CFD Solution Subsonic-Supersonic Isentropic Nozzle

Diego Pisco Araujo, Sara Ruano Ferrer, Gerard Soldevilla Claramunt
UPC-EETAC, Simulación

CONTENTS

***Abstract*- La dinámica de fluidos siempre ha tenido problemas para determinar lo que sucede en los estados más transitorios de cualquier problema, y las toberas no son una excepción. Para ello, la dinámica de fluidos computacional lo intenta resolver a través del método numérico de MacCormack. El objetivo de este trabajo es simular el estado transitorio de una tobera convergente-divergente para determinar si hay algún valor extremo en sus propiedades físicas. Finalmente, el trabajo también pretende determinar el valor óptimo de la relación entre el área del *reservorio* y el área del cuello.**

Fluid dynamics has always had issues to determine what happens in the non-stationary states of any problem, and nozzles are not an exception. For this purpose, computational fluid dynamics tries to solve it with MacCormack's numerical method. The aim of this project is to simulate the transitory state of a convergent-divergent nozzle in order to determine whether there is an extreme value in its physical properties. Finally, the project also intends to define the optimal value of the reservoir-to-throat area ratio.

I. Description of the problem and its relevance

The intent of this project is to simulate the performance of the physical properties in a Subsonic-Supersonic Isentropic Nozzle flow, also called Convergent-Divergent Isentropic Nozzle flow. In this kind of nozzle two different sections can be distinguished- an area-convergent segment separated from a divergent one by the throat, that is, the position in the nozzle with the smallest cross-section area.

At the beginning of the convergent section of the nozzle there is the reservoir, considered to be the initial point of the simulated flow, and the point where the initial parameters of the problem- pressure, temperature, density, and velocity- are defined.

Throughout the development of the problem, a quasi-one-dimensional nozzle will be assumed, that means that the variation of these initially defined properties is treated as one-dimensional along the nozzle. Despite that, the longitudinal variation of the cross-section area of the nozzle will be considered as a tridimensional effect, and thus, it should be added as another variable to the problem.

Our aim is to determine pressure, temperature, density, and velocity of the gases along the nozzle from an initial point when the flow enters the nozzle- like the one in Figure 1- until a stationary regime has been reached. Even though the stationary regime has an easily computed solution, the resolution of the non-stationary problem- the purpose of the simulation- needs to be performed with a numerical analysis.

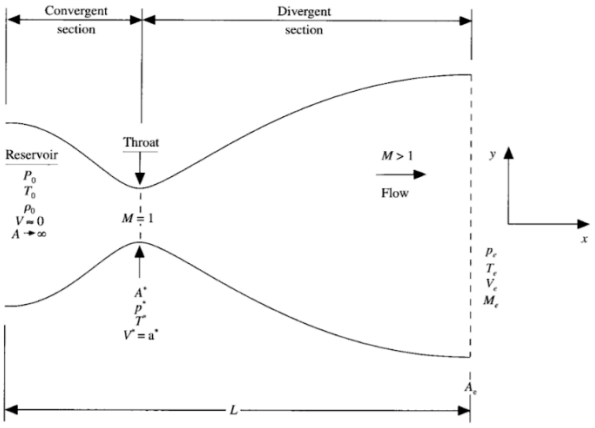

Figure 1: Problem outline

The relevance of this problem remains in the knowledge of the values of these properties on the non-steady regime of the problem. With it, we will be able to determine if some extremes can be found, and act accordingly when the materials of said nozzle are chosen, so as to avoid any dangerous degradation of them.

II. Mathematical aspects

At this point, computational fluid dynamics (CFD) is introduced in order to solve the non-stationary regime of this problem. With it, the nozzle domain is discretized, and all the fluid properties are issued on a discrete finite grid collection of points temporally and spatially separated as finite differences of time (Δt) and space (Δx). With this scenario, a set of values for all the variables will be computed along the whole domain of time and space.

These properties, as usually happens in fluid computations, are expressed in terms of dimensionless variables for them to be easily interpreted at the end of the simulation. This way, pressure, temperature, and density will be nondimensionalized using their values in the reservoir; the cross-section area will be divided by its value on the nozzle throat, the velocity by the speed of sound in the

reservoir, space by one third of the nozzle length, and time by the product of the formerly mentioned speed of sound and nozzle length.

The discrete values of these nondimensionalized properties will be computed using MacCormack's technique, an accurate second-order numerical method both in space and time. Using this method, the initial and boundary values will need to be known in order to be able to compute each and every one of the parameters along the grid for each step forward in time.

The nozzle will be considered to have a shape specified in the expression $A(x) = 1 + 2.2(1 - 1.5)^2 \, , for \; 0 \leq x \leq 3$. This expression defines the nozzle throat to be in the middle of it. In case we wanted to modify the position of this throat, the value to be modified would have to be the "1.5" term.

The initial conditions for the remaining properties are expressed in the following equations:

$$\rho(x) = 1 - 0.3146x \quad ; \quad T(x) = 1 - 0.2314x \quad ; \quad V(x) = (0.1 + 1.09x)\sqrt{T(x)}$$

The equations that will need to be discretized are the governing flow equations of the problem. Initially, five equations are defined- the continuity equation, the momentum equation, the energy equation, the equation of state and the expression of a calorically perfect gas. The former two equations are combined with the momentum equation and the energy equation to simplify the problem into a 3-equaton system:

$$\frac{\partial \rho'}{\partial t'} = -\rho' \frac{\partial V'}{\partial x'} - \rho' V' \frac{\partial (\ln A')}{\partial x'} - V' \frac{\partial \rho'}{\partial x'} \quad ; \quad \frac{\partial V'}{\partial t'} = -V' \frac{\partial V'}{\partial x'} - \frac{1}{\gamma}\left(\frac{\partial T'}{\partial x'} + \frac{T'}{\rho'}\frac{\partial \rho'}{\partial x'}\right)$$

$$\frac{\partial T'}{\partial t'} = -V' \frac{\partial T'}{\partial x'} - (\gamma - 1)T'\left[\frac{\partial V'}{\partial x'} + V' \frac{\partial (\ln A')}{\partial x'}\right]$$

Pressure will be computed with the values of density and temperature found in each time-step using the non-dimensionalized equation of state:

$$p' = T'\rho'$$

Note that these variables are already non-dimensionalized. In the following computations all variables will be non-dimensionalized.

These equations will be the ones discretized to find how density, velocity and temperature vary in time for every position of our discrete grid. To obtain a first solution of the parameters at the next time-step, we will establish the spatial derivatives as forward differences:

$$\left(\frac{\partial \rho}{\partial t}\right)_i^t = -\rho_i^t \frac{V_{i+1}^t - V_i^t}{\Delta x} - \rho_i^t V_i^t \frac{\ln A_{i+1} - \ln A_i}{\Delta x} - V_i^t \frac{\rho_{i+1}^t - \rho_i^t}{\Delta x}$$

$$\left(\frac{\partial V}{\partial t}\right)_i^t = -V_i^t \frac{V_{i+1}^t - V_i^t}{\Delta x} - \frac{1}{\gamma}\left(\frac{T_{i+1}^t - T_i^t}{\Delta x} + \frac{T_i^t}{\rho_i^t}\frac{\rho_{i+1}^t - \rho_i^t}{\Delta x}\right)$$

$$\left(\frac{\partial T}{\partial t}\right)_i^t = -V_i^t \frac{T_{i+1}^t - T_i^t}{\Delta x} - (\gamma - 1)T_i^t\left[\frac{V_{i+1}^t - V_i^t}{\Delta x} + V_i^t \frac{\ln A_{i+1} - \ln A_i}{\Delta x}\right]$$

Using these time differences, a first set of predicted values will be computed:

$$\bar{\rho}_i^{t+\Delta t} = \rho_i^t + \left(\frac{\partial \rho}{\partial t}\right)_i^t \Delta t \quad ; \quad \bar{V}_i^{t+\Delta t} = V_i^t + \left(\frac{\partial V}{\partial t}\right)_i^t \Delta t \quad ; \quad \bar{T}_i^{t+\Delta t} = T_i^t + \left(\frac{\partial T}{\partial t}\right)_i^t \Delta t$$

The second step of this method consists in performing the same equations using rearward differences and these predicted values:

$$\left(\overline{\frac{\partial \rho}{\partial t}}\right)_i^{t+\Delta t} = -\bar{\rho}_i^{t+\Delta t} \frac{\bar{V}_i^{t+\Delta t} - \bar{V}_{i-1}^{t+\Delta t}}{\Delta x} - \bar{\rho}_i^{t+\Delta t} \bar{V}_i^{t+\Delta t} \frac{\ln A_i - \ln A_{i-1}}{\Delta x} - \bar{V}_i^{t+\Delta t} \frac{\bar{\rho}_i^{t+\Delta t} - \bar{\rho}_{i-1}^{t+\Delta t}}{\Delta x}$$

$$\left(\overline{\frac{\partial V}{\partial t}}\right)_i^{t+\Delta t} = -\bar{V}_i^{t+\Delta t} \frac{\bar{V}_i^{t+\Delta t} - \bar{V}_{i-1}^{t+\Delta t}}{\Delta x} - \frac{1}{\gamma}\left(\frac{\bar{T}_i^{t+\Delta t} - \bar{T}_{i-1}^{t+\Delta t}}{\Delta x} + \frac{\bar{T}_i^{t+\Delta t}}{\bar{\rho}_i^{t+\Delta t}} \frac{\bar{\rho}_i^{t+\Delta t} - \bar{\rho}_{i-1}^{t+\Delta t}}{\Delta x}\right)$$

$$\left(\overline{\frac{\partial T}{\partial t}}\right)_i^{t+\Delta t} = -\bar{V}_i^{t+\Delta t} \frac{\bar{T}_i^{t+\Delta t} - \bar{T}_{i-1}^{t+\Delta t}}{\Delta x} - (\gamma - 1)\bar{T}_i^{t+\Delta t}\left[\frac{\bar{V}_i^{t+\Delta t} - \bar{V}_{i-1}^{t+\Delta t}}{\Delta x} + \bar{V}_i^{t+\Delta t} \frac{\ln A_i - \ln A_{i-1}}{\Delta x}\right]$$

Finally, using the computed time differences- both forward and rearward- a definitive time difference is computed as an average of these two.

$$\left(\frac{\partial \rho}{\partial t}\right)_{i\,av}^t = 0.5\left(\left(\frac{\partial \rho}{\partial t}\right)_i^t + \left(\overline{\frac{\partial \rho}{\partial t}}\right)_i^{t+\Delta t}\right) \quad ; \quad \left(\frac{\partial V}{\partial t}\right)_{i\,av}^t = 0.5\left(\left(\frac{\partial V}{\partial t}\right)_i^t + \left(\overline{\frac{\partial V}{\partial t}}\right)_i^{t+\Delta t}\right)$$

$$\left(\frac{\partial T}{\partial t}\right)_{i\,av}^{t} = 0.5\left(\left(\frac{\partial T}{\partial t}\right)_{i}^{t} + \left(\frac{\overline{\partial T}}{\partial t}\right)_{i}^{t+\Delta t}\right)$$

And with these final time differences, the final values of the parameters can be computed along the grid for this time-step.

$$\rho_i^{t+\Delta t} = \rho_i^t + \left(\frac{\partial \rho}{\partial t}\right)_{i\,av}^{t}\Delta t \quad ; \quad V_i^{t+\Delta t} = V_i^t + \left(\frac{\partial V}{\partial t}\right)_{i\,av}^{t}\Delta t \quad ; \quad T_i^{t+\Delta t} = T_i^t + \left(\frac{\partial T}{\partial t}\right)_{i\,av}^{t}\Delta t$$

Note that for each time-step, the magnitude of this time-step must be computed using Courant's condition. This condition states the following:

$$\Delta t_i^t = C\frac{\Delta x}{a_i^t + V_i^t}$$

C, the Courant number, must be smaller than 1 for the computation to remain stable, for this problem a value of 0.5 will be used. This condition must be met at each position of the nozzle. Therefore, for each time-step, this value must be computed for all positions and the minimum of these values must be chosen.

For each time-step, in the subsonic inflow boundary, density and temperature are the same value as the ones in the reservoir, so their respective nondimensional values will be the unity 1. As for the velocity, it must be computed considering a constant slope with the following two positions of the grid. In the supersonic outflow boundary, density, velocity, and temperature must be computed considering a constant slope with the last two positions of the grid.

The assumption that in the inflow boundary, temperature, density, and pressure are the same as in the reservoir is only correct if the Mach number is 0. This is not the case, but the difference in the results is negligible. Besides, the discretization of the nozzle into a grid of 31 positions may vary the steady analytical results. Therefore, at the end of the simulation, when the nozzle reaches the steady solution, these results will be compared, and a percentage of error will be computed.

III. Code Organization and User's Interface

In our visual studio file, we have a source code that provides the definition and implementation of classes, structures, methods, and interfaces, which we named *NozzleLib*. Additionally, we added an assembly (.NET Framework) that allows us the development of interaction interfaces for doing our simulation, which we named *SimuladorNozzle*.

A. Classes

Inside the *NozzleLib* file we have two different classes:

Position Class

This class contains the definition of all the attributes of position, temperature, density, velocity, pressure, and area of a nozzle-division. Then, the class has a constructor setting the values and different kinds of methods which allow to extract or set the attributes' values. Afterwards, there is a Mathematic-methods section, with methods that compute the natural logarithm, the speed of sound, delta time and Mach number.

The attributes and methods in this class are listed in Table 1.

