# Lecture 4.1

## Stability of numerical methods

Numerical methods can be **unstable** i.e. the numerical solution diverges from the correct solution. e.g. the undamped harmonic oscillator $\ddot{x} + x = 0$ or $\dot{x} = v$ and $\dot{v} = -x$:

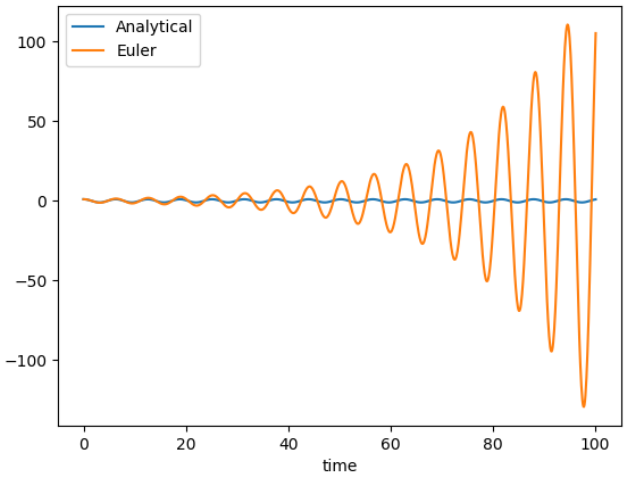
```
def euler(f, x0, dt):
    tn = 0
    xn = x0
    while True:
        yield tn, xn
        xn = xn + f(tn,xn)*dt
        tn = tn + dt

import numpy as np
import matplotlib.pyplot as plt

def f(t, X):
    x,v = X
    xdot = v
    vdot = -x
    return np.array([xdot, vdot])

X0 = np.array([1,0])
dt = 1e-1
solver = euler(f,X0,dt)
ts, Xs = list(), list()
for t,X in solver:
    ts.append(t)
    Xs.append(X)
    if t > 100:
        break

Xs = np.array(Xs)
plt.plot(ts, np.cos(ts), label='Analytical')
plt.plot(ts, Xs[:,0], label='Euler')
plt.legend(); plt.xlabel('time')
plt.savefig('figs/euler_undamped_ho.png')
plt.close('all')
```




Euler is always unstable for the undamped harmonic oscillator.

Other methods can be stable for the undamped harmonic oscillator. In general, stability depends on the method, ODE, and time-step. The question of stability can be tackled theoretically, but is beyond the scope of this course.

A method may be stable provided the time-step is less than some upper bound. e.g. Euler is stable for $\dot{x} + \alpha x = 0$ if $\delta t < 2/\alpha$.

Related concept is **accuracy**. The method may not diverge, but **how** close is it to the correct answer? Disagreement is proportional to $(\delta t)^n$. Beware that this disagreement may be biased. i.e. a method may consistently over- or under-estimate a solution.

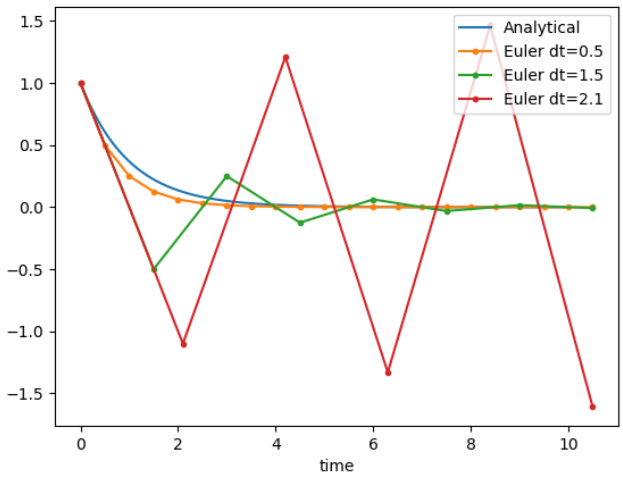
For example, consider $\dot{x} + \alpha x = 0$ with $\alpha = 1$ and different time-steps $\delta t$. For $\delta t = 0.5$ it's clearly stable, although it is **biased**. For a $\delta t = 1.5$ it oscillates about the correct answer, but is stable. For $\delta t = 2.1$ the numerics diverges from the correct answer.

```
def f(t,x):
    return -x

ta = np.linspace(0,10,1001)
plt.plot(ta, np.exp(-ta), label='Analytical')

for dt in [0.5, 1.5, 2.1]:
    X0 = np.array([1])
    solver = euler(f,X0,dt)
    ts, Xs = list(), list()
    for t,X in solver:
        ts.append(t)
        Xs.append(X)
        if t > 10:
            break
    plt.plot(ts, Xs, '.-', label='Euler dt={}'.format(dt))

plt.legend(); plt.xlabel('time')
plt.savefig('figs/euler_stability.png')
plt.close('all')
```




Also consider **efficiency**. Euler takes less computation than Runge–Kutta, so there is a trade-off.

