presents a method of adaptive system control based on genetic algorithms. The method consists of a population of controllers evolving towards an optimum controller through the use of probabilistic genetic operators. A brief overview of genetic algorithms is first given. The remainder of the paper identifies the problems associated with genetic algorithms controllers, and addresses the key issue of stability. A theoretical analysis of the proposed genetic algorithm controller shows the population converges to stable controllers under fitness-proportionate selection pressure. The minimization of the effects of instability is also discussed.

On the Stability of Genetic Algorithm Based Controllers

Genetic algorithms are stochastic search techniques that guide a population of solutions using the principles of evolution and natural genetics. In recent years, genetic algorithms have become a popular optimization tool for many areas of research, including the field of system control and control design. Significant research exists concerning genetic algorithms for control design and off-line controller analyses. However, little work has been done with on-line genetic algorithm controls primarily because of the problems associated with instability in early stages of the controllers evolution. Also, until now the stability of controllers based on genetic algorithms has not been researched in detail.

This study presents a genetic algorithm controller that consists of a population of controllers, each of which control the system for a specified time period. A brief overview of genetic algorithms and a history of genetic algorithms in

system controls is provided, followed by a detailed discussion of the developed controller. The scope of the research encompasses an analysis of the stability of the resulting control system with respect to the convergence of the genetic algorithm. The results of several simulations are given in support of the developed theory. Also, issues related to practical considerations of the genetic algorithm controller are discussed for future development.

Genetic Algorithms and Parallel Processing

Genetic algorithms are stochastic search techniques based on the principles of evolution. Extensive research has been performed exploiting the robust properties of genetic algorithms and demonstrating their capabilities across a broad range of problems. These evolutionary methods have gained recognition as general problem solving techniques in many applications, including function optimization, image processing, classification and machine learning, training of neural networks, and system control.

Code –

genetic.py:-

```
import random

import statistics

import sys

import time

def _generate_parent(length, geneSet, get_fitness):

    genes = []

    while len(genes) < length:

        sampleSize = min(length - len(genes), len(geneSet))

        genes.extend(random.sample(geneSet, sampleSize))

    fitness = get_fitness(genes)

    return Chromosome(genes, fitness)

def _mutate(parent, geneSet, get_fitness):

    childGenes = parent.Genes[:]

    index = random.randrange(0, len(parent.Genes))

    newGene, alternate = random.sample(geneSet, 2)

     childGenes[index] = alternate if newGene == childGenes[index] else
newGene
```

```
    fitness = get_fitness(childGenes)
    return Chromosome(childGenes, fitness)
def _mutate_custom(parent, custom_mutate, get_fitness):
    childGenes = parent.Genes[:]
    custom_mutate(childGenes)
    fitness = get_fitness(childGenes)
    return Chromosome(childGenes, fitness)
def get_best(get_fitness, targetLen, optimalFitness, geneSet, display,
             custom_mutate=None, custom_create=None):
    if custom_mutate is None:
        def fnMutate(parent):
            return _mutate(parent, geneSet, get_fitness)
    else:
        def fnMutate(parent):
            return _mutate_custom(parent, custom_mutate, get_fitness)
    if custom_create is None:
        def fnGenerateParent():
            return _generate_parent(targetLen, geneSet, get_fitness)
    else:
        def fnGenerateParent():
            genes = custom_create()
            return Chromosome(genes, get_fitness(genes))
    for improvement in _get_improvement(fnMutate, fnGenerateParent):
        display(improvement)
        if not optimalFitness > improvement.Fitness:
            return improvement
def _get_improvement(new_child, generate_parent):
    bestParent = generate_parent()
    yield bestParent
```

```
    while True:
        child = new_child(bestParent)
        if bestParent.Fitness > child.Fitness:
            continue
        if not child.Fitness > bestParent.Fitness:
            bestParent = child
            continue
        yield child
        bestParent = child
class Chromosome:
    def __init__(self, genes, fitness):
        self.Genes = genes
        self.Fitness = fitness
class Benchmark:
    @staticmethod
    def run(function):
        timings = []
        stdout = sys.stdout
        for i in range(100):
            sys.stdout = None
            startTime = time.time()
            function()
            seconds = time.time() - startTime
            sys.stdout = stdout
            timings.append(seconds)
            mean = statistics.mean(timings)
            if i < 10 or i % 10 == 9:
                print("{} {:3.2f} {:3.2f}".format(
                    1 + i, mean,
```

```
                    statistics.stdev(timings, mean) if i > 1 else 0))
```