Table 1: Attributes and methods for Position class

	Types	**Name**	**Meaning**
Attributes	double	x	position in x coordinates
	double	T	temperature
	double	ro	density
	double	V	velocity
	double	p	pressure
	double	A	area
	int	i	divisions of the nozzle
	double	M	Mach number
	double	R	gas constant

	Types	Arguments	Meaning
Constructors	Position	5	help us to define the position class and set default values for our simulation.
	Position	6	

	Types	Name	Meaning
Math Methods	double	LnA	calculate natural logarithm
	double	Speedofsound	calculate the speed of sound
	double	Deltatime	calculate the variation of delta time, with the input of Courant value and delta position x
	double	MachNumber	calculate de number of Mach
Functions (Variable Extraction)	double	GetX	returns the value x
	double	GetTemperature	returns the value T
	double	GetDensity	returns the value ro
	double	GetVelocity	returns the value V
	double	GetPressure	returns the value p
	double	GetArea	returns the value A
Functions (Variable Setting)	void	SetX	set the value x
	void	SetTemperature	set the value T
	void	SetDensity	set the value D
	void	SetVelocity	set the value V
	void	SetPressure	set the value P
	void	SetA	set the value A

Nozzle Class

This class contains all the position classes with their attributes of each of the divisions in the nozzle and some parameters regarding the simulation, such as the courant number. It also computes how these values change with time.

First, all the attributes, parameters and nozzle are defined. These parameters are gamma, the gas constant, the number of spatial divisions of the nozzle, delta time, delta x, throat position and Courant value. The most important property created in this section is the matrix containing the time steps in rows and space divisions in columns.

Then, there is a constructor which defines the characteristics of the nozzle (length, temperature, density, Courant, divisions). After that, we have some functions obtaining the values of the matrix, for instance, a certain position or a certain row containing the nozzle in a particular time-step. After that, a function is defined to compute delta time at a specific time using the courant condition and ensuring it is met in every position of the nozzle.

Finally, the main method of this class is defined as *ComputeNextTime* and it is intended to compute the nozzle properties at each position after a certain delta time. For that, it takes a list of each property- temperature, velocity, pressure, and density- at each position and it computes the difference of each of them using the MacCormack technique described in the previous chapter. In the end, it creates a new row of position classes in the matrix of the nozzle.

The attributes and methods in this class are listed in Table 2.

Table 2: Attributes and Methods for Nozzle class.

	Types	Name	Meaning
Attributes	double	gamma	gamma constant

	double	R	gas constant
	double	deltax	spatial increment
	double	deltatime	temporary increase
	List<double>	TimeList	list of time which its first value is a zero
	double	C	Courant value
	Position	malla	matrix whose rows are time steps and columns are the space divisions
	int	N	number of space divisions
	double	dimensionalvalues	vector to save the dimensional values of L, T0, a0, p0, ro0
	Position	dimensionalPos	to save the new position(class) for dimensional values
	double	throatposition	position of the throat

	Types	Arguments	Meaning
Constructors	Nozzle	6	help us to define the Nozzle class and set default values for our simulation.
	Nozzle	7	
	Nozzle	0	

	Types	Name	Meaning
Functions (Extraction and setting)	Position	getPosition	returns the position for an specific row and column
	int	getDivisions	returns the value of N
	void	setPosition	set a specific Position in an indicated row and column inside the matrix (malla)
	int	getN	returns the value of N
	double	getCourant	returns the Courant value
	List<double>	getTimeList	returns the list of TimeList
	double	getDimensionalValues	returns a vector of the dimensional values
	Position	getDimensionalPosition	returns the position(class) for dimensional values
	Position	getmalla	returns the matrix (malla) of Position
Other functions	List<double>	getTimeList	Returns a list of time steps from an initial step to a final step, as indicated in the input.
	List<double>	createListArea	Returns a list of all the areas for the same time step and for all the spatial divisions in the nozzle
	List<double>	createListDensity	Returns a list of all the densities for the same time step and for all the spatial divisions of the nozzle
	List<double>	createListVelocity	Returns a list of all the velocities for the same time step and for all the spatial divisions of the nozzle
	List<double>	createListTemperature	Returns a list of all the temperatures for the same time step and for all the spatial divisions of the nozzle

	void	SetDimensionalValues	This function set the initial values to obtain dimensional values for these magnitudes [L, T0, a0, p0, ro0]
	List<Position>	GetRow	Returns a list for all the attributes (that are set in the class Position) for a specific row.
	List<Position>	GetColumn	Returns a list for all the attributes (that are set in the class Position) for a specific column.
	List<double>	GetRowPar	Returns a list for the attribute that you specified in the input for a specific row.
	List<double>	GetColumnPar	Returns a list for the attribute that you specified in the input for an specific column, this list has a specific length between (step and the final time step that you indicate in the input)
	double	ComputeDeltaTime	Returns the maximum value of the delta time for a specific time step.
	void	SetDeltaTime	Set the value of a Delta Time in our attributes
	void	ComputeNextTime()	Calculates the next step of the simulation, obtains the new values of the position class (as temperature, density, speed, and area) and modifies them. To find these new values we use MacCormack's formulas.
	void	ComputeUntilPos	Calculates all the next times until the last position (in our case is after 1401 step)
	void	SetNewArea	set the value of the Area, by changing in the attributes.

B. Simulator

In order to build this simulator, a WPF project (windows presentation foundation) is created. It uses the XAML language to provide a declarative model for application programming. This WPF platform supports a rich set of application development features, including an application model, resources, controls, graphics, and layouts. Once both classes have been detailed, it is time for the simulation implementation using those classes.

First an object of the class Nozzle is defined, the parameters of which could be partially decided by the user, as it will be see further on. Once the object nozzle is created, the simulation begins and all variables' values on both spatial and temporal domains are computed. An integer value regarding the time position of the simulation changes as does time in the simulation; it also shows the time step the simulation is currently on.

When the simulator is initiated, the appearance of the simulator is incomplete, this is because some meaningless information or other constructions are hidden until the simulation begins. There are other features, such as buttons, which are always visible; even so, some transparent rectangles are distributed along all the window in order to cover some of these parts, so the user will not be able to click a button when it is not necessary. Once a certain feature is available, the rectangle covering it disappears to enable the user to make use of it. This system ensures that buttons where the cursor appearance is a hand will provide correct actions. Additionally, in the textboxes where the user has to introduce some value, the simulator checks whether the information is incorrect and displays some helpful information for assisting the user in solving the error.

Also, there is a timer that will be initiated at the beginning of the simulation, which the auto button does not stop or initiate, in order to be able to measure intervals of time during all the simulation. This is useful in cases like blinking labels or waiting times so as not to saturate the simulation in cases of high workload like the function SetChart(), explained later.

In order to organize the simulator's design, the tab control tool is used. This tab control- as seen in Figure 2- was divided into three different tabs- the simulator, the report, and the Anderson tables, clicking on each of them, the users can switch between them.

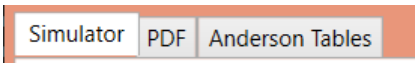

Figure 2: Simulator's tabs

Simulation Tab

The simulation tab is the most extended and complex part of the code. Regarding this Tab, it is divided into three stack panels. The first one- on the left- contains a section to define the initial settings- like the one in Figure 3- and at the bottom there is a table that is hidden or visible depending on the user actions, the initial settings allows to modify the divisions of the nozzle and the Courant value, needed to compute the delta time.

