On Synchronization Algorithms for the Optimization of Resources in Agriculture

A01242223 – Leonardo D. Garcia  
Dr. Camilo Lozoya

EP00811 Research Internship  
Tecnológico de Monterrey, Campus Chihuahua, México  
[A01242223@tec.mx](mailto:A01242223@tec.mx)  
December 3­rd, 2021

**Abstract**

This report examines a systems-based approach on the synchronization and optimization of resources in the agriculture sector. By providing an algorithmic aspect inspired by the dining philosophers problem from operating systems theory, it is possible to generate a concurrent program that will simulate the system dynamics of the plant models considering the sharing of limited resources such as water.

**Keywords**: multithreading, irrigation, algorithm, resources.

# Objectives

The objectives of this paper focus on the capability of the present individual to formalize a contemporary viewpoint into the techniques that can be used in control engineering for the regulation and stability of dynamical systems.

Operating systems are the software pillars that holds together the organized aspects of the computer architecture. Although a computer by itself can work properly, the operating system controls and coordinates the management of physical resources inside the machine to provide the swiftest and best regulated of computing procedures.

Given the accelerated abilities of the processors, it has been proposed that one can employ the very same algorithms that the OS utilizes for their handling of shared data and memory locations in a similar way that generates an analogy to the inputs to withhold the stability of a plant—an irrigation area that must share its water with other adjacent sectors.

In essence, the purpose of this experiment is in the adaptation of a concurrent, synchronization algorithm from operating systems theory into a simple yet elegant solution that can be used for the reservation of input energy.

# Theoretical Framework

## Irrigation Areas

An irrigation area is a finite-sized polygonal space where crops are grown throughout the year’s seasons. Like other real functional systems, they can be modeled mathematically as a linear-time invariant (LTI) system. A disadvantage in trying to do so lies on the fact that real systems usually present nonlinearities which complicate their analysis.

An alternative to preserve the advantages and strengths of an LTI system is by linearizing the plant on different operating points. As a result, a piecewise model of the plant is generated.

For this LTI models, a state-space approach is taken instead of a transfer function model to consider the controllability and observability of the plant for a MIMO model [1]. The state equations for these cases are presented in Equations 1 & 2.

(1)

(2)

is the state matrix, **B** is the input matrix, is the output matrix, and is the transmission matrix. To avoid any major complication with the study of the irrigation areas, second-order models were identified from each of four distinct areas in previous studies from Dr. Camilo Lozoya.

These models depend thus on two state variables and two input variables. The state variables are the soil moisture and the rate of change of the moisture with respect to time . The input variables are from the other hand the irrigation level and the evapotranspiration (ETO) of the soil. The ETO is modified by a proportional parameter called , which is denominated the crop constant and is a dimensional constant to regulate every area. These changes from area to area.

In the end, the vector expressions for these variables are displayed in Equations 3 & 4.

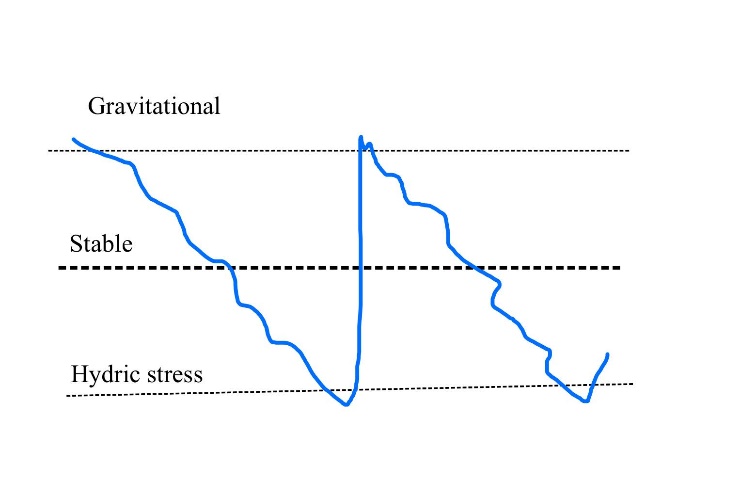
(1)

(2)

The actual state equations must be separated according to three levels. The denominated previous investigations from Dr. Lozoya categorize the advancements of water consumption unto two thresholds. The first threshold separates what is considered the gravitational water, which is the excess water that remains after an irrigation cycle. It can be found at the top of the moisture level diagram.

The next phase is the stable region, which is the most preferrable because it keeps in place the safety of the plant. No hazardous risks relating to water shortage occur at this point. There is a middle bias line that represents the most optimal operating point for the area. This specific offset depends on the irrigation area and most of the times can only be estimated experimentally.

