Assignment 33, Numerical integration of simple harmonic oscillator equations of motion

Euler's method uses the following iterative update rule:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

.

where $x_n = x_0 + nh$ for $n \in \mathbb{N}$.

Here is the meaning of the above. Let $y'(x) = f(x, y)$ be an ode of the first order (not necessarily linear), and suppose that $y(x_0) = y_0$. Then we obtain a local linear approximation to the value of the unknown function y about x_0 as

$$y = y_0 + \Delta x f(x_0, y_0)$$

h - or Δx - is the *step size*. It denotes a certain distance along the axis of the independent variable. Given the ODE and a point on the (unkown) curve y, we reach the approximate next y value by starting at y_0, and then travelling h times the *slope* of the curve as given by the value of its derivative evaluated at the point (x_0, y_0). This, of course, gives us a new point: (x_1, y_1), where $x_1 = x_0 + h$ and $y_1 = y_0 + hf(x_0, y_0)$.

The process above can be repeated. In order to make progress, we essentially tell a small lie. The price we pay for the small lie we tell is that we have to resign ourselves to an appoximate value for each y_n for $n \geq 1$. The lie is this: we next calculate the slope of the tangent line at the point (x_1, y_1). This is a fiction, since the point in question doesn't actually belong to the unknonwn curve y. But we can think of $f(x_1, y_1)$ as coming from one of the infnitely many curves in the family of curves comprising the general solution to the differential equation we started with. For a small enough step size, this curve will be 'close to' the actual curve, and so the slope of the line tangent to it at the point (x_1, y_1) will in general be a good guide to the direction we ought to move in order to rach the next (approximate) point on the correct curve.

As an example, suppose that $y' = 0.1\sqrt{y} + 0.4x^2$ and that $y(2) = 4$.

In [1]:
```
# jupyter nbconvert --to webpdf notebook.ipynb
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_theme(style="darkgrid")
from tabulate import tabulate


def y(y0, x0, h, f):
    """
    euler's method update rule:

    y0: float
        previous y value
    x0: float
        previous x value
```

```
    h: float
        step size
    f: function
        float --> float
        y' = f(x, y)
    """
    return y0 + h * f(x0, y0)


def f(x, y):
    return 0.1 * math.sqrt(y) + 0.4 * x**2


X = np.linspace(2.0, 2.5, 6)


Y = [4]

for i in range(0, len(X)):
    Y.append(round(y(Y[i], X[i], 0.1, f), 4))

table = list(zip(X, Y))

print(tabulate(table, headers="XY", tablefmt="fancy_grid"))


X = np.linspace(2.0, 2.5, 11)

Y = [4]

for i in range(0, len(X)):
    Y.append(round(y(Y[i], X[i], 0.05, f), 4))

table = list(zip(X, Y))

print(tabulate(table, headers="XY", tablefmt="fancy_grid"))
```

X	Y
2	4
2.1	4.18
2.2	4.3768
2.3	4.5913
2.4	4.8243
2.5	5.0767

X	Y
2	4
2.05	4.09
2.1	4.1842
2.15	4.2826
2.2	4.3854
2.25	4.4927
2.3	4.6045
2.35	4.721
2.4	4.8423
2.45	4.9685
2.5	5.0997

We can proceed in an analagous way in the event that the unknown function is a vector-valued function of time:

$$\mathbf{x}' = \frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(t, \mathbf{x}(t))$$

with initial conditions given by $t = t_0, \mathbf{x}_0 = \mathbf{x}(t_0)$.

The update rule for obtaining the next approximate value of the unknown function $\mathbf{x}$ from the previous value and the slope of the (approx) tangent at the previous point can then be written as:

$$\mathbf{k}_n = \Delta t \mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{k}_n$$

where $t_n = t_0 + n\Delta t$.

As a particular example, consider the dimensionless form of the Hamiltonian of the simple harmonic oscilator:

$$\mathcal{H} = \frac{1}{2}x^2 + \frac{1}{2}p^2$$

where x and p are dimensionless position and momentum.

