

MAS212 Scientific Computing and Simulation

Dr. Sam Dolan

School of Mathematics and Statistics,
University of Sheffield

Autumn 2016

<http://sam-dolan.staff.shef.ac.uk/mas212/>

G30 Hicks Building
s.dolan@sheffield.ac.uk

Today's lecture

- **Well-conditioned** and **ill-conditioned** problems
- Example: Wilkinson's polynomial
- Matrices, norms and **condition number**
- Iterative methods:
 - Convergence and spectral radius
 - Solving matrix equations iteratively

Conditioning

- Suppose we have some **problem** with parameters $\lambda_j \dots$
- \dots and a **solution** represented by functions $f_k(\lambda_j)$.
- We say that the problem is:
 - **well-conditioned** if small changes in λ_j produce small changes in f_k .
 - **ill-conditioned** if small changes in λ_j produce some **large** or **non-smooth** changes in f_k .
- The **condition number** C is an attempt to quantify the sensitivity of the solution to changes in the parameters.
- As small changes in parameters can be produced by numerical errors (e.g. round-off error), numerical solutions of ill-conditioned problems are **unreliable**.

Example: Wilkinson's polynomial

- Consider the polynomial

$$P_n(x) = (x - 1)(x - 2) \dots (x - n) = \prod_{k=1}^n (x - k)$$

- The equation $P_n(x) = 0$ has **roots** $x = 1, 2, \dots, n$.
- Now consider a **slightly-perturbed** polynomial, for example,

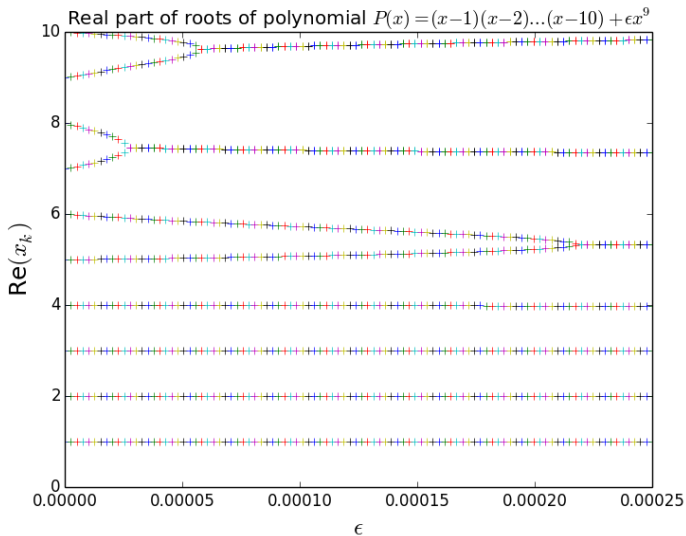
$$\tilde{P}_n(x) = P_n(x) + \epsilon x^{n-1}, \quad \epsilon \ll 1.$$

- How do the roots of $\tilde{P}_n(x)$ differ from the roots of $P_n(x)$?
- Let us investigate by plotting the roots x_k as a function of ϵ , for the cases $n = 10$ and $n = 15$.

Example: Wilkinson's polynomial

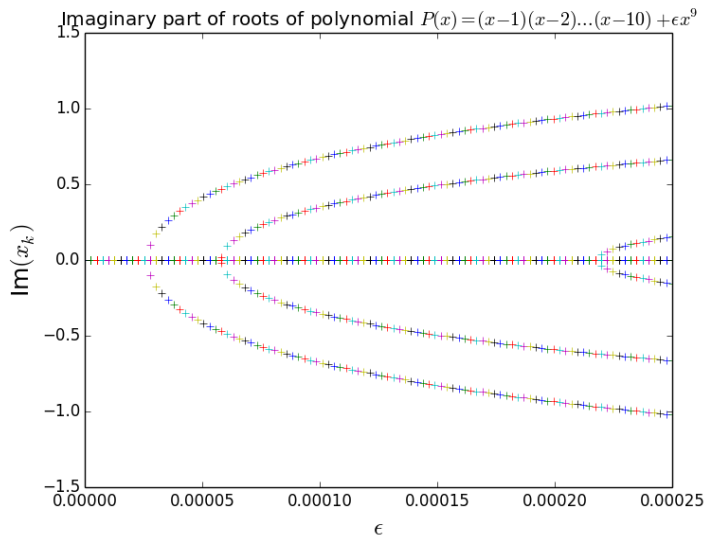
```
import numpy as np
import matplotlib.pyplot as plt
n = 10
# Coefficients of Wilkinson's polynomial for n=10
a = np.array([1,-55,1320,-18150,157773,-902055,3416930,
              -8409500,12753576,-10628640,3628800], dtype=np.float64)
print "The unperturbed roots are : ", np.roots(a)
b = np.copy(a) # for the modified coefficients
eps = np.linspace(0.0, 2.5e-4, 100)
for ep in eps:
    b[1] = a[1] + ep # make a small change to  $x^{n-1}$  coefficient
    roots = np.roots(b) # find the roots of the modified polynomial
    plt.plot(ep*np.ones(n), roots.real, '+')
plt.show()
```

