# BACHELOR'S THESIS

Safe and unsafe stability loss in systems with input saturation

by

Oleksii Yeromenko

in partial fulfilment of the requirements for the degree of

Bachelor of Science (BSc)

in Mathematics

Supervisor: Prof. Ivan Ovsyannikov

Date of Submission: 20.05.2025
Date of Defense: 16.05.2025
School of Computer Science & Engineering

CONSTRUCTOR UNIVERSITY

# Statutory Declaration

| | |
|---|---|
| Family Name, Given/First Name | Yeromenko, Oleksii |
| Matriculation number | 30006509 |
| What kind of thesis are you submitting: Bachelor-, Master- or PhD-Thesis | Bachelor Thesis |

**English: Declaration of Authorship**

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

**German: Erklärung der Autorenschaft (Urheberschaft)**

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung  oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

19.05.2025 [illegible]

Date, Signature

# Contents

**Summary.** We will consider a problem of stability and control of a specific system with input saturation and its variations. Based on previous discoveries and motivations we expect to observe safe and unsafe scenarios in the system, as the latter can lead to catastrophes. For different parameter values and function choices we systematize its behavior, produce diagrams describing the stability of the solutions, compare them with theoretical results where available, and bring some other interesting observations to the attention of a reader. For this purpose numerical algorithms and technics will be utilized, as well as analytical studies of the system.

## 1 Introduction

It is common for many real-world systems to have a desirable base state, but under certain scenarios the stability of this state is lost and it leads to devastating catastrophes.

The motivation comes from aviation, namely pilot control of fighter jets. A couple of accidents have occurred where after some sequences of pilot's actions they the control over their aircraft was lost.[1] Something that is referred to as pilot-induced oscillation (PIO) took place, but the direct cause was yet to be discovered. It is known that under certain control system parameter alterations or manipulations the plane behavior, which the parameter was responsible for, ceases to be revertible. It then leads to unexpected outcomes including plane crashes and similar. It is therefore essential to study such cases and develop a theoretical understanding of them.

In essence, in the systems describing the above examples the transition between two stable states, one of which is assumed to be desirable in reality, from A to B and from B to A was completely different. That is, there is such a bifurcation point after which the base state loses its stability safely or unsafely. It is important to note that studying this kind of system linearly will only give us an idea of stability loss itself of our base state, but not the way this loss takes place. Nonlinearly, we can distinguish between safe or unsafe stability losses. In the safe scenario, the situation is under control, the bifurcation is revertible. But in the unsafe scenario, the system loses control and cannot be returned to the original state by reversing the actions. This may lead to some unexpected catastrophic scenarios and such typical scenarios as described above can also be referred to with a more general term *hysteresis*. It is defined as the dependence of the state of a system on its history. Moreover, under some conditions the return path from one trajectory to another can be unreachable, usually because of the engineering specificity of system construction. But we do not focus on discussing the latter topic in this paper.

The safety of the bifurcation of states can be determined by system parameters or even outer conditions. Several studies on non-linear and switched systems deal with the same type of problem and focus on testing stability, the range of parameters where this stability is preserved, as well as utilizing other techniques for this purpose.[2] A study specifically on planes was also conducted. For instance, this can be aerodynamic airfoil characteristics that can be investigated through experiments, etc.[3]

[1] C. A. Shifrin, "Sweden seeks cause of Gripen crash," *Aviation Week and Space Technology* 139, no. 7 (1993); M. A. Dornhein, "Report pinpoints factors leading to YF-22 crash," *Aviation Week & Space Technology* 9 (1992).

[2] G.A. Leonov & J.E. Rooda R.A. van den Berg A.Yu. Pogromsky, "Design of Convergent Switched Systems," *Springer*, 2006, Alexander Yu Pogromsky and Alexey S Matveev, "A non-quadratic criterion for stability of forced oscillations," *Systems & Control Letters* 62, no. 5 (2013).

[3] Mathew Martin Zifeng Yang Hirofumi Igarashi and Hui Hu, "An Experimental Investigation on Aerodynamic Hysteresis of a Low-Reynolds Number Airfoil," Preprint, available online, *Springer*, 2008, https:

Here we focus on studying a particular saturated system, which we introduce further, for its base state stability loss scenarios. We do not explain exactly the reason behind the choice of saturation, but this topic has been studied elsewhere.[4] Firstly, we will consider a "simplified" version of a system with singularity on a specific $y$-axis region and discontinuous periodic signal. Afterwards a singularity is removed by opening up a region around $y$-axis with the help of a different function choice and a smooth sinusoidal signal is used instead of the discontinuous one. This way we move from cases where analytical computations are comparatively easily performed, allowing us to compare the practical simulation results with the theoretical ones, to the more complicated but closer to real-world ones. In this way, we are able to develop some more concrete understanding of behavior in the first few system variations and then use them to merely observe the similar picture for the more relevant choice of functions.

## 2 System behavior description and visualization

We consider the following system:

$$\begin{aligned} \dot{x} &= y - a\left(F(x) + f(t)\right) \\ \dot{y} &= -b\left(F(x) + f(t)\right) \end{aligned} \tag{1}$$

This serves as a general form and, as discussed, we will have two choices per functions $F, f$. Here $f$ plays the role of an input saturation or an external signal, which we fix to have a period of $2\tau$ with amplitude equal to $\mu$. In all further cases we will have $a, b, \mu > 0$.

We expect the system to have certain attracting solutions. That is, after sufficient time $t$ the system will get as close as $\epsilon$, $\forall \epsilon > 0$, to this trajectory. Since $f(t)$ is periodic we expect to have them in the form of attracting cycles.

We construct a stroboscopic map by the following way: consider a 3D space $(x, y, t)$ and take a set of planes $T_n : \{t = t^* + 2\tau n, n \in \mathbb{Z}\}$. This way the map $g_n : T_n \to T_{n+1}$ is defined along the trajectories of (1). Because of periodicity of $f(t)$ the periodic points of the map $g$ correspond to the periodic trajectories of the different periods of the system (1). In particular, the fixed points correspond to $2\tau$-periodic trajectories. We also note that if we make a $\tau$ shift in time and perform the reflection $(x, y) \mapsto (-x, -y)$, the system (1) will not change.

In this paper we focus namely on the fixed point of the constructed map. However, we are able to observe higher periodicity points during numerical simulations as well. The illustrations for them and other examples are provided further.

### 2.1 $F$ - closed, $f$ - meander

#### 2.1.1 Analytical description

In this case we consider:

$$F(x) = \operatorname{sign}(x), \quad f(t) = \begin{cases} -\mu, & (2n-1)\tau \le t < 2n\tau \\ \mu, & 2n\tau \le t < (2n+1)\tau \end{cases}, \quad n \in \mathbb{Z} \tag{2}$$

The system (1) then has a discontinuity along the line $\{x = 0\}$, so there will be a so called "sliding zone" on a region of the line. We define the vector field inside that zone to be the

//doi.org/10.2514/6.2008-315.

[4] Karl Johan Astrom and Lars Rundqwist, "Integrator Windup and How to Avoid It," in *1989 American Control Conference* (1989), https://doi.org/10.23919/ACC.1989.4790464.

average between vector fields on the left and on the right of the region:

$$\dot{y} = \frac{-b(-1+f(t)) - b(1+f(t))}{2}.$$

We look for the fixed points by the following way: we fix a point with coordinates $(0, -y_0)$, $y_0 > 0$ at the moment denoted as $t_0 < \tau$. At time $\tau$ the function (after $\tau - t_0$ time) the function $f(t)$ changes its sign and we try to find such $y_0$ and $t_0$ such that this trajectory is at the point $(0, y_0)$ at the moment $t_0 + \tau$. The symmetry that we mentioned above guarantees us that at the moment $t_0 + 2\tau$ the trajectory will return to $(0, -y_0)$.

In the time interval $t_0 \le t < \tau$ the system (1) has the form

$$\begin{aligned} \dot{x} &= y - a\left(-1+\mu\right) = y + a - a\mu \\ \dot{y} &= -b\left(-1+\mu\right) = b - b\mu \end{aligned} \tag{3}$$

and we assume $y_0$ to be big enough for $\dot{x}$ to be negative and we start not in the sliding zone. The solution of the system (3) with initial conditions $x(t_0) = 0,\ y(t_0) = -y_0$ is:

$$x(t) = (b-b\mu)\left(\frac{t^2}{2} - \frac{t_0^2}{2}\right) - (b-b\mu)(t-t_0)t_0 + (-y_0+a-a\mu)(t-t_0),\ y(t) = (b-b\mu)(t-t_0) - y_0 \tag{4}$$

Then for $\tau \le t \le t_0 + \tau$ we integrate the system

$$\begin{aligned} \dot{x} &= y - a\left(-1-\mu\right) = y + a + a\mu \\ \dot{y} &= -b\left(-1-\mu\right) = b + b\mu \end{aligned} \tag{5}$$

with initial conditions $x(\tau) = (b-b\mu)\left(\frac{\tau^2}{2} - \frac{t_0^2}{2}\right) - (b-b\mu)(\tau-t_0)t_0 + (-y_0+a-a\mu)(\tau - t_0),\ y(\tau) = (b-b\mu)(\tau-t_0) - y_0$. The solution obtained is:

$$\begin{aligned} x(t) &= (b-b\mu)\left(\frac{\tau^2}{2} - \frac{t_0^2}{2}\right) - (b-b\mu)(\tau-t_0)t_0 - y_0(\tau-t_0) \\ &\quad + (a-a\mu)(\tau-t_0) + (b+b\mu)\left(\frac{t^2}{2} - \frac{\tau^2}{2}\right) - (b+b\mu)(t-\tau)\tau \\ &\quad + (b-b\mu)(\tau-t_0)(t-\tau) - t_0(t-\tau) + (a+a\mu)(t-\tau) \\ y(t) &= (b+b\mu)(t-\tau) + (b-b\mu)(\tau-t_0) - y_0 \end{aligned} \tag{6}$$

Now taking $x(t_0+\tau) = 0,\ y(t_0+\tau) = y_0$ we derive equations for $y_0$ and $t_0$. For the equation for $y$ we get:

$$(b+b\mu)t_0 + (b-b\mu)(\tau-t_0) - y_0 = y_0 \implies y_0 = b\mu t_0 + (b-b\mu)\frac{\tau}{2}. \tag{7}$$

Substituting to the one for $x$ and simplifying we obtain:

$$b\mu t_0^2 + (2a\mu - b\mu\tau)t_0 + (a-a\mu)\tau = 0. \tag{8}$$

Note that $y_0(t_0)$ is a monotonically increasing function. The equation (8) is always quadratic under condition that $b, \mu \ne 0$, which is indeed satisfied by our assumptions in the beginning. It can have up to two roots and when we manipulate the parameters so that the roots are born a bifurcation takes place. Consequently, stable or unstable fixed points are born.