The equations of motion are then given by:

$$\begin{cases} \frac{dx}{dt} = \frac{\partial \mathcal{H}}{\partial p} = p \\ \frac{dp}{dt} = -\frac{\partial \mathcal{H}}{\partial x} = -x \end{cases} \tag{1}$$

So let,

$$\mathbf{x}(x_1, x_2) = \mathbf{x}(x, p)$$

.

Then

$$\mathbf{f}(f_1, f_2) = \mathbf{f}(x_2, -x_1) = \mathbf{f}(p, -x)$$

.

Now, for simple harmonic motion, we know that x is a sinusoidal function of t. WLOG, then, let us assume that

$$x(t) = \cos(t)$$

Then we also have, from the equations of motion above, that

$$p(t) = -\sin(t)$$

The above both clearly have periods of $\Delta t = 2\pi$.

We take here the 'natural' set of initial conditons: $t_0 = 0, x_0 = cos(t_0) = 1, p_0 = -sin(t_0) = 0$. This set of initial conditions represents the case wherein we have 'streched' the 'spring' to its fullest extent (here, 1 'unit' from its unperturbed position) and not yet released it.

Note that, with these choices of x and p, the equation expressing the hamiltonian becomes:

$$\mathcal{H} = \frac{1}{2}\cos^2(t) + \frac{1}{2}\sin^2(t)$$

which is the parametric equation of a circle in $x - p$ space, centered at the origin and with radius $r = \frac{1}{2}$.

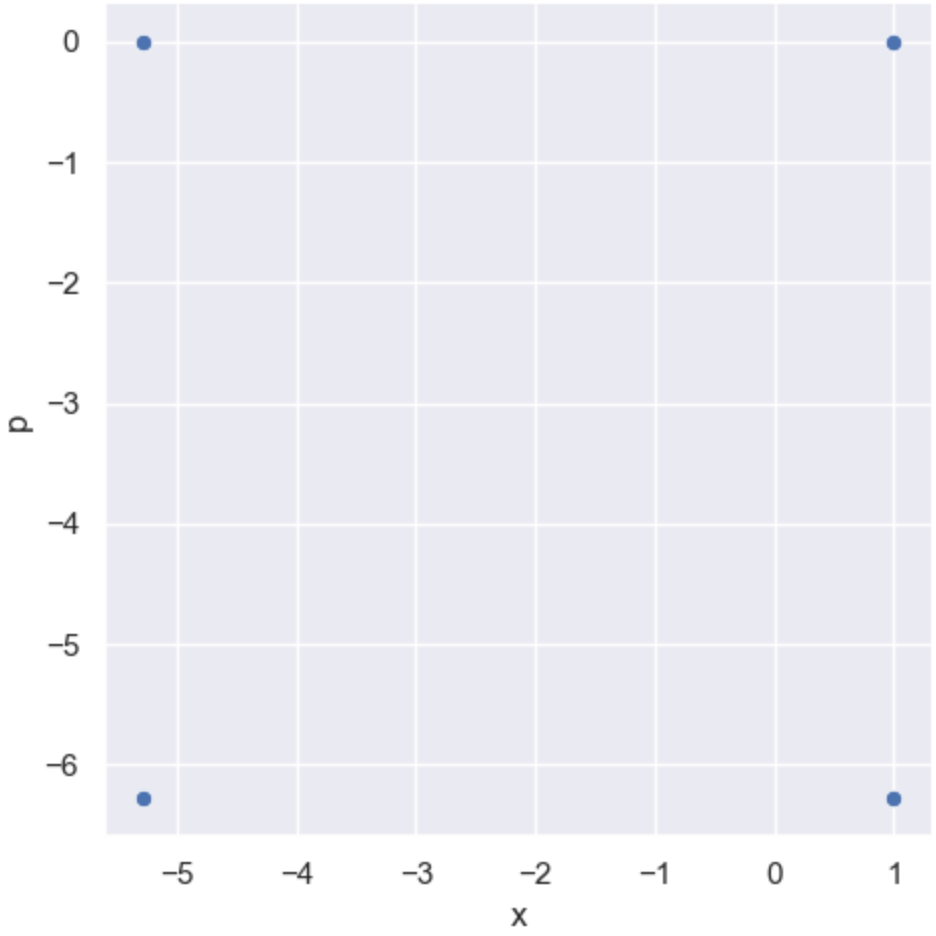
In [2]:
```
def x1(t):
    return math.cos(t)


def x2(t):
    return -math.sin(t)


def f1(t):
    return x2(t)
```

```
def f2(t):
    return -x1(t)


def euler(ti, tf, dt, x1, x2, f1, f2):
    """
    ti: float
        initial time
    tf: float
        final time
    dt: int
        determiner of size of time increments
    x1: function: float --> float
        first component of X
    x2: function: float --> float
        second component of X
    f1: function: float --> float
        first component of F
    f2: function: float --> float
        second component of F
    """
    t = np.linspace(ti, tf, dt)
    X = [np.array([x1(t[0]), x2(t[0])])] * len(t)
    F = [None] * len(t)
    for i in range(1, len(t)):
        # present value of f is obtained by evaluating
        # its compents at the previous time step
        f = np.array([f1(t[i - 1]), f2(t[i - 1])])
        F[i - 1] = f
        k = (tf - ti) / dt * F[i - 1]
        x = X[i - 1] + k
        X[i] = x
    return X


table = euler(0.0, 10 * math.pi, 5, x1, x2, f1, f2)

print("       tf - ti = 2pi")
print(tabulate(table, headers="xp", tablefmt="fancy_grid"))
fig1 = sns.relplot(data=pd.DataFrame(table, columns=["x", "p"]), x="x", y="p")
plt.show()
plt.close()

table = euler(0.0, 10 * math.pi, 11, x1, x2, f1, f2)

print("\n   tf - ti = pi")
print(tabulate(table, headers="xp", tablefmt="fancy_grid"))
fig2 = sns.relplot(data=pd.DataFrame(table, columns=["x", "p"]), x="x", y="p")
plt.show()
plt.close()

table = euler(0.0, 10 * math.pi, 23, x1, x2, f1, f2)

print("\n        tf - ti = pi/2")
print(tabulate(table, headers="xp", tablefmt="fancy_grid"))
fig3 = sns.relplot(data=pd.DataFrame(table, columns=["x", "p"]), x="x", y="p")
plt.show()
plt.close()

table = euler(0.0, 10 * math.pi, 10000, x1, x2, f1, f2)

print("\n                  X vs P for tf - ti = pi/20,000")
```

```
fig4 = sns.relplot(data=pd.DataFrame(table, columns=["x", "p"]), x="x", y="p")
plt.show()
plt.close()
```

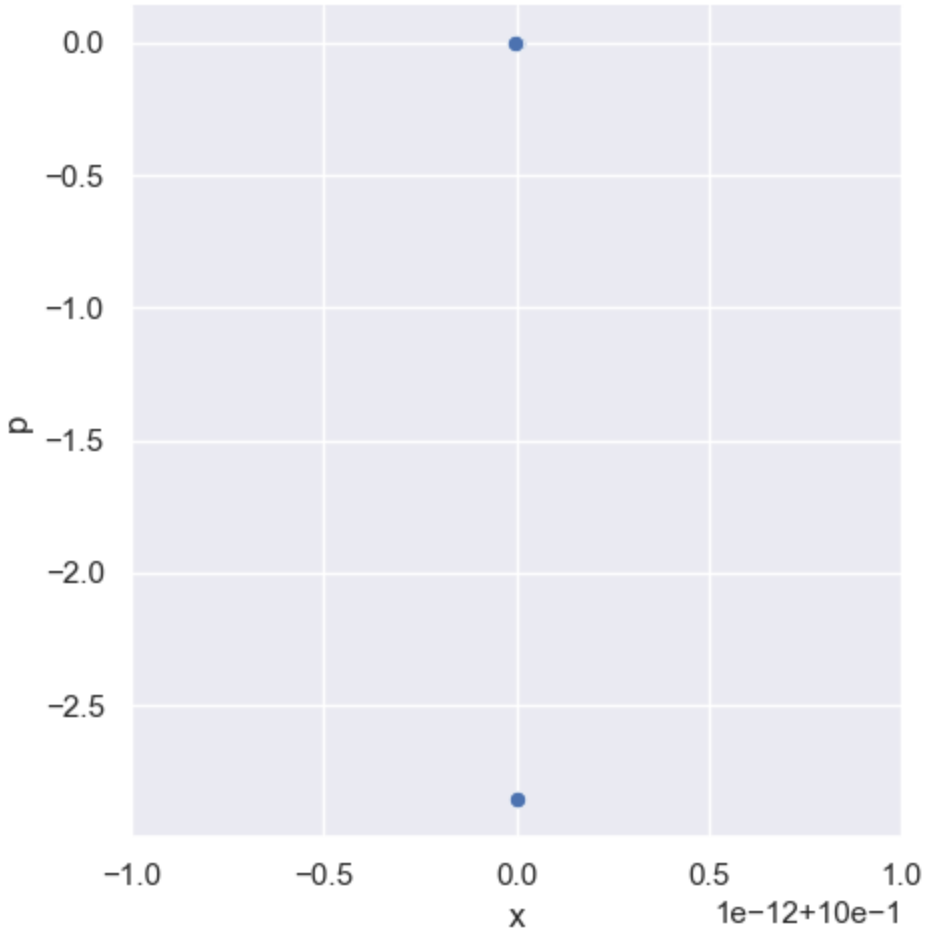
tf - ti = 2pi

x	p
1	-0
1	-6.28319
-5.28319	-6.28319
-5.28319	-1.77636e-15
1	1.51558e-14



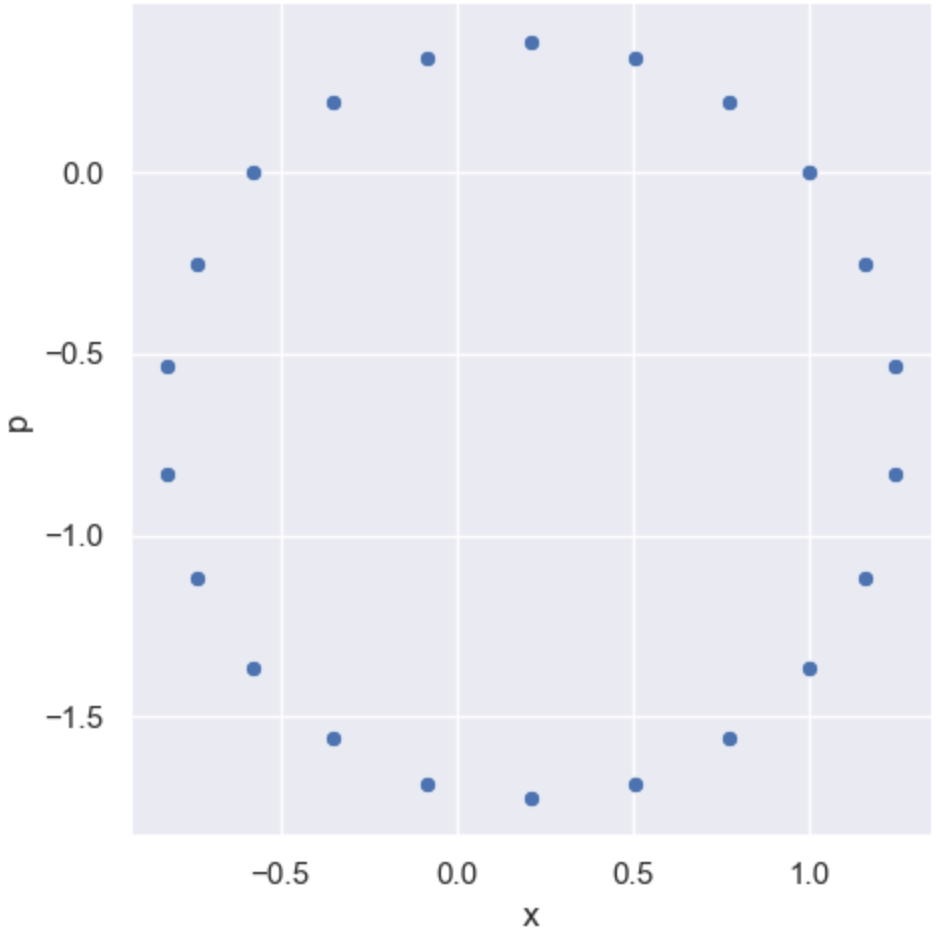
tf - ti = pi

x	p
1	-0
1	-2.85599
1	0
1	-2.85599
1	0
1	-2.85599
1	0
1	-2.85599
1	0
1	-2.85599
1	0

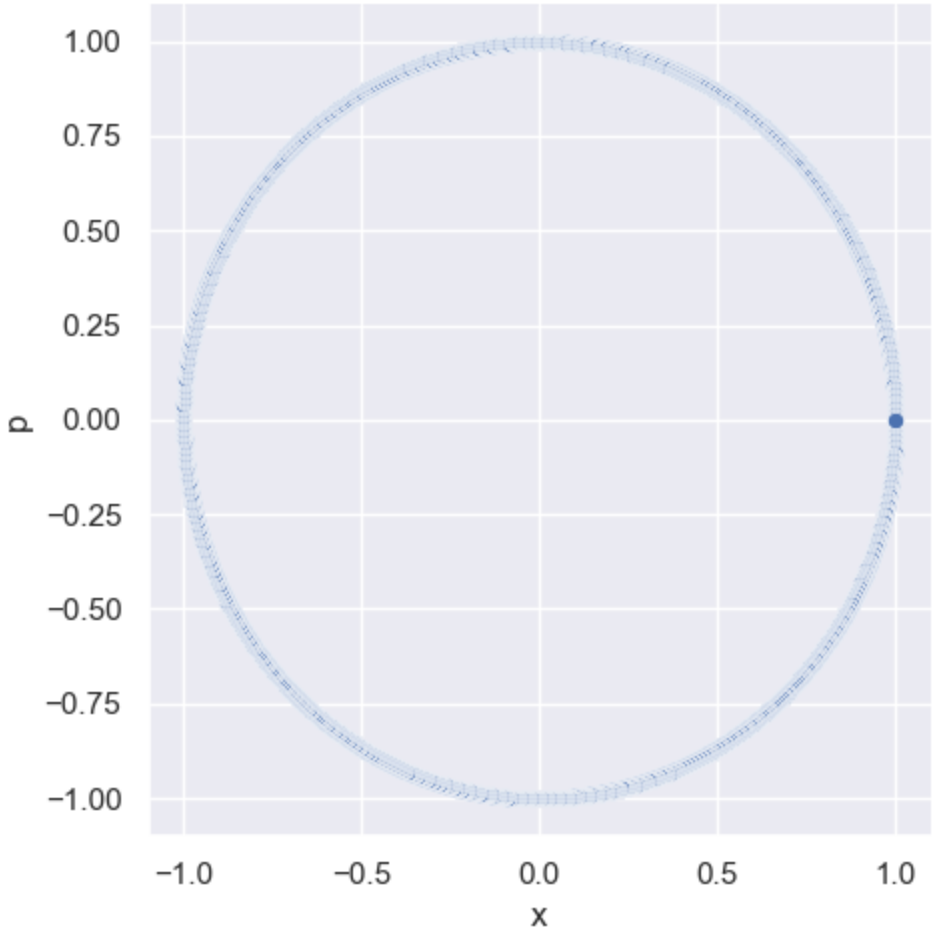


tf - ti = pi/2

x	p
1	-0
1	-1.36591
-0.352007	-1.5603
-0.736828	-0.249718
0.505647	0.317701
1.24411	-0.831375
0.211828	-1.72586
-0.820458	-0.831375
-0.0819912	0.317701
1.16048	-0.249718
0.775663	-1.5603
-0.576344	-1.36591
-0.576344	1.33227e-15
0.775663	0.194389
1.16048	-1.11619
-0.0819912	-1.68361
-0.820458	-0.534535
0.211828	0.359946
1.24411	-0.534535
0.505647	-1.68361
-0.736828	-1.11619
-0.352007	0.194389
1	-3.9968e-15



X vs P for tf − ti = pi/20,000

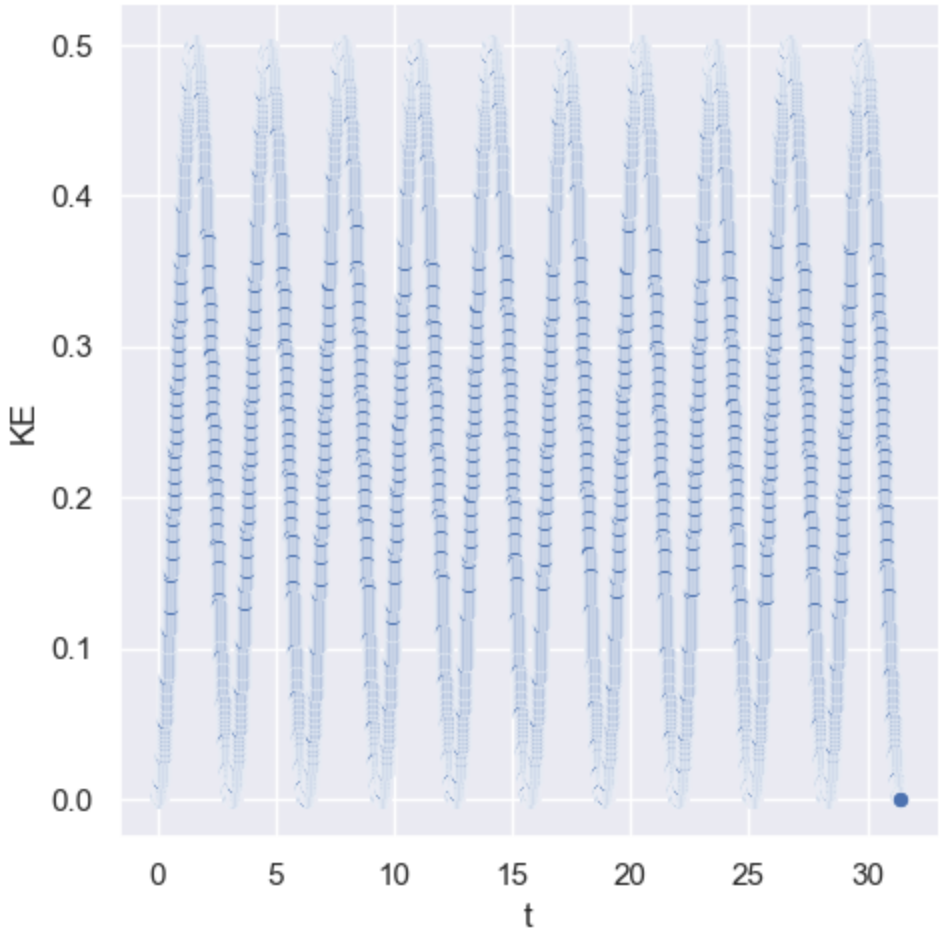


Now, all of the energy which is not *potential* is entirely kinetic (i.e., it is energy of *motion*, associated with and deriving from the momentum of the oscillator at each instant of time). Thus, we have it that the

kinetic energy of the oscillator as a function of time, t is given by the second term in the expression for the hamiltonian above:

$$KE = \frac{1}{2}p(t)^2$$

In [3]:
```
P = 0.5 * np.array([table[i][-1] ** 2 for i in range(len(table))])

TP = list(zip(np.linspace(0, 10 * math.pi, 10000), P))

fig5 = sns.relplot(data=pd.DataFrame(TP, columns=["t", "KE"]), x="t", y="KE")
plt.show()
plt.close()
```

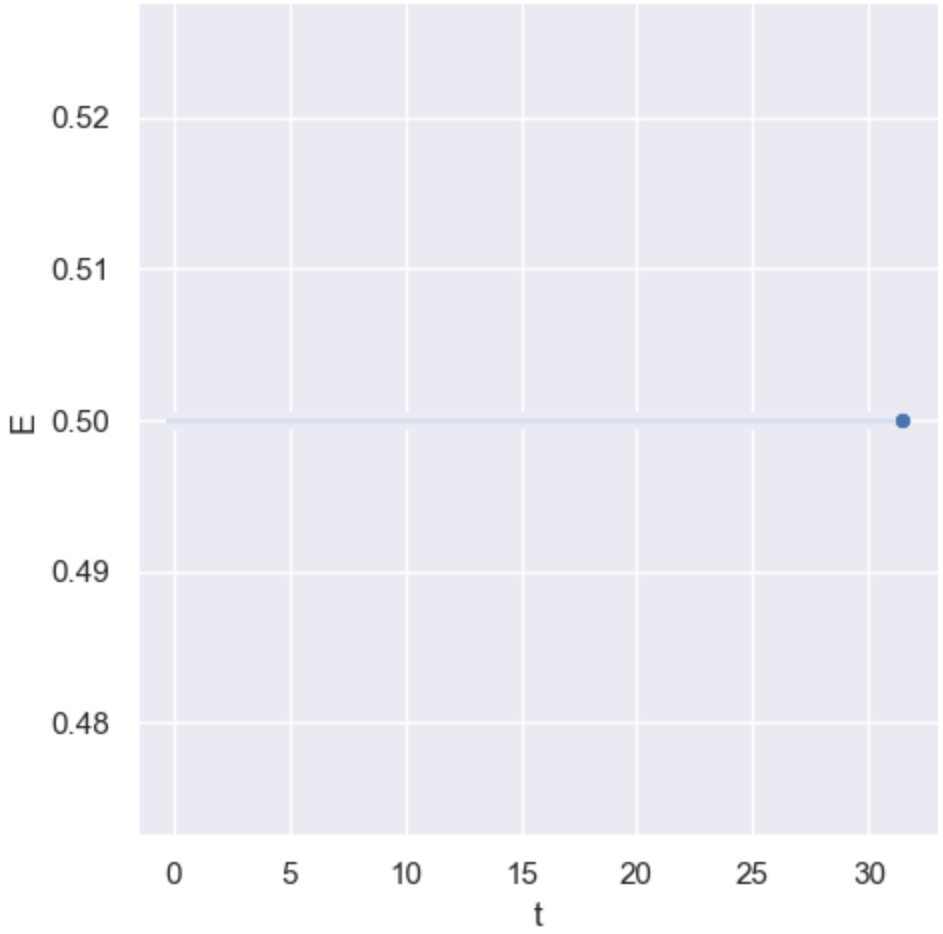


The total energy is given by the sum of the terms in the expression for the hamiltonian:

$$E = \frac{1}{2}x(t)^2 + \frac{1}{2}p(t)^2$$

In [4]:
```
E = 0.5 * np.array([round((table[i][0]**2 + table[i][-1]**2), 2) for i in range(len(tabl
print(E)

TE = list(zip(np.linspace(0, 10 * math.pi, 10000), E))

fig6 = sns.relplot(data=pd.DataFrame(TE, columns=["t", "E"]), x="t", y="E")
plt.show()
plt.close()
```

```
[0.5 0.5 0.5 ... 0.5 0.5 0.5]
```



We can improve upon the above implementations by noting that the euler update rule is always the same, and can always be reduced to the simplest case by applying it to each of the components of a differential equation of a vector valued function. With these observations in hand, we can simplify and generalize our above implementation by defining a new euler function that - given the *previous* point, the value of the derivative at that point, and the value of the function at that point - returns the result of one iteration of the euler update rule:

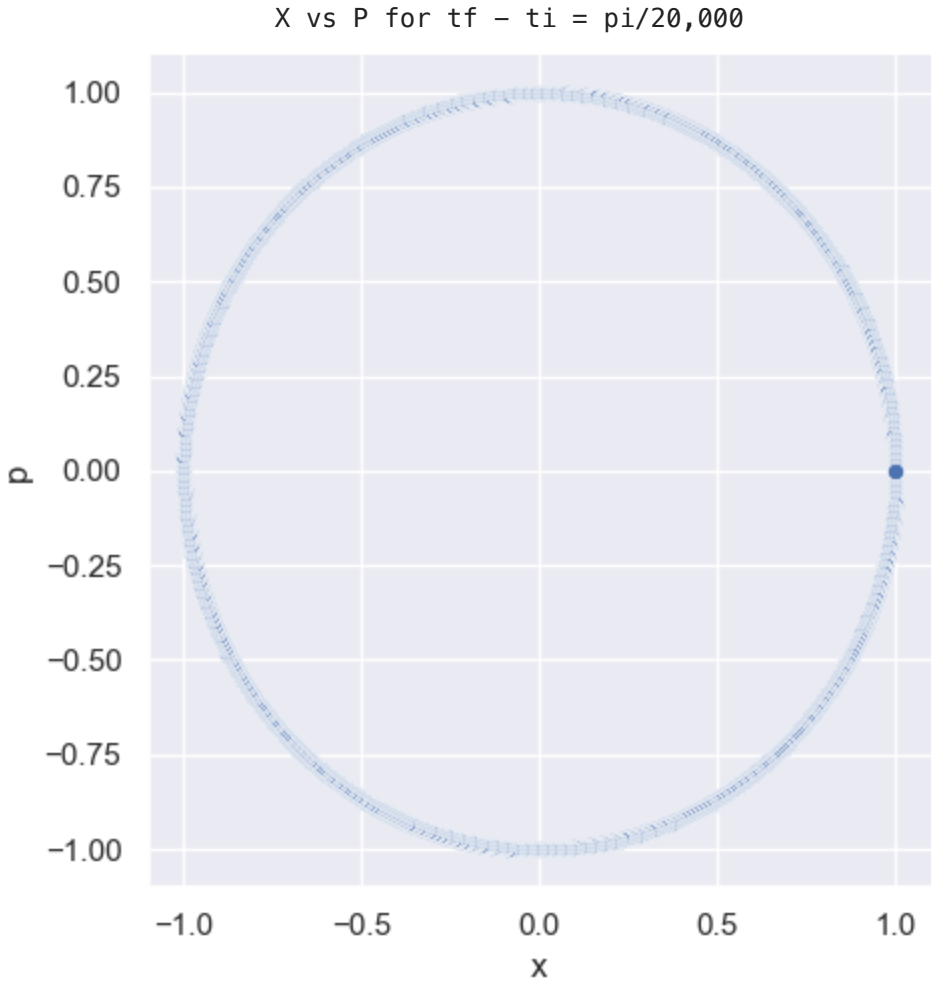
---

euler's rule:

`def euler(t, x, dt, f): return dt*f(x, t)` ___

In [24]:
```
def euler(t, x, dt, f):
    return dt * f(x, t)

def solve_euler(ti, tf, intrvls, n, *cmps):
    """
    ti: float
        initial time
    tf: float
        final time
    intrvls: int
        determiner of size of time increments
    n: int
        number of components
    *cmps: function float --> float
        one or more functions (i.e., derivatives) required
        to implement the euler update rule
    """
    t = np.linspace(ti, tf, intrvls)
```

```
    X = np.array([tuple([cmps[i](t[0]) for i in range(0, n)])] * len(t))
    F = [None] * len(t)
    for i in range(1, len(t)):
        k = [euler(t[i - 1], None, (tf - ti) / intrvls, cmps[j]) for j in range(n, 2*n)]
        x = X[i - 1] + k
        X[i] = x
    return X


def x1(t):
    return math.cos(t)


def x2(t):
    return -math.sin(t)


def f1(x, t):
    return x2(t)


def f2(x, t):
    return -x1(t)


table = solve_euler(0.0, 10 * math.pi, 10000, 2, x1, x2, f1, f2)

print("\n                    X vs P for tf - ti = pi/20,000")
fig4 = sns.relplot(data=pd.DataFrame(table, columns=["x", "p"]), x="x", y="p")
plt.show()
plt.close()
```



In [ ]: