



UNIVERSITÀ
DI 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	2
2	Numerical solution	4
2.1	Time step size	4
2.2	Tolerance setting	5
2.3	Comparison with multiscale method	5
A	Code listing	7
A.1	main.cpp	7
A.2	plot.py	9
A.3	my_formatter.py	14

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:

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

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

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

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, $\omega \gg \sqrt{g/l}$.

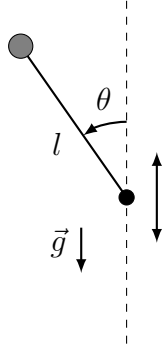


Figure 1: Kapitza's pendulum.

The equation can be rewritten highlighting the potential:

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

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:

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

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 $\theta = 0$:

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

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

$$\sigma < \frac{a^2}{2} \quad (5)$$

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 the 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

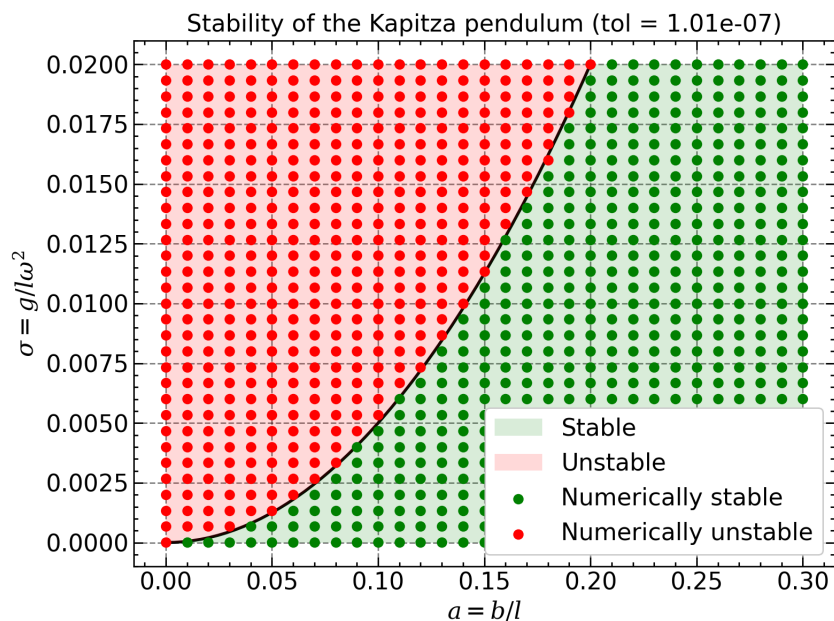


Figure 2: Stability of the upper fixed point.

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.

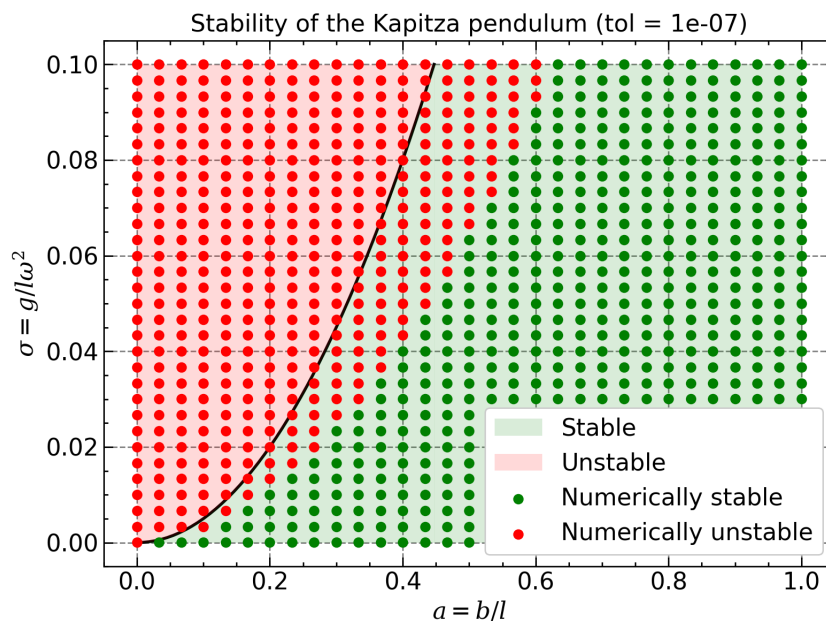


Figure 3: Stability of the upper fixed point beyond multiscale domain.

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

```
1 #include "/Users/francescomarchisotti/Documents/Uni/Magistrale/Algoritmi/Libs/  
   include/ode_solver.hpp"  
2  
3 #include <cmath>  
4 #include <fstream>  
5 #include <iomanip>  
6 #include <iostream>  
7  
8 using std::cerr;  
9 using std::cin;  
10 using std::cout;  
11 using std::endl;  
12  
13 void rangeArray(double arr[], const double &min, const double &max,  
14                 const int &n = 1000);  
15  
16 void RHS(const double &t, double Y[], double R[]);  
17  
18 int main() {  
19     // Create parameter grid  
20     const int nPoints = 30;  
21     const int nStep    = 10000;  
22  
23     const double minA = 0.0;  
24     const double maxA = 0.3; // a of order epsilon  
25     double gridA[nPoints + 1];  
26     rangeArray(gridA, minA, maxA, nPoints);  
27  
28     const double minSigma = 1.0e-5;  
29     const double maxSigma = 2.0e-2; // sigma of order epsilon^2  
30     double gridSigma[nPoints + 1];  
31     rangeArray(gridSigma, minSigma, maxSigma, nPoints);  
32  
33     std::ofstream stability, trajectory;  
34     stability.open("data/stability.csv");  
35     trajectory.open("data/trajectory.csv");  
36     if (!stability || !trajectory) {  
37         cerr << "Error: unable to open file" << endl;  
38         return 1;  
39     }  
40  
41     // Init model parameters
```



```

42     const double theta0 = 1.0e-7;
43     const double omega0 = 0.0; // omega0 is the initial angular velocity, not
44                                // the frequency of oscillation of the fulcrum
45     const double mu = 0.1;     // mu of order epsilon
46     const int nEq = 5;
47
48     const double tmin = 0.0;
49     const double dt =
50         1.0e-2; // with the current parameters, the fulcrum oscillates with
51                 // omega of order 10, so a step size of 1.0e-2 should be
52                 // enough to capture the dynamics (the period of oscillation
53                 // is 2pi/omega = 0.6)
54
55     stability << "a,sigma,endpoint" << endl;
56     trajectory << "t,theta,a,sigma" << endl;
57     for (int iA = 0; iA <= nPoints; iA++) {
58         for (int iSigma = 0; iSigma <= nPoints; iSigma++) {
59             const double a = gridA[iA];
60             const double sigma = gridSigma[iSigma];
61
62             double Y[nEq] = {theta0, omega0, sigma, a, mu};
63             double t = tmin;
64
65             trajectory << std::setprecision(16) << t << "," << Y[0] << "," << a
66                 << "," << sigma << endl;
67             for (int iStep = 0; iStep < nStep; iStep++) {
68                 // Integration step
69                 rk4Step(t, Y, RHS, dt, nEq);
70                 t += dt;
71
72                 // This is probably not necessary
73                 if (fabs(Y[0]) >= M_PI) Y[0] = fmod(Y[0], 2 * M_PI);
74
75                 trajectory << std::setprecision(16) << t << "," << Y[0] << "," <<
76                     << a << "," << sigma << endl;
77             }
78
79             stability << std::setprecision(16) << a << "," << sigma << "," <<
80                 << Y[0] << endl;
81         }
82     }
83
84     stability.close();
85     trajectory.close();
86
87     return 0;
88 }
89
90 void rangeArray(double arr[], const double &min, const double &max,