Now let us calculate the bifurcation condition. We do that with the help of the discriminant of the equation (8):

$$D = (2a\mu - b\mu\tau)^2 - 4ab\mu\tau + 4b\mu^2\tau = \mu^2(4a^2 + b^2\tau^2) - 4ab\mu\tau. \tag{9}$$

$D$ is itself a quadratic polynomial in $\mu$ with positive main coefficient and roots $\mu = 0$ and $\mu = \mu_0 := \dfrac{4ab\tau}{b^2\tau^2 + 4a^2}$. Note that by Cauchy inequality $b^2\tau^2 + 4a^2 \geq 2\sqrt{4a^2b^2\tau^2} = 4ab\tau$, so $\mu_0 \leq 1$. In general, when $0 < \mu < \mu_0$ there are no real roots of (8), hence no fixed points of the map and when $\mu > \mu_0$ the equation has two roots.

For further checks, let us denote the left part of equation (8) as a function $h(t_0) = b\mu t_0^2 + (2a\mu - b\mu\tau)t_0 + (a - a\mu)\tau$ and compute the values $h_1 := h(0) = a\tau(1-\mu)$ and $h_2 := h(\tau) = a\tau(1+\mu)$. This function is quadratic with a positive main coefficient (assuming our conditions) and has its extremum point at $t_e = \dfrac{\tau}{2} - \dfrac{a}{b}$.

Let us now check for further conditions for the fixed points to exist. The first condition is $0 \leq t_0 \leq \tau$. The second means that the fixed point can not be at the sliding zone: $-y_0 \leq a(-1+\mu)$. Substituting the formula (7) we can rewrite this condition in terms of $t_0$: $t_0 \geq t_{cr} := \dfrac{1}{\mu}\left(\dfrac{a}{b} - \dfrac{\tau}{2}\right) + \dfrac{\tau}{2} - \dfrac{a}{b}$. We notice that depending on the sign of $\dfrac{a}{b} - \dfrac{\tau}{2}$ two different cases of behavior of $t_{cr}$ are obtained.

**Case 1.** $\dfrac{\tau}{2} > \dfrac{a}{b}$.

Let us see what we get by increasing $\mu$ from zero. When $\mu \to +0$ we have $t_{cr} \to -\infty$ and $t_{cr}(\mu)$ is obviously a monotonically increasing function.

When $0 < \mu < \mu_0$ the equation (8) has not real roots and $t_{cr} < 0$.

When $\mu_0 < \mu < 1$ the equation has two roots between 0 and $\tau$, because $0 = \dfrac{\tau}{2} - \dfrac{\tau}{2} < \dfrac{\tau}{2} - \dfrac{a}{b} < t_e < \tau$ and both $h_1, h_2$ are positive. Here $t_{cr} < 0$. **Conclusion:** after $\mu$ crossed the value $\mu_0$ two fixed points are born.

When $\mu > 1$ the equation (8) has two roots, but now $h_1 < 0$, $h_2 > 0$. So the smaller root lies outside the interval $(0, \tau)$. Additionally, $t_{cr} > 0$, but the bigger root is greater than $t_{cr}$, since $t_{cr} < \dfrac{\tau}{2} - \dfrac{a}{b}$ and the bigger root is $\dfrac{\tau}{2} - \dfrac{a}{b} + \dfrac{\sqrt{D}}{2b\mu}$ which is bigger than that value. **Conclusion:** after $\mu$ crossed 1 the first fixed point at the moment $\mu = 1$ reached the boundary of the sliding zone and ($t_{cr}(\mu = 1) = 0$) and disappeared.

**Case 2.** $\dfrac{\tau}{2} < \dfrac{a}{b}$.

Now $t_{cr}(\mu)$ is a decreasing function and $t_{cr} \to +\infty$ as $\mu \to +0$.

When $0 < \mu < \mu_0$ there are no roots in (8) and $t_{cr} > 0$.

When $\mu_0 < \mu < 1$ the equation (8) has two real roots, but both lying outside the interval, because $t_e < 0$ and $h_1, h_2 > 0$. Here $t_{cr} > 0$. **Conclusion:** there are not fixed points.

When $\mu > 1$ $h_1 < 0$, $h_2 > 0$, so the bigger root of (8) is inside $(0, \tau)$. Moreover, $t_{cr} < 0$. **Conclusion:** after $\mu$ crosses 1 there is one fixed point which is born at the moment $\mu = 1$ from the boundary of the sliding zone.

### 2.1.2 Numerical results and comparison

Here we provide the results obtained through numerical simulations and compare them with theoretical ones where possible. A more detailed description of the underlying functionality of the developed software tool set is provided in one of the sections below. From now on, all further illustrations will be produced with the step size of a numerical algorithm $h = 0.01$, end time of simulation $t_{end} = 1000$. For pictures of safe and unsafe scenarios parameter values $a = 10,\ b = 2,\ \tau = 4$ and $a = b = 1,\ \tau = 10$ will be used respectively. Thus, $\frac{\tau}{2} > \frac{a}{b}$ and $\frac{\tau}{2} > \frac{a}{b}$ for both types of behavior in return, so we can potentially see a correspondence with our theoretical results.

The results will be conveyed with the help of diagrams that try to capture hysteresis of the system when present.

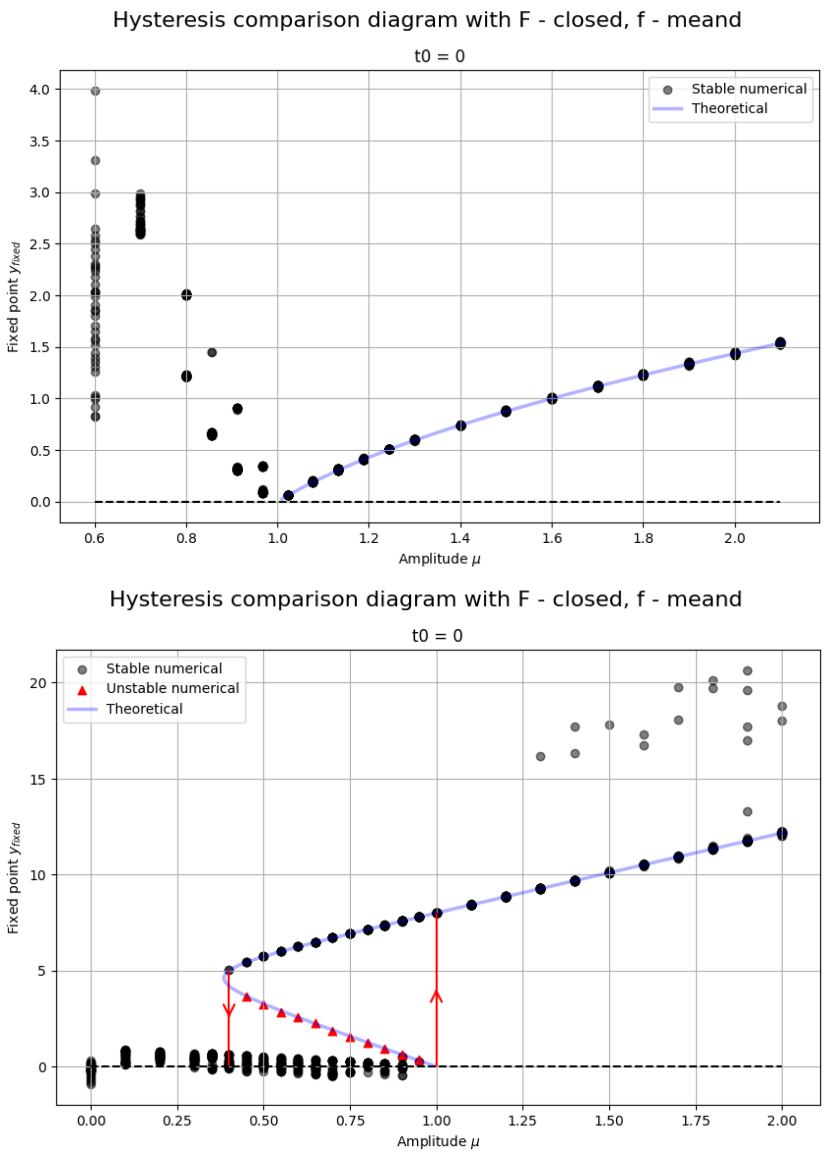


Figure 1: Safe (image above) and unsafe (image below) cases. $F$-closed, $f$-meander

On the figure 1 a safe and unsafe cases for our mentioned function choice are illustrated. We can observe the development of system states over the increasing amplitude $\mu$, the scaling coefficient for our signal $f(t)$. The system states are characterized by the fixed point $y_0$, which was constructed in the previous subsection, and we see it on the vertical axis.

First important observation we make is the following: according to the performed calculations the system itself does not give us an explicit solution for a base case. Namely, for $\frac{\tau}{2} > \frac{a}{b}$ the first fixed point are born only at $\mu_0$ with the smaller one already being unstable (as we will confirm with the diagram); and for $\frac{\tau}{2} < \frac{a}{b}$ the only fixed point is born at $\mu = 1$. However, on both diagrams in the regions where no fixed points can be found we still see an outlined trajectory of points. In fact, because the sliding zone, which is present in this case, can reach up to $a(1 + \mu)$ and taking in to account the values of $a$ for each case it is safe to assume that those points are precisely the points in the sliding zone. Even though this region is somewhat ill-defined (violates uniqueness of solutions, continuity of the system) it can be viewed as our system's initial state, or base state. Also, since the way the points that end up in the sliding zone are captured in the underlying algorithm is somewhat random one should not rely too much on the exact positions of those points. It is enough to see that they are within the zone boundaries.

Taking these specificity into account, on both diagrams we can indeed see the behavior that we would expect. In the safe scenario we have our base state in the form of the sliding zone points after which a new stable state is born and one can revert the transition from one state to another with ease. In the unsafe case, however, an unstable state is born along with another stable one. When this unstable state intersects the base state the system is forced to transition to the new stable state. The reversion in this case is much harder to achieve, since one would require to decrease $\mu$ all the way to the point when this stable state was born ($\mu = \mu_0$). Only then is the base state reached and the change reverted.

The transition from the base state to the previously born stable state is similar to what perhaps occurred in the pilot catastrophe example. Moreover, it is believed that after crossing the bifurcation point ($\mu = 1$ in this case) due to restrictions of the engineering design of the jets the value required to revert the state ($\mu = \mu_0$ in this case) could not be reached.

Additionally, it is apparent that the theoretical solution indeed matches the numerical results rather precisely. Of course, a smaller step size $h$ or end time $t_{end}$ can be chosen to increase the accuracy.

Plotting the actual trajectories for values of $\mu$ around the bifurcation point in the unsafe scenario one can see how they differ, like it is done on the figure 2.

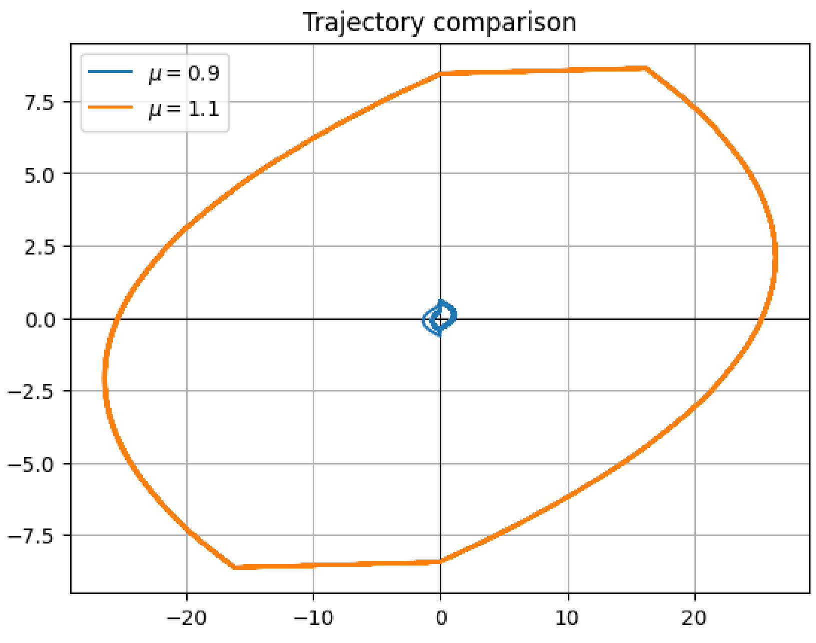


Figure 2: Solution trajectory comparison

Another crucial observation lies in the "outlying" points in the top-right corner of the unsafe diagram. If we try looking at the trajectory corresponding to one of those points we can see something as on the figure 3. If examined closely, the trajectory on this figure is attracted to a cycle with a period $6\tau$. Being a multiplicity of the $2\tau$ periodic trajectories that we focus on, we can talk about a parametric resonance taking place here. In broad terms, the frequencies of the trajectories synchronize and consequently we observe these higher period points on the diagram with greater values of $y_0$. This is a direction that potentially requires closer attention.

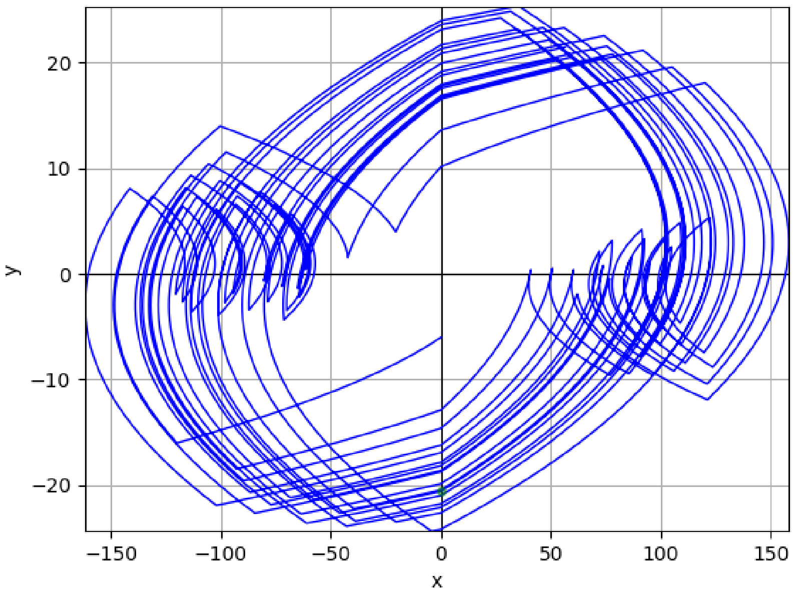


Figure 3: Trajectory of period $6\tau$

## 2.2 $F$ - closed, $f$ - sinusoid

### 2.2.1 Analytical description

In this case we have:

$$F(x) = \operatorname{sign}(x), \quad f(t) = \mu \sin\left(\frac{\pi t}{\tau}\right) \tag{10}$$

The sliding zone along the line $\{x = 0\}$ remains, but now the saturation is described by a sinusoid with amplitude $\mu$ and period $2\tau$. We repeat similar steps as for the previous variation. Only this time we only need to integrate over one interval in time, because of the smoothness of sin.

In the time interval $t_0 \le t < t_0 + \tau$ the system (1) becomes:

$$\begin{aligned} \dot{x} &= y - a\left(-1 + \mu \sin\left(\frac{\pi t}{\tau}\right)\right) = y + a - a\mu \sin\left(\frac{\pi t}{\tau}\right) \\ \dot{y} &= -b\left(-1 + \mu \sin\left(\frac{\pi t}{\tau}\right)\right) = b - b\mu \sin\left(\frac{\pi t}{\tau}\right) \end{aligned} \tag{11}$$

With initial conditions $x(t_0) = 0$, $y(t_0) = -y_0$ the system (11) solves to:

$$
\begin{aligned}
x(t) &= b\left(\frac{t^2}{2}-\frac{t_0^2}{2}\right) - bt_0(t-t_0) + \frac{b\mu\tau^2}{\pi^2}(\sin\left(\frac{\pi t}{\tau}\right) - \sin\left(\frac{\pi t_0}{\tau}\right)) - \frac{b\mu\tau}{\pi}\cos\left(\frac{\pi t_0}{\tau}\right)(t-t_0) \\
&\quad - y_0(t-t_0) + a(t-t_0) + \frac{a\mu\tau}{\pi}(\cos\left(\frac{\pi t}{\tau}\right) - \cos\left(\frac{\pi t_0}{\tau}\right)) \\
y(t) &= b(t-t_0) + \frac{b\mu\tau}{\pi}(\cos\left(\frac{\pi t}{\tau}\right) - \cos\left(\frac{\pi t_0}{\tau}\right)) - y_0
\end{aligned}
\tag{12}
$$

Now setting conditions $x(t_0+\tau)=0$, $y(t_0+\tau)=y_0$ we derive. For the formula for $y_0$:

$$
b\tau + \frac{b\mu\tau}{\pi}(\cos\left(\frac{\pi(t_0+\tau)}{\tau}\right) - \cos\left(\frac{\pi t_0}{\tau}\right)) - y_0 = y_0 \implies y_0 = \frac{b\tau}{2} - \frac{b\mu\tau}{\pi}\cos\left(\frac{\pi t_0}{\tau}\right) \tag{13}
$$

Substituting (13) into the equation for $x$ and simplifying we obtain:

$$
b\mu\tau^2\sin\left(\frac{\pi t_0}{\tau}\right) + a\mu\tau\pi\cos\left(\frac{\pi t_0}{\tau}\right) = \frac{a\tau\pi^2}{2}. \tag{14}
$$

Our equation has the form $A\sin(x)+B\cos(x)=C$. The method for solving this type of equation is to rewrite it in the equivalent form $R\sin(x+\alpha)=C$, where $R=\sqrt{A^2+B^2}$ and $\alpha=\arctan\left(\frac{B}{A}\right)$.

In our case $R=\sqrt{(\mu\tau)^2+(\frac{a}{b}\mu\pi)^2}$, $\alpha=\arctan\left(\frac{a\pi}{b\tau}\right)$, $C=\frac{a\pi^2}{2b}$. We can now find the roots of the equation, but for them to exist we need to ensure $\left|\frac{C}{R}\right|<1$. Since in our case $\mu>0$ it is enough to have $\frac{C}{R}<1$. Solving the condition for $\mu$ we obtain:

$$
\mu > \mu_0 := \frac{\frac{a\pi}{2b}}{\sqrt{(\frac{\tau}{\pi})^2+(\frac{a}{b})^2}} \tag{15}
$$

Notice that we always have $\mu_0<\frac{\pi}{2}$. The actual roots then are:

$$
\begin{aligned}
t_1 &= \frac{\tau}{\pi}\left(\arcsin\frac{\mu_0}{\mu} - \arctan\left(\frac{a\pi}{b\tau}\right)\right) \\
t_2 &= \frac{\tau}{\pi}\left(\pi - \arcsin\frac{\mu_0}{\mu} - \arctan\left(\frac{a\pi}{b\tau}\right)\right)
\end{aligned}. \tag{16}
$$

Further conditions for fixed points are $0\le t_0\le\tau$ and similar to previous case $-y_0\le a(-1+\mu\sin\left(\frac{\pi t_0}{\tau}\right))$. Using (13) we rewrite the second condition to:

$$
\frac{\mu\tau}{\pi}\cos\left(\frac{\pi t_0}{\tau}\right) - \frac{a}{b}\mu\sin\left(\frac{\pi t_0}{\tau}\right) \le \frac{\tau}{2} - \frac{a}{b}. \tag{17}
$$

As above we can compute $R=\sqrt{(\frac{\mu\tau}{\pi})^2+(\frac{a}{b}\mu)^2}$, $\alpha=\arctan\left(-\frac{b\tau}{a\pi}\right)$, $C=\frac{\tau}{2}-\frac{a}{b}$. Hence, we require:

$$
\begin{aligned}
t_0 &\geq \frac{\tau}{\pi}\left(\arcsin \frac{\mu_0}{\mu}\frac{\frac{a}{b}-\frac{\tau}{2}}{\frac{a\pi}{2b}} + \arctan\left(\frac{b\tau}{a\pi}\right)\right) =: t_{cr1} \\
t_0 &\leq \frac{\tau}{\pi}\left(\pi - \arcsin \frac{\mu_0}{\mu}\frac{\frac{a}{b}-\frac{\tau}{2}}{\frac{a\pi}{2b}} + \arctan\left(\frac{b\tau}{a\pi}\right)\right) =: t_{cr2}
\end{aligned}
. \tag{18}
$$

Notice that the quantity $\frac{\tau}{2} - \frac{a}{b}$ will determine the behavior of (16).

**Case 1.** $\frac{\tau}{2} > \frac{a}{b}$.

Let us start increasing $\mu$ from 0. When $\mu \to +0$ we have $t_{cr1}, t_{cr2}$ do not exist. Notice that $\frac{C}{R}$ is a monotonically decreasing function.

When $0 < \mu < \mu_0$ (14) does not have any roots and $t_{cr1}, t_{cr2}$ may or may not exist depending on when $\mu_0 > \frac{C}{R} \iff \mu > \frac{(\frac{a}{b}-\frac{\tau}{2})}{\frac{a\pi}{2b}} = \frac{2}{\pi} - \frac{b\tau}{a\pi}$, which is $> 0$ when $\frac{\tau}{2} > \frac{a}{b}$ which is the case here. **Conclusion:** no fixed points.

Let us now set $\mu > \mu_0$ and verify the conditions for the roots $t_1$, $t_2$. It is not hard to check that we always have $t_{cr1} < t_2 < t_{cr2}$. However for $t_1$ we can discover the condition when $t_1 < t_{cr1}$. The solution gives us:

$$
\mu > \mu_0 \sqrt{1 + \left(\frac{\frac{a}{b}-\frac{\tau}{2}}{\frac{a\pi}{2b}}\right)^2} := \mu_1. \tag{19}
$$

It can also be shown that $\mu_0 < \mu_1 < \frac{\pi}{2}$. After $\mu_0 > \frac{\pi}{2}$ the root $t_1$ becomes negative, hence outside of $(0, \tau)$, while the second root remains.

That is, when $\mu_0 < \mu < \mu_1$ (14) has both roots. **Conclusion**: there are two fixed points.

When $\mu > \mu_1$ one root disappears in the sliding zone. **Conclusion**: one fixed point.

