Week 6: The bad news: ill conditioned and ill posed problems

- Conditioning of linear systems and polynomials, well posedness of ODEs

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

Plan for today

1. Review of implicit integration methods, how to deal with non linear equations in implicit methods
2. Solving Ax=b without inverting the matrix A
3. Ill conditioned problems
4. Well posedness of ODEs
5. This week's tutorial (today we will see this as we go)

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.

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

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

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

The linear problem - C just contains numbers

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:

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

The non linear problem - C contains f(x,y)

Consider the Van der Pol oscillator

$$\frac{d^2y}{dt^2} - 2a(1-y^2)\frac{dy}{dt} + y = 0$$

How do we write this in matrix form?

The non linear problem - C contains f(x,y)

Consider the Van der Pol oscillator

$$\frac{d^2y}{dt^2} - 2a(1-y^2)\frac{dy}{dt} + y = 0$$

Decompose into 2 first order equations, then:

$$\frac{d}{dt}\begin{bmatrix} y \\ v \end{bmatrix} = -\begin{bmatrix} 0 & -1 \\ 1 & -2a(1-y^2) \end{bmatrix}\begin{bmatrix} y \\ v \end{bmatrix}$$

But C contains functions of y, help!

The non linear problem - C contains f(x,y)

Consider the Van der Pol oscillator

$$\frac{d}{dt}\begin{bmatrix} y \\ v \end{bmatrix} = -\begin{bmatrix} 0 & -1 \\ 1 & -2a(1-y^2) \end{bmatrix}\begin{bmatrix} y \\ v \end{bmatrix}$$

The backward Euler method with step size h is

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

Rearrange this so that:

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

I can't isolate x_{k+1} ! What should I do?

Iterate and hope for the best…

Algorithm:

Initially use the last value as a first try:

$$x_{k+1}^{guess(0)} = (I + hC(x_k))^{-1} x_k$$

Now use this new guess as the value in the matrix and repeat:

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

I stop when:

$$| x_{k+1}^{guess(i)} - x_{k+1}^{guess(i-1)} | < \epsilon$$

Iterate and hope for the best…

```
        y_of_t_old   = self._solution_y[:,itime-1]
        y_of_t_old_matrix = np.matrix(y_of_t_old)

        # Remember we have to copy if we don't want to actually
        # amend the elements of y_of_t_old_matrix
        y_of_t_guess = np.copy(y_of_t_old_matrix)

        # Implement the iterative scheme described in the lecture
        error = 100.0
        error_threshold = 1.0e-6
        while (error > error_threshold) :

            # Get the C matrix using the guess of y_new
            C_matrix_t = self.get_C_matrix(y_of_t_guess)
            I_plus_hC = np.eye(2) + h * C_matrix_t
            I_plus_hC_inv = np.linalg.inv(I_plus_hC)

            # Get the new guess using the C matrix of the old one
            y_of_t_new = I_plus_hC_inv * y_of_t_old_matrix.transpose()

            # If the new guess is the same as the old one we have converged
            error = np.linalg.norm(y_of_t_guess - y_of_t_new)
            y_of_t_guess = y_of_t_new

            # Check it is not getting worse!
            assert error < 1000.0, 'Non linear iterations not converging!'

        # Now (assuming it has converged) assign the
        # value we found to to solution and continue the time iteration
        y_of_t_new_final = y_of_t_guess.transpose()
        self._solution_y[:,itime] = y_of_t_new_final
```

We will see this in the tutorial.

What is the value of ϵ that we require here?

Iterate and hope for the best…

```
        y_of_t_old   = self._solution_y[:,itime-1]
        y_of_t_old_matrix = np.matrix(y_of_t_old)

        # Remember we have to copy if we don't want to actually
        # amend the elements of y_of_t_old_matrix
        y_of_t_guess = np.copy(y_of_t_old_matrix)

        # Implement the iterative scheme described in the lecture
        error = 100.0
        error_threshold = 1.0e-6
        while (error > error_threshold) :

            # Get the C matrix using the guess of y_new
            C_matrix_t = self.get_C_matrix(y_of_t_guess)
            I_plus_hC = np.eye(2) + h * C_matrix_t
            I_plus_hC_inv = np.linalg.inv(I_plus_hC)

            # Get the new guess using the C matrix of the old one
            y_of_t_new = I_plus_hC_inv * y_of_t_old_matrix.transpose()

            # If the new guess is the same as the old one we have converged
            error = np.linalg.norm(y_of_t_guess - y_of_t_new)
            y_of_t_guess = y_of_t_new

            # Check it is not getting worse!
            assert error < 1000.0, 'Non linear iterations not converging!'

        # Now (assuming it has converged) assign the
        # value we found to to solution and continue the time iteration
        y_of_t_new_final = y_of_t_guess.transpose()
        self._solution_y[:,itime] = y_of_t_new_final
```

We will see this in the tutorial.

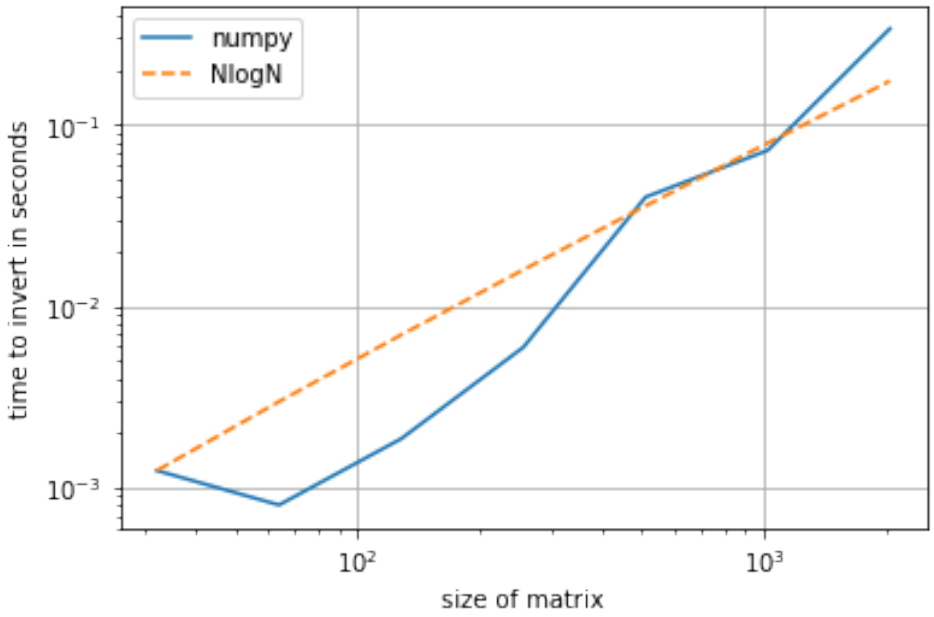
It is 10^{-6}

Plan for today

1. ~~Review of implicit integration methods, how to deal with non linear equations in implicit methods~~
2. Solving Ax=b without inverting the matrix A
3. Ill conditioned problems
4. Well posedness of ODEs
5. This week's tutorial (today we will see this as we go)

Solving Ax=b without inverting A

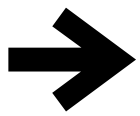
- The equation Ax=b comes up a lot in numerical methods!
- Not just in implicit integration of ODEs, but also in fitting functions, …
- As you will have seen in the week 6 lab, inverting a large matrix is expensive - naively the cost scales as N^2 but this can be reduced to NlogN.
- Can we invert without inverting?



Solving Ax=b without inverting A

- Method 1: Gauss Jordan elimination - covered in most linear algebra courses, we won't discuss more but it can be implemented as an algorithm

$$\mathbf{A} = \begin{pmatrix} 0 & 2 & 1 \\ 2 & -1 & 1 \\ 1 & 3 & 2 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 7 \\ 3 \\ 13 \end{pmatrix},$$



$$\left(\begin{array}{ccc|ccc|c} 0 & 2 & 1 & 1 & 0 & 0 & 7 \\ 2 & -1 & 1 & 0 & 1 & 0 & 3 \\ 1 & 3 & 2 & 0 & 0 & 1 & 13 \end{array}\right)$$

A $\qquad\qquad$ b

Many tedious row operations later…

$$\left(\begin{array}{ccc|ccc|c} 1 & 0 & 0 & -5 & -1 & 3 & 1 \\ 0 & 1 & 0 & -3 & -1 & 2 & 2 \\ 0 & 0 & 1 & 7 & 2 & -4 & 3 \end{array}\right)$$

A^{-1} $\quad$ x

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

1. Choose a matrix A_0 that is roughly equal to A, but that is easy to invert.

What could you choose in this example?