knightsTest.py

```
import datetime
import random
import unittest
import genetic
def get_fitness(genes, boardWidth, boardHeight):
    attacked = set(pos
                   for kn in genes
                   for pos in get_attacks(kn, boardWidth, boardHeight))
    return len(attacked)
def display(candidate, startTime, boardWidth, boardHeight):
    timeDiff = datetime.datetime.now() - startTime
    board = Board(candidate.Genes, boardWidth, boardHeight)
    board.print()
    print("{}\n\t{}\t{}".format(
        ' '.join(map(str, candidate.Genes)),
        candidate.Fitness,
        timeDiff))
def     mutate(genes,     boardWidth,     boardHeight,     allPositions,
nonEdgePositions):
    count = 2 if random.randint(0, 10) == 0 else 1
    while count > 0:
        count -= 1
        positionToKnightIndexes = dict((p, []) for p in allPositions)
        for i, knight in enumerate(genes):
                    for  position  in  get_attacks(knight,  boardWidth,
boardHeight):
                positionToKnightIndexes[position].append(i)
```

```
        knightIndexes = set(i for i in range(len(genes)))
        unattacked = []
        for kvp in positionToKnightIndexes.items():
            if len(kvp[1]) > 1:
                continue
            if len(kvp[1]) == 0:
                unattacked.append(kvp[0])
                continue
            for p in kvp[1]:  # len == 1
                if p in knightIndexes:
                    knightIndexes.remove(p)
        potentialKnightPositions = \
            [p for positions in
             map(lambda x: get_attacks(x, boardWidth, boardHeight),
                 unattacked)
             for p in positions if p in nonEdgePositions] \
                if len(unattacked) > 0 else nonEdgePositions
        geneIndex = random.randrange(0, len(genes)) \
            if len(knightIndexes) == 0 \
            else random.choice([i for i in knightIndexes])
        position = random.choice(potentialKnightPositions)
        genes[geneIndex] = position
def create(fnGetRandomPosition, expectedKnights):
    genes = [fnGetRandomPosition() for _ in range(expectedKnights)]
    return genes
def get_attacks(location, boardWidth, boardHeight):
    return [i for i in set(
        Position(x + location.X, y + location.Y)
        for x in [-2, -1, 1, 2] if 0 <= x + location.X < boardWidth
```

```
        for y in [-2, -1, 1, 2] if 0 <= y + location.Y < boardHeight
        and abs(y) != abs(x))]
class KnightsTests(unittest.TestCase):
    def test_3x4(self):
        width = 4
        height = 3
        # 1,0   2,0   3,0
        # 0,2   1,2   2,2
        # 2      N N N .
        # 1      . . . .
        # 0      . N N N
        #        0 1 2 3
        self.find_knight_positions(width, height, 6)
    def test_8x8(self):
        width = 8
        height = 8
        self.find_knight_positions(width, height, 14)
    def test_10x10(self):
        width = 10
        height = 10
        self.find_knight_positions(width, height, 22)
    def test_12x12(self):
        width = 12
        height = 12
        self.find_knight_positions(width, height, 28)
    def test_13x13(self):
        width = 13
        height = 13
        self.find_knight_positions(width, height, 32)
```

```
    def test_benchmark(self):
        genetic.Benchmark.run(lambda: self.test_10x10())
          def   find_knight_positions(self,   boardWidth,   boardHeight,
expectedKnights):
        startTime = datetime.datetime.now()
        def fnDisplay(candidate):
            display(candidate, startTime, boardWidth, boardHeight)
        def fnGetFitness(genes):
            return get_fitness(genes, boardWidth, boardHeight)
        allPositions = [Position(x, y)
                        for y in range(boardHeight)
                        for x in range(boardWidth)]
        if boardWidth < 6 or boardHeight < 6:
            nonEdgePositions = allPositions
        else:
            nonEdgePositions = [i for i in allPositions
                                if 0 < i.X < boardWidth - 1 and
                                0 < i.Y < boardHeight - 1]
        def fnGetRandomPosition():
            return random.choice(nonEdgePositions)
        def fnMutate(genes):
            mutate(genes, boardWidth, boardHeight, allPositions,
                   nonEdgePositions)
        def fnCreate():
            return create(fnGetRandomPosition, expectedKnights)

        optimalFitness = boardWidth * boardHeight
           best = genetic.get_best(fnGetFitness, None, optimalFitness,
None,
                                fnDisplay, fnMutate, fnCreate)
```

```
        self.assertTrue(not optimalFitness > best.Fitness)
class Position:
    def __init__(self, x, y):
        self.X = x
        self.Y = y
    def __str__(self):
        return "{},{}".format(self.X, self.Y)
    def __eq__(self, other):
        return self.X == other.X and self.Y == other.Y
    def __hash__(self):
        return self.X * 1000 + self.Y
class Board:
    def __init__(self, positions, width, height):
        board = [['.'] * width for _ in range(height)]

        for index in range(len(positions)):
            knightPosition = positions[index]
            board[knightPosition.Y][knightPosition.X] = 'N'
        self._board = board
        self._width = width
        self._height = height
    def print(self):
        # 0,0 prints in bottom left corner
        for i in reversed(range(self._height)):
            print(i, "\t", ' '.join(self._board[i]))
        print(" \t", ' '.join(map(str, range(self._width))))
if __name__ == '__main__':
    unittest.main()
```

Output –

```
Microsoft Windows [Version 10.0.19041.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\nick_pc\Desktop\CI PRACS\geneticcontroller>genetic.py

C:\Users\nick_pc\Desktop\CI PRACS\geneticcontroller>knightsTests.py
9       . . . . . . . . . .
8       . . . . . . N . N .
7       . . . . . . N . . .
6       . N . N . N . . . .
5       . N . . N . . N . .
4       . N N . . . . . . .
3       . N N . . . . . N .
2       . . N N . . . N . .
1       . . . . . . N N . .
0       . . . . . . . . . .
        0 1 2 3 4 5 6 7 8 9
8,3 6,1 4,5 2,2 8,8 1,5 1,6 6,8 6,1 2,3 6,7 7,1 1,4 2,4 1,3 7,2 3,2 7,2 7,5 3,2 5,6 3,6
        81      0:00:00
9       . . . . . . . . . .
8       . . . . . . N . N .
7       . . . N . . N . . .
6       . N . N . N . . . .
5       . N . . N . . N . .
4       . N N . . . . . . .
3       . N N . . . . . N .
2       . . N N . . . N N .
1       . . . . . . N N . .
0       . . . . . . . . . .
        0 1 2 3 4 5 6 7 8 9
8,3 3,7 4,5 2,2 8,8 1,5 1,6 6,8 6,1 2,3 6,7 7,1 1,4 2,4 1,3 8,2 3,2 7,2 7,5 3,2 5,6 3,6
        87      0:00:00.004985
```

Conclusion – In this experiment I learnt what are genetic controller, how it is used to, I used my own dataset for the experiment, I was able to perform various operations and apply rules on the controller and learned defuzzification too.