**Case 2.** $\frac{\tau}{2} < \frac{a}{b}$.

Similarly, when $0 < \mu < \mu_0$ no roots exist. Additionally, $t_{cr1}, t_{cr2}$ already exist by the time $\mu$ reaches $\mu_0$, as $\frac{(\frac{a}{b}-\frac{\tau}{2})}{\frac{a\pi}{2b}} < 1$ for this case. **Conclusion:** not fixed points.

Here we expect the greater root to be born from the sliding zone boundary. So let us compute the condition for $t_2 > t_{cr1}$. In fact, the solution gives us the same $\mu_1$ as was defined before. The only difference is that here it is not necessarily that $\mu_1 < \frac{\pi}{2}$. We can in fact show that $\mu_1 < \frac{\pi}{2} \iff \tau > \frac{a}{b}$.

When $\mu_0 < \mu < \mu_1$ we notice that $t_1, t_2 < t_{cr1}$ . **Conclusion:** no fixed points.

When $\mu > \mu_1$ still $t_1 < t_{cr1}$, but now $t_2 > t_{cr1}$. Also, $t_2 < t_{cr2}$ for this set of amplitudes. **Conclusion:** one fixed point is born from the boundary of the sliding zone.

### 2.2.2 Numerical results and comparison

The diagrams for this case are provided on the figure 4. One can verify that the numerical and theoretical trajectories match with good accuracy. This case is not particularly different from the previous except for the actual values where roots are born and the bifurcation occurs. We do still observe the higher periodicity points on the unsafe plot.

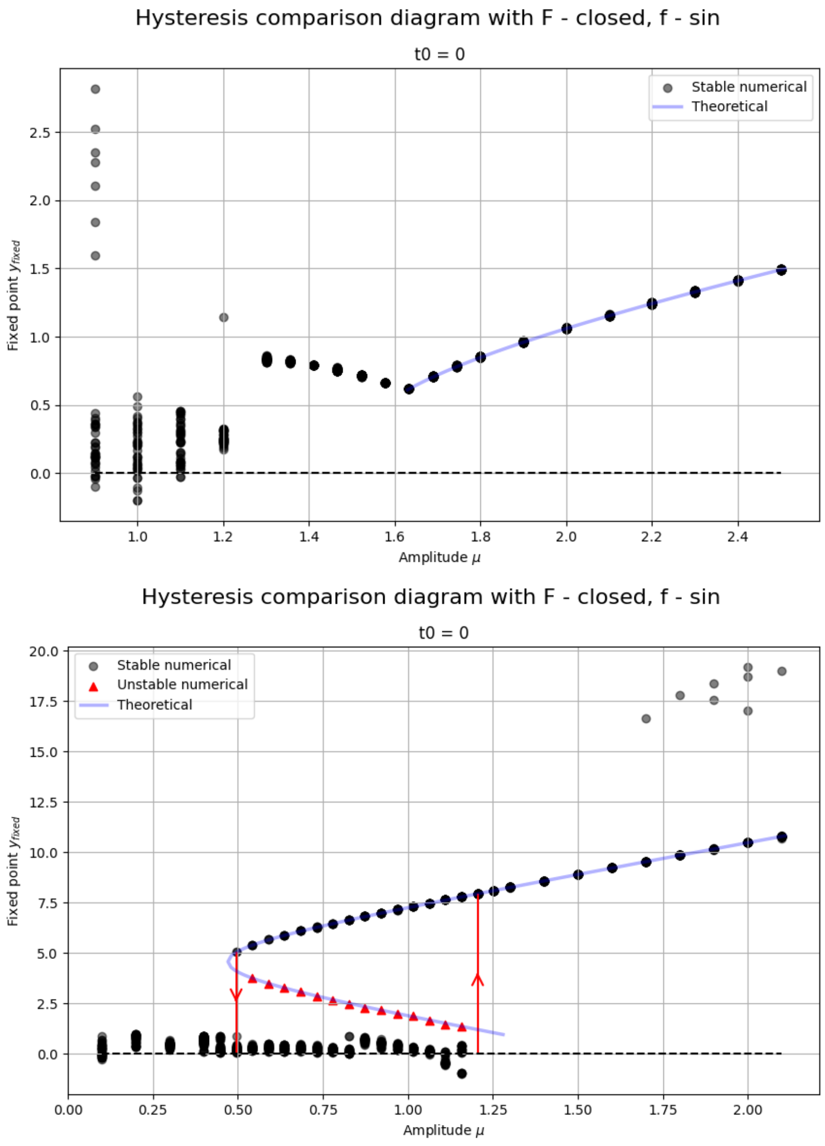
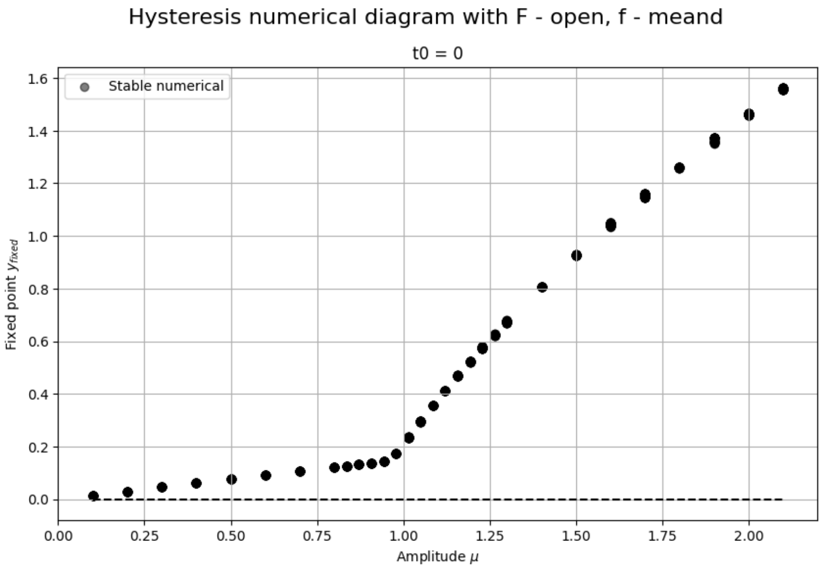


Figure 4: Safe (image above) and unsafe (image below) cases. $F$-closed, $f$-sin

## 2.3 $F$-open, $f$-meander

### 2.3.1 Numerical results and comparison

Although the analytical derivations for this particular case are not provided we still manage to observe the similar safe and unsafe scenarios for this function variation on the figure 5. The overall behavior is as we expect for the scenarios of our interest. The essential detail that we note from these plots is that one can observe an explicit enough base state plotted numerically. Due to absence of the sliding zone, because of the choice of $F$, the system is continuous, although not smooth. This is enough for apparently an explicit solution for base state to now exist. It seems like it only exists for those fixed points whose trajectory in $x$ is within $(-2, 2)$. But one would require analytical proof, which can still be done in this case with not too much difficulty.



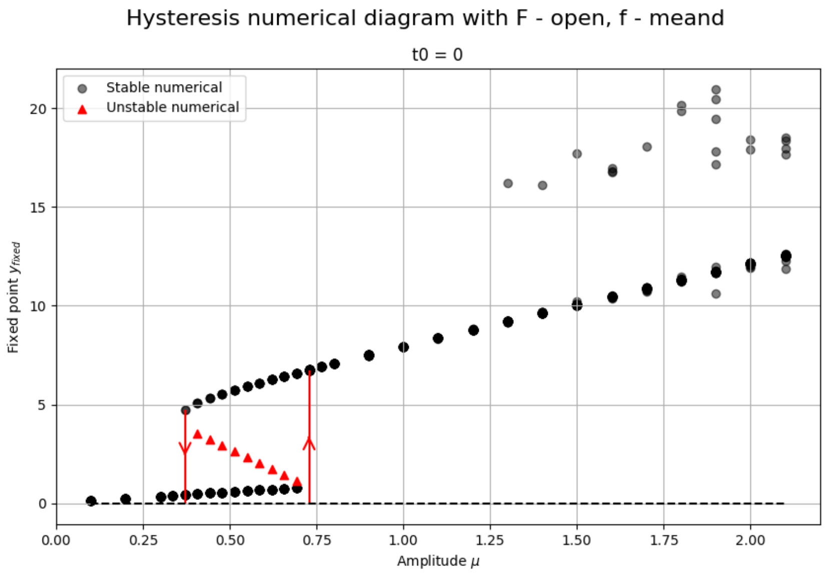


Figure 5: Safe (image above) and unsafe (image below) cases. $F$-open, $f$-meander

# 3 Methods and tools

For the purposes of this research a numerical tool set in Python was developed. Here information regarding its structure, functionality, etc, is presented. The reader is welcome to experiment with the full source code and use it for their own ideas by following the link to the GitHub repository (also see Appendix A for the core code listings.).

The overall structure is as follows.

```
functions.py
analytical.py
num_methods.py
plotting.py
toolset.py
```

- `functions.py`: Contains general definitions of functions used during the research.
- `analytical.py`: Functions for computing most of the derived values ($\mu_0$, $\mu_1$, etc) from the analytical system studies.
- `num_methods.py`: Contains numerical algorithms used throughout the research.
- `plotting.py`: Functions for visualizing results.
- `toolset`: Functions for producing the diagrams capturing system's stability.

All modules were written to enable reproducibility and flexibility in experimenting with model parameters. To understand how the modules are utilized in practice a Jupyter notebook with examples is included in the repository.

We will say a few words about the most important code parts in the following subsections.

## 3.1 Numerical Methods

In this subsection we discuss some details regarding the numerical algorithms used. This concerns the algorithms for numerical differential equation solving. The two methods utilized were Euler's and Runge-Kutta 4, which we assume the reader is familiarized with. The Euler's algorithm was used predominantly in the earlier research stages. With time it was replaces with a more accurate Runge-Kutta 4, so we focus mainly on its underlying functionality.

The main difference from the usual algorithm of this kind originates from the fact that, as discussed, in some cases our system involves a sliding zone, which requires further care in the code.

Firstly, on each iteration step the algorithm readjusts its step in case of crossing $y$-axis. By simple linear interpolation, a line between the point before crossing and after is constructed. Afterwards the $y$-intercept of that line is taken as a new value for $y$ for this iteration. The step $h$ for this iteration is adjusted accordingly.

```
if x*x_new < 0
    slope := (y_new - y)/(x_new - x)
    y_tilda := slope*(-x)+y
    h_coef := Euclid_distance((0,y_tilda), (x_new, y_new))
    h_tilda := h*h_coef
    x_new := 0
    y_new := y_tilda
    t -:= h - h_tilda
```

On one hand, this increases algorithm's accuracy for our particular case of discontinuity along the $y$-axis. But more importantly, if on the next iteration it turns out that the vector fields around the computed value point at each other, it indicates that we are in the sliding zone and the vector field defined for this specific region is used instead.

```
if F_type is 'closed'
        if x = 0 and (x_vec_field epsilon to the right < 0)
        and (x_vec_field epsilon to the left > 0)
            Runge-Kutta_step()
```

Additionally, a step is adjusted in case the trajectory escapes the sliding zone. This further ensures the discontinuity does not impact the steps too much.

```
if y_new_in_sliding < sliding_zone_bottom_edge():
    y_new := sliding_zone_bottom_edge() - epsilon
    h_coef := abs((y_new - y_prev)/(y_new_in_sliding - y_prev))
    h_tilda := h*h_coef
    t -:= h - h_tilda
elif y_new_in_sliding > sliding_zone_top_edge():
    y_new := sliding_zone_top_edge() + epsilon
    h_coef := abs((y_new - y_prev)/(y_new_in_sliding - y_prev))
    h_tilda := h*h_coef
    t -:= h - h_tilda
```

## 3.2 Visualization

Here we explain how the visualization tools were implemented. Two functions serve for this purpose in the source code: `plot_basic` makes a simple plot of a trajectory obtained from the arrays of values from one of the previously mentioned numerical methods, `plot_anim` produces an animation of a solving trajectory developing in time. Although being more demanding in terms of computational resources, the latter function proved to be rather useful for developing the intuition for the system's behavior, and generally better perception of the trajectories.