```

```

91         const int &n) {
92     const double delta = (max - min) / n;
93     for (int i = 0; i <= n; i++) arr[i] = min + i * delta;
94 }
95
96 void RHS(const double &t, double Y[], double R[]) {
97     const double theta = Y[0];
98     const double omega = Y[1];
99     const double sigma = Y[2];
100    const double a      = Y[3];
101    const double mu      = Y[4];
102
103    R[0] = omega;
104    R[1] = (sigma + a * cos(t)) * sin(theta) - mu * omega;
105    R[2] = 0.0;
106    R[3] = 0.0;
107    R[4] = 0.0;
108 }

```

A.2 plot.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4 from scipy.optimize import minimize_scalar
5
6 from my_formatter import multiple_formatter
7
8 plt.style.use(["grid", "science", "notebook", "mylegend"])
9
10 SAVE_FIGURES = True
11
12
13 def check_stability(a: float, sigma: float) -> bool:
14     """
15     Check if the fixed point is stable according to the Multiscale
16     Method.
17     """
18     return a**2 > 2 * sigma
19
20
21 def preprocess_stability(tol: float = 1e-2) -> pd.DataFrame:
22     """Preprocess the stability data."""
23
24     df = pd.read_csv("data/stability.csv")
25
26     df["stability"] = np.abs(df["endpoint"]) < tol
27     df["is_stable"] = check_stability(df["a"], df["sigma"])
28

```

```

29     return df
30
31
32 def num_errors(tol: float, df: pd.DataFrame) -> int:
33     """Count the number of errors in the stability data."""
34     df["stability"] = np.abs(df["endpoint"]) < tol
35     return np.sum(df["stability"] != df["is_stable"])
36
37
38 def physical_tol(tol_guess: float = 1e-7, max_step: int = 1000) -> float:
39     """
40     Find the tolerance (to determine the numerical stability) such
41     that if a = 0 the fixed point is always unstable.
42     """
43
44     df = pd.read_csv("data/stability.csv")
45     filt = df["a"] == 0
46     df = df[filt]
47
48     tol = 10 * tol_guess
49
50     df["stability"] = np.abs(df["endpoint"]) < tol
51
52     # Take big steps until only one fixed point is stable
53     while df["stability"].sum() > 1:
54         tol *= 0.5
55         df["stability"] = np.abs(df["endpoint"]) < tol
56
57     if df["stability"].sum() != 1:
58         raise ValueError("Failed to find the physical tolerance.")
59
60     # Take small steps until the fixed point becomes unstable
61     n_step = 0
62     while df["stability"].sum() == 1:
63         if n_step > max_step:
64             raise ValueError("Failed to find the physical tolerance.")
65         tol *= 0.999
66         df["stability"] = np.abs(df["endpoint"]) < tol
67         n_step += 1
68
69     return tol, num_errors(tol, preprocess_stability(tol))
70
71
72 def optimize_tol(tol_guess: float = 1e-7) -> float:
73     """
74     Find the tolerance (to determine the numerical stability) that
75     minimizes the number of errors.
76     """
77

```

```

78     df = preprocess_stability(-1)
79
80     optim = minimize_scalar(
81         num_errors,
82         args=(df,),
83         bounds=(0.1 * tol_guess, 2 * tol_guess),
84         method="bounded",
85     )
86
87     if not optim.success:
88         raise ValueError(f"Optimization failed: {optim.message}")
89
90     errors = num_errors(optim.x, df)
91
92     return optim.x, errors
93
94
95 def plot_stability(tol: float = 1e-2, skip: int = 1) -> None:
96     """Plots the stability of the fixed point on the a-sigma plane."""
97
98     df = preprocess_stability(tol)[::skip]
99
100    stable_index = df["stability"]
101    stable = df[stable_index]
102    unstable = df[~stable_index]
103
104    fig, ax = plt.subplots(1, 1)
105
106    # Plot multiscale method separation line and stability regions
107    a_max = np.sqrt(2 * df["sigma"].max())
108    a = np.linspace(df["a"].min(), a_max)
109    ax.plot(a, 0.5 * a**2, c="k", zorder=-1)
110    ax.fill_between(
111        a,
112        0.5 * a**2,
113        color="green",
114        linewidth=0,
115        alpha=0.15,
116        label="Stable",
117    )
118    ax.fill_between(
119        np.linspace(a_max, df["a"].max()),
120        df["sigma"].max(),
121        color="green",
122        linewidth=0,
123        alpha=0.15,
124    )
125    ax.fill_between(
126        a,

```