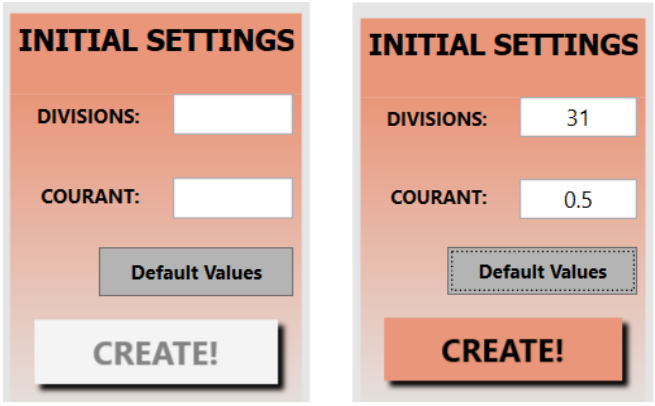

Figure 3: Initial Settings interface.

The second stack panel -in the middle- includes the nozzle visualization where all the space divisions of the nozzle are shown for a specific simulation time. The simulation carries on in a simulator like the one in Figure 4.

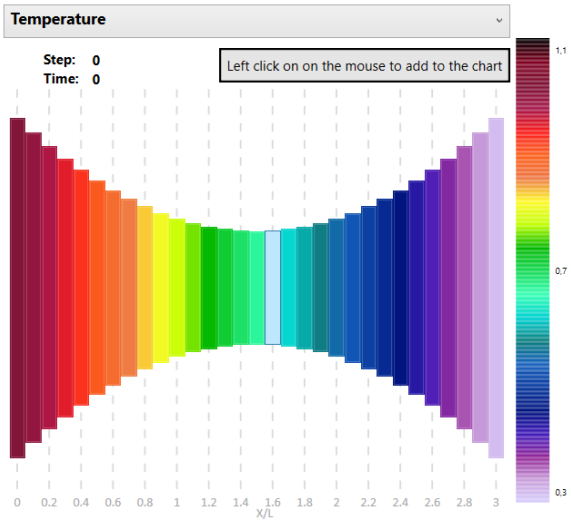

Figure 4: Initial conditions of temperature in a nozzle simulated in the simulator.

This part is distributed with a grid containing the color indicator and a canvas, in which the divisions of the nozzle are painted according to the values of the selected property. Above the canvas, there is a combobox enabling to select the desired property to simulate. In the right part of this grid, there is the color indicator which allows the user to know which is the color equivalence. The colors chosen in order to depict the nozzle properties are intuitive for the user to be able to tell whether a certain value is high or low.

If suddenly the user moves the cursor over some of the divisions of the nozzle, a table on the first column, on the left, appears on the bottom of this section, which can be fixed with a right click. This table is like the one depicted in Figure 5.

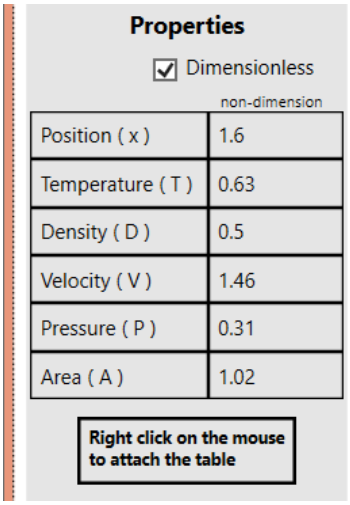

Figure 5: Properties' values table when the 1,6 m position is selected.

The simulator controls will be allocated in the inferior part of this second stack panel. These buttons will let the user control how the simulation carries on. Particularly, the user will be able to advance the simulation step-by-step or automatically. If this former option is selected, it is also possible to control the simulation velocity with a slider bar. The third button allows the user to restart the simulation, while the last two buttons download and load a certain simulation. The controls are distributed like the ones depicted in Figure 6.

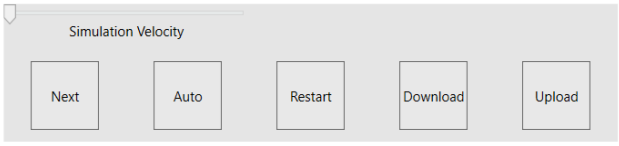

Figure 6: Simulation controls

Afterwards, on the right stackPannel, the charts representing the evolution of velocity, temperature, density, and pressure along the time in a certain position of the nozzle, are found. The extension used for this purpose is LiveCharts.wpf. On the top of this section there is a matrix of buttons -with the corresponding position of the nozzle written- that allows the user to select one specific position of the nozzle to be plotted. When the user clicks some button, a line appears below this button, the color of which is used as a legend since this position will be plotted on the charts with it. Some controls are available, such as maximizing one specific chart or showing dimensions of the values. The resulting charts look like the ones in Figure 7.

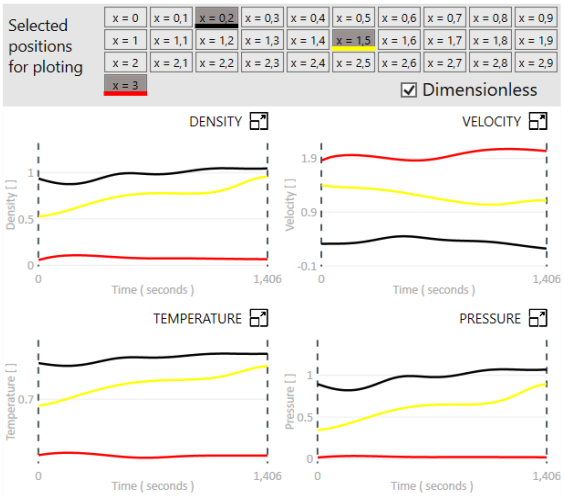

Figure 7: Charts of a simulation when 0.2m, 1.5m and 3m positions are selected to be plotted.

Finally, below the charts there will be the advanced study controls, which will be explained later on.

The complete design of the simulation tab is represented in Figure 8.

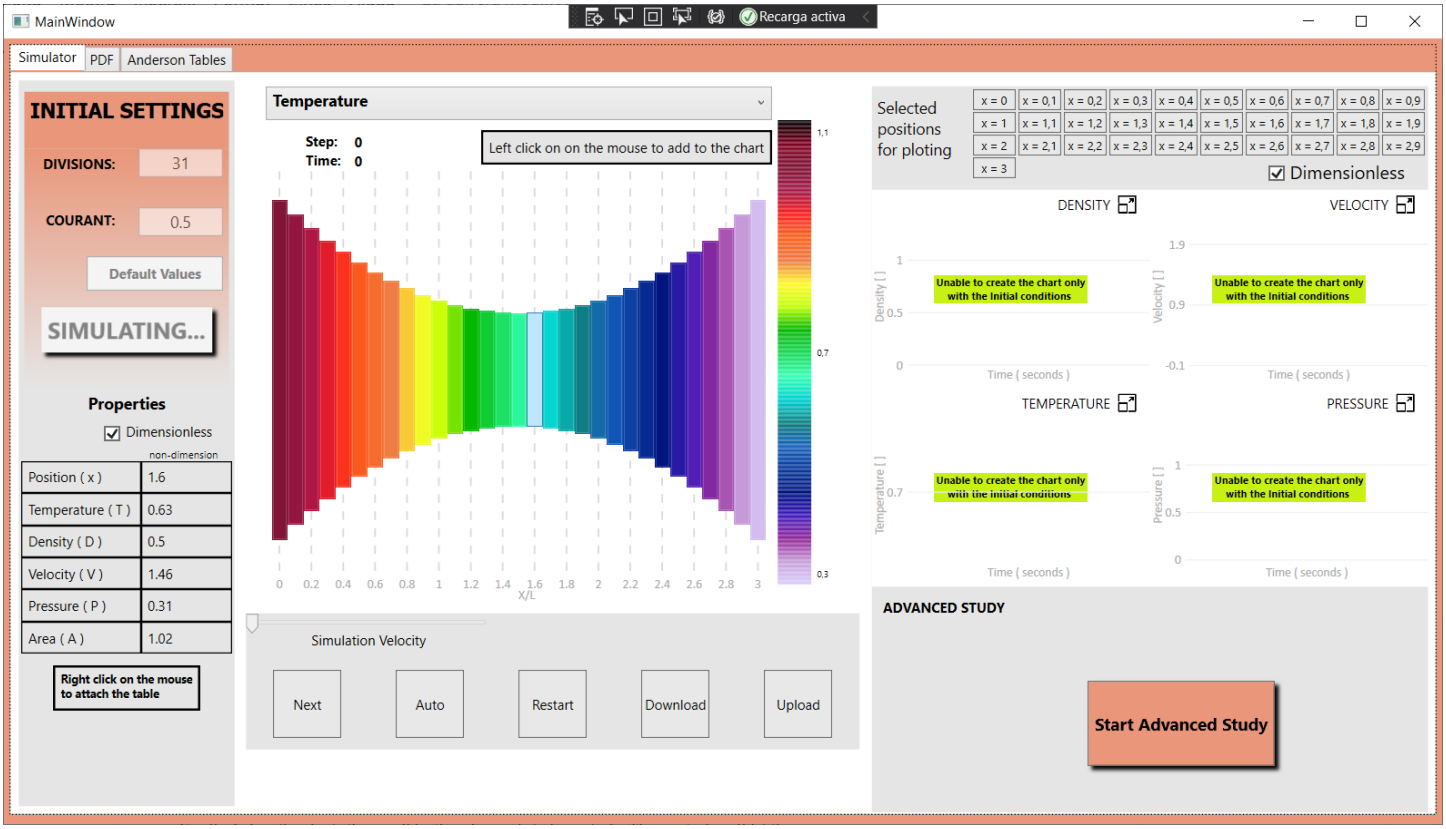

Figure 8: Complete design of the simulator tab.

Example of complex and well-documented function *(The annotation '[number]' is referred to a specific line of the code in the annex)*

An example of a well-documented function and one of the most complex in the simulation is the function SetChart(). This function is called for uploading the chart information. It will create four lists of lists of decimal values [5-8] (List<List<double>>) referred to each property, and then, it will plot this collection of values on each chart. Each component of each list of lists is referred to each position of the nozzle's divisions. So, each component will depict the value of some property on a specific position of the nozzle along the time, from the beginning until the current simulation time.

Representing a plot on LiveCharts is not automatic, its computational cost is not low, and for that reason, the charts cannot instantaneously plot if the simulation is running fast and there are many positions to compute. And so first, in order to reduce computational time, some values of those lists are dismissed using the following rule: the higher the simulation time is, the more values are dismissed [9-24]. For example, if we have 20 Temperature values for a specific position, we will plot all of them, but if we are on step 1200, for each value we plot, 40 values will be excluded and missed (the variable stepsChart is set as 40). In any case the results will be apparently the same.

After explaining some procedures of the function, some information that SetChart is based on must be kept in mind. First, there is a list of Brushes (List<Brush> brushesList), which is a group of ten colors that used to plot the charts. Secondly there is a List of integer numbers (List<int> brushesPos) that saves the information of which positions are selected for plotting and which color are referred to. This list is initially set with '-1' on all its positions, which means that the corresponding position is not selected for plotting, any other value would mean that the corresponding position is selected for plotting and its value refers to the color that represents it on brushesList. So, referring to SetCharts() function, a new list of Brushes will be created (List<Brush> ListBrush [27]) and it has the same length as divisions. Now, each position of brushesPos will be checked [40] and if some position is selected for plotting (different to '-1'), one new List of values is added on the list of lists of values [45-48] referred to each property, with the function GetColumnPar considering the finStep. Additionally, on ListBrush, the Brush recorded on brushesList corresponding to the desired position is added [49].

Once all positions are checked, the lines below each button referred to each position are painted [63]. Now all the information is ready to be plotted, but it is still not so simple. As mentioned, the plotting of the charts is not automatic, when the chart is updated it disappears and then the new plot is lifted from below to its position. It takes some fractions of second or even more than a second in extreme cases. So, if the simulation were too fast no stable plots would be displayed, that is, they would be removed so fast that the user could not see anything useful. To solve that, the function will not update the information of the charts all the times it is called. It

would update the information only after 1 or 2 seconds [71] of the last update, depending on the number of positions selected for plotting [64-70].

There is also another condition [71], there is a boolean that is changed to true when charts need to be updated because setChart is called on every interval of the clock changes, but that does not mean that the simulation is running. Consequently, if all conditions are met, the charts are updated, first we create an array of time values for the horizontal axis [76-83] and then all four charts are updated with the function createChart called four times [84-87] with different parameters. This function creates as many series as divisions [107-118], those collections corresponding to the positions desired to plot have some Brush color contained on ListBrush, and those not selected for plotting have an empty list of values so no value will be represented for those positions.

PDF Tab

This tab will enable the user to read the report of this project and understand how the simulator works.

Anderson tables tab

Finally, in the Anderson tab will let the user compare the current property values of each of the nozzle positions in the simulation with the ones in the Anderson tables. In the first step of the simulation, the Anderson table displayed will be Table 7.2, whereas the rest of the steps will be compared with Table 7.3 related to the 1400^{th} step.

IV. CORRECT RESULTS PROOF

Before building the simulator, some proof that the results obtained by the code computations are the correct ones must be obtained. The reference in this project to compare the results is J. D. Anderson's Computational Fluid Dynamics, the basics with applications. This book contains two table with results to compare- flow-field variables after the first time step and after 1400 time steps.

The following table describes the results obtained in the computations and the error they have compared to the results in the Anderson book.