There are two main features that we point out about the animating function. First of them is the ability to plot the sliding zone region dynamically during the animation. In some cases it was essential to see how the zone behaves along with the trajectories. For instance on figure 6 one can verify that the trajectory is stuck in the sliding zone by seeing the region itself in red and zooming into it by setting the corresponding parameters. Of course, in Jupyter notebook interface one obtains an actual animation, here we only provide a final frame image.

The second feature is the functionality of plotting a $y$-intercept point at each animation frame. This is done like in the pseudo-code:

```
if frame = 0 and x_values_array[frame] = 0
    intercept_frame := 0
    set_point(0, y_values_array[intercept_frame])
else:
    if len(x_values_array) - frame + frame_step) < 0
        frame_step := len(x_values) - frame
        frame := frame + frame_step - 1
    if x_values_array[max(frame-frame_step, 0)]*x_values_array[frame] <= 0
        intercept_frame := argmax(x_values_array[max(frame-frame_step, 0)
```

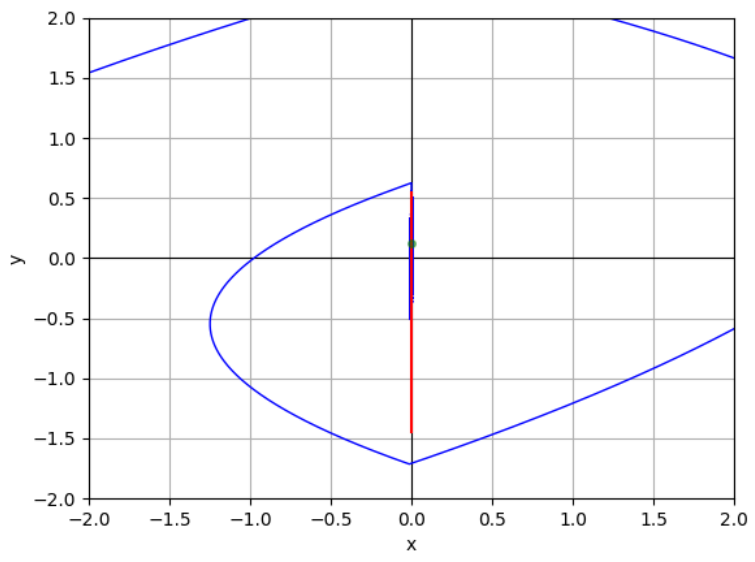


Figure 6: Trajectory visualization

```
        to frame+1] = 0) + max(frame−frame_step, 0)
    if intercept_frame is not None:
        set_point(0, y_values_array[intercept_frame])
    else:
        set_point(None)
```

## 3.3 Diagrams

In this subsection we discuss the underlying functionality of the diagram creation. The main functions for this purpose are: `create_diagram_hyst_theoretical`, `create_diagram_hyst_numerical` and `create_diagram_hyst`. The names of the first two stand for themselves, while the last one combines the functionality of the two by allowing options of plotting one of them or both at the same time. We focus on the first two functions.

In the `create_diagram_hyst_theoretical` previously analytically derived formulas are used to depict the states of the system. The overall functionality is fairly simple and is implemented like in the pseudocode:

```
if (tau/2 − a/b) > 0
    compute lower & upper branches
else:
    compute one stable branch
```

Of course the values used in the computation depend on which $f$ function is desired.

A more interesting implementation is in `create_diagram_hyst_numerical`. The function receives a list of initial conditions for $y$ and list of values for $\mu$ for which we would like to obtain the fixed points. Firstly, the approximation for fixed points are computed. It is much more efficient to use parallelization techniques for these computations, as they can take a substantial amount of time:

```
for mu in mu_array and y0 in y0_array:
    results := Parallelized(compute_y_intercept)(mu, y0)
```

The additional functions used in this computation are as in the pseudocode:

```
def find_first_zero_index(x_array)
    for i from 1 to len(x_array)+1
        if x_values[-i] = 0:
            return i
    return len(x_values) - 1

def compute_y_intercept():
    x0 := 0
    x_array, y_array := Runge_Kutta()
    y_index := find_first_zero_index(x_array)

    if y_array[-y_index] >= 0
        y_intercept := y_values[-y_index]
    else
        y_index_new := find_first_zero_index(x_values[:-y_index])
        y_intercept := y_values[:-y_index][-y_index_new]

    return y_intercept
```

Afterwards is it required to determine the region where the instable branch lives (if it exists in the first place). That is done like so:

```
for mu in mu_array
    diff := difference(results_grouped_by_mu['mu'])
    if F_type is 'closed'
        upper_bound := compute_slide_topedge_max()
    elif F_type is 'open'
        upper_bound := 2*b
    if any(diff > upper_bound) and (mu_unstable_left is None)
        mu_unstable_left := mu
        y_max_left := max(results_grouped_by_mu['mu_unstable_left'])
    elif all(grouped_results[mu]) > upper_bound and
    (mu_unstable_right is None)
        mu_unstable_right := mu
        y_max_right := min(results_grouped_by_mu['mu_unstable_right'])
        break
```

Note that the upper limiting value that determines whether an unstable branch was born is chosen depending on $F$ and, in fact, might require a more careful consideration in future.

After identifying the unstable branch range we compute some trajectories within this range and for initial conditions in $y$ within the height of the branch for each $\mu$. That is:

```
for mu in mu_unstable_array and
y0 between 0 & maximal_fixed_point(mu) with step:=small_step
    unstable_results := Parallelized(compute_y_intercept)(mu, y0)
```

Now the technique for identifying the approximate value of the fixed point corresponding to the unstable state is the following: since we chose $y_0$ evenly for each $\mu$ in the unstable range, there must be two consecutive $y_0$ values such that their trajectories drift apart to the base state and the newly born stable state each. Hence, the unstable fixed point is approximately the mean of those two $y_0$ values. This is implemented like so:

```
for mu in unstable_mu_array
    unstable_diff = difference(unstable_results_grouped_by_mu['mu'])
    if any(unstable_diff > 0.5*maximal_fixed_point(mu):
        max_diff_idx := argmax(unstable_diff >
        max(unstable_results_grouped_by_mu['mu'])/2)
        unstable_y_array.append(mean(y0_array(max_diff_idx)))
```

It is only left to plot those values accordingly.

In addition, there is a function `create_diagram_point` that does not account for the unstable branch and only plots the stable states. This can also be useful for some initial graphs where the stability loss complexity is not yet needed.

# Bibliography

## Primary Literature

Astrom, Karl Johan, and Lars Rundqwist. "Integrator Windup and How to Avoid It." In *1989 American Control Conference*. 1989. https://doi.org/10.23919/ACC.1989.4790464.

Dornhein, M. A. "Report pinpoints factors leading to YF-22 crash." *Aviation Week & Space Technology* 9 (1992).

Pogromsky, Alexander Yu, and Alexey S Matveev. "A non-quadratic criterion for stability of forced oscillations." *Systems & Control Letters* 62, no. 5 (2013).

R.A. van den Berg, G.A. Leonov & J.E. Rooda, A.Yu. Pogromsky. "Design of Convergent Switched Systems." *Springer*, 2006.

Shifrin, C. A. "Sweden seeks cause of Gripen crash." *Aviation Week and Space Technology* 139, no. 7 (1993).

Zifeng Yang, Mathew Martin, Hirofumi Igarashi, and Hui Hu. "An Experimental Investigation on Aerodynamic Hysteresis of a Low-Reynolds Number Airfoil." Preprint, available online, *Springer*, 2008. https://doi.org/10.2514/6.2008-315.

# A Core Code Listings

## A.1 functions.py

```
# Mathematical function definitions
import numpy as np


def F_closed(x):
    if x < 0:
        return -1
    if x > 0:
        return 1
    else:
        return 0


def F_open(x):
    if x < -2:
        return -1
    if x > 2:
        return 1
    else:
        return x/2


def f_meand(t, mu, tau):
    if 0 <= t % (2*tau) < tau:
        return mu
    if tau <= t % (2*tau) <= 2*tau:
        return -mu


def sin(t, mu, tau):
    return mu*np.sin(np.pi*t / tau)


def dx_dt(t, x, y, a, F, f, mu, tau):
    return y - a*(F(x) + f(t, mu, tau))


def dy_dt(t, x, y, b, F, f, mu, tau):
    return -b*(F(x) + f(t, mu, tau))
```

