Week 5: Matrices and implicit methods for linear ODEs

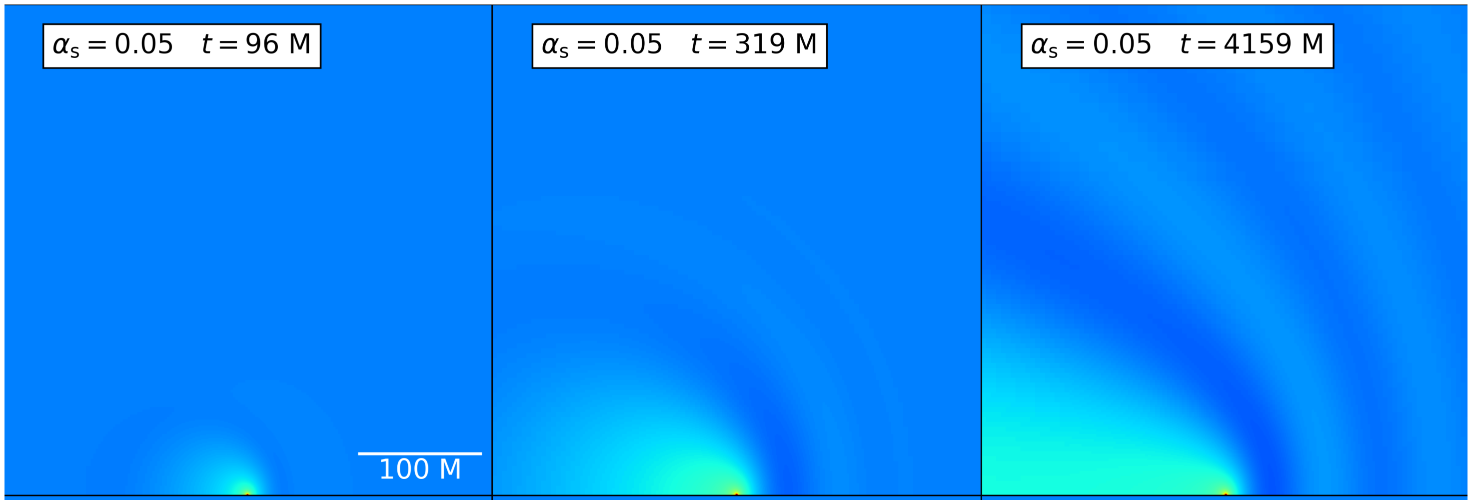
- Matrices in python, backwards Euler method for linear systems

Dr K Clough, Topics in Scientific computing, Autumn term 2023

Coursework announcement - animation and displaying results

I have uploaded an example animation with the Week 4 Tutorial Solutions, but you do not have to use animation in the coursework.

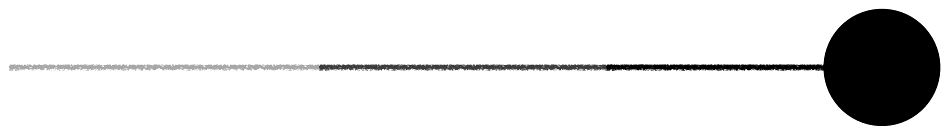
Another nice method of seeing evolution is to use a series of snapshots.



Coursework announcement - animation and displaying results

I have uploaded an example animation with the Week 4 Tutorial Solutions, but you do not have to use animation in the coursework.

How could you give a star a “tail”? Investigate the parameter “alpha” that changes the opacity of the line, noting that it can be a vector…



Sometimes pictures easier to share than movies so don’t discount them!

Plan for today

1. Motivation - revision of coupled linear ODEs and illustration of stiff functions
2. How to do linear algebra with python - Sympy versus Numpy
3. Solution - solving a stiff linear ODE system with an implicit method
4. This week's tutorial - matrices and harmonic oscillator solution with implicit methods

Revision of coupled linear ODEs

Consider this simple first order dimension 2 linear ODE:

$$\dot{x} = 998x + 1998y \quad x(0) = 1$$

$$\dot{y} = -999x - 1999y \quad y(0) = 0$$

How do I solve this equation (analytically)?

Revision of coupled linear ODEs

Consider this simple first order dimension 2 linear ODE:

$$\dot{x} = 998x + 1998y \quad x(0) = 1$$

$$\dot{y} = -999x - 1999y \quad y(0) = 0$$

Recall that we can write a linear system of equations as a matrix equation

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}$$

Find the eigenvalues λ_i and their associated eigenvectors v_i, then solution is

$$X = \sum_i A_i v_i e^{\lambda_i}$$ and we determine the coefficients A_i using the initial conditions

Revision of coupled linear ODEs

$$\frac{d}{dt}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}998 & 1998\\-999 & -1999\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

The two eigenvalues and their associated eigenvectors are

$$\lambda_i = (-1000, -1) \qquad v_i = \begin{bmatrix}-1\\1\end{bmatrix}, \begin{bmatrix}-2\\1\end{bmatrix}$$

Why two?

so solution is:

$$\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}-2Ae^{-t} + -Be^{-1000t}\\Ae^{-t} + Be^{-1000t}\end{bmatrix}$$

and using the initial conditions A = -1 B =1

Revision of coupled linear ODEs

The two eigenvalues and their associated eigenvectors are

$$\lambda_i = (-1000, -1) \qquad v_i = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

What does this tell me (physically) about the solution?

Revision of coupled linear ODEs

The two eigenvalues and their associated eigenvectors are

$$\lambda_i = (-1000, -1) \qquad v_i = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

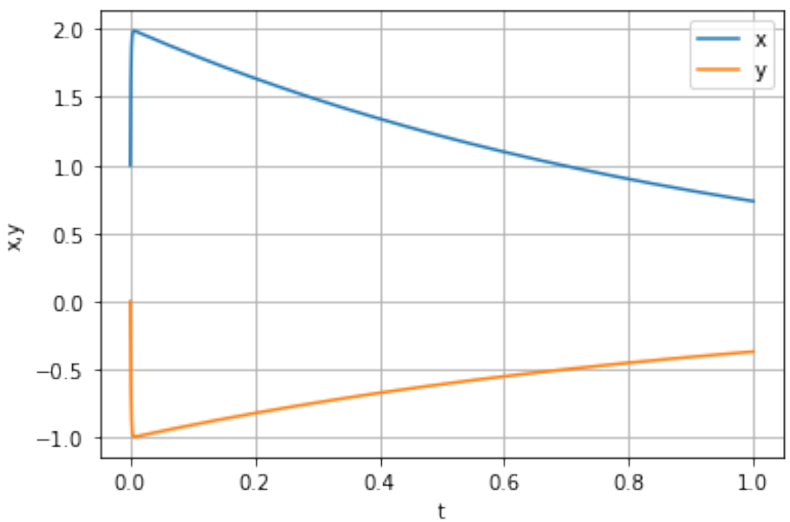
Two modes, one with a very short timescale, one with a much longer one ($\tau \sim 1/\lambda_i$)

Modes go in “opposite directions” for x and y

Revision of coupled linear ODEs

The two eigenvalues and their associated eigenvectors are

$$\lambda_i = (-1000, -1) \qquad v_i = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

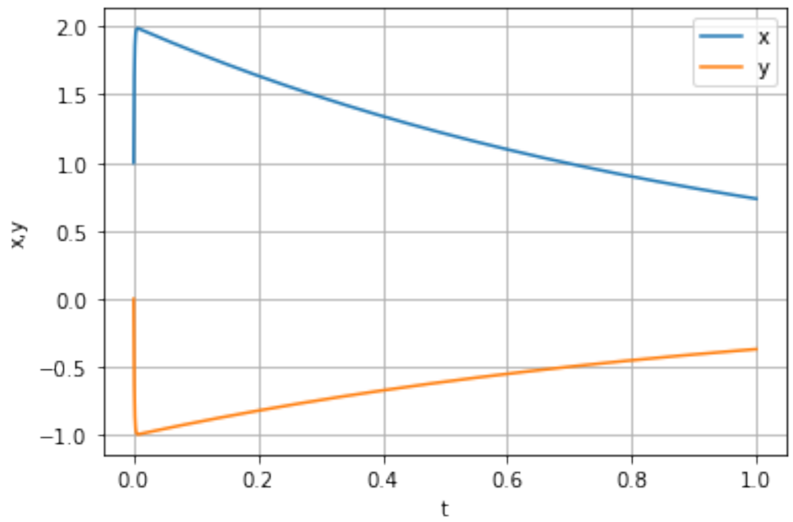


Two modes, one with a very short timescale, one with a much longer one ($\tau \sim 1/\lambda_i$)

Modes go in "opposite directions" for x and y

The trouble with explicit solutions

Why will our explicit methods have trouble with this system?



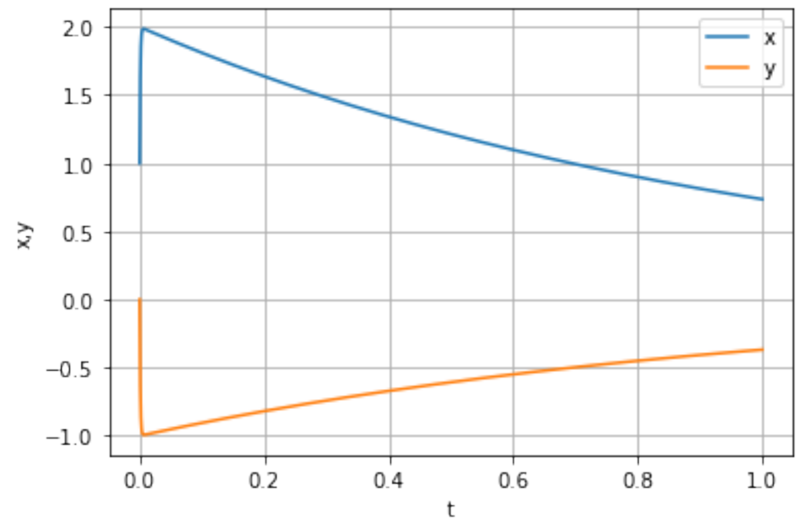
Two modes, one with a very short timescale, one with a much longer one ($\tau \sim 1/\lambda_i$)

Recall that we called these **"stiff" systems**

The trouble with explicit solutions for stiff systems

Let's call the matrix C, and assume that it has only positive eigenvalues, so:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = -C\begin{bmatrix} x \\ y \end{bmatrix}$$



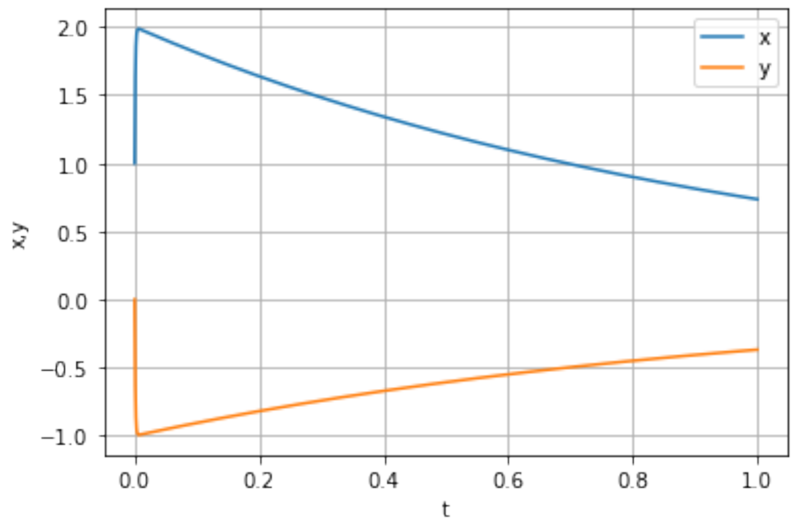
Why do I need it to only have positive eigenvalues?

The trouble with explicit solutions for stiff systems

Let's call the matrix C, and assume that it has only positive eigenvalues, so:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = -C\begin{bmatrix} x \\ y \end{bmatrix}$$

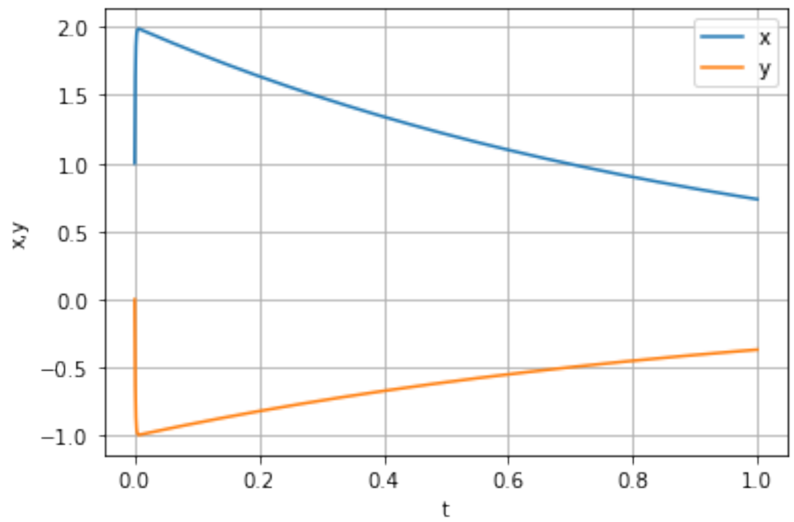
We assume the system is stable, so modes decay over time, therefore the eigenvalues (noting the minus sign introduced above) need to be positive



The trouble with explicit solutions for stiff systems

Let's call the matrix C, and assume that it has only positive eigenvalues, so:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = -C\begin{bmatrix} x \\ y \end{bmatrix}$$



The forward Euler method with step size h is

$$x_{k+1} = x_k + h(-Cx_k)$$

So we see that any x_k is obtained from the initial state x_0 by *k* applications of the matrix $(I - hC)$

$$x_k = (I - hC)^k x_0$$

The trouble with explicit solutions for stiff systems

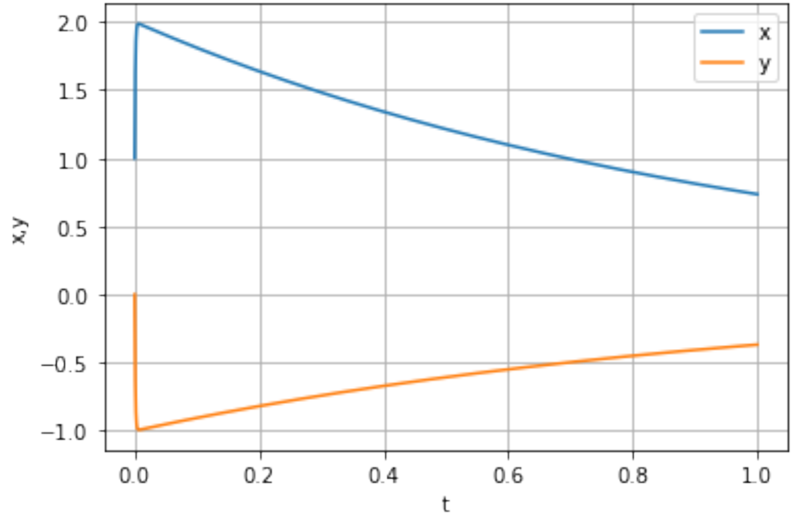
Knowing that C is positive definite, this means that it can be decomposed as

$$C = A^{-1} \Lambda A$$

with Λ a diagonal matrix of the eigenvalues.

A bit of matrix algebra gives:

$$(I - hC)^k = A^{-1}(I - h\Lambda)^k A$$

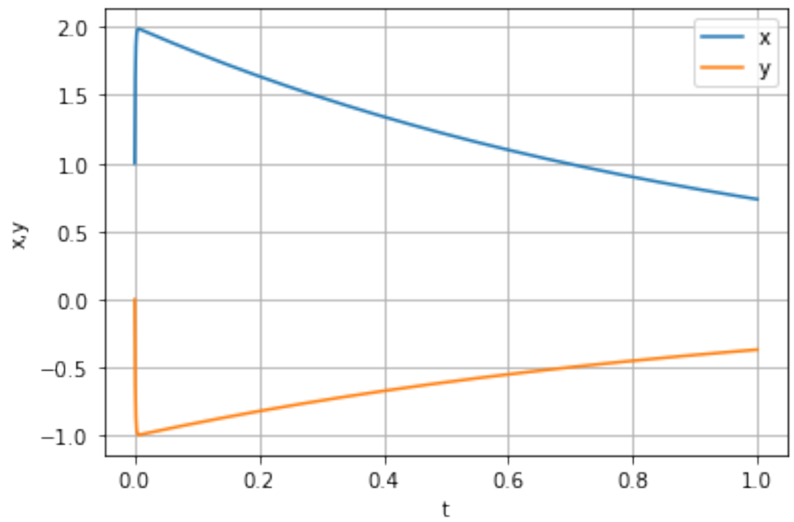


Where do the A^ks go? Remember that for matrices $(AB)^2 = ABAB$

The trouble with explicit solutions for stiff systems

Any x_k is obtained from the initial state x_0 as:

$$x_k = A^{-1}(I - h\Lambda)^k A x_0$$



If we want this to converge for all the elements of the matrix we need that

$$|1 - h\lambda_{max}| < 1$$

And so we need a step size h given by

$$h < \frac{2}{\lambda_{max}}$$

(which for $\lambda_{max} = 1000$ is very small compared to the other timescale in the problem of 1)

Plan for today

1. ~~Motivation - revision of coupled linear ODEs and illustration of stiff functions~~
2. How to do linear algebra with python - Sympy versus Numpy
3. Solution - solving a stiff linear ODE system with an implicit method
4. This week's tutorial - matrices and harmonic oscillator solution with implicit methods

Linear algebra using Python

Q: When should we use sympy and when numpy/scipy?

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

```python
import numpy as np

C_matrix = np.matrix([[998, 1998], [-999, -1999]])
C_inverse = np.linalg.inv(C_matrix)
eigenvalues, eigenvectors = np.linalg.eig(C_matrix)

print("The matrix is ", C_matrix)
print("Its inverse is ", C_inverse)
print("Eigenvalues are ", eigenvalues)
print("Eigenvectors are ", eigenvectors)
```

```
The matrix is  [[  998  1998]
 [ -999 -1999]]
Its inverse is  [[-1.999 -1.998]
 [ 0.999  0.998]]
Eigenvalues are  [   -1. -1000.]
Eigenvectors are  [[ 0.89442719 -0.70710678]
 [-0.4472136   0.70710678]]
```

Linear algebra using Python

Sympy when we expect whole number answers, or symbolic math
For numerics, mostly use numpy or scipy

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

```python
import numpy as np

C_matrix = np.matrix([[998, 1998], [-999, -1999]])
C_inverse = np.linalg.inv(C_matrix)
eigenvalues, eigenvectors = np.linalg.eig(C_matrix)

print("The matrix is ", C_matrix)
print("Its inverse is ", C_inverse)
print("Eigenvalues are ", eigenvalues)
print("Eigenvectors are ", eigenvectors)
```

```
The matrix is  [[  998  1998]
 [ -999 -1999]]
Its inverse is  [[-1.999 -1.998]
 [ 0.999  0.998]]
Eigenvalues are  [   -1. -1000.]
Eigenvectors are  [[ 0.89442719 -0.70710678]
 [-0.4472136   0.70710678]]
```

Sympy

For symbolic math and algebra. Useful for checking simple algebra, for more advanced symbolic maths I recommend SageMath or Mathematica

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

Usually import the whole class or function you need. Can also do:

```python
import sympy as sp
from sympy import Matrix, pprint
```

And use sp.function for less frequently used functions

Sympy

For symbolic math and algebra. Useful for checking simple algebra, for more advanced symbolic maths I recommend SageMath or Mathematica

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

Matrix is a class in sympy. Here we are instantiating an object of the Matrix class - setting its attributes (basically its size and entries) with the values given.

Sympy

For symbolic math and algebra. Useful for checking simple algebra, for more advanced symbolic maths I recommend SageMath or Mathematica

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

The Matrix class contains most of the methods you want for getting properties of the matrix - its inverse, determinant, eigenvalues etc.
Since they are methods (functions) and not attributes we need the brackets after them ().

Remember to think of these methods as saying,
“Hey C_matrix, give me your inverse!”

Sympy

For symbolic math and algebra. Useful for checking simple algebra, for more advanced symbolic maths I recommend SageMath or Mathematica

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

```
The matrix is
⎡998   1998 ⎤
⎢           ⎥
⎣-999  -1999⎦

 Its inverse is
⎡-1999   -999 ⎤
⎢──────  ─────⎥
⎢ 1000    500 ⎥
⎢             ⎥
⎢ 999     499 ⎥
⎢ ────    ─── ⎥
⎣ 1000    500 ⎦

 Eigenvalues and eigenvectors are
⎡⎛             ⎡⎡-1⎤⎤⎞  ⎛         ⎡⎡-2⎤⎤⎞⎤
⎢⎜-1000, 1,   ⎢⎢  ⎥⎥⎟, ⎜-1, 1,   ⎢⎢  ⎥⎥⎟⎥
⎣⎝             ⎣⎣1 ⎦⎦⎠  ⎝         ⎣⎣1 ⎦⎦⎠⎦
```

pprint() is a useful function for printing off sympy algebra in a nice way

Sympy

For symbolic math and algebra. Useful for checking simple algebra, for more advanced symbolic maths I recommend SageMath or Mathematica

```python
from sympy import Matrix, pprint

C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)
```

The matrix is

$$\begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

Its inverse is

$$\begin{bmatrix} \frac{-1999}{1000} & \frac{-999}{500} \\ \frac{999}{1000} & \frac{499}{500} \end{bmatrix}$$

Eigenvalues and eigenvectors are

$$\left[\left(-1000,\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right]\right),\ \left(-1,\ 1,\ \left[\begin{bmatrix} -2 \\ 1 \end{bmatrix}\right]\right)\right]$$

Q: What is the 1 in the middle here?

Sympy

Another example

```
M = Matrix([[3, -2,  4, -2], [5,  3, -3, -2], [5, -2,  2, -2], [5, -2, -3,  3]])
pprint(M.eigenvects())
```

$$\left[\left(-2,\ 1,\ \left[\begin{bmatrix}0\\1\\1\\1\end{bmatrix}\right]\right),\ \left(3,\ 1,\ \left[\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\right]\right),\ \left(5,\ 2,\ \left[\begin{bmatrix}1\\1\\1\\0\end{bmatrix},\ \begin{bmatrix}0\\-1\\0\\1\end{bmatrix}\right]\right)\right]$$

Q: What is the 2 in the middle here?

Sympy

Another example

```
M = Matrix([[3, -2,  4, -2], [5,  3, -3, -2], [5, -2,  2, -2], [5, -2, -3,  3]])
pprint(M.eigenvects())
```

$$\left[\left(-2,\ 1,\ \left[\begin{bmatrix}0\\1\\1\\1\end{bmatrix}\right]\right),\ \left(3,\ 1,\ \left[\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\right]\right),\ \left(5,\ 2,\ \left[\begin{bmatrix}1\\1\\1\\0\end{bmatrix},\ \begin{bmatrix}0\\-1\\0\\1\end{bmatrix}\right]\right)\right]$$

Value of the eigenvectors for each repeated eigenvalue

Value of the eigenvalue

This is the *multiplicity* of the eigenvalue (if we have repeated eigenvalues, it is > 1)

Sympy

```
from sympy import symbols, Eq, Function, pprint
from sympy.solvers.ode.systems import dsolve_system

x = Function("x")
y = Function("y")
t = symbols("t")

my_equations = [Eq(x(t).diff(t), 998*x(t) + 1998*y(t)),
                Eq(y(t).diff(t), -999*x(t) - 1999*y(t))]

solution = dsolve_system(my_equations)
pprint(solution[0][0])
pprint(solution[0][1])
```

```
                -1000⋅t          -t
x(t) = - C₁⋅e          - 2⋅C₂⋅e
              -1000⋅t        -t
y(t) = C₁⋅e          + C₂⋅e
```

Sympy can solve simple ODEs, but usually prefers to formulate them as a system of coupled algebraic expressions

Sympy

```
from sympy import symbols, Eq, Function, pprint
from sympy.solvers.ode.systems import dsolve_system

x = Function("x")
y = Function("y")
t = symbols("t")

my_equations = [Eq(x(t).diff(t), 998*x(t) + 1998*y(t)),
                Eq(y(t).diff(t), -999*x(t) - 1999*y(t))]

solution = dsolve_system(my_equations)
pprint(solution[0][0])
pprint(solution[0][1])
```

```
                -1000⋅t          -t
x(t) = - C₁⋅e          - 2⋅C₂⋅e
              -1000⋅t        -t
y(t) = C₁⋅e          + C₂⋅e
```

Again here Function is a class that sympy uses to represent variables that are functions of another variable.
Instead symbol is used for the independent variable t.

The Function class has a method that allows us to differentiate the function

Sympy

```
from sympy import symbols, Eq, Function, pprint
from sympy.solvers.ode.systems import dsolve_system

x = Function("x")
y = Function("y")
t = symbols("t")

my_equations = [Eq(x(t).diff(t), 998*x(t) + 1998*y(t)),
                Eq(y(t).diff(t), -999*x(t) - 1999*y(t))]

solution = dsolve_system(my_equations)
pprint(solution[0][0])
pprint(solution[0][1])
```

```
             -1000⋅t         -t
x(t) = - C₁⋅e        - 2⋅C₂⋅e
           -1000⋅t       -t
y(t) = C₁⋅e        + C₂⋅e
```

Another class is Eq for an equation LHS = RHS

Eq (RHS, LHS)

Sympy

```
from sympy import symbols, Eq, Function, pprint
from sympy.solvers.ode.systems import dsolve_system

x = Function("x")
y = Function("y")
t = symbols("t")

my_equations = [Eq(x(t).diff(t), 998*x(t) + 1998*y(t)),
                Eq(y(t).diff(t), -999*x(t) - 1999*y(t))]

solution = dsolve_system(my_equations)
pprint(solution[0][0])
pprint(solution[0][1])
```

```
                 -1000⋅t          -t
x(t) = - C₁⋅e          - 2⋅C₂⋅e
              -1000⋅t        -t
y(t) = C₁⋅e          + C₂⋅e
```

```
solution_with_ics = dsolve_system(my_equations, ics={x(0): 1, y(0): 0})
pprint(solution_with_ics[0][0])
pprint(solution_with_ics[0][1])
```

```
            -t    -1000⋅t
x(t) = 2⋅e    - e
            -t    -1000⋅t
y(t) = - e    + e
```

Can also feed in the initial conditions to dsolve_system()

Numpy and Scipy

```
import numpy as np

C_matrix = np.matrix([[998, 1998], [-999, -1999]])
C_inverse = np.linalg.inv(C_matrix)
eigenvalues, eigenvectors = np.linalg.eig(C_matrix)

print("The matrix is ", C_matrix)
print("Its inverse is ", C_inverse)
print("Eigenvalues are ", eigenvalues)
print("Eigenvectors are ", eigenvectors)
```

```
The matrix is  [[  998  1998]
 [ -999 -1999]]
Its inverse is  [[-1.999 -1.998]
 [ 0.999  0.998]]
Eigenvalues are  [   -1. -1000.]
Eigenvectors are  [[ 0.89442719 -0.70710678]
 [-0.4472136   0.70710678]]
```

Again numpy has a matrix class (small m!)

However, now most of the functions to get things like eigenvalues or inverses live not as methods in the class, but instead as methods in the library of functions **np.linalg**. These functions expect to act on objects of type “matrix”.

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
print(my_matrix**(-1))
```

```
print(my_array**(-1.0))
```

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
print(my_matrix**(-1))
```

```
[[ 0.66666667  0.33333333]
 [-0.33333333 -0.66666667]]
```

"Proper" matrix inverse

```
print(my_array**(-1.0))
```

```
[[ 0.5  1. ]
 [-1.  -0.5]]
```

1/element for each entry

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
my_vector_array * my_vector_array
```

```
my_vector_matrix * my_vector_matrix
```

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
my_vector_array * my_vector_array
```

```
array([ 9, 16])
```

Each entry in turn

```
my_vector_matrix * my_vector_matrix
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/p9/hydj_8nx5w3c8rkwjmgvty5r0000gp/T/ipykernel_38788/567733534.py in <module
>
----> 1 my_vector_matrix * my_vector_matrix

/opt/homebrew/anaconda3/lib/python3.9/site-packages/numpy/matrixlib/defmatrix.py in __mu
l__(self, other)
    216         if isinstance(other, (N.ndarray, list, tuple)) :
    217             # This promotes 1-D vectors to row vectors
--> 218             return N.dot(self, asmatrix(other))
    219         if isscalar(other) or not hasattr(other, '__rmul__') :
    220             return N.dot(self, other)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (1,2) and (1,2) not aligned: 2 (dim 1) != 1 (dim 0)
```

```
my_vector_matrix * my_vector_matrix.transpose()
```

```
matrix([[25]])
```

```
my_vector_matrix.transpose() * my_vector_matrix
```

```
matrix([[ 9, 12],
        [12, 16]])
```

Need to respect matrix shape rules and ordering

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
my_vector_matrix + my_vector_matrix
```

```
my_vector_array + my_vector_array
```

Numpy array versus matrix

```
import numpy as np
my_matrix = np.matrix([[2, 1], [-1, -2]])
my_array = np.array([[2, 1], [-1, -2]])
my_vector_matrix = np.matrix([3, 4])
my_vector_array = np.array([3, 4])

print(my_matrix)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_array)
```

```
[[ 2  1]
 [-1 -2]]
```

```
print(my_vector_matrix)
```

```
[[3 4]]
```

```
print(my_vector_array)
```

```
[3 4]
```

Q: How do the return values differ?

