UNIVERSITÀ DI TORINO

Fisica Torino

Dynamical Systems

Francesco Marchisotti

Stability Analysis of the Upper Fixed Point of Kapitza's Pendulum in the (a, σ)–plane

Professor:
Guido Boffetta

academic year 2024/2025

Contents

1 Problem statement

The Kapitza pendulum is a pendulum in which the pivot point oscillates up and down. Experimental observations show that, unlike a regular pendulum, this system can exhibit a stable fixed point in the inverted position (where the mass is directly above the pivot) in addition to the stable fixed point of the regular pendulum.

In particular, the pivot point oscillates with a frequency much greater than the characteristic frequency of oscillation of the pendulum. Let the y-coordinate of the pivot point be described as:

$$y_P = b \cos \omega t$$

and its acceleration as:

$$a_P = \frac{\mathrm{d}^2 y_P}{\mathrm{d}t^2} = -b\omega^2 \cos \omega t$$

This results in the following equation of motion for the system:

$$\ddot{\theta} - \frac{1}{l} \left(g + b\omega^2 \cos \omega t \right) \sin \theta + c\dot{\theta} = 0 \tag{1}$$

where θ denotes the angle between the pendulum's arm and the vertical direction when the system is in the upper fixed point position, as shown in Figure 1. Here, $\omega \gg \sqrt{g/l}$.

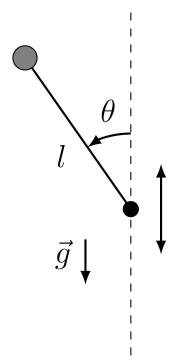

Figure 1: Kapitza's pendulum.

The equation can be rewritten highlighting the potential:

$$\ddot{\theta} = -c\dot{\theta} - \frac{\partial V}{\partial \theta}, \quad V(\theta, t) = \frac{1}{l} \left(g + b\omega^2 \cos \omega t \right) \cos \theta \tag{2}$$

The dynamics of the system feature two distinct behaviours: the fast oscillations of the pivot point and the slow oscillations of the pendulum. Given this dual nature, the multiscale method is particularly well-suited to determine the system's trajectory.

To identify the actual free parameters of the system, it is useful to eliminate the physical units from the equation. The dimensionless form is given by:

$$\ddot{\theta} - (\sigma + a\cos t')\sin\theta + \mu\dot{\theta} = 0 \tag{3}$$

where the parameters are defined as $\mu = c/\omega$, $a = b/\omega$, and $\sigma = g/l\omega^2$. To apply the multiscale method, it is assumed that the parameters scale as $\mu \sim \epsilon$, $a \sim \epsilon$ and $\sigma \sim \epsilon^2$, where ϵ is a small parameter.

By applying the multiscale method, a solvability condition similar to the original equation (Eq (2)) is obtained:

$$\ddot{\theta}_0 = -\mu\dot{\theta}_0 - \frac{\partial V_{\text{eff}}}{\partial\theta}, \quad V_{\text{eff}}(\theta) = \sigma\cos\theta - \frac{a^2}{8}\cos 2\theta$$

In this expression, the non-autonomous potential $V(\theta, t)$ is replaced by an effective autonomous potential $V_{\text{eff}}(\theta)$, which represents a time-averaged version of the original potential. Stability of the fixed point $\theta = 0$ is determined by analysing the curvature of the effective potential at $\theta = 0$:

$$\left.\frac{\partial^2 V_{\text{eff}}}{\partial\theta^2}\right|_0 = -\sigma + \frac{a^2}{2} \tag{4}$$

The fixed point is stable if the curvature is positive, leading to the condition:

$$\sigma < \frac{a^2}{2} \tag{5}$$

The primary objective of this project is to investigate the numerical stability of the upper fixed point in the (a, σ)–plane and compare the results with the predictions obtained using the multiscale method.

2 Numerical solution

Method To determine the stability of the upper fixed point, a fourth-order Runge-Kutta method was employed to evolve a trajectory with the initial condition $\theta_0 \simeq 0$. After a specified amount of dimensionless time had elapsed, the fixed point was defined as numerically stable if the trajectory remained within a prescribed tolerance of the fixed point $\theta = 0$.