## A.2 analytical.py

```
# Derived formulas
import functions as fn
```

```python
import numpy as np


# Compute mu0 (roots t0 born)
def compute_mu0(a, b, tau, f_type):
    if f_type == 'meand':
        mu0 = (4*a*b*tau)/(b**2 * tau**2 + 4*a**2)
    elif f_type == 'sin':
        mu0 = (a*np.pi/b)/(2*np.sqrt((tau/np.pi)**2 + (a/b)**2))

    return mu0

# Compute mu1 (in case of sin value when one root remains)
def compute_mu1(a, b, tau):
    mu1 = compute_mu0(a, b, tau, 'sin') * np.sqrt(1 + ((a/b - tau
      ↪ /2)/(a*np.pi/(2*b)))**2)
    return mu1

# Compute t_critical (roots t0 have to be > than t critical)
def compute_tcr(mu, a, b, tau, f_type):
    if f_type == 'meand':
        tcr = (1/mu)*(a/b - tau/2) + tau/2 - a/b

    elif f_type == 'sin':
        R = mu*np.sqrt((tau/np.pi)**2 + (a/b)**2)
        C = a/b - tau/2
        if np.abs(C/R) < 1:
            alpha = np.arctan((b*tau)/(a*np.pi))
            tcr1 = (np.arcsin(C/R) + alpha) * (tau/np.pi)
            tcr2 = (np.pi - np.arcsin(C/R) + alpha) * (tau/np.pi)
            tcr = [tcr1, tcr2]
        else:
            tcr = []
    return tcr


# Compute roots t0 (time when f(t) changes its sign)
def compute_roots_t0(mu, a, b, tau, f_type, tcr_filter=True):
    if f_type == 'meand':
        discrim = (2*a*mu - b*mu*tau)**2 - 4*b*mu*(a - a*mu)*tau
        root1 = ((b*mu*tau - 2*a*mu) - np.sqrt(discrim)) / (2*b*mu)
        root2 = ((b*mu*tau - 2*a*mu) + np.sqrt(discrim)) / (2*b*mu)
        solutions = np.array([root1, root2], dtype=float)
        if tcr_filter:
            solutions = solutions[(solutions > compute_tcr(mu, a, b
              ↪ , tau, f_type))]
        return solutions
    elif f_type == 'sin':
        R = np.sqrt((b*mu*tau**2)**2 + (a*mu*tau*np.pi)**2)
        C = (a*tau*np.pi**2)/2
        if np.abs(C/R) <= 1 + 1e-8:
            alpha = np.arctan((a*np.pi)/(b*tau))
            root1 = (np.arcsin(compute_mu0(a, b, tau, 'sin')/mu) -
               ↪ alpha) * (tau/np.pi)
            root2 = (np.pi - np.arcsin(compute_mu0(a, b, tau, 'sin'
               ↪ )/mu) - alpha) * (tau/np.pi)
            solutions = np.array([root1, root2], dtype=float)
            if tcr_filter:
                if compute_tcr(mu, a, b, tau, f_type) != []:
                    solutions = solutions[(solutions > compute_tcr(
                      ↪ mu, a, b, tau, f_type)[0]) & (solutions <
                      ↪ compute_tcr(mu, a, b, tau, f_type)[1])]
                else:
                    pass
        else:
            solutions = []
        return solutions

# Compute y0 (y-intercept of attracting cycle)
def compute_y0(t0, mu, b, tau, f_type):
    if f_type == 'meand':
        y0 = b*mu*t0 + (b - b*mu)*tau/2
    elif f_type == 'sin':
        y0 = (b*tau)/2 - np.cos(np.pi*t0/tau)*(b*mu*tau)/np.pi

    return y0


def compute_slide_window(a, mu):
    dist = a*(mu - 1) - a*(-mu + 1)
    return dist


def compute_slide_topedge(t, a, mu, tau, f_type):
    f  = None

    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin

    point = a*(f(t, mu, tau) + 1)

    return point
```

```
def compute_slide_botedge(t, a, mu, tau, f_type):
    f  = None

    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin

    point = a*(f(t, mu, tau) - 1)

    return point


def compute_slide_topedge_max(a, mu):
    point = a*(mu + 1)

    return point

def compute_slide_botedge_max(a, mu):
    point = a*(-mu - 1)

    return point
```

## A.3 analytical.py

```
# Derived formulas
import functions as fn
import numpy as np


# Compute mu0 (roots t0 born)
def compute_mu0(a, b, tau, f_type):
    if f_type == 'meand':
        mu0 = (4*a*b*tau)/(b**2 * tau**2 + 4*a**2)
    elif f_type == 'sin':
        mu0 = (a*np.pi/b)/(2*np.sqrt((tau/np.pi)**2 + (a/b)**2))

    return mu0

# Compute mu1 (in case of sin value when one root remains)
def compute_mu1(a, b, tau):
    mu1 = compute_mu0(a, b, tau, 'sin') * np.sqrt(1 + ((a/b - tau
      ↪ /2)/(a*np.pi/(2*b)))**2)
    return mu1

# Compute t_critical (roots t0 have to be > than t critical)
def compute_tcr(mu, a, b, tau, f_type):
    if f_type == 'meand':
        tcr = (1/mu)*(a/b - tau/2) + tau/2 - a/b

    elif f_type == 'sin':
        R = mu*np.sqrt((tau/np.pi)**2 + (a/b)**2)
        C = a/b - tau/2
        if np.abs(C/R) < 1:
            alpha = np.arctan((b*tau)/(a*np.pi))
            tcr1 = (np.arcsin(C/R) + alpha) * (tau/np.pi)
            tcr2 = (np.pi - np.arcsin(C/R) + alpha) * (tau/np.pi)
            tcr = [tcr1, tcr2]
        else:
            tcr = []
    return tcr


# Compute roots t0 (time when f(t) changes its sign)
def compute_roots_t0(mu, a, b, tau, f_type, tcr_filter=True):
    if f_type == 'meand':
        discrim = (2*a*mu - b*mu*tau)**2 - 4*b*mu*(a - a*mu)*tau
        root1 = ((b*mu*tau - 2*a*mu) - np.sqrt(discrim)) / (2*b*mu)
        root2 = ((b*mu*tau - 2*a*mu) + np.sqrt(discrim)) / (2*b*mu)
        solutions = np.array([root1, root2], dtype=float)
        if tcr_filter:
            solutions = solutions[(solutions > compute_tcr(mu, a, b
              ↪ , tau, f_type))]
        return solutions

    elif f_type == 'sin':
        R = np.sqrt((b*mu*tau**2)**2 + (a*mu*tau*np.pi)**2)
        C = (a*tau*np.pi**2)/2
        if np.abs(C/R) <= 1 + 1e-8:
            alpha = np.arctan((a*np.pi)/(b*tau))
            root1 = (np.arcsin(compute_mu0(a, b, tau, 'sin')/mu) -
              ↪ alpha) * (tau/np.pi)
            root2 = (np.pi - np.arcsin(compute_mu0(a, b, tau, 'sin'
              ↪ )/mu) - alpha) * (tau/np.pi)
            solutions = np.array([root1, root2], dtype=float)
            if tcr_filter:
                if compute_tcr(mu, a, b, tau, f_type) != []:
                    solutions = solutions[(solutions > compute_tcr(
                      ↪ mu, a, b, tau, f_type)[0]) & (solutions <
                      ↪ compute_tcr(mu, a, b, tau, f_type)[1])]
                else:
                    pass
        else:
            solutions = []
        return solutions
```

```python
# Compute y0 (y-intercept of attracting cycle)
def compute_y0(t0, mu, b, tau, f_type):
    if f_type == 'meand':
        y0 = b*mu*t0 + (b - b*mu)*tau/2
    elif f_type == 'sin':
        y0 = (b*tau)/2 - np.cos(np.pi*t0/tau)*(b*mu*tau)/np.pi

    return y0


def compute_slide_window(a, mu):
    dist = a*(mu - 1) - a*(-mu + 1)
    return dist


def compute_slide_topedge(t, a, mu, tau, f_type):
    f   = None

    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin

    point = a*(f(t, mu, tau) + 1)

    return point

def compute_slide_botedge(t, a, mu, tau, f_type):
    f   = None

    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin

    point = a*(f(t, mu, tau) - 1)

    return point


def compute_slide_topedge_max(a, mu):
    point = a*(mu + 1)

    return point

def compute_slide_botedge_max(a, mu):
    point = a*(-mu - 1)
    return point
```

## A.4 num_methods.py

```
# Numerical Algortihms
import numpy as np
import functions as fn
import analytical as an


class NoConverge(Exception):
    """Custom exception for something specific."""
    pass


def euler(fx, fy, t0, x0, y0, h, t_end):
    t_values = [t0]
    x_values = [x0]
    y_values = [y0]

    t = t0
    x = x0
    y = y0

    while t <= t_end:
        x_new = x + h * fx(t, x, y)
        y_new = y + h * fy(t, x, y)

        t += h

        t_values.append(t)
        x_values.append(x_new)
        y_values.append(y_new)

        x = x_new
        y = y_new

    return np.array(t_values), np.array(x_values), np.array(
      ↪ y_values)


def euler_slide(fx, fy, t0, x0, y0, h, t_end):
    t_values = [t0]
    x_values = [x0]
    y_values = [y0]
    slide_zone = []

    t = t0
    x = x0
    y = y0

    while t <= t_end:
        if (x == 0) and (fx(t, 10e-5, y) < 0) and (fx(t, -10e-5, y)
          ↪ > 0):
            if limit_check(fx, t, y):
                x_new = 0
                y_new = (fy(t, 10e-5, y) + fy(t, -10e-5, y))/2
            else:
                raise(KeyError)
        else:
            x_new = x + h * fx(t, x, y)
            y_new = y + h * fy(t, x, y)

        if x*x_new < 0:
            h_tilda = -x/fx(t, x, y)
            x_new = 0
            y_new = y + h_tilda*fy(t, x, y)
            t -= h - h_tilda

        slide_zone.append(((fx(t, -10e-5, y)-y)*(-1), (fx(t, 10e-5,
          ↪ y)-y)*(-1)))

        t += h

        t_values.append(t)
        x_values.append(x_new)
        y_values.append(y_new)

        x = x_new
        y = y_new

    slide_zone.append(0)
    return np.array(t_values), np.array(x_values), np.array(
      ↪ y_values), slide_zone


def runge_kutta(fx, fy, t0, x0, y0, h, t_end):
    t_values = [t0]
    x_values = [x0]
    y_values = [y0]

    t = t0
    x = x0
    y = y0

    while t <= t_end:
        k1_x = fx(t, x, y)
        k2_x = fx(t + h/2, x + h/2 * k1_x, y)
        k3_x = fx(t + h/2, x + h/2 * k2_x, y)
        k4_x = fx(t + h, x + h * k3_x, y)

        k1_y = fy(t, x, y)
        k2_y = fy(t + h/2, x, y + h/2 * k1_y)
        k3_y = fy(t + h/2, x, y + h/2 * k2_y)
        k4_y = fy(t + h, x, y + h * k3_y)

        x_new = x + h/6 * (k1_x + 2*k2_x + 2*k3_x + k4_x)
        y_new = y + h/6 * (k1_y + 2*k2_y + 2*k3_y + k4_y)

        t += h

        t_values.append(t)
        x_values.append(x_new)
        y_values.append(y_new)

        x = x_new
        y = y_new

    return np.array(t_values), np.array(x_values), np.array(
      ↪ y_values)


def runge_kutta_slide(fx, fy, t0, x0, y0, h, t_end, F_type, f_type,
  ↪  a, b, mu, tau):
    t_values = [t0]
    x_values = [x0]
    y_values = [y0]

    F, f  = None, None

    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin
    else:
        raise NameError

    if F_type == 'closed':
        F = fn.F_closed
    elif F_type == 'open':
        F = fn.F_open
    elif F_type == 'tanh':
        F = np.tanh
    else:
        raise NameError

    t = t0
    x = x0
    y = y0


    while t < t_end:
        if F_type == 'closed':
            if (x == 0) and (fx(t, 10e-5, y, a, F, f, mu, tau) < 0)
             ↪  and (fx(t, -10e-5, y, a, F, f, mu, tau) > 0):
                x_new = 0
                k1_y = fy(t, -10e-5, y, b, F, f, mu, tau)
                k2_y = fy(t + h/2, -10e-5, y + h/2 * k1_y, b, F, f,
                  ↪  mu, tau)
                k3_y = fy(t + h/2, -10e-5, y + h/2 * k2_y, b, F, f,
                  ↪  mu, tau)
                k4_y = fy(t + h, -10e-5, y + h * k3_y, b, F, f, mu,
                  ↪  tau)

                y_incr_left = h/6 * (k1_y + 2*k2_y + 2*k3_y + k4_y)

                k1_y = fy(t, 10e-5, y, b, F, f, mu, tau)
                k2_y = fy(t + h/2, 10e-5, y + h/2 * k1_y, b, F, f,
                  ↪ mu, tau)
                k3_y = fy(t + h/2, 10e-5, y + h/2 * k2_y, b, F, f,
                  ↪ mu, tau)
                k4_y = fy(t + h, 10e-5, y + h * k3_y, b, F, f, mu,
                  ↪ tau)

                y_incr_right = h/6 * (k1_y + 2*k2_y + 2*k3_y + k4_y
                  ↪ )
                if ((y_incr_left + y_incr_right) / 2) < an.
                  ↪ compute_slide_botedge(t, a, mu, tau, f_type):
                    y_new = an.compute_slide_botedge(t, a, mu, tau,
                      ↪  f_type) - 10e-5
                    h_coef = np.abs((y_new - y)/(((y_incr_left +
                      ↪ y_incr_right) / 2) - y))
                    h_tilda = h*h_coef
                    t -= h - h_tilda
                elif ((y_incr_left + y_incr_right) / 2) > an.
                  ↪ compute_slide_topedge(t, a, mu, tau, f_type):
                    y_new = an.compute_slide_topedge(t, a, mu, tau,
                      ↪  f_type) + 10e-5
                    h_coef = np.abs((y_new - y)/(((y_incr_left +
                      ↪ y_incr_right) / 2) - y))
                    h_tilda = h*h_coef
                    t -= h - h_tilda
        else:
            y_new = y + (y_incr_left + y_incr_right) / 2
    else:
        k1_x = fx(t, x, y, a, F, f, mu, tau)
        k2_x = fx(t + h/2, x + h/2 * k1_x, y, a, F, f, mu,
          ↪ tau)
        k3_x = fx(t + h/2, x + h/2 * k2_x, y, a, F, f, mu,
          ↪ tau)
        k4_x = fx(t + h, x + h * k3_x, y, a, F, f, mu, tau)

        k1_y = fy(t, x, y, b, F, f, mu, tau)
        k2_y = fy(t + h/2, x, y + h/2 * k1_y, b, F, f, mu,
          ↪ tau)
        k3_y = fy(t + h/2, x, y + h/2 * k2_y, b, F, f, mu,
          ↪ tau)
        k4_y = fy(t + h, x, y + h * k3_y, b, F, f, mu, tau)

        x_new = x + h/6 * (k1_x + 2*k2_x + 2*k3_x + k4_x)
        y_new = y + h/6 * (k1_y + 2*k2_y + 2*k3_y + k4_y)

else:
    k1_x = fx(t, x, y, a, F, f, mu, tau)
    k2_x = fx(t + h/2, x + h/2 * k1_x, y, a, F, f, mu, tau)
    k3_x = fx(t + h/2, x + h/2 * k2_x, y, a, F, f, mu, tau)
    k4_x = fx(t + h, x + h * k3_x, y, a, F, f, mu, tau)

    k1_y = fy(t, x, y, b, F, f, mu, tau)
    k2_y = fy(t + h/2, x, y + h/2 * k1_y, b, F, f, mu, tau)
    k3_y = fy(t + h/2, x, y + h/2 * k2_y, b, F, f, mu, tau)
    k4_y = fy(t + h, x, y + h * k3_y, b, F, f, mu, tau)

    x_new = x + h/6 * (k1_x + 2*k2_x + 2*k3_x + k4_x)
    y_new = y + h/6 * (k1_y + 2*k2_y + 2*k3_y + k4_y)

if x*x_new < 0:
    slope =(y_new - y)/(x_new - x)
    y_tilda = slope*(-x)+y
    h_coef = np.sqrt((0-x)**2+(y_tilda-y)**2)/np.sqrt((
      ↪ x_new-x)**2+(y_new-y)**2)
    h_tilda = h*h_coef
    x_new = 0

    y_new = y_tilda
    print(y_new)

    t -= h - h_tilda
```

```
        t += h

        t_values.append(t)
        x_values.append(x_new)
        y_values.append(y_new)

        x = x_new
        y = y_new


    return np.array(t_values), np.array(x_values), np.array(
    ↪ y_values)
```