Hydric stress then becomes an urgent problem once the plant leaves the stable region. It is at this point where the lack of water becomes significative to the vegetation and therefore can spawn visible damage sooner or later pertaining to the drought of the crop region. Fig. 1 depicts a crude diagram of these operating zones for a sample model.



**Figure 1**. Levels of soil moisture for the plant.

In total, three state matrices and three input matrices were formed depending on the level of water consumption present. These are presented as Equations 5 to 10 here below.

(5)

(6)

(7)

(8)

(9)

(10)

## Multithreading

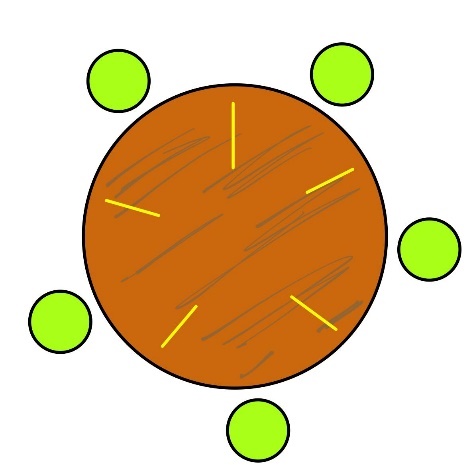
A process thread is a unit of processor utilization that permits the running of a set of operations within the core of the computer. Modern computers permit the capability of running multiple threads in a single processor. This allows the event of running various processes simultaneously.

While this seems to propose the opportunity to run processes in parallel, it does not work like that. Threads are concurrent: they each advance step by step waiting for the others to continue. Because they interrupt each other little by little at the high frequencies of the computer clock, it can be felt that they operate asynchronously.

The act of running different threads in a single core to apparent a parallel process is called multithreading. Because they do not work in parallel, frequently problems arise relating to the synchronization of resources, where threads may quarrel between each other of the utilization of one. Another problem can be if the threads wait each other to finish, creating a cycle that will never end. This event is termed a deadlock in computer science.

### Dining Philosophers Problem

To solve the problem, a mind experiment was performed by computer scientists Edsger W. Dijkstra and Tony Hoare back in the 1960s. Let there be philosophers seated in a round table. Before them come plates of noodles, each for each philosopher [2]. Fig. 2 exemplifies this.



**Figure 2**. Dining philosophers table.

To eat their noodles, they must use a pair of chopsticks. Nonetheless, they find a problem: everyone must share at least one chopstick with their adjacent neighbor. Therefore, one’s right neighbor shares their left chopstick, and the parallel with the left neighbor. Because one must use the whole pair to be able to eat, there comes the question: who eats?

The solution is very simple, although it sounds complex to implement whoever reaches the chopsticks first gets to eat first. All others who did not reach them must wait for the others to eat, hoping not to starve while waiting. When those who are eating finish, they leave the chopsticks back in the table, and let others (whoever reaches them first) to eat. To implement it in code, one must use a concept called a mutex.

A mutex is an abstract object that can be blocked so that if other threads attempt to grab them, they are unable and must wait for the process to release the mutex. Hence, they permit the synchronization of concurrent processes without the fear of robbing the resources at an inappropriate moment.

The operation to collect the mutexes if available is called , because one waits for them to be free. To release them, one simply the computer that they are free once again. The algorithm, short and easy, is shown below.

Text

Description automatically generated

# Experiment

## Program Description

The program can therefore be passed to a real computer language where the practical aspects of coding in the real world are put to the test. The language picked for the task was Python, because it is a very simple yet useful language with a broad range of available libraries for things like numeric operations and multithreading.

As a first job, it is important to define the object to simulate. It is proposed that the irrigation area be defined as a class, where one group several instances to recreate the adjacent groups of irrigation areas. This class must possess various attributes, including the crop constant and threshold levels for simulation.

The methods recommended for the class (besides the constructor) include irrigation dynamics function that returns the next state for the plant according to the input variables and the previous states of the system. According to the soil moisture level, the state equation can take one of the many forms planted in Equations 5 to 10.

When running the simulation, the control action depends on the region where the humidity finds itself, and whether other resources are available. Because all irrigation areas use the same water, this water source will be blocked as a mutex. As a result, when one is in the hydric stress region, it can only be watered if no other area is under irrigation. When it reaches the gravitational water region, it releases the mutex for the rest of the areas to fight for them.

Section 3.2 presents a view into the Python script, including documentation and comments that serve as explanatory aid for the reader.