```

127         0.5 * a**2,
128         y2=df["sigma"].max(),
129         color="red",
130         linewidth=0,
131         alpha=0.15,
132         label="Unstable",
133     )
134
135     # Plot numerical stability points
136     ax.scatter(
137         stable["a"],
138         stable["sigma"],
139         color="green",
140         label="Numerically stable",
141         marker="o",
142     )
143     ax.scatter(
144         unstable["a"],
145         unstable["sigma"],
146         color="red",
147         label="Numerically unstable",
148         marker="o",
149     )
150
151     ax.set_title(f"Stability of the Kapitza pendulum (tol = {tol:.3g})")
152     ax.set_xlabel(r"$a = b/l$")
153     ax.set_ylabel(r"$\sigma = g / l \ \omega^2$")
154
155     ax.legend(loc="lower right", framealpha=1)
156
157     fig.tight_layout()
158
159     if SAVE_FIGURES:
160         fig.savefig(f"images/stability_{tol:.3g}.png", dpi=200)
161
162
163 def plot_trajectories(tol: float = 1e-2, skip: int = 2) -> None:
164     """
165     Plots the numerically stable trajectories of the Kapitza
166     pendulum for various values of a-sigma.
167     """
168
169     df = pd.read_csv("data/trajectory.csv")
170
171     theta_max: float = 0
172
173     fig, ax = plt.subplots(1, 1)
174     for a in df["a"].unique()[::skip]:
175         for sigma in df["sigma"].unique()[::skip]:

```

```

176         filt = (df["a"] == a) & (df["sigma"] == sigma)
177         traj = df[filt]
178         is_stable = check_stability(a, sigma)
179         if abs(traj["theta"].iloc[-1]) < tol:
180             theta_max = max(theta_max, abs(traj["theta"]).max())
181             ax.plot(
182                 traj["t"],
183                 traj["theta"],
184                 ls="--" if not is_stable else "-",
185                 alpha=0.5 if not is_stable else 1,
186             )
187
188     ax.set_title(
189         f"Numerically stable trajectories\nof the Kapitza pendulum (tol = {tol
190         :.3g})"
191     )
192     ax.set_xlabel(r"$t$")
193     ax.set_ylabel(r"$\theta(t)$")
194
195     if theta_max > np.pi / 6:
196         ax.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 6))
197         ax.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter(6)))
198
199     fig.tight_layout()
200
201     if SAVE_FIGURES:
202         fig.savefig(f"images/stable_trajs_{tol:.3g}.png", dpi=200)
203
204 def plot_errors(tol: float = 1e-2, skip: int = 2) -> None:
205     """
206     Plots the numerically unstable but analitically stable trajectories
207     of the Kapitza pendulum for various values of a-sigma.
208     """
209
210     df = pd.read_csv("data/trajectory.csv")
211
212     theta_max: float = 0
213
214     fig, ax = plt.subplots(1, 1)
215     for a in df["a"].unique()[::skip]:
216         for sigma in df["sigma"].unique()[::skip]:
217             filt = (df["a"] == a) & (df["sigma"] == sigma)
218             traj = df[filt]
219             is_stable = check_stability(a, sigma)
220             if abs(traj["theta"].iloc[-1]) > tol:
221                 if is_stable:
222                     theta_max = max(theta_max, abs(traj["theta"]).max())
223                     ax.plot(traj["t"], traj["theta"])

```

```

224
225     ax.set_title(
226         f"Numerically unstable but actually stable trajectories\n"
227         f"of the Kapitza pendulum (tol = {tol:.3g})"
228     )
229     ax.set_xlabel(r"$t$")
230     ax.set_ylabel(r"$\theta(t)$")
231
232     if theta_max > np.pi / 6:
233         ax.yaxis.set_major_locator(plt.MultipleLocator(np.pi / 6))
234         ax.yaxis.set_major_formatter(plt.FuncFormatter(multiple_formatter(6)))
235
236     fig.tight_layout()
237
238     if SAVE_FIGURES:
239         fig.savefig(f"images/errors_{tol:.3g}.png", dpi=200)
240
241
242 def main() -> None:
243     """Main function."""
244
245     skip = 1
246
247     phys_tol, n_errors = physical_tol()
248     print(f"Physical tolerance: {phys_tol:.3g}, number of errors: {n_errors}")
249     plot_stability(phys_tol)
250     plot_trajectories(phys_tol, skip)
251     plot_errors(phys_tol, skip)
252
253     opt_tol, n_errors = optimize_tol(phys_tol)
254     print(f"Optimal tolerance: {phys_tol:.3g}, number of errors: {n_errors}")
255     plot_stability(opt_tol)
256     # plot_trajectories(opt_tol, skip)
257     # plot_errors(opt_tol, skip)
258
259     plt.show()
260
261
262 if __name__ == "__main__":
263     main()

```

A.3 my_formatter.py

```

1  """Provides a function to format the axis labels of a plot"""
2
3  from typing import Callable
4
5  import numpy as np
6

```

```

7
8 def gcd(a: int, b: int) -> int:
9     """Compute the greatest common divisor of a and b"""
10    while b:
11        a, b = b, a % b
12    return a
13
14
15 def simplify_fraction(numerator: int, denominator: int) -> tuple[int, int]:
16     """Simplify a fraction"""
17     common = gcd(numerator, denominator)
18     return numerator // common, denominator // common
19
20
21 def latex_frac(numerator: int, denominator: int) -> str:
22     """Return a LaTeX representation of a fraction"""
23     return r"\frac{" + str(numerator) + "}" + str(denominator) + "}"
24
25
26 def multiple_formatter(
27     denominator: int = 12,
28     multiple: float = np.pi,
29     latex_multiple: str = r"\pi",
30 ) -> Callable[[float, float], str]:
31     """Produce a multiple formatter"""
32
33     def _multiple_formatter(x: float, _: float) -> str:
34         den = denominator
35         num = int(np rint(den * x / multiple))
36         num, den = simplify_fraction(num, den)
37
38         if den == 1:
39             if num == 0:
40                 return r"$0$"
41             elif num == 1:
42                 return rf"${latex_multiple}$"
43             elif num == -1:
44                 return rf"$-{latex_multiple}$"
45             else:
46                 return rf"${num}{latex_multiple}$"
47         else:
48             if num == 1:
49                 return r"$\frac{%s}{%s}$" % (latex_multiple, den)
50             elif num == -1:
51                 return r"$\frac{-%s}{%s}$" % (latex_multiple, den)
52             else:
53                 return r"$\frac{%s}{%s}$" % (num, latex_multiple, den)
54
55     return _multiple_formatter

```



```
56
57
58 if __name__ == "__main__":
59     print(f"{gcd(12, 8) = }")
60     print(f"{gcd(8, 13) = }")
61     print(f"{gcd(12, 2) = }")
62
63     print(f"{simplify_fraction(12, 8) = }")
64     print(f"{simplify_fraction(8, 13) = }")
65     print(f"{simplify_fraction(12, 2) = }")
66     print(f"{simplify_fraction(2, 12) = }")
```