## A.5 plotting.py

```
# Visualization functionality
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import functions as fn
import analytical as an


def plot_basic(x_values, y_values):
    fig, ax = plt.subplots()
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('Euler␣Method␣with␣Sliding␣Zone')

    ax.axhline(0, color='black', linewidth=0.8)  # x-axis
    ax.axvline(0, color='black', linewidth=0.8)  # y-axis

    # Plot the continuous line and dashed segments
    ax.plot(x_values, y_values)

    plt.grid()
    plt.show()


def plot_anim(t_values, x_values, y_values, title, frame_step, xlim
↪ = None, ylim = None, include_y_intecept=False, include_slide=
↪ False, vec_field=False,
                normed=True, a=None, b=None, mu=None, tau=None,
                ↪ f_type=None, F_type=None):
    # Set up the figure and axis
    fig, ax = plt.subplots()
    ax.set_xlabel('x')
    ax.set_ylabel('y')
ax.set_title(title)

# Plot the x and y axes
ax.axhline(0, color='black', linewidth=0.8)  # x-axis
ax.axvline(0, color='black', linewidth=0.8)  # y-axis

x = np.linspace(-10, 10, 10)
y = np.linspace(-2, 2, 4)
X, Y = np.meshgrid(x, y)

# Initialize the line object
line, = ax.plot([], [], 'b', lw=1)
slide_zone, = ax.plot([], [], 'r')
y_intercept, = ax.plot([], [], 'go', markersize=4, alpha=0.5)
if vec_field:
    if f_type == 'meand':
        f = fn.f_meand
    elif f_type == 'sin':
        f = fn.sin
    if F_type == 'closed':
        F = fn.F_closed
    elif F_type == 'open':
        F = fn.F_open
    dx_dt = lambda t, x, y: fn.dx_dt(t, x, y, a, F, f)
    dy_dt = lambda t, x, y: fn.dy_dt(t, x, y, b, F, f)
    quiver = ax.quiver(X, Y, np.zeros_like(X), np.zeros_like(Y)
      ↪ , scale=25, width=0.004, color='purple')

# Set limits based on data
x_left = np.min(x_values) if xlim == None else xlim[0]
x_right = np.max(x_values) if xlim == None else xlim[1]
y_left = np.min(y_values) if ylim == None else ylim[0]
y_right = np.max(y_values) if ylim == None else ylim[1]

ax.set_xlim(x_left, x_right)
ax.set_ylim(y_left, y_right)


# Initialization function for the animation
def init():
    line.set_data([], [])
    slide_zone.set_data([], [])
    y_intercept.set_data([], [])
    return line, slide_zone, y_intercept

intercept_frame = None

# Update function for the animation
```

```
def update(frame):
    # Update line data up to the current frame
    line.set_data(x_values[:frame], y_values[:frame])

    nonlocal intercept_frame
    nonlocal frame_step

    if vec_field:
        dx_dt_v = np.vectorize(dx_dt)
        dy_dt_v = np.vectorize(dy_dt)

        U = dx_dt_v(t_values[frame], X, Y)
        V = dy_dt_v(t_values[frame], X, Y)
        magnitude = np.sqrt(U**2 + V**2)
        if normed:
            U = U / magnitude
            V = V / magnitude
        quiver.set_UVC(U, V)

    res_list = [line]

    if include_slide:
        slide_zone.set_data([0,0], [an.compute_slide_botedge(
          ↪ t_values[frame], a, mu, tau, f_type),
                                    an.compute_slide_topedge(
                                      ↪ t_values[frame], a, mu
                                      ↪ , tau, f_type)])
        res_list.append(slide_zone)

    if vec_field:
        res_list.append(quiver)

    if include_y_intecept:
        if (frame == 0) and (x_values[frame] == 0):
            intercept_frame = 0
            y_intercept.set_data([0], [y_values[intercept_frame
              ↪ ]])
        else:
            if (len(x_values) - (frame+frame_step)) < 0:
                frame_step = len(x_values) - frame
                frame = frame + (frame_step) - 1
            if (x_values[np.max([frame-frame_step, 0])]*
              ↪ x_values[frame] <= 0):
                intercept_frame = np.argmax(x_values[np.max([
                  ↪ frame-frame_step, 0]):frame+1] == 0) + np.
                  ↪ max([frame-frame_step, 0])
            if intercept_frame is not None:
                y_intercept.set_data([0], [y_values[
                      ↪ intercept_frame]])
                else:
                    y_intercept.set_data([], [])
            res_list.append(y_intercept)

        return res_list

    # Create the animation
    anim = FuncAnimation(fig, update, frames=range(0, len(t_values)
      ↪ , frame_step), init_func=init, interval=10, blit=True)
    x = np.arange(-10, 10)
    y = np.arange(-10, 10)

    plt.grid()
    plt.close(fig)

    return anim
```