Table 3: First step computations and errors

I	x/L	A/A*	ρ/ρ_0	e(ρ)	V/a_0	e(V)	T/T_0	e(T)	p/p_0	M
1	0,000	5,950	1,000000	0,000%	0,111484	0,436%	1	0,000%	1	0,111484
2	0,100	5,312	0,954898	-0,011%	0,211579	-0,199%	0,971757	-0,025%	0,927929	0,214632
3	0,200	4,718	0,926919	-0,009%	0,311675	-0,104%	0,950209	0,022%	0,880767	0,319737
4	0,300	4,168	0,899521	-0,053%	0,411231	0,056%	0,928904	-0,010%	0,835569	0,426678
5	0,400	3,662	0,872052	0,006%	0,508275	0,054%	0,907588	-0,045%	0,791464	0,533525
6	0,500	3,200	0,844472	0,056%	0,602756	-0,040%	0,886244	0,028%	0,748408	0,640273
7	0,600	2,782	0,816717	-0,035%	0,694623	-0,054%	0,864846	-0,018%	0,706334	0,74693
8	0,700	2,408	0,788688	-0,040%	0,783824	-0,022%	0,843354	0,042%	0,665143	0,85352
9	0,800	2,078	0,760241	0,032%	0,870304	0,035%	0,821706	-0,036%	0,624695	0,960092
10	0,900	1,792	0,731172	0,024%	0,954008	0,001%	0,799808	-0,024%	0,584797	1,066741
11	1,000	1,550	0,701212	0,030%	1,034876	-0,012%	0,777537	-0,060%	0,545218	1,173621
12	1,100	1,352	0,670050	0,007%	1,112843	-0,014%	0,754737	-0,035%	0,505712	1,280962
13	1,200	1,198	0,637402	0,063%	1,187826	-0,015%	0,731253	0,035%	0,466102	1,389054
14	1,300	1,088	0,603141	0,023%	1,259719	-0,022%	0,706986	-0,002%	0,426412	1,498195
15	1,400	1,022	0,567462	0,081%	1,328387	0,029%	0,681971	-0,004%	0,386993	1,608576
16	1,500	1,000	0,530957	-0,008%	1,393685	-0,023%	0,656435	0,066%	0,348539	1,72016
17	1,600	1,022	0,494491	0,099%	1,455483	0,033%	0,630762	-0,038%	0,311906	1,832628
18	1,700	1,088	0,458903	-0,021%	1,513689	-0,021%	0,605366	0,060%	0,277804	1,945484
19	1,800	1,198	0,424711	-0,068%	1,568235	0,015%	0,580546	-0,078%	0,246564	2,058225
20	1,900	1,352	0,392027	0,007%	1,619045	0,003%	0,556414	0,074%	0,218129	2,170501
21	2,000	1,550	0,360658	-0,095%	1,666017	0,001%	0,532925	-0,014%	0,192204	2,282162
22	2,100	1,792	0,330277	0,084%	1,709016	0,001%	0,509951	-0,010%	0,168425	2,393216
23	2,200	2,078	0,300551	-0,149%	1,747886	-0,007%	0,487344	0,071%	0,146472	2,503775
24	2,300	2,408	0,271206	0,076%	1,782449	0,025%	0,464974	-0,006%	0,126104	2,613983
25	2,400	2,782	0,242038	0,016%	1,81252	-0,026%	0,442741	-0,059%	0,10716	2,724006
26	2,500	3,200	0,212904	-0,045%	1,837904	-0,005%	0,420572	-0,102%	0,089542	2,834019
27	2,600	3,662	0,183709	-0,158%	1,858408	0,022%	0,398417	0,105%	0,073193	2,944232
28	2,700	4,168	0,154389	0,253%	1,873863	-0,007%	0,376242	0,064%	0,058088	3,05495
29	2,800	4,718	0,124902	-0,079%	1,884182	0,010%	0,354023	0,006%	0,044218	3,166703
30	2,900	5,312	0,095217	0,229%	1,889555	-0,024%	0,331739	-0,079%	0,031587	3,28066
31	3,000	5,950	0,065533	-0,708%	1,894928	-0,004%	0,309456	0,147%	0,02028	3,406384

Table 4: 1400th step results and errors

I	x/L	A/A*	ρ/ρ_0	e(ρ)	V/a_0	e(V)	T/T_0	e(T)	p/p_0	M
1	0,000	5,950	1,000000	0,000%	0,099144	0,145%	1	0,000%	1	0,099144
2	0,100	5,312	0,997534	-0,047%	0,112141	0,126%	0,999008	0,001%	0,996544	0,112196
3	0,200	4,718	0,996979	-0,002%	0,125138	0,110%	0,99879	-0,021%	0,995773	0,125214
4	0,300	4,168	0,994093	0,009%	0,142659	-0,239%	0,99763	-0,037%	0,991737	0,142828
5	0,400	3,662	0,991548	-0,046%	0,162225	0,139%	0,99661	-0,039%	0,988186	0,1625
6	0,500	3,200	0,987190	0,019%	0,186569	-0,231%	0,994855	-0,015%	0,982111	0,18705
7	0,600	2,782	0,981697	-0,031%	0,215423	0,197%	0,99264	-0,036%	0,974472	0,21622
8	0,700	2,408	0,973650	-0,036%	0,250768	-0,092%	0,989379	0,038%	0,963309	0,252111
9	0,800	2,078	0,962500	-0,052%	0,293558	-0,150%	0,984839	-0,016%	0,947908	0,295809
10	0,900	1,792	0,946540	-0,049%	0,345774	-0,065%	0,978285	0,029%	0,925986	0,349591
11	1,000	1,550	0,924014	0,001%	0,409007	0,002%	0,968929	-0,007%	0,895303	0,415513
12	1,100	1,352	0,892380	0,043%	0,485017	0,004%	0,955566	-0,045%	0,852728	0,496166
13	1,200	1,198	0,849068	0,008%	0,57475	-0,043%	0,936819	-0,019%	0,795423	0,593815
14	1,300	1,088	0,791968	-0,004%	0,677987	-0,002%	0,911224	0,025%	0,72166	0,710246
15	1,400	1,022	0,720879	-0,017%	0,792503	-0,063%	0,877764	-0,027%	0,632762	0,845887
16	1,500	1,000	0,638606	-0,062%	0,913983	-0,002%	0,836416	0,050%	0,53414	0,999372
17	1,600	1,022	0,550955	-0,008%	1,036626	-0,036%	0,788563	-0,055%	0,434463	1,167358
18	1,700	1,088	0,464924	-0,016%	1,15458	-0,036%	0,736765	-0,032%	0,34254	1,345116
19	1,800	1,198	0,386303	0,079%	1,263376	0,030%	0,683987	-0,002%	0,264227	1,527597
20	1,900	1,352	0,318339	0,106%	1,360554	-0,033%	0,6328	-0,032%	0,201445	1,71034
21	2,000	1,550	0,261803	-0,075%	1,445546	-0,031%	0,584919	-0,014%	0,153133	1,890096
22	2,100	1,792	0,215897	-0,048%	1,518928	-0,005%	0,541238	0,044%	0,116852	2,064634
23	2,200	2,078	0,179050	0,028%	1,582036	0,002%	0,501966	-0,007%	0,089877	2,232951
24	2,300	2,408	0,149635	-0,244%	1,636144	0,009%	0,466996	-0,001%	0,069879	2,394225
25	2,400	2,782	0,126083	0,066%	1,682906	-0,006%	0,4359	-0,023%	0,05496	2,548981
26	2,500	3,200	0,107196	0,184%	1,72307	0,004%	0,408378	0,093%	0,043777	2,696323
27	2,600	3,662	0,091851	-0,162%	1,758519	-0,027%	0,383782	-0,057%	0,035251	2,838604
28	2,700	4,168	0,079466	0,589%	1,788643	-0,020%	0,362086	0,024%	0,028773	2,972471
29	2,800	4,718	0,069038	0,056%	1,816674	-0,018%	0,342266	0,078%	0,023629	3,105242
30	2,900	5,312	0,060944	-0,091%	1,839206	0,011%	0,325262	0,080%	0,019823	3,224883
31	3,000	5,950	0,052850	-0,282%	1,861737	-0,014%	0,308258	0,084%	0,016292	3,353216

It is clear that all computations are correct since errors are less than 1% in all parameters for all positions. This minimal error is due to the 3-decimal rounding in the Anderson book.

These calculations can be proved in the simulator in the Anderson tables' tab. This tab contains the comparison between the results in the simulator and the Anderson's tables in both cases. Moreover, values can be checked in the simulator itself- when the mouse is hovered over a certain cell, its properties' values appear in the bottom left part of the simulator.

V. ADVANCED STUDY

Once the nozzle has been simulated and all the values have been proven, it is time for the simulator to do something more. The purpose of this section is to go a little bit further into the study of the non-steady solution of a nozzle. In order to do this, the simulator includes the possibility of changing the reservoir-to-throat area ratio.

This functionality is located in the bottom-left part of the simulator, just below the charts. When the user wants to access it, the Start Advanced Study button must be clicked. If so, a new interface will appear, enabling the user to input the new reservoir-to-throat ratio and check the comparison with the default value graphically. This interface has the appearance like the one depicted in Figure 9. Then, the simulation button below must be clicked to start the new simulation with the new reservoir-to-throat value. The simulation will run the same as the default one.

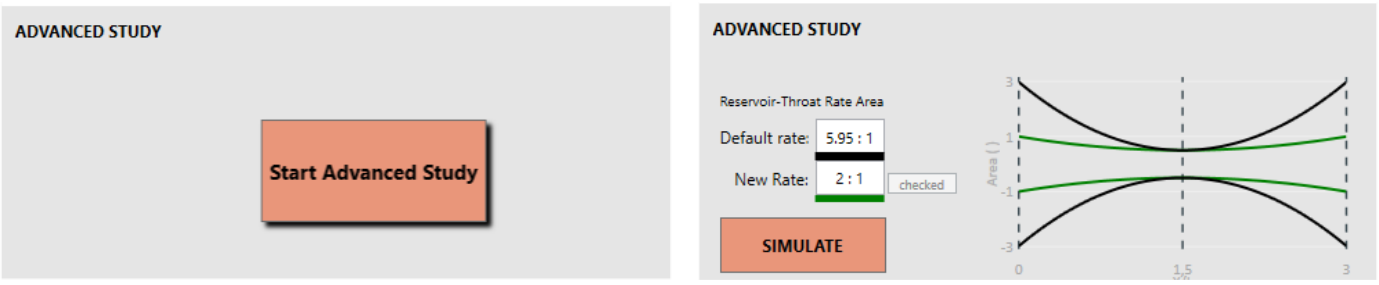

Figure 9: Advanced study interface.

With this possibility, the aim is to observe how all the properties vary along the nozzle when the reservoir-to-throat area ratio is modified. Particularly, maximum values are compared in order to find out whether there are some dangerous extremes. The exit velocity is also compared. These values are collected and compared in charts of Figure 10 and Figure 11.

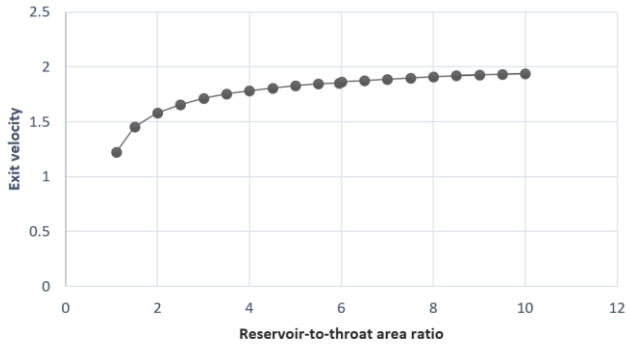

Figure 10: Exit velocity vs. Reservoir-to-throat ratio.

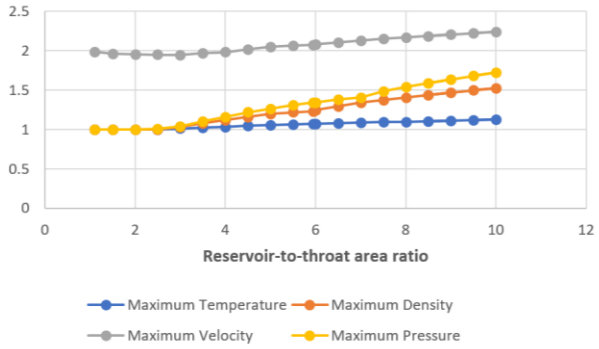

Figure 11: Nozzle properties' maximum values vs. Reservoir-to-throat ratio.

From these charts we can affirm that reducing this parameter decreases the exit velocity while relatively maintaining similar maximum values of temperature and the other physical properties. Nevertheless, when increasing the reservoir-to-throat ratio, exit velocity increases slightly while maximum values of the physical properties are much greater, creating extremes.

The obtained results indicate that increasing this parameter does not maximize temperature as much as previously thought, but rather pressure and density. Maybe this could point to certain improvements to resist such values during the transitory state as means to reaching greater exit velocity values.

These results also lead us to conclude that the given default value of reservoir-to-throat ratio is the optimal one to obtain the best values in exit velocity, and maximum temperature, pressure, and density.

VI. User's Guide

ANNEX A: FUNCTION USED TO UPDATE CHARTS

```
1.  public void SetChart()
2.      {
3.        if (nozzlesim.Getmalla() != null)   // if the matrix spacial-time is empty charts would be empty
4.        {
5.          List<List<double>> listV = new List<List<double>>();                    // The four list of properties: imagine there are 31 divisions, for each object (listV,listP ...)
6.          List<List<double>> listP = new List<List<double>>();                    // there will be a list of 31 List of doubles, one for each division. Each of this 31 list are collections of
7.          List<List<double>> listT = new List<List<double>>();                    // values of some property along the time, form the initial time until the actual time
8.          List<List<double>> listD = new List<List<double>>();
9.          int stepsChart;             // For a large value of steps, there are many values of each property, plotting that amount will be non-efficient, so we dismiss some of them.
10.                                     // For each value that we plot [stepCharet] values will be dismissed, in order to keep relatively low the amount of values plotted.
11.         if (this.steps > 1000)
12.           stepsChart = 40;
13.         else if (this.steps > 500)
14.           stepsChart = 20;
15.         else if (this.steps > 250)
16.           stepsChart = 10;
17.         else if (this.steps > 100)
18.           stepsChart = 5;
19.         else if (this.steps > 50)
20.           stepsChart = 2;
21.         else if (this.steps > 25)
22.           stepsChart = 1;
23.         else
24.           stepsChart = 0;
25.         int i = 0;
26.         int finStep = steps;
27.         List<Brush> ListBrush = new List<Brush>();      // This list has the same length as divisions of the nozzle, for the selected positions for plotting, there will be a Brush color
28.                                                         // different to Transparent that would be the case in which the position is not selected for plotting
29.         int posBrushes = 0;
30.         Position dimens;                        // This values are referred to the dimensional values of the simulation
31.         if (DimensionlessButton.IsChecked == false)
32.         {
33.           // T V P D
34.           dimens = nozzlesim.getDimensionalPosition();
35.         }
36.         else
37.         {
38.           dimens = new Position(0, 1, 1, 1, 0, 0);
39.         }
40.         foreach (int pos in brushesPos)             // brushesPos contains the information that indicates which positions are selected for plotting
41.         {
42.           if (pos != -1)                        // brushesPos are a list of integers, this number corresponds to the position of the Brush of BrushesList, so
43.           {                                     // when the number is different to -1, a column of values will be added to the list of each one of the lists
44.                                                 // (listV, listP...) and The corresponding Brush will be added to ListBrush
45.             listV.Add(nozzlesim.GetColumnPar(i, "V", stepsChart, dimens, finStep));
46.             listP.Add(nozzlesim.GetColumnPar(i, "P", stepsChart, dimens, finStep));
47.             listT.Add(nozzlesim.GetColumnPar(i, "T", stepsChart, dimens, finStep));
48.             listD.Add(nozzlesim.GetColumnPar(i, "D", stepsChart, dimens, finStep));
49.             ListBrush.Add(brushesList[brushesPos[i]]);
50.             posBrushes++;
51.           }
52.           else                  // in case of == -1 means some position is not selected
53.           {
54.             listV.Add(new List<double>());
55.             listP.Add(new List<double>());
56.             listT.Add(new List<double>());
57.             listD.Add(new List<double>());
58.             ListBrush.Add(Brushes.Transparent);
59.           }
60.           i++;
61.         }
62.
63.         createRecColors(ListBrush);  // this function paints the rectangles below the buttons of the positions of some color that let us to connect it with the chart
64.         int sel = 0;                // sel will count the amount of selected positions
65.         foreach (int pos in brushesPos)
66.           if (pos != -1)
67.             sel++;
68.         int maxUpdate = 1;          // it refers to the time that the function will wait for updating the charts since the last update
69.         if (sel > 2)
70.           maxUpdate = 2;
71.         if (lastChartUpdate > new TimeSpan(maxUpdate * 10000000) && plotChanged == true)     // plotChanged is a boolean that is set to true everytime charts need to be updated because the values changed
72.         {
73.           plotChanged = false;
74.           lastChartUpdate = new TimeSpan(0);
75.           // create the array of times
76.           List<double> timeList = nozzlesim.getTimeList(stepsChart, finStep);           // Those are the labels of the horizontal axis
77.           var times = new string[timeList.Count];
```

```
                i = 0;
                foreach (double time in timeList)
                {
                    times[i] = (Math.Round(time, 3)).ToString();
                    i++;
                }
                createChart(chartV, listV, ListBrush, xAxisV, times);      // those functions create each of the charts using the information previously computed
                createChart(chartP, listP, ListBrush, xAxisP, times);
                createChart(chartT, listT, ListBrush, xAxisT, times);
                createChart(chartD, listD, ListBrush, xAxisD, times);    //
            }


        }
        else
        {
            chartV.Series = new SeriesCollection();
            chartP.Series = new SeriesCollection();
            chartT.Series = new SeriesCollection();
            chartD.Series = new SeriesCollection();
        }
    }
```

ANNEX B: FUNCTION USED TO CREATE A SPECIFIC CHART WHEN ALL THE DATA IS COMPUTED AND READY FOR PLOTTING

```
    public void createChart(CartesianChart chart, List<List<double>> listV, List<Brush> ListBrush, Axis xAxis, string[] times)
    {

        chart.Series = new SeriesCollection();
        int i = 0;
        while (i < listV.Count)                    // This while creates each of the series for each division, if some serie is not selected for plotting listV is empty
        {                                          // so no plot will be seen regarding this position
            LineSeries linSerie = new LineSeries
            {
                Title = "x = " + (i / 10.0).ToString(),
                Name = "lineChart_" + i.ToString(),
                Values = new ChartValues<double>(listV[i]),
                PointGeometry = null,
                Fill = Brushes.Transparent,
                Stroke = ListBrush[i],
            };
            chart.Series.Add(linSerie);
            i++;
        }

        if (times.Count() > 1)             // Those lines creates the labels of the horizontal axis
        {
            xAxis.MaxValue = times.Count() - 1;
            xAxis.Separator.Step = times.Count() - 1;
            xAxis.Labels = times;
        }

        DataContext = this;
    }
```

ANNEX C: FUNCTION USED TO CREATE A LIST OF VALUES OF SOME SPECIFIC PROPERTY

```
public List<double> GetColumnPar(int col, string parameter, int steps, Position dimens, int finStep)
    {
        List<double> columna = new List<double>();
        int i = 0;
        int initStep = 0;
        while (i < malla.GetLength(0) && i <= finStep)                  // steps are the number of dismissed position
        {                                                               // dimens is an object position that contains the dimensional values of each property
            if (initStep == 0) // only initStep = 0 is added, others are dismissed, later some condition will make initStep return to 0
            {
                Position pos = GetPosition(i, col);
                if (pos != null)
                {
                    double value;
                    if (parameter == "x")
                        value = Math.Round(pos.GetX(), 4);
                    else if (parameter == "T")
                        value = Math.Round(pos.GetTemperature() * dimens.GetTemperature(), 4);
                    else if (parameter == "D")
                        value = Math.Round(pos.GetDensity() * dimens.GetDensity(), 4);
                    else if (parameter == "V")
                        value = Math.Round(pos.GetVelocity() * dimens.GetVelocity(), 4);
                    else if (parameter == "P")                                  // dimensional P units are hPa
                    {
                        if (dimens.GetPressure() != 1)
                            value = Math.Round(pos.GetPressure() * dimens.GetPressure(), 4) * dimens.R / 100;
```

```
                    else
                        value = Math.Round(pos.GetPressure() * dimens.GetPressure(), 4);
                }
                else if (parameter == "A")
                    value = Math.Round(pos.GetArea(), 4);
                else if (parameter == "M")
                    value = Math.Round(pos.MachNumber(), 4);
                else
                    value = -2;

                if (value != -2)
                    columna.Add(value);
            }
            else
            {
                break;
            }
        }
        if (initStep == steps)      // here we code the way we dismiss values, the next position of the while will has initStep = 0, that are the only cases added
        {
            initStep = -1;
        }
        initStep++;
        i++;
    }
    return columna;
}
```