This process was repeated for a range of (a, σ) values, and te results were plotted alongside the multiscale method prediction. For simplicity, the value of μ was assumed constant throughout the simulation.

Parameter range The value of θ_0 must be sufficiently close to the fixed point to lie within its attraction basin, yet sufficiently far to allow feasible numerical analysis. The value chosen was $\theta_0 = 10^{-7}$, and the angular velocity ω_0 was assumed to be 0.

As the multiscale method requires μ to be of order ϵ, a value of $\mu = 0.1$ was chosen and held constant throughout the simulations.

The parameter a, which must be of order $\epsilon \simeq 0.1$, was varied within the range $0 - 0.3$, while σ, of order $\epsilon^2 \simeq 0.01$, was varied within the range $10^{-5} - 0.02$. In this range, 30 equally spaced values were chosen for both a and σ. This resolution was sufficient to capture the required behaviour while maintaining manageable computational run-times.

Table 1 summarizes the simulation parameters.

θ_0	10^{-7}
ω_0	0
μ	0.1
a	$0 - 0.3$
σ	$10^{-5} - 0.02$

Table 1: Parameter ranges.

2.1 Time step size

The time step was selected to be sufficiently small to resolve the system's fast dynamics. Given that $\mu = c/\omega = 0.1$, it follows that $\omega \sim 1/\mu = 10$, so the period of the pivot oscillations is of order $2\pi/\omega \sim 0.6$. To ensure accuracy, the time step was set approximately one order of magnitude smaller than this period, resulting in a time step of 1×10^{-2}.

To observe the fixed point's behaviour over a sufficiently large dimensionless time, the simulation was executed for 10 000, corresponding to a final dimensionless time of $t = 100$.

2.2 Tolerance setting

The tolerance for determining the stability of the fixed point was chosen based on a physical constraint: when $a = 0$ (i.e., $b = 0$), the Kapitza pendulum reduces to a regular pendulum with a stationary pivot. In this case, the upper fixed point is always unstable.

To ensure this condition was satisfied, a simple algorithm was devised to determine a suitable tolerance. The relevant implementation can be found in the code listing starting at line 38 (Listing A.2).

The tolerance used was 1.01×10^{-7}.

2.3 Comparison with multiscale method

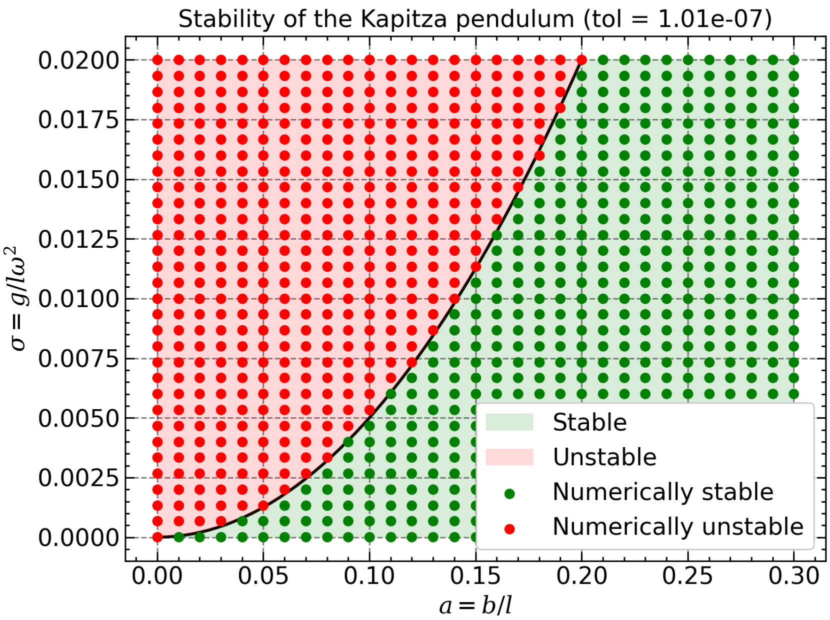

Figure 2: Stability of the upper fixed point.

The results of the simulation are shown in Figure 2. With the exception of three points along the boundary of stability $\sigma = a^2/2$ (represented by the black line), the numerical results are in consistent with the predictions of the multiscale method.

The comparison demonstrates that the perturbative multiscale method, truncated to second order in ϵ, correctly predicts the stability region, provided that the numerical simulations accurately replicate the physical behaviour of the system. It is also of interest to investigate the stability of the fixed point beyond the region of validity of the multiscale method. For instance, running simulations with the parameters shown in Table 2 reveals deviations from the multiscale prediction, as illustrated in Figure 3.

θ_0	10^{-7}
ω_0	0
μ	1
a	$0-1$
σ	$10^{-5}-0.1$

Table 2: Parameter ranges.

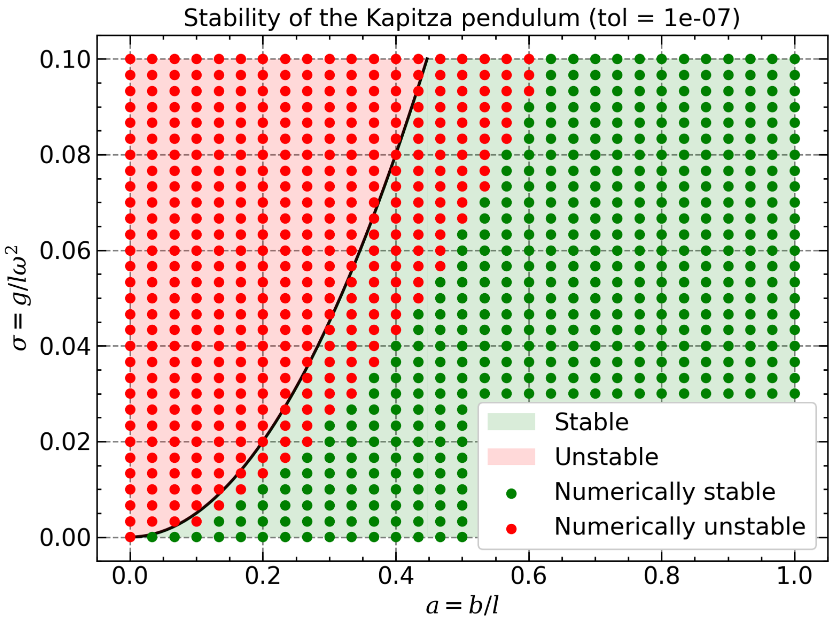

Figure 3: Stability of the upper fixed point beyond multiscale domain.

These results suggest that a higher-order multiscale approximation is required to accurately capture the physical behaviour of the Kapitza pendulum when the parameters extend beyond the validity region of the second-order perturbative solution. Alternatively, it may indicate that the system cannot be effectively studied using a perturbative method in this regime.

A Code listing

GitHub repository: https://github.com/marchfra/kapitza.git

A.1 main.cpp

```cpp
#include "/Users/francescomarchisotti/Documents/Uni/Magistrale/Algoritmi/Libs/
    include/ode_solver.hpp"

#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>

using std::cerr;
using std::cin;
using std::cout;
using std::endl;

void rangeArray(double arr[], const double &min, const double &max,
                const int &n = 1000);

void RHS(const double &t, double Y[], double R[]);

int main() {
    // Create parameter grid
    const int nPoints = 30;
    const int nStep   = 10000;

    const double minA = 0.0;
    const double maxA = 0.3;  // a of order epsilon
    double gridA[nPoints + 1];
    rangeArray(gridA, minA, maxA, nPoints);

    const double minSigma = 1.0e-5;
    const double maxSigma = 2.0e-2;  // sigma of order epsilon^2
    double gridSigma[nPoints + 1];
    rangeArray(gridSigma, minSigma, maxSigma, nPoints);

    std::ofstream stability, trajectory;
    stability.open("data/stability.csv");
    trajectory.open("data/trajectory.csv");
    if (!stability || !trajectory) {
        cerr << "Error: unable to open file" << endl;
        return 1;
    }

    // Init model parameters
```

```
    const double theta0 = 1.0e-7;
    const double omega0 = 0.0;  // omega0 is the initial angular velocity, not
                                // the frequency of oscillation of the fulcrum
    const double mu = 0.1;      // mu of order epsilon
    const int nEq   = 5;

    const double tmin = 0.0;
    const double dt =
        1.0e-2;  // with the current parameters, the fulcrum oscillates with
                 // omega of order 10, so a step size of 1.0e-2 should be
                 // enough to capture the dynamics (the period of oscillation
                 // is 2pi/omega = 0.6)

    stability << "a,sigma,endpoint" << endl;
    trajectory << "t,theta,a,sigma" << endl;
    for (int iA = 0; iA <= nPoints; iA++) {
        for (int iSigma = 0; iSigma <= nPoints; iSigma++) {
            const double a     = gridA[iA];
            const double sigma = gridSigma[iSigma];

            double Y[nEq] = {theta0, omega0, sigma, a, mu};
            double t      = tmin;

            trajectory << std::setprecision(16) << t << "," << Y[0] << "," << a
                       << "," << sigma << endl;
            for (int iStep = 0; iStep < nStep; iStep++) {
                // Integration step
                rk4Step(t, Y, RHS, dt, nEq);
                t += dt;

                // This is probably not necessary
                if (fabs(Y[0]) >= M_PI) Y[0] = fmod(Y[0], 2 * M_PI);

                trajectory << std::setprecision(16) << t << "," << Y[0] << ","
                           << a << "," << sigma << endl;
            }

            stability << std::setprecision(16) << a << "," << sigma << ","
                      << Y[0] << endl;
        }
    }

    stability.close();
    trajectory.close();

    return 0;
}

void rangeArray(double arr[], const double &min, const double &max,
```

```
                const int &n) {
    const double delta = (max - min) / n;
    for (int i = 0; i <= n; i++) arr[i] = min + i * delta;
}

void RHS(const double &t, double Y[], double R[]) {
    const double theta = Y[0];
    const double omega = Y[1];
    const double sigma = Y[2];
    const double a     = Y[3];
    const double mu    = Y[4];

    R[0] = omega;
    R[1] = (sigma + a * cos(t)) * sin(theta) - mu * omega;
    R[2] = 0.0;
    R[3] = 0.0;
    R[4] = 0.0;
}
```

A.2 plot.py

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.optimize import minimize_scalar

from my_formatter import multiple_formatter

plt.style.use(["grid", "science", "notebook", "mylegend"])

SAVE_FIGURES = True


def check_stability(a: float, sigma: float) -> bool:
    """
    Check if the fixed point is stable according to the Multiscale
    Method.
    """
    return a**2 > 2 * sigma


def preprocess_stability(tol: float = 1e-2) -> pd.DataFrame:
    """Preprocess the stability data."""

    df = pd.read_csv("data/stability.csv")

    df["stability"] = np.abs(df["endpoint"]) < tol
    df["is_stable"] = check_stability(df["a"], df["sigma"])

```

```
    return df


def num_errors(tol: float, df: pd.DataFrame) -> int:
    """Count the number of errors in the stability data."""
    df["stability"] = np.abs(df["endpoint"]) < tol
    return np.sum(df["stability"] != df["is_stable"])


def physical_tol(tol_guess: float = 1e-7, max_step: int = 1000) -> float:
    """
    Find the tolerance (to determine the numerical stability) such
    that if a = 0 the fixed point is always unstable.
    """

    df = pd.read_csv("data/stability.csv")
    filt = df["a"] == 0
    df = df[filt]

    tol = 10 * tol_guess

    df["stability"] = np.abs(df["endpoint"]) < tol

    # Take big steps until only one fixed point is stable
    while df["stability"].sum() > 1:
        tol *= 0.5
        df["stability"] = np.abs(df["endpoint"]) < tol

    if df["stability"].sum() != 1:
        raise ValueError("Failed to find the physical tolerance.")

    # Take small steps until the fixed point becomes unstable
    n_step = 0
    while df["stability"].sum() == 1:
        if n_step > max_step:
            raise ValueError("Failed to find the physical tolerance.")
        tol *= 0.999
        df["stability"] = np.abs(df["endpoint"]) < tol
        n_step += 1

    return tol, num_errors(tol, preprocess_stability(tol))


def optimize_tol(tol_guess: float = 1e-7) -> float:
    """
    Find the tolerance (to determine the numerical stability) that
    minimizes the number of errors.
    """

```

```python
    df = preprocess_stability(-1)

    optim = minimize_scalar(
        num_errors,
        args=(df,),
        bounds=(0.1 * tol_guess, 2 * tol_guess),
        method="bounded",
    )

    if not optim.success:
        raise ValueError(f"Optimization failed: {optim.message}")

    errors = num_errors(optim.x, df)

    return optim.x, errors


def plot_stability(tol: float = 1e-2, skip: int = 1) -> None:
    """Plots the stability of the fixed point on the a-sigma plane."""

    df = preprocess_stability(tol)[::skip]

    stable_index = df["stability"]
    stable = df[stable_index]
    unstable = df[~stable_index]

    fig, ax = plt.subplots(1, 1)

    # Plot multiscale method separation line and stability regions
    a_max = np.sqrt(2 * df["sigma"].max())
    a = np.linspace(df["a"].min(), a_max)
    ax.plot(a, 0.5 * a**2, c="k", zorder=-1)
    ax.fill_between(
        a,
        0.5 * a**2,
        color="green",
        linewidth=0,
        alpha=0.15,
        label="Stable",
    )
    ax.fill_between(
        np.linspace(a_max, df["a"].max()),
        df["sigma"].max(),
        color="green",
        linewidth=0,
        alpha=0.15,
    )
    ax.fill_between(
        a,
```

```
        0.5 * a**2,
        y2=df["sigma"].max(),
        color="red",
        linewidth=0,
        alpha=0.15,
        label="Unstable",
    )

    # Plot numerical stability points
    ax.scatter(
        stable["a"],
        stable["sigma"],
        color="green",
        label="Numerically stable",
        marker="o",
    )
    ax.scatter(
        unstable["a"],
        unstable["sigma"],
        color="red",
        label="Numerically unstable",
        marker="o",
    )

    ax.set_title(f"Stability of the Kapitza pendulum (tol = {tol:.3g})")
    ax.set_xlabel(r"$a = b/l$")
    ax.set_ylabel(r"$\sigma = g / l \omega^2$")

    ax.legend(loc="lower right", framealpha=1)

    fig.tight_layout()

    if SAVE_FIGURES:
        fig.savefig(f"images/stability_{tol:.3g}.png", dpi=200)


def plot_trajectories(tol: float = 1e-2, skip: int = 2) -> None:
    """
    Plots the numerically stable trajectories of the Kapitza
    pendulum for various values of a-sigma.
    """

    df = pd.read_csv("data/trajectory.csv")

    theta_max: float = 0

    fig, ax = plt.subplots(1, 1)
    for a in df["a"].unique()[::skip]:
        for sigma in df["sigma"].unique()[::skip]:
```

```python
            filt = (df["a"] == a) & (df["sigma"] == sigma)
            traj = df[filt]
            is_stable = check_stability(a, sigma)
            if abs(traj["theta"].iloc[-1]) < tol:
                theta_max = max(theta_max, abs(traj["theta"]).max())
                ax.plot(
                    traj["t"],
                    traj["theta"],
                    ls="--" if not is_stable else "-",
                    alpha=0.5 if not is_stable else 1,
                )

    ax.set_title(
        f"Numerically stable trajectories\nof the Kapitza pendulum (tol = {tol:.3g})"
    )
    ax.set_xlabel(r"$t$")
    ax.set_ylabel(r"$\theta(t)$")

    if theta_max > np.pi / 6:
        ax.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 6))
        ax.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter(6)))

    fig.tight_layout()

    if SAVE_FIGURES:
        fig.savefig(f"images/stable_trajs_{tol:.3g}.png", dpi=200)


def plot_errors(tol: float = 1e-2, skip: int = 2) -> None:
    """
    Plots the numerically unstable but analitically stable trajectories
    of the Kapitza pendulum for various values of a-sigma.
    """

    df = pd.read_csv("data/trajectory.csv")

    theta_max: float = 0

    fig, ax = plt.subplots(1, 1)
    for a in df["a"].unique()[::skip]:
        for sigma in df["sigma"].unique()[::skip]:
            filt = (df["a"] == a) & (df["sigma"] == sigma)
            traj = df[filt]
            is_stable = check_stability(a, sigma)
            if abs(traj["theta"].iloc[-1]) > tol:
                if is_stable:
                    theta_max = max(theta_max, abs(traj["theta"]).max())
                    ax.plot(traj["t"], traj["theta"])
```

```python

    ax.set_title(
        f"Numerically unstable but actually stable trajectories\n"
        f"of the Kapitza pendulum (tol = {tol:.3g})"
    )
    ax.set_xlabel(r"$t$")
    ax.set_ylabel(r"$\theta(t)$")

    if theta_max > np.pi / 6:
        ax.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 6))
        ax.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter(6)))

    fig.tight_layout()

    if SAVE_FIGURES:
        fig.savefig(f"images/errors_{tol:.3g}.png", dpi=200)


def main() -> None:
    """Main function."""

    skip = 1

    phys_tol, n_errors = physical_tol()
    print(f"Physical tolerance: {phys_tol:.3g}, number of errors: {n_errors}")
    plot_stability(phys_tol)
    plot_trajectories(phys_tol, skip)
    plot_errors(phys_tol, skip)

    opt_tol, n_errors = optimize_tol(phys_tol)
    print(f"Optimal tolerance: {phys_tol:.3g}, number of errors: {n_errors}")
    plot_stability(opt_tol)
    # plot_trajectories(opt_tol, skip)
    # plot_errors(opt_tol, skip)

    plt.show()


if __name__ == "__main__":
    main()
```

A.3 my_formatter.py

```python
"""Provides a function to format the axis labels of a plot"""

from typing import Callable

import numpy as np

```

```python

def gcd(a: int, b: int) -> int:
    """Compute the greatest common divisor of a and b"""
    while b:
        a, b = b, a % b
    return a


def simplify_fraction(numerator: int, denominator: int) -> tuple[int, int]:
    """Simplify a fraction"""
    common = gcd(numerator, denominator)
    return numerator // common, denominator // common


def latex_frac(numerator: int, denominator: int) -> str:
    """Return a LaTeX representation of a fraction"""
    return r"\frac{" + str(numerator) + "}{" + str(denominator) + "}"


def multiple_formatter(
    denominator: int = 12,
    multiple: float = np.pi,
    latex_multiple: str = r"\pi",
) -> Callable[[float, float], str]:
    """Produce a multiple formatter"""

    def _multiple_formatter(x: float, _: float) -> str:
        den = denominator
        num = int(np.rint(den * x / multiple))
        num, den = simplify_fraction(num, den)

        if den == 1:
            if num == 0:
                return r"$0$"
            elif num == 1:
                return rf"${latex_multiple}$"
            elif num == -1:
                return rf"$-{latex_multiple}$"
            else:
                return rf"${num}{latex_multiple}$"
        else:
            if num == 1:
                return r"$\frac{%s}{%s}$" % (latex_multiple, den)
            elif num == -1:
                return r"$\frac{-%s}{%s}$" % (latex_multiple, den)
            else:
                return r"$\frac{%s%s}{%s}$" % (num, latex_multiple, den)

    return _multiple_formatter
```

```python


if __name__ == "__main__":
    print(f"{gcd(12, 8) = }")
    print(f"{gcd(8, 13) = }")
    print(f"{gcd(12, 2) = }")

    print(f"{simplify_fraction(12, 8) = }")
    print(f"{simplify_fraction(8, 13) = }")
    print(f"{simplify_fraction(12, 2) = }")
    print(f"{simplify_fraction(2, 12) = }")
```