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

1. Choose a matrix A_0 that is roughly equal to A, but that is easy to invert.

We could choose for example

$$A_0 = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 2.0 \end{bmatrix}$$

Remember: The inverse of a diagonal matrix just inverts the elements

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

1. Choose a matrix A_0 that is roughly equal to A, but that is easy to invert.

2. Work out its inverse A_0^{-1} and the difference between $\Delta A = A - A_0$

What would these be in our example?

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

1. Choose a matrix A_0 that is roughly equal to A, but that is easy to invert.

2. Work out its inverse A_0^{-1} and the difference between $\Delta A = A - A_0$

We have

$$A_0 = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 2.0 \end{bmatrix} \quad \text{so } A_0 = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 0.5 \end{bmatrix} \quad \text{and } \Delta A = \begin{bmatrix} -0.1 & -0.2 \\ -0.3 & 0.1 \end{bmatrix}$$

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

1. Choose a matrix A_0 that is roughly equal to A, but that is easy to invert.

2. Work out its inverse A_0^{-1} and the difference between $\Delta A = A - A_0$

3. Plugging this into Ax=b, we obtain the condition on the solution x that

$$x = A_0^{-1}(b - \Delta A x)$$

-> This suggests an iterative scheme $x_{k+1} = A_k^{-1}(b - \Delta A_k x_k)$

Solving Ax=b without inverting A

- Method 2: Iterative improvement

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Algorithm

We can write it as

$$x_{k+1} = Rx_k + c$$

Where we define the residual matrix $R = -A_0^{-1}\Delta A_0$ and $c = A_0^{-1}b$

How do we know if the iterations will converge?

Solving Ax=b without inverting A

Test for convergence

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$x_{k+1} = Rx_k + c$$

Since

$$x_1 = Rx_0 + c$$

$$x_2 = R(Rx_0 + c) + c$$

...

$$x_n = R^n x_0 + (1 + R + R^2 \ldots R^{n-1})\ c \qquad \Longrightarrow\ R^n \to 0 \text{ as } n \to \infty$$

Solving Ax=b without inverting A

Theorem

$R^n \to 0 \text{ as } n \to \infty$

if and only if

$\rho(R) < 1$ where

$\rho(R) = \max |\lambda_i|$ (the maximum of the absolute values of the eigenvalues of the matrix) is called ***the spectral radius of R***

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

What can we do to try to make the spectral radius of R small?

Solving Ax=b without inverting A

Theorem

$R^n \to 0 \text{ as } n \to \infty$

$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

if and only if

$\rho(R) < 1$ where

$\rho(R) = \max |\lambda_i|$ (the maximum of the absolute values of the eigenvalues of the matrix) is called ***the spectral radius of R***

$R = -A_0^{-1}\Delta A_0$ so we want to choose A_0 to be close to A, so that the difference ΔA_0 is small

Iterative improvement in the tutorial

ACTIVITY 2:

Inverting without inverting!

In the lecture we discussed how to invert a matrix without using the python linalg.inv function by iterating from an initial guess.

Here you should implement an algorithm to invert the matrix we saw in the tutorial:

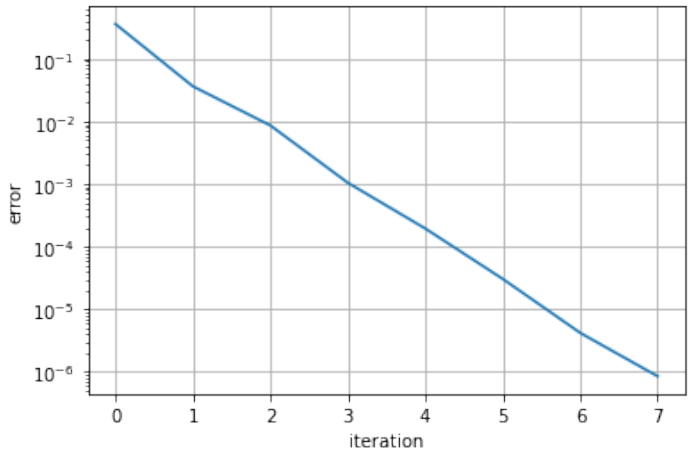
$$\begin{bmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

You can start with an initial guess $x_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

You can implement this using functional methods or using classes, but your solutions should:

1. Display the outputs at every iteration
2. Plot the decrease in the change between each iteration (using a matrix norm like linalg.norm())
3. Calculate and output the spectral radius of the residual matrix R
4. Stop once a certain level of accuracy is obtained - how many iterations are required to get an accuracy of 10^{-6} ? How strongly does this depend on the initial guess?