Take home: methods are **approximate**; think about accuracy and stability!

## Boundary value problems and Partial differential equations

The solution of these problems is beyond the scope of this course.

An example of a boundary value problem is time-independent Schrödinger:

$$\left(-\hbar^2\frac{\partial^2}{\partial x^2}+V(x)\right)\psi = E\psi$$

with the condition that $\psi(0)=0$ and $\psi(L)=0$ where $L$ is larger than any length-scale in the problem.

A partial differential equation is a differential equation where differentials are with respect to more than one variable. An example is the time-dependent Schrödinger equation:

$$\left(-\hbar^2\frac{\partial^2}{\partial x^2}+V(x)\right)\psi = i\hbar\frac{\partial}{\partial t}\psi.$$

Solution of PDEs is a rich and interesting area. The most straight-forward method is to set up a grid in each variable (e.g. $x$ and $t$) and then use methods related to Euler etc. This is the Finite Difference Method.

## Multiprocessing

I've been encoraging you to wrap your code up in functions.

```
def f(x):
    return x**2

y = [f(n) for n in range(10)]
```

rather than

```
y = list()
for n in range(10):
    y.append(n**2)
```

because the former allows you to build complex behaviour from simple, testable parts.

The former is also *parallelisable*. When you want to work on lots of data, this is essential.

```
from multiprocessing import Pool

def f(x):
    return x**2

p = Pool()
y = p.map(f, range(10))
p.close()
```

It is important to close resources when you no longer need them, whether they are connections to multiprocessing, or network connections, or files.

A slightly better way to write this kind of thing is

```
with Pool() as p:
    y = p.map(f, range(10))
```

The `with` construct handles closing and can be used for network connections and files.

For problems which are *embarrasingly parallel*, that's all there is to it. For problems where you have to share data between processes, then that is well beyond the scope.

For most practical problems I encounter in my experimental work, the above is sufficient. e.g. I get 1000 data files each 10M points and I need to do numerically-intensive processing. With the above, I get 8x speedup on a single machine, with negligible additional effort.

## Functional promises

When using multiprocessing, you must ensure that the function always gives the same output for the same input. e.g.

```
def f(x):
    return x**2
```

`f(2)` always gives `4`.

We can break this by referring to a mutable object which exists outside the function. e.g.

```
def incsum(A):
    A[0] += 1
    return A[0]

B = [1]
[incsum(B) for n in range(10)]
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

# Classes

Everything in Python is an object.

An object has *properties* and *methods*, which are functions attached to that object.

```
R = np.random.random((3,3))
R.shape # a property
R.mean() # a method
```

```
0.606582352458
```

Sometimes, we want to make our own objects.

```
class Fork:
    def __init__(self, tines=3):
        self.tines = tines
    def snap(self):
        if self.tines > 0:
            self.tines -= 1
    def __str__(self):
        # str should be human readable
        return "This is a fork with {n} tines".format(n=self.tines)
    def __repr__(self):
        # repr should be unambiguous
        return "Fork({n})".format(n=self.tines)

normal_fork = Fork(4)
dessert_fork = Fork(3)

dessert_fork.snap()
dessert_fork.tines
```

```
2
```

- Keyword `class` used to make a new object.
- Function `__init__` is called when you make a new instance of this object (e.g. `dessert_fork`).
- Other specially-named functions, such as `__str__` exist
  - `__str__` should return a human-readable string describing the object
  - `__repr__` should return an unambiguous description, ideally code which could be used to make this object
- Write whatever functions are necessary to do sensible things to this object, e.g. `snap`.
- The first argument of any function inside this definition (conventionally called `self`) is the name by which this object refers to itself.
  - e.g. The function `snap` can refer to the `tines` property of the specific object using `self.tines`.

```
normal_fork
```

```
This is a fork with 4 tines
```

```
repr(normal_fork)
```

```
Fork(4)
```

```
cutlery = [Fork(4) for n in range(10)]
cutlery[0].snap()
cutlery
```

```
[Fork(3), Fork(4), Fork(4), Fork(4), Fork(4), Fork(4), Fork(4), Fork(4), Fork(4), Fork(4)]
```