```
my_vector_matrix + my_vector_matrix
```

```
matrix([[6, 8]])
```

```
my_vector_array + my_vector_array
```

```
array([6, 8])
```

The same!

Plan for today

1. ~~Motivation - revision of coupled linear ODEs and illustration of stiff functions~~
2. ~~How to do linear algebra with python - Sympy versus Numpy~~
3. Solution - solving a stiff linear ODE system with an implicit method
4. This week's tutorial - matrices and harmonic oscillator solution with implicit methods

Explicit versus implicit methods

An explicit method is one where the variable we want at the next step y_{k+1} can be written explicitly in terms of quantities we know at the current step $y_k,\ t_k$, e.g.

$$y_{k+1} = y_k + h\, f(y_k, t_k) \qquad \text{“\textit{forward Euler - explicit}”}$$

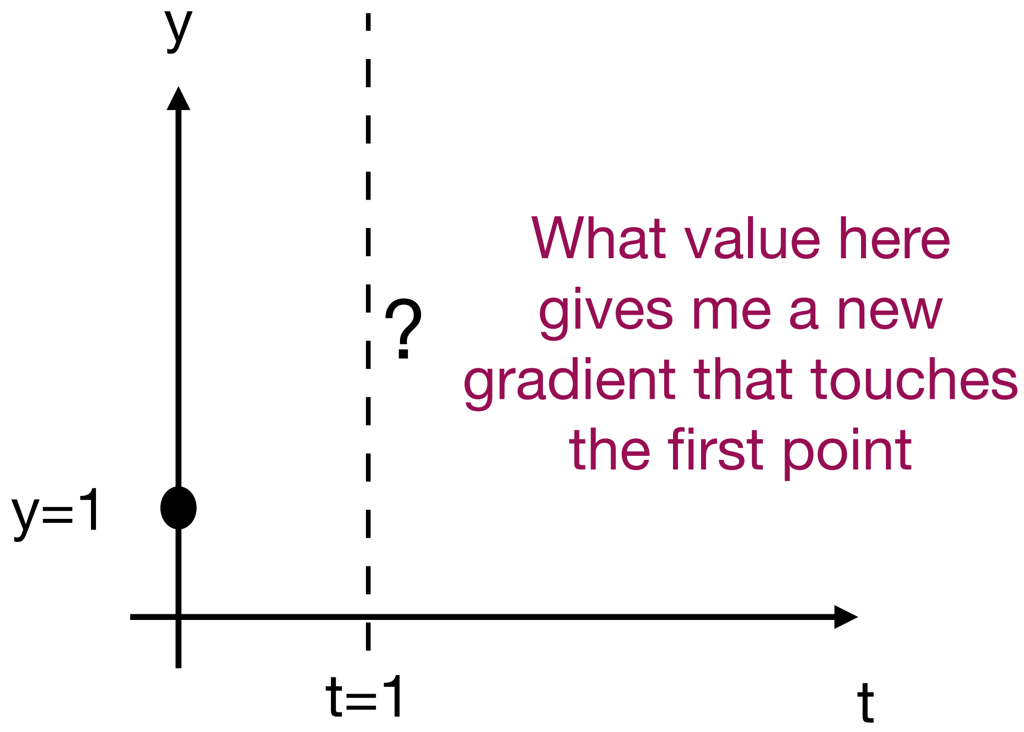
Implicit methods will instead result in equations where we cannot easily isolate and solve for the quantity we want,
e.g.

$$y_{k+1} = y_k + h\, f(y_{k+1}, t_{k+1}) \qquad \text{“\textit{backward Euler - implicit}”}$$

Change in paradigm for implicit methods

$$\frac{dy}{dt} = y^2 + y - 1$$

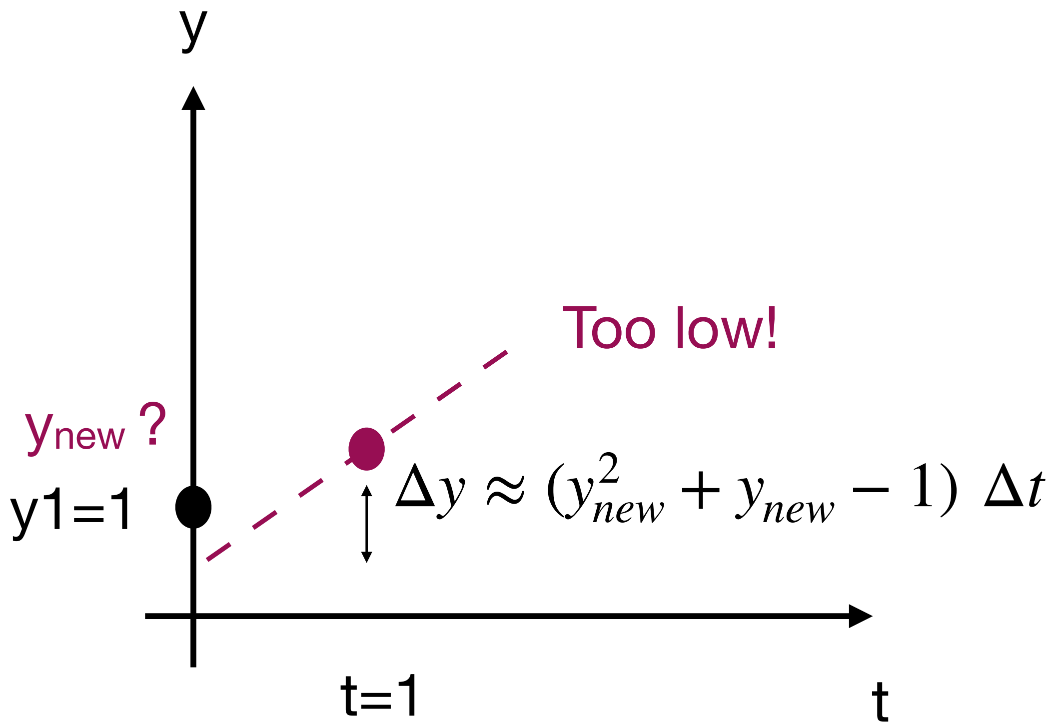
$$y(t = 0) = 1$$



Backward Euler's method

$$\frac{dy}{dt} = y^2 + y - 1$$

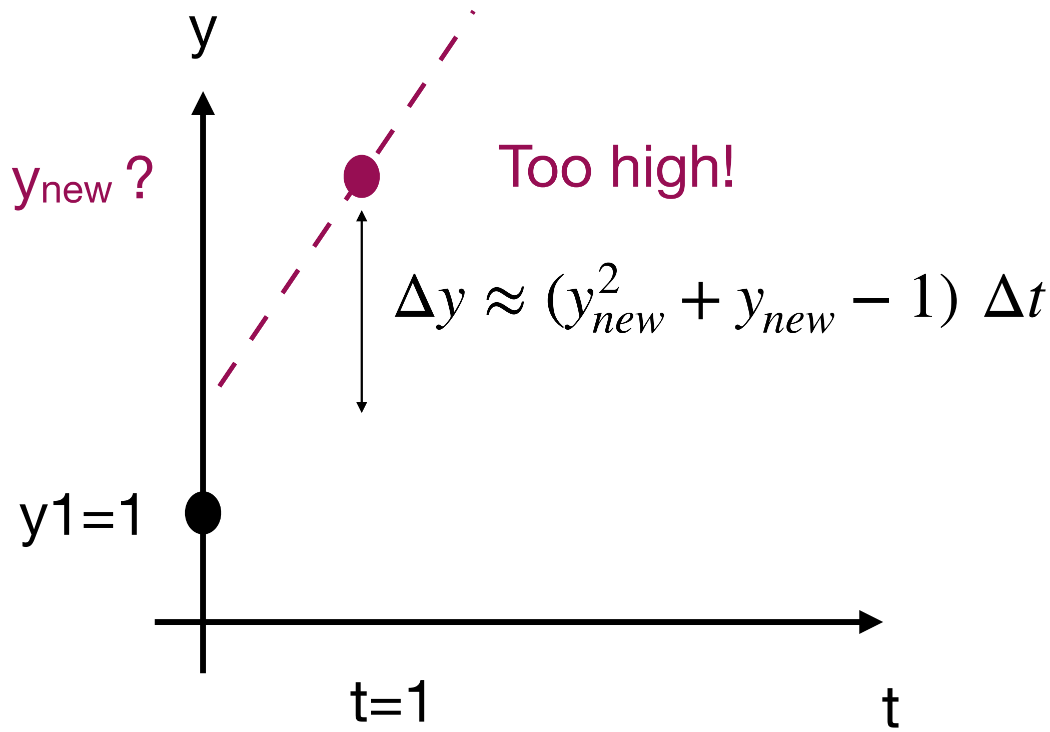
$$y(t = 0) = 1$$



Backward Euler's method

$$\frac{dy}{dt} = y^2 + y - 1$$

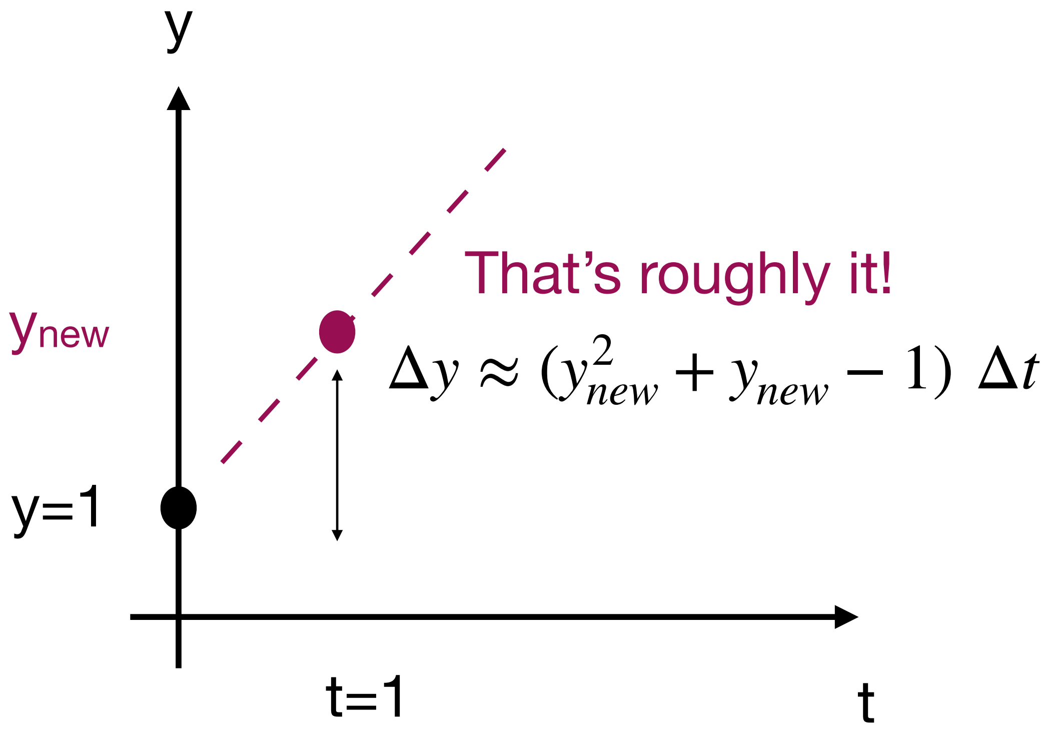
$$y(t=0) = 1$$



Backward Euler's method

$$\frac{dy}{dt} = y^2 + y - 1$$

$$y(t=0) = 1$$



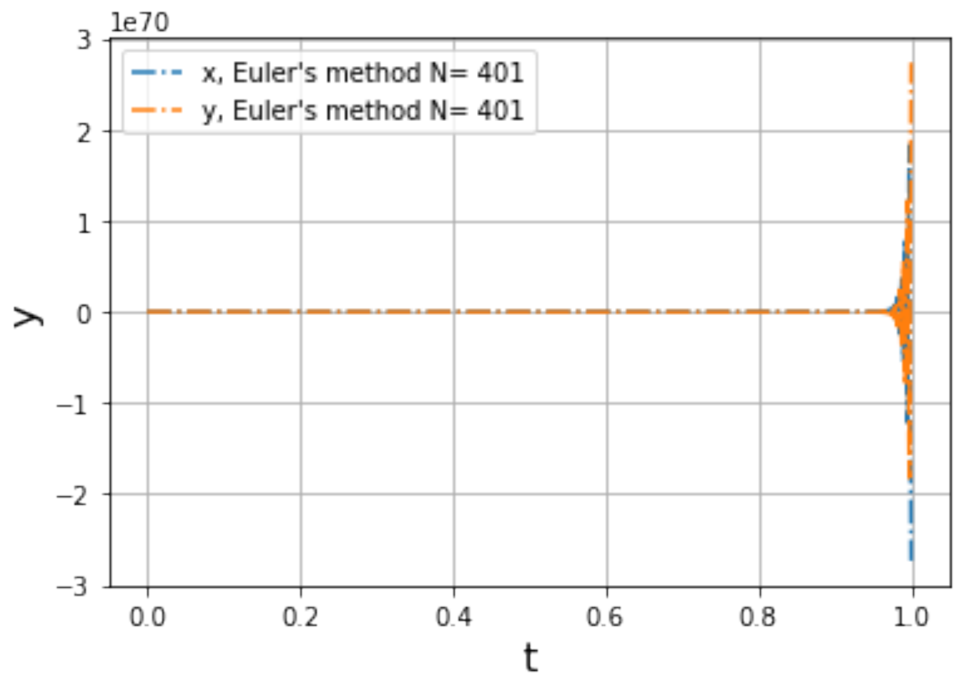
Let's solve our initial problem this way

Consider this simple first order dimension 2 linear ODE:

$$\dot{x} = 998x + 1998y \quad x(0) = 1$$

$$\dot{y} = -999x - 1999y \quad y(0) = 0$$

This is using Euler's forward method with 400 points



Let’s solve our initial problem this way

Let’s call the matrix C, and assume that it has only positive eigenvalues, so:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = -C\begin{bmatrix} x \\ y \end{bmatrix}$$

The backward Euler method with step size h is

$$x_{k+1} = x_k + h(-Cx_{k+1})$$

With a bit of matrix algebra can rearrange this so that:

$$x_{k+1} = (I + hC)^{-1}x_k$$

So we see that any x_k is obtained from the initial state x_0 by *k* applications of the matrix

$$x_k = (I + hC)^{-k}x_0$$

Let's solve our initial problem this way

Again, knowing that C is positive definite, this means that
it can be decomposed as

$$C = A^{-1} \Lambda A$$

with Λ a diagonal matrix of the eigenvalues.

A bit of matrix algebra gives:

$$(I + hC)^{-k} = A^{-1}(I + h\Lambda)^{-k} A$$

Now for convergence we need $\left| \frac{1}{1 + h\lambda_i} \right| < 1$ for all λ_i

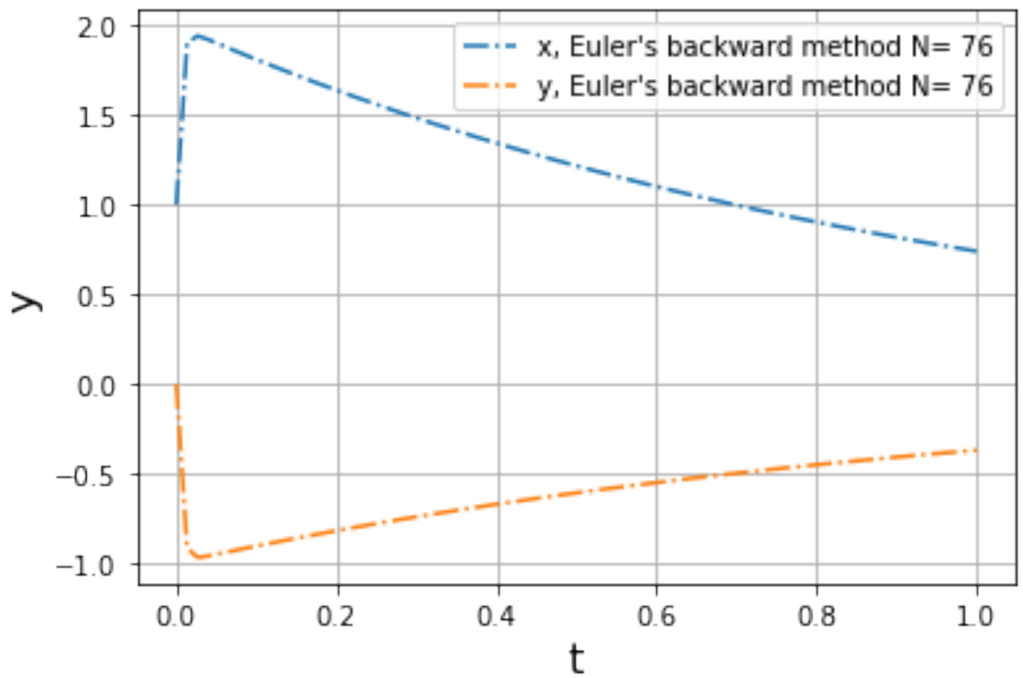
But this is always the case since h and λ_i are positive! Unconditional convergence!

Let's solve our initial problem this way

This is using Euler's backwards method with 75 points

This seems too good to be true!

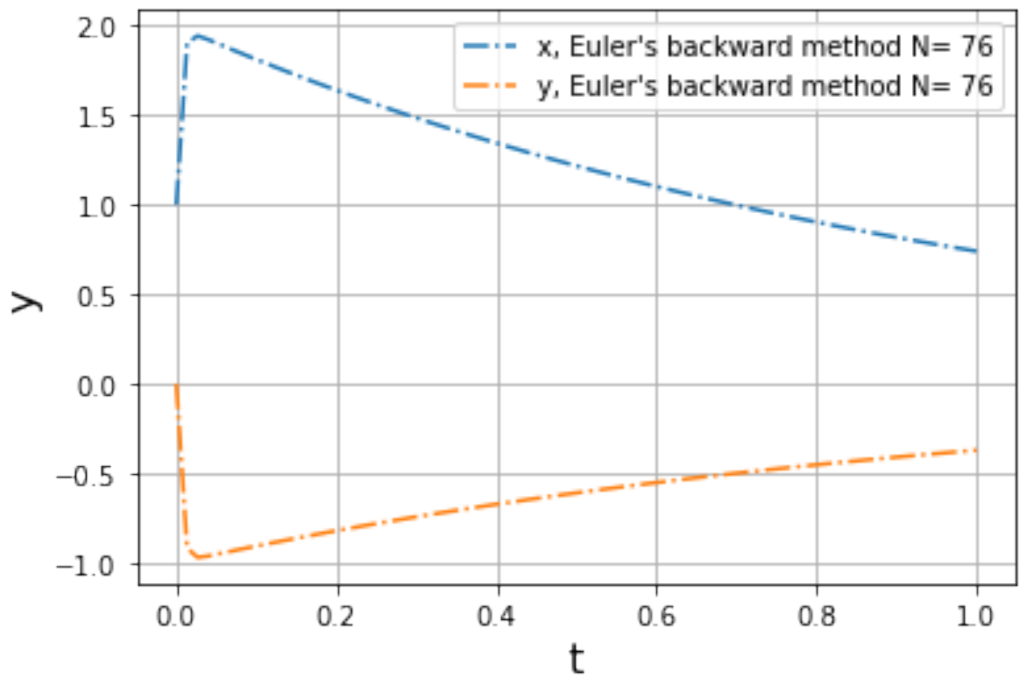
What's the catch?



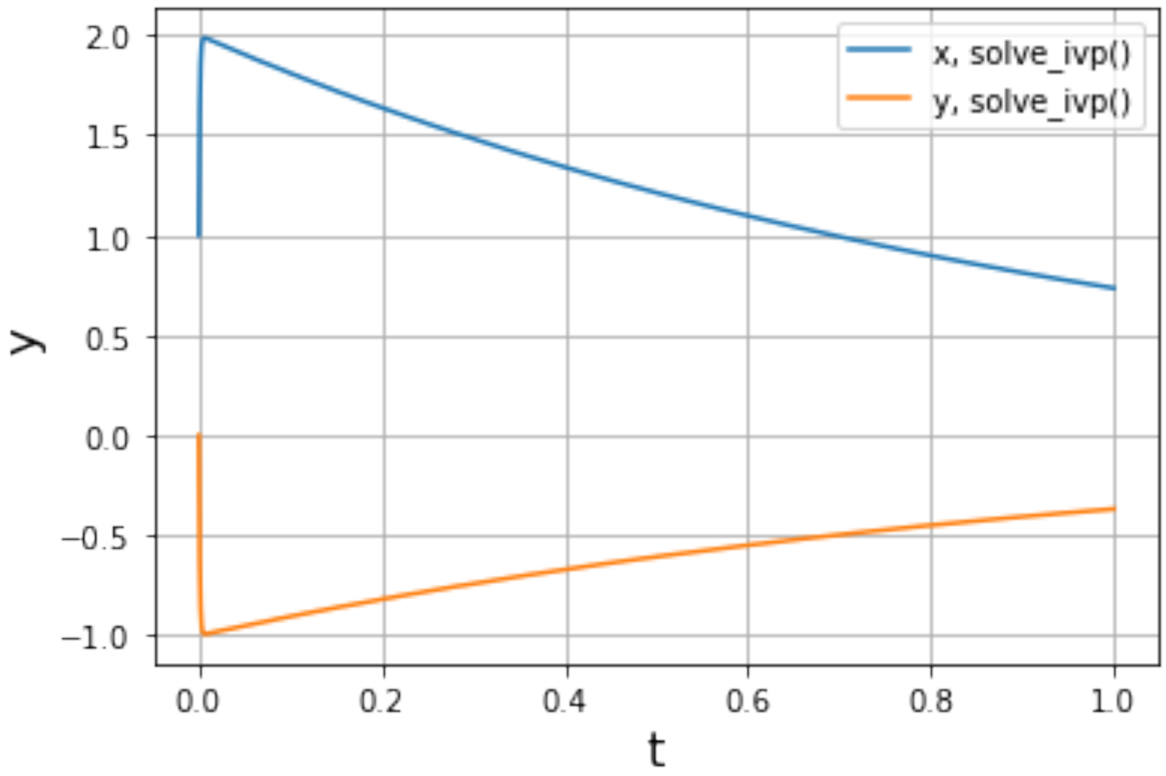
Let’s solve our initial problem this way

We had to invert a matrix to get this solution. That is trivial for a 2x2 system, but for higher dimension systems (that we we encounter with PDEs) it is VERY costly.

Also we have only considered linear systems, for a good reason! Non linear systems will be harder and will require iteration at each time step (more next week)



What about solve_ivp() ?



solve_ivp() detects the stiff system and either takes smaller steps initially or switches to another method (LSODA rather than RK45)

Plan for today

1. ~~Motivation - revision of coupled linear ODEs and illustration of stiff functions~~
2. ~~How to do linear algebra with python - Sympy versus Numpy~~
3. ~~Solution - solving a stiff linear ODE system with an implicit method~~
4. This week's tutorial - matrices and harmonic oscillator solution with implicit methods

Tutorial this week

ACTIVITY 1:

I have written a class below for integrating linear equations that implements the (explicit) forward Euler method using matrix methods. Update it to include the (implicit) backwards Euler method. Be sure to add in asserts to sense check what the class is doing.

The class is applied to the system we saw in the lectures:

$$\dot{x} = 998x + 1998y \quad x(0) = 1$$

$$\dot{y} = -999x - 1999y \quad y(0) = 0$$

which can also be written as

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}$$

Determine:

1. What is the maximum step size we can take while still keeping the Forward Euler method *stable*?
2. Is this consistent with the bounds we saw in the lecture?
3. What step size do we need to take in order to keep the Backward Euler method *stable*?
4. Is this consistent with the bounds we saw in the lecture?
5. What step size do we need to take in order to render the Backward Euler method *accurate*?

Implement and test the backwards Euler method

Tutorial this week

ACTIVITY 2:

Now apply the integrator to the following coupled, second order harmonic oscillator system.

HINT You first need to think carefully about what dimension this needs to be, and how to cast it into first order matrix form:

$$m_1\ddot{x}_1 = -kx_1 + k(x_2 - x_1) = -2kx_1 + kx_2$$
$$m_2\ddot{x}_2 = -kx_2 + k(x_1 - x_2) = -2kx_2 + kx_1$$

where k is the spring constant and m_1 and m_2 are the mass of the oscillators. Set the initial conditions as

$x_1 = 1 \quad \dot{x}_1 = 0$

$x_2 = 0 \quad \dot{x}_2 = 2$

Set $k = 1$ and $m_1 = 0.1$ and $m_2 = 10$

```
# Integrator for the coupled harmonic oscillator

# UPDATE ME!
```

Apply it to the coupled harmonic oscillator

Tutorial this week

ACTIVITY 3

Now we will try solving the systems with sympy. Below is the code for the lecture example. Update it to solve for the coupled harmonic oscillator above, checking against your numerical solution. Is that equation stiff or not? How can you tell?

```
# Solution of coupled linear equations using sympy

import sympy as sp
from sympy import symbols, Eq, Function, pprint, Matrix
from sympy.solvers.ode.systems import dsolve_system

# Compare the eigenvalue decomposition
C_matrix =  Matrix([[998, 1998],[-999, -1999]])
C_inverse = C_matrix.inv()
eigenvalues_and_vectors = C_matrix.eigenvects()

print("\n The matrix is ")
pprint(C_matrix)
print("\n Its inverse is ")
pprint(C_inverse)
print("\n Eigenvalues and eigenvectors are ")
pprint(eigenvalues_and_vectors)

# solve the linear system of ODEs
x = Function("x")
y = Function("y")
t = symbols("t")

my_equations = [Eq(x(t).diff(t), 998*x(t) + 1998*y(t)),
                Eq(y(t).diff(t), -999*x(t) - 1999*y(t))]
```

Try out some sympy

Tutorial this week

ACTIVITY 4

Which is faster, sympy or numpy?

Generate an NxN matrix containing random integers both sympy and numpy.

Compute the inverse using both libraries and calculate the time taken to do this. Repeat this for a range of N and see which one scales better - make a plot of your results. What do you conclude?

HINT Recall that we talked about timing functions in the Week 2 lecture.

```
# UPDATE ME!

my_matrix = numpy.random.randint(low=0, high=10, size=[3,3])

print(my_matrix)

print(np.linalg.inv(my_matrix))
```

```
[[8 6 0]
 [3 0 3]
 [6 9 8]]
[[ 0.10714286  0.19047619 -0.07142857]
 [ 0.02380952 -0.25396825  0.0952381 ]
 [-0.10714286  0.14285714  0.07142857]]
```

Compare the speed of sympy and numpy in inverting matrices