## Python Code

"""

File name: main.py

Version: 1.0

Author: Leonardo D. Garcia

Date: 12/03/2021

Description:

Algorithmic aspect inspired by the dining philosophers problem from operating systems theory used to generate a concurrent program that will simulate the system dynamics of an irrigation area model considering the sharing of limited resources such as water.

"""

import time

import random

import threading

import numpy as np

import matplotlib.pyplot as plt

from scipy.io import loadmat

class AreaSimulation:

    """

    Class for the simulation of the irrigation area.

    ...

    Attributes

    ----------

    xk\_initial : np.ndarray

        Initial values for the state variables.

    xk\_array : np.ndarray

        Array to collect all measurements in the state variables.

    ir\_array : np.ndarray

        Array for the input variables.

    Kc : float

        Crop constant.

    control\_high : float

        High control level of humidity.

    control\_low : float

        Low control level of humidity.

    irrigation\_value : float

        Irrigation value for the area.

    eto\_array : np.ndarray

        Array to collect ETO information from the plant.

    t\_array : np.ndarray

        Time array.

    Methods

    -------

    irrigation\_dynamics\_function(xk\_initial, Kc, control\_high, control\_low, irrigation\_value)

        Returns the next state for the system.

    run(filename, mutex)

        Runs the simulation to collect information for the plant in the simulation period.

    display(title)

        Display a graph with the system dynamics.

    """

    def \_\_init\_\_(self, xk\_initial: np.ndarray, Kc: float, control\_high: float,

                 control\_low: float, irrigation\_value: float):

        """

        Parameters

        ----------

        xk\_initial : np.ndarray

            Initial values for the state variables.

        Kc : float64

            Crop constant.

        control\_high : float

            High control level of humidity.

        control\_low : float

            Low control level of humidity.

        irrigation\_value : float

            Irrigation value for the area.

        """

        self.xk\_initial = xk\_initial

        self.xk\_array = np.empty(0)

        self.ir\_array = np.empty(0)

        self.Kc = Kc

        self.control\_high = control\_high

        self.control\_low = control\_low

        self.irrigation\_value = irrigation\_value

        self.eto\_array = np.empty(0)

        self.t\_array = np.empty(0)

    def irrigation\_dynamics\_function(self, xk\_initial: np.ndarray, ir\_act: np.ndarray,

                                     eto\_act: float, Kc: float) -> np.ndarray:

        """

        Parameters

        ----------

        xk\_initial : np.ndarray

            Initial values for the state variables.

        ir\_act : np.ndarray

            Values for the input variables.

        eto\_act : float

            Present value of the ETO.

        Kc : float

            Crop constant.

        """

        # State and input matrices for the different levels of earth moisture.

        A\_above = np.array([[0.98501, 0.00001], [0.00001, 0.98501]])

        A\_middle = np.array([[1.00000, 0.00001], [0.00001, 0.98501]])

        A\_below = np.array([[1.00000, 0.00001], [0.00001, 0.98501]])

        B\_above = np.array([[0.00245, -0.00004], [0.00300, -0.00020]])

        B\_middle = np.array([[0.00135, -0.00060], [0.00325, -0.00050]])

        B\_below = np.array([[0.00125, -0.00005], [0.00150, -0.00035]])

        # Not used.

        C = np.array([1, 0])

        # Threshold levels.

        high\_limit = 7.0

        low\_limit = -7.0

        high\_threshold = 5.0

        low\_threshold = -5.0

        # State and input variables.

        xk\_act = xk\_initial

        uk\_act = np.array([ir\_act, Kc\*eto\_act], dtype=object)

        xk\_next = np.empty(2)

        # Control action.

        if xk\_act[0] > high\_threshold:

            xk\_next = np.dot(A\_above, xk\_act) + np.dot(B\_above, uk\_act)

        elif xk\_act[0] > low\_threshold:

            xk\_next = np.dot(A\_middle, xk\_act) + np.dot(B\_middle, uk\_act)

        else:

            xk\_next = np.dot(A\_below, xk\_act) + np.dot(B\_below, uk\_act)

        if xk\_next[0] > high\_limit:

            xk\_next[0] = high\_limit

        elif xk\_next[0] < low\_limit:

            xk\_next[0] = low\_limit

        return xk\_next

    def run (self, filename: str, mutex) -> None:

        """

        Parameters

        ----------

        filename : str

            Path location for the .mat with the ETO information.

        mutex : Lock

            Locks the irrigation.