In the tutorial you will implement this method in code for this simple matrix.



Plan for today

1. ~~Review of implicit integration methods, how to deal with non linear equations in implicit methods~~
2. ~~Solving Ax=b without inverting the matrix A~~
3. Ill conditioned problems
4. Well posedness of ODEs

} "Problems with the problem"

5. This week's tutorial (today we will see this as we go)

Problems with the problem

- We have already discussed problems with the ***methods*** used to solve ODEs, but we have assumed, without much justification, that the ***problem itself*** can always be solved in a satisfactory way, so long as we apply the right method.
- Bad news: real life is not so kind!
- We will now discuss 2 distinct problems:
  - **1. Ill conditioned problems** - this is about the sensitivity of the solution to small changes in the coefficients
  - **2. Ill posed problems** - this is about whether an ODE (later we will see the same for PDEs) actually has a solution, and whether the solution is unique

Ill conditioned problems

- We have some equations (any equations - a polynomial, a matrix, an ODE…) with parameters λ_i.
- The equations have solutions represented by k functions $f_k(\lambda_i)$.
- The problem is said to be ***ill conditioned*** if *small changes in λ_i result in large or non smooth changes in $f_k(\lambda_i)$.*

This doesn't mean that the problem doesn't have a solution, so why is this an issue?

Ill conditioned problems

- We have some equations (any equations - a polynomial, a matrix, an ODE…) with parameters λ_i.
- The equations have solutions represented by k functions $f_k(\lambda_i)$.
- The problem is said to be ***ill conditioned*** if *small changes in λ_i result in large or non smooth changes in $f_k(\lambda_i)$.*

> *Numerically we will only ever be able to represent λ_i with finite precision, so if the solution changes wildly for a small difference in the values, we won't be able to trust it.*

Ill conditioned problems - test for Ax=b

- The ***condition number C*** tells us how much (using some norm) a small change in the input parameters changes the solution:

$$\frac{\|\Delta f_k(\lambda_i)\|}{\|f_k(\lambda_i)\|} \leq C \frac{\|\Delta \lambda_i)\|}{\|\lambda_i\|}$$

- For a matrix equation Ax=b we will use the ***row-sum norm*** and we will see that:

$$\frac{\|\Delta x)\|}{\|x\|} \leq C \frac{\|\Delta b)\|}{\|b\|} \quad \textit{with } C = \|A\|\|A^{-1}\|$$

The matrix equation is ill conditioned if $C > 10^n$ where n is the number of equations (usually the number of elements in x)

Ill conditioned problems - test for Ax=b

- The ***row-sum norm*** of an m x n matrix is:

$$\|A\| = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|$$

- *In words:* for each row, compute the sum of the absolute values of the elements, then take the maximum of these sums.

- Useful property of this norm: $\|AB\| \le \|A\|\|B\|$

Ill conditioned problems - test for Ax=b

- The ***row-sum norm*** of an m x n matrix is:

$$\|A\| = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|$$

- *In words:* for each row, compute the sum of the absolute values of the elements, then take the maximum of these sums.

- Useful property of this norm: $\|AB\| \le \|A\|\|B\|$

Why do I need this norm? Why can't I just use the matrix entries?

Ill conditioned problems - test for Ax=b

- Time for some matrix algebra! We perturb the input b by some Δb which gives a change in the solution Δx:

$$
\begin{aligned}
& Ax = b \\
\implies & A(x + \Delta x) = b + \Delta b \\
\implies & A(\Delta x) = \Delta b \\
\implies & \Delta x = A^{-1}\Delta b
\end{aligned}
$$

Ill conditioned problems - test for Ax=b

- Time for some matrix algebra! We perturb the input b by some Δb which gives a change in the solution Δx:

$$\Delta x = A^{-1}\Delta b \quad \textit{plus the row sum norm inequality}$$

$$\implies \|\Delta x\| = \|A^{-1}\Delta b\| \leq \|A^{-1}\|\|\Delta b\|$$

but also $\|b\| = \|Ax\| \leq \|A\|\|x\|$ *so then*

$$\implies \|\Delta x\|\|b\| \leq \|A^{-1}\|\|\Delta b\|\|A\|\|x\|$$

Why do I need this norm? Why can't I just use the matrix entries?

Ill conditioned problems - test for Ax=b

- I need the norm so I can rearrange this equation, ie, divide through by the norm values (which are just numbers - how would I divide by a matrix?!)
- Then

$$\|\Delta x\|\|b\| \leq \|A^{-1}\|\|\Delta b\|\|A\|\|x\|$$

gives $\dfrac{\|\Delta x)\|}{\|x\|} \leq C\dfrac{\|\Delta b)\|}{\|b\|}$ *with* $C = \|A\|\|A^{-1}\|$ *as stated before*

Ill conditioned problems - test for Ax=b

- Example: $$\begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7.999 \end{bmatrix}$$

In the tutorial you will show that $A = \begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix}$, $A^{-1} = \begin{bmatrix} -3999 & 2000 \\ 2000 & 1000 \end{bmatrix}$

What is the condition number $C = \|A\|\|A^{-1}\|$?

Is A ill conditioned? ($C > 10^n$)

Ill conditioned problems - test for Ax=b

- Example: $$\begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7.999 \end{bmatrix}$$

In the tutorial you will show that $A = \begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix}$, $A^{-1} = \begin{bmatrix} -3999 & 2000 \\ 2000 & 1000 \end{bmatrix}$

What is the condition number

$$C = \|A\|\|A^{-1}\|?$$

Is A ill conditioned? ($C > 10^n$)

The condition number is 5.999 x 5999 = 35988, which is bigger that 100 (10^2), so yes it is.

Iterative improvement in the tutorial

ACTIVITY 3

This activity is on checking ill conditioning in a matrix problem $Ax = b$.

Here you should implement an algorithm to check ill conditioned matrices and test it on the matrix problem we saw in the lecture:

$$\begin{bmatrix} 1 & 2 \\ 2 & 3.999 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7.999 \end{bmatrix}$$

You can implement this using functional methods or using classes, but your solutions should:

1. Have a function or method to calculate the row sum norm of a given matrix as defined in the lecture
2. Have a function or method to calculate the condition number $C = \|A\|\|A^{-1}\|$ of a given matrix
3. Write a test function that gives an assert error when it detects an ill conditioned matrix
4. Test explicitly the effect of one or two examples of small changes in b on the solution x, and verify that they respect the inequality

$$\frac{\|\Delta x)\|}{\|x\|} \leq C \frac{\|\Delta b)\|}{\|b\|}$$

In the tutorial you will write code to solve the above problem and check the conditioning of any matrix

Plan for today

1. ~~Review of implicit integration methods, how to deal with non linear equations in implicit methods~~
2. ~~Solving Ax=b without inverting the matrix A~~
3. ~~Ill conditioned problems~~
4. Well posedness of ODEs

} "Problems with the problem"

5. This week's tutorial (today we will see this as we go)

Well posed problems

- An initial value problem is well posed if:
  - A solution exists
  - The solution is unique
  - The solution depends continuously on the initial data

What does it mean for a solution to not exist?
How could this happen?

Well posed problems - conditions for ODEs

- In general cases (PDEs) it is often not easy to diagnose whether an initial value problem is well posed - many standard examples are solved, but others are not.
- For ODEs there is a condition that can be tested, and that is to check that the function is ***Lipschitz continuous***. Whilst it applies for ODEs of all dimensions, we will focus here on the case with only dimension 1 for clarity, that is, the initial value problem:

$$\frac{dy}{dt} = f(y,t) \quad \text{defined on a subset of the reals } D \subset \mathbb{R}^2$$

$$D = \{(t,y) \mid a \leq t \leq b,\ -\infty \leq y \leq \infty\} \text{ with } y(a) = \alpha$$

Well posed problems - conditions for ODEs

- ***If the function f is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial value problem is well posed*.**

$$\frac{dy}{dt} = f(y,t) \quad \text{defined on a subset of the reals } D \subset \mathbb{R}^2$$

- What is a Lipschitz condition? If a finite constant $L > 0$ exists such that:

$$|f(y_1,t) - f(y_2,t)| \leq L\,|y_1 - y_2| \quad \forall (t,y1),(t,y2) \in D$$

We say that f satisfies a Lipschitz condition with Lipschitz constant L.

Well posed problems - conditions for ODEs

- ***If the function f is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial value problem is well posed*.**

Example 1: $$\frac{dy}{dt} = y - t^2 + 1 \quad 0 \le t \le 2 \quad y(0) = 0.5$$

Since $\frac{\partial f}{\partial y} = 1 \, \forall (t, y)$ then $|f(y_1, t) - f(y_2, t)| \le 1 \times |y_1 - y_2|$

So f is Lipschitz continuous with L = 1 and so the initial value problem is well posed.

Well posed problems - conditions for ODEs

- ***If the function f is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial value problem is well posed*.**

Example 2: $$\frac{dy}{dt} = y^2 t \quad y(t_0) = \alpha > 0$$

with $D = \{(t, y) \mid t_0 \leq t \leq T, \ -\infty \leq y \leq \infty\}$

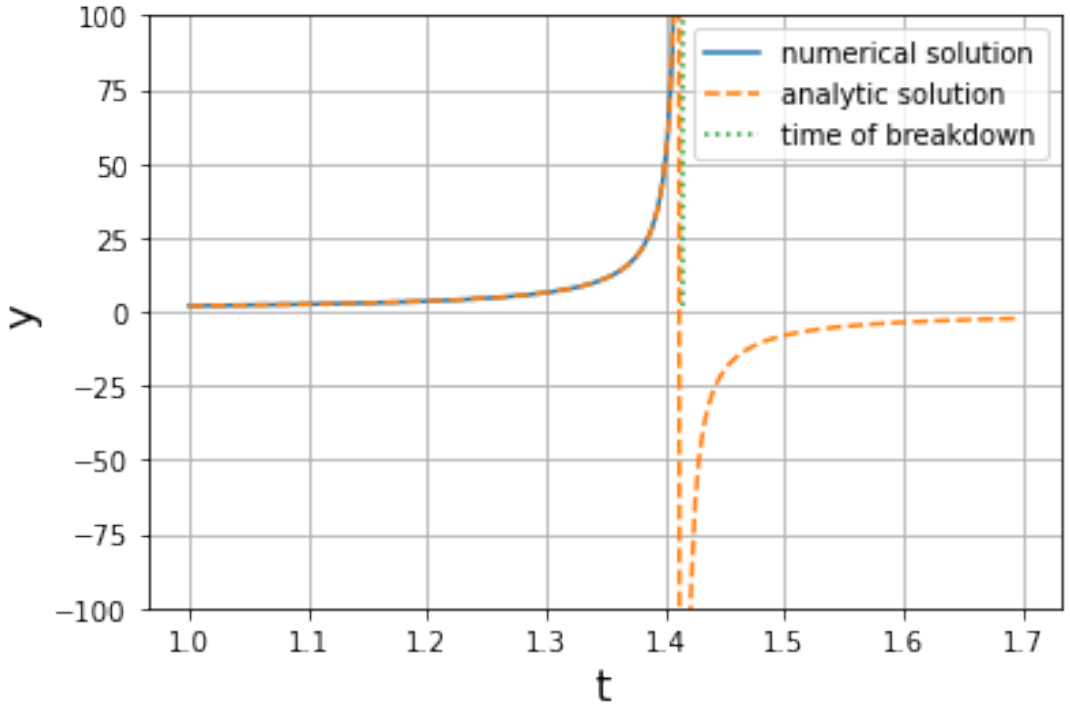
- The inequality is unbounded as it can depend on y, so there is no Lifshitz condition. The initial value problem is ill posed.

Well posed problems - conditions for ODEs

- The problem here is that although y has a unique solution, it is unbounded and breaks down at

$$t = \sqrt{\frac{2}{\alpha} + t_0^2}$$

- If we restrict the time domain to less than this value of t, we are ok, but if we try to cross it the solution (formulated as an initial value problem) will fail. We will see this in the tutorial.



Well posed problems - conditions for ODEs

- ***What does the Lipschitz condition do for us?***

$$|f(y_1, t) - f(y_2, t)| \leq L\,|y_1 - y_2| \quad \forall (t, y1), (t, y2) \in D$$

We can show from this (via Grönwall's inequality) that for any solutions $x_1(t), x_2(t)$ to the ODE

$$|x_1(t) - x_2(t)| \leq e^{Lt}\,|x_1(0) - x_2(0)|$$

- If $x_1(0) = x_2(0)$ this tells us that the solution is ***unique*** since the RHS is zero.
- If $x_1(0) = a, x_2(0) = a + \delta$ it tells us that the solution changes by an amount that is bounded by $\delta\ e^{Lt}$ - this is the meaning of "***depends continuously on the initial data***".

What other "well posed" condition have I not explicitly proved here?

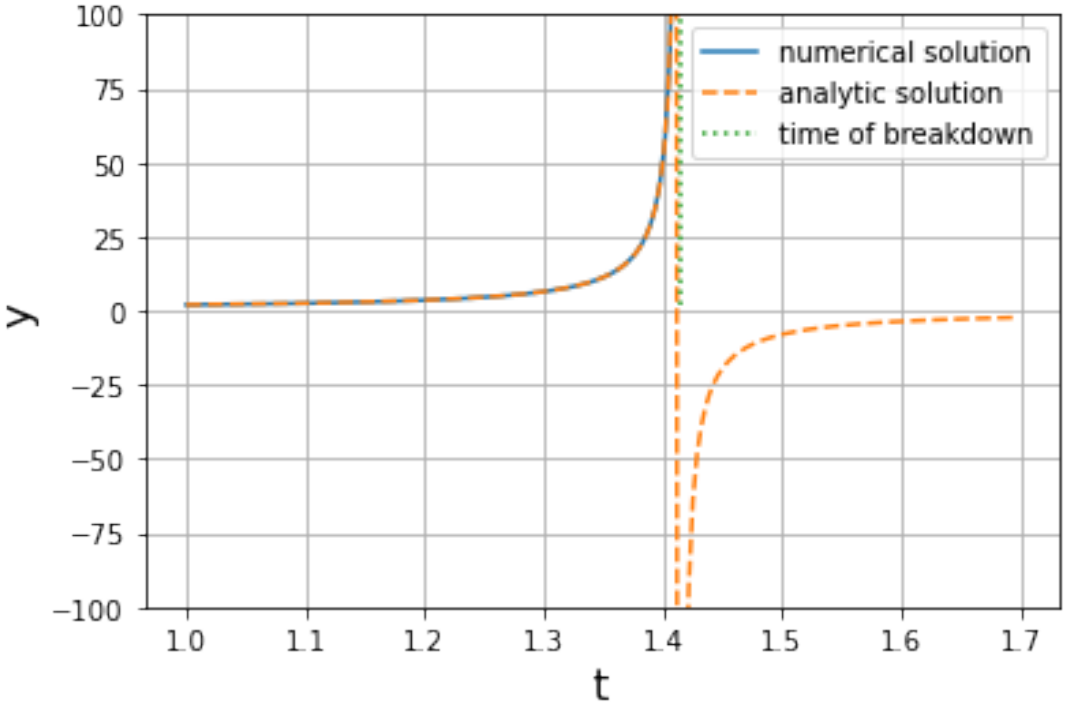
Well posed problems - conditions for ODEs

- ***What does the Lipschitz condition do for us?***

$$|f(y_1, t) - f(y_2, t)| \leq L|y_1 - y_2| \quad \forall (t, y1), (t, y2) \in D$$

I haven't proved ***existence*** of solutions, but it can also be derived (less straightforwardly) from this condition. One needs to use Banach's fixed point theorem, and this takes us away from the main points of the course, but do look it up if you are interested!

Well posed problems in the tutorial



In the tutorial you will write code to try to solve the ODE example we considered and show how it breaks down

Well posed problems - very active area of QMUL research!

- QMUL Maths is one of the leading places for solving issues of well-posedness in modified gravity theories!

Aron Kovacs
“the well-posed guy”

Well-Posed Formulation of Scalar-Tensor Effective Field Theory
Áron D. Kovács and Harvey S. Reall
Phys. Rev. Lett. 124, 221101

Summary of problems with problems

In principle:

- Before you start working on a problem, consider whether it could be ill posed or ill conditioned.
- Build checks into operations to look for ill conditioned matrices, etc

In practise:

- When something doesn’t work despite your best efforts, consider whether there is a fundamental issue with what you are trying to do!

Plan for today

1. Review of implicit integration methods, how to deal with non linear equations in implicit methods
2. Solving Ax=b without inverting the matrix A
3. Ill conditioned problems
4. Well posedness of ODEs

} "Problems with the problem"

5. This week's tutorial (today we will see this as we go)