## A.6 toolset.py

```
# Diagram creation functionality
import numpy as np
import matplotlib.pyplot as plt
import functions as fn
import analytical as an
import num_methods as nm
from joblib import Parallel, delayed, parallel_backend
from collections import defaultdict


def find_first_zero_index(x_values):
    for i in range(1, len(x_values)+1):
        if x_values[-i] == 0:
            return i
    return len(x_values) - 1


def compute_y_intercept(t0, y0, mu, a, b, tau, F_type, f_type, h,
  ↪ t_end):
    x0 = 0
    _, x_values, y_values = nm.runge_kutta_slide(fn.dx_dt, fn.dy_dt
      ↪ , t0, x0, y0, h, t_end,
                                                       F_type, f_type,
                                                         ↪ a, b, mu,
                                                         ↪ tau)
    y_index = find_first_zero_index(x_values)

    if y_values[-y_index] >= 0:
        y_intercept = y_values[-y_index]
    else:
        y_index_new = find_first_zero_index(x_values[:-y_index])
        y_intercept = y_values[:-y_index][-y_index_new]

    return y_intercept


def create_diagram_point(t0_list, y0_list, mu_list, a, b, tau, F, f
    ↪ , h, t_end):
    num_plots = len(t0_list)
    ncols = 2 if num_plots > 1 else 1
    nrows = (num_plots + 1) // 2

    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(10,
      ↪ 6))
    fig.suptitle(f"Hysteresis␣diagram␣with␣F␣=␣{F},␣f␣=␣{f}",
      ↪ fontsize=16)
    if num_plots == 1:
        ax = np.array([ax])
    else:
        ax = np.ravel(ax)

    for i in range(len(t0_list)):
        with parallel_backend("loky", inner_max_num_threads=1):   #
          ↪ More efficient threading
            results = Parallel(n_jobs=-1, batch_size=20)(
                delayed(compute_y_intercept)(t0_list[i], y0, mu, a,
                  ↪ b, tau, F, f, h, t_end)
                for mu in mu_list
                for y0 in y0_list
            )

        mu_array = np.repeat(mu_list, len(y0_list))
        y_points = np.array(results)

        sc = ax[i].scatter(mu_array, y_points, c=np.tile(y0_list,
          ↪ len(mu_list)), cmap='viridis', alpha=0.5)
        fig.colorbar(sc, ax=ax[i], label=r'$y_0$␣values')   # Attach
          ↪ colorbar to the subplot
        ax[i].plot(mu_list, np.zeros(len(mu_list)), 'k--')
        ax[i].set_title(f"t0␣=␣{t0_list[i]}")
        ax[i].grid()

    plt.tight_layout()
    plt.show()
```

```
def create_diagram_hyst_theoretical(ax, mu_list_max, a, b, tau,
  ↪ f_type, alpha=0.8):
    mu_list = np.arange(0, mu_list_max+0.01, 0.01)
    mu_list = np.append(mu_list, [an.compute_mu0(a, b, tau, f_type)
      ↪ ])
    mu_list = np.sort(mu_list)
    m0 = None
    m1 = None
    if f_type == 'meand':
        m0 = an.compute_mu0(a, b, tau, f_type)
        m1 = 1
    if f_type == 'sin':
        m0 = an.compute_mu0(a, b, tau, f_type)
        m1 = an.compute_mu1(a, b, tau)
    if (tau/2 - a/b) > 0:
        solutions_t0 = {float(mu) : an.compute_roots_t0(mu, a, b,
          ↪ tau, f_type) for mu in mu_list}
        solutions_y0_smaller = {float(mu) : np.min([an.compute_y0(t
          ↪ , mu, b, tau, f_type) for t in solutions_t0[mu]]) for
          ↪ mu in mu_list[(mu_list >= m0) & (mu_list < m1)]}
        solutions_y0_greater = {float(mu) : np.max([an.compute_y0(t
          ↪ , mu, b, tau, f_type) for t in solutions_t0[mu]]) for
          ↪ mu in mu_list[(mu_list > m0)]}
        mu_plot = list(solutions_y0_smaller.keys())[::-1]
        mu_plot += list(solutions_y0_greater.keys())
        y_plot = list(solutions_y0_smaller.values())[::-1] + list(
          ↪ solutions_y0_greater.values())
    else:
        solutions_t0 = {float(mu) : an.compute_roots_t0(mu, a, b,
          ↪ tau, f_type)[0] for mu in mu_list[mu_list > m1]}
        solutions_y0 = [an.compute_y0(t, mu, b, tau, f_type) for t,
          ↪  mu in zip(solutions_t0.values(), solutions_t0.keys())
          ↪ ]
        mu_plot = mu_list[mu_list > m1]
        y_plot = solutions_y0

    ax.plot(mu_plot, y_plot, c='blue', alpha=alpha, linewidth=2.5,
      ↪ label='Theoretical')
    ax.legend()

    ax.set_xlabel(r'Amplitude␣$\mu$')
    ax.set_ylabel(r'Fixed␣point␣$y_{fixed}$')


def create_diagram_hyst_numerical(ax, t0, y0_list, mu_list, a, b,
  ↪ tau, F_type, f_type, h, t_end, coloring=False):
    mu_unstable_left = None
    ind_unstable_left = 0
```

```
y_max_left = 0

mu_unstable_right = None
ind_unstable_right = 0
y_max_right = 0
if (t0 % (2*tau)) < tau:
    with parallel_backend("loky", inner_max_num_threads=1):  #
     ↪ More efficient threading
        results_flat = Parallel(n_jobs=-1, batch_size=10)(
            delayed(compute_y_intercept)(t0, y0, mu, a, b, tau,
              ↪ F_type, f_type, h, t_end)
            for mu in mu_list
            for y0 in y0_list
        )
else:
    with parallel_backend("loky", inner_max_num_threads=1):
        results_flat = Parallel(n_jobs=-1, batch_size=10)(
            delayed(compute_y_intercept)(t0, y0, mu, a, b, tau,
              ↪ F_type, f_type, h, t_end)
            for mu in mu_list
            for y0 in -y0_list
        )

# Group results by mu after computing
grouped_results = defaultdict(list)
idx = 0
for mu in mu_list:
    for _ in y0_list:
        grouped_results[mu].append(results_flat[idx])
        idx += 1

for j, mu in enumerate(mu_list):
    diff = np.diff(grouped_results[mu])
    upper_bound = None
    if F_type == 'closed':
        upper_bound = an.compute_slide_topedge_max(a, mu)
    elif F_type == 'open':
        upper_bound = 2*b
    elif F_type == 'tanh':
        upper_bound = 2*b
    if (np.any(diff > upper_bound) and (mu_unstable_left ==
     ↪ None)):
        mu_unstable_left = mu
        ind_unstable_left = j
        y_max_left = np.max(grouped_results[mu_unstable_left])
    elif (np.all(np.array(grouped_results[mu], dtype=float) >
     ↪ upper_bound) and (mu_unstable_right == None)):
        mu_unstable_right = mu_list[j]
```

```
            ind_unstable_right = j
            y_max_right = np.min(grouped_results[mu_unstable_right
              ↪ ])
            break

    nbh = 0.1
    unstable_mu_range = mu_list[ind_unstable_left + 1:
      ↪ ind_unstable_right]
    if (t0 % (2*tau)) < tau:
        unstable_y0_grouped = {mu : np.arange(0, -np.max(
          ↪ grouped_results[mu]), -nbh) for mu in
          ↪ unstable_mu_range}
    else:
        unstable_y0_grouped = {mu : np.arange(0, np.max(
          ↪ grouped_results[mu]), nbh) for mu in unstable_mu_range
          ↪ }
    unstable_y_list = []

    with parallel_backend("loky", inner_max_num_threads=1):
        unstable_results_flat = Parallel(n_jobs=-1, batch_size=10)(
            delayed(compute_y_intercept)(np.min(an.compute_roots_t0
              ↪ (mu, a, b, tau, f_type, tcr_filter=False)), y0, mu
              ↪ , a, b, tau, F_type, f_type, h, t_end)
            for mu in unstable_mu_range
            for y0 in unstable_y0_grouped[mu]
        )

    grouped_unstable_results = defaultdict(list)
    idx = 0
    for mu in unstable_mu_range:
        for _ in unstable_y0_grouped[mu]:
            grouped_unstable_results[mu].append(
              ↪ unstable_results_flat[idx])
            idx += 1

    for mu in unstable_mu_range:
        unstable_diff = np.diff(grouped_unstable_results[mu])
        if np.any(unstable_diff > np.abs((unstable_y0_grouped[mu
          ↪ ][-1]/2))):
            max_diff_idx = np.argmax(unstable_diff > (np.max(
              ↪ grouped_unstable_results[mu])) / 2)
            if (t0 % (2*tau)) < tau:
                unstable_y_list.append(-(unstable_y0_grouped[mu][
                  ↪ max_diff_idx + 1] + unstable_y0_grouped[mu][
                  ↪ max_diff_idx]) / 2)
            else:
                unstable_y_list.append((unstable_y0_grouped[mu][
                  ↪ max_diff_idx + 1] + unstable_y0_grouped[mu][
                ↪ max_diff_idx]) / 2)


mu_array = np.repeat(mu_list, len(y0_list))
y_points = np.array(results_flat)
print(unstable_mu_range, unstable_y_list)
if coloring:
    sc = ax.scatter(mu_array, y_points, c=np.tile(y0_list, len(
      ↪ mu_list)), cmap='viridis', alpha=0.5, label='Stable␣
      ↪ numerical')
else:
    sc = ax.scatter(mu_array, y_points, c='k', alpha=0.5, label
      ↪ ='Stable␣numerical')
if unstable_mu_range.size != 0:
    ax.scatter(unstable_mu_range, unstable_y_list, color='red',
      ↪  marker='^', label='Unstable␣numerical')
    ax.vlines(mu_unstable_left, 0, y_max_left, color='red')
    ax.vlines(mu_unstable_right, 0, y_max_right, color='red')
    arrow_length = 0.1 * (y_max_left/2)
    ax.annotate(
        '',
        xy=(mu_unstable_left, y_max_left/2 + arrow_length / 2),
        xytext=(mu_unstable_left, y_max_left/2 - arrow_length /
          ↪  2),
        arrowprops=dict(
            arrowstyle='<-',
            color='red',
            linewidth=1.5,
            mutation_scale=20
        )
    )
    arrow_length = 0.1 * (y_max_right/2)
    ax.annotate(
        '',
        xy=(mu_unstable_right, y_max_right/2 + arrow_length /
          ↪ 2),
        xytext=(mu_unstable_right, y_max_right/2 - arrow_length
          ↪  / 2),
        arrowprops=dict(
            arrowstyle='->',
            color='red',
            linewidth=1.5,
            mutation_scale=20
        )
    )
ax.plot(mu_list, np.zeros(len(mu_list)), 'k--')
ax.set_title(f"t0␣=␣{t0}")
ax.set_xlabel(r'Amplitude␣$\mu$')
```

```
    ax.set_ylabel(r'Fixed␣point␣$y_{fixed}$')
    ax.legend()

    return ax, sc


def create_diagram_hyst(diag_type, t0_list=None, y0_list=None,
  ↪ mu_list=None, a=None, b=None, tau=None, F_type=None, f_type=
  ↪ None, h=None, t_end=None, coloring=False):
    if (t0_list != None) and ((diag_type == 'numerical') or (
      ↪ diag_type == 'comparison')):
        num_plots = len(t0_list)
        ncols = 2 if num_plots > 1 else 1
        nrows = (num_plots + 1) // 2

        fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize
          ↪ =(10, 6))
        fig.suptitle(f"Hysteresis␣{diag_type}␣diagram␣with␣F␣-␣{
          ↪ F_type},␣f␣-␣{f_type}", fontsize=16)
        if num_plots == 1:
            axs = np.array([axs])
        else:
            axs = np.ravel(axs)
    else:
        fig, axs = plt.subplots(nrows=1, ncols=1, figsize=(10, 6))
        fig.suptitle(f"Hysteresis␣{diag_type}␣diagram␣with␣F␣-␣{
          ↪ F_type},␣f␣-␣{f_type}", fontsize=16)

    sc = None
    if diag_type == 'numerical':
        for i in range(len(t0_list)):
            axs[i].grid()
            _, sc = create_diagram_hyst_numerical(axs[i], t0_list[i
              ↪ ], y0_list, mu_list, a, b, tau, F_type, f_type, h,
              ↪ t_end)

    elif diag_type == 'theoretical':
        axs.grid()
        create_diagram_hyst_theoretical(axs, mu_list[-1], a, b, tau
          ↪ , f_type)

    elif diag_type == 'comparison':
        for i in range(len(t0_list)):
            axs[i].grid()
            _, sc = create_diagram_hyst_numerical(axs[i], t0_list[i
              ↪ ], y0_list, mu_list, a, b, tau, F_type, f_type, h,
              ↪ t_end)
            create_diagram_hyst_theoretical(axs[i], mu_list[-1], a,
        ↪ b, tau, f_type, alpha=0.3)

if ((diag_type == 'numerical') or (diag_type == 'comparison')):
    if coloring:
        cbar_ax = fig.add_axes([0.92, 0.15, 0.02, 0.7])
        fig.colorbar(sc, cax=cbar_ax, label=r'$y_0$␣values')

plt.show()
```