        """

        # Load file into 'eto\_array' variable.

        data = loadmat(filename)

        self.eto\_array = data['eto\_array']

        simulation\_period = self.eto\_array.size

        self.t\_array = np.arange(simulation\_period)

        # Initialize variables for the simulation.

        self.xk\_array = np.empty([2, simulation\_period])

        self.ir\_array = np.empty([simulation\_period])

        self.ir\_array[0] = 0.0

        # Run over the simulation period.

        for t in range(simulation\_period):

            xk\_next = self.irrigation\_dynamics\_function(self.xk\_initial, self.ir\_array[t],

                                                        self.eto\_array[t], self.Kc)

            self.xk\_initial = xk\_next

            self.xk\_array[0, t] = xk\_next[0]

            self.xk\_array[1, t] = xk\_next[1]

            if t == simulation\_period-1:

                break

            if xk\_next[0] > self.control\_high:

                self.ir\_array[t+1] = 0.0

                if mutex.locked():

                    mutex.release()

            elif xk\_next[0] < self.control\_low and not mutex.locked():

                self.ir\_array[t+1] = self.irrigation\_value

                mutex.acquire()

            else:

                self.ir\_array[t+1] = self.ir\_array[t]

    def display(self, title: str) -> None:

        """

        Parameters

        ----------

        title : str

            Name for the plot title.

        """

        simulation\_period = self.eto\_array.size

        # Generate a plot with the information from the system dynamics.

        plt.title(title)

        plt.plot(self.t\_array, self.xk\_array[0], 'b-',

                 self.t\_array, np.zeros(simulation\_period), 'k--',

                 self.t\_array, self.control\_high\*np.ones(simulation\_period), 'k:',

                 self.t\_array, self.control\_low\*np.ones(simulation\_period), 'k:')

        plt.axis([0, simulation\_period, -8, 8])

        plt.show()

random.seed(time.time())

# Initialize empty array for the four areas.

Areas = list()

# Values for the parameters in all areas.

xk\_initial = [np.array([5.0, 0.0]), np.array([2.0, 0.0]),

               np.array([2.0, 0.0]), np.array([-2.0, 0.0])]

Kc = [1.0, 1.2, 1.4, 0.8]

control\_high = [5.0, 1.0, 2.0, 4.0]

control\_low = [-5.0, -1.0, -2.0, -4.0]

irrigation\_value = [40.0, 45.0, 35.0, 40.0]

# Path location for the ETO .mat file.

filename = r"C:\Users\leodg\Documents\TecMTY\Sem\_9\Research Internship\notebook\DataSet\_ETO\_45\_days.mat"

# Mutex object.

mutex = threading.Lock()

threads = list()

# Initialize, run, and display the results of the individual areas with multithreading and mutexing.

for i in range(4):

    Areas.append(AreaSimulation(xk\_initial=xk\_initial[i], Kc=Kc[i],

                                control\_high=control\_high[i], control\_low=control\_low[i],

                                irrigation\_value=irrigation\_value[i]))

    threads.append(threading.Thread(target=Areas[i].run, args=[filename, mutex]))

# Run the threads.

for t in threads:

    t.start()

for t in threads:

    t.join()

for i, area in enumerate(Areas):

    area.display(f"Area {i+1}")

# Results

After running the simulation, the following four charts (Fig. 3) were delivered by the program. They are for the four areas defined in lines 246 to 268 of the script. Fig. 3a and 3d do not seem off by any means. The swift changes that do look visible in plain eye are in Fig. 3b & 3d.

There are some huge notches where the plants are under some harsh hydric stress states. The reason for this is because, as water becomes utilized by any other plant, they must wait for the resource to be liberated in order to replenish. Additionally, because the areas do not follow any priority queue, it is a tournament to see who reaches first the water.

What the experiment helped demonstrate is that these operating system concepts can be employed in any other area, and one must seize these algorithms to create an opportunity of optimization and synchronization in the daily aspects of the industrial life.

A picture containing diagram

Description automatically generated

**Figure 3a**. Irrigation area (1).

A picture containing shape

Description automatically generated

**Figure 3b**. Irrigation area (2).

Histogram

Description automatically generated

**Figure 3c**. Irrigation area (3).

Diagram

Description automatically generated with low confidence

**Figure 3d**. Irrigation area (4).

# Conclusions

To conclude, the experiment can be considered a huge success in term of adapting a simulation model for a real system that can make use of synchronization techniques of computer science for the applications of automatic control engineering.

The next step in this arduous scientific process delves into the formalization of the experiment using a real-time embedded system architecture in an actual irrigation area. This will help to solidify the validity of the algorithm and discover any other perturbance that may be affecting the plant.

# References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts,* 10th ed. Hoboken, NJ: Wiley, 2018, pp. 293-295.

[2] K. Ogata, *Modern Control Engineering,* 5th ed. Upper Saddle River, NJ: Pearson, 2010, pp. 29-35.