Example: Wilkinson's polynomial



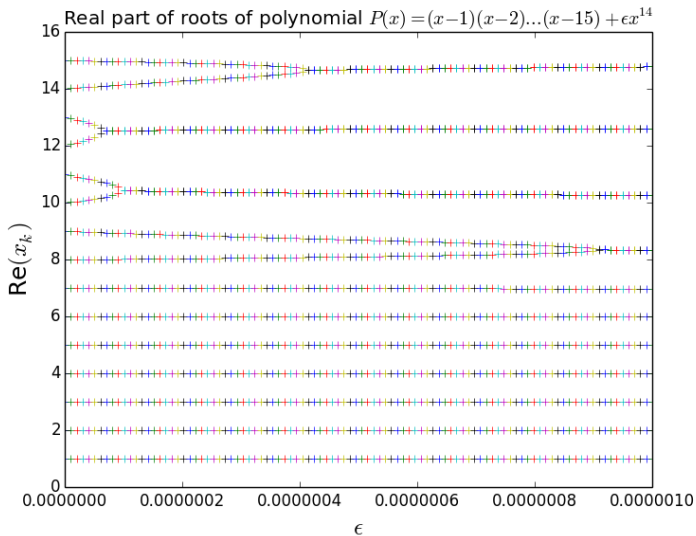
$n = 10$, real part of root

Example: Wilkinson's polynomial



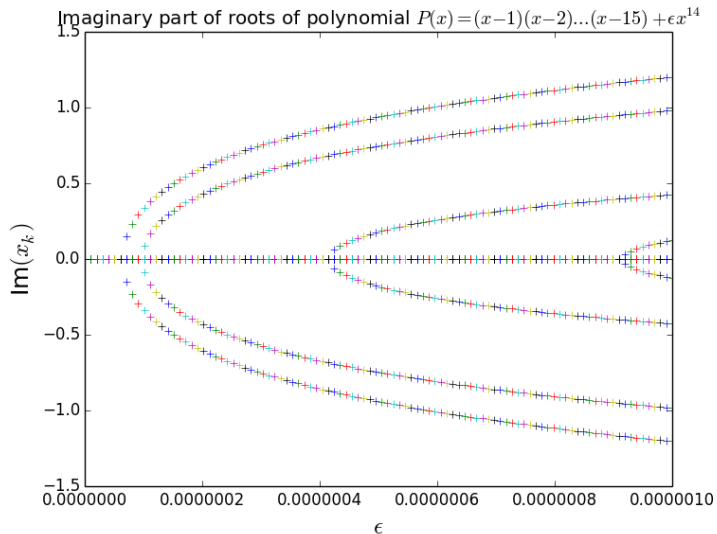
$n = 10$, imaginary part of root

Example: Wilkinson's polynomial



$n = 15$, real part of root

Example: Wilkinson's polynomial



$n = 15$, imaginary part of root

Roots of Wilkinson's polynomial

- For $n = 15$ case, the first merger of roots occurs around $\epsilon \sim 10^{-7}$
- For $n = 20$ case, a tiny change $\epsilon \sim 10^{-10}$ has a large effect on at least one pair of roots.
- As n increases, the roots become **extremely sensitive** to the polynomial coefficients
⇒ problem is **ill-conditioned**.
- Wilkinson used this polynomial to illustrate the **ubiquity** of ill-conditioned problems, and later commented on the impact of the discovery:

In *The Perfidious Polynomial* (1984)

“Speaking for myself I regard it as the most traumatic experience in my career as a numerical analyst.”

A cautionary tale

- Suppose we wished to find the **eigenvalues** λ of a large $n \times n$ matrix A .
- The eigenvalues are the roots of the **characteristic polynomial** $p(x)$ defined by

$$p(x) = \det(A - xI)$$

- As Wilkinson's example shows, large- n polynomials can be **ill-conditioned**, even when the roots are not close together (i.e. even when eigenvalue problem is well-conditioned).

Matrix equations

- In numerical analysis, we frequently encounter problems of the form

$$\mathbf{Ax} = \mathbf{b}$$

where \mathbf{A} is an $n \times n$ matrix, \mathbf{b} is vector of known values, and \mathbf{x} is a vector of unknowns.

- For example, in fitting a linear model to a data set, we found **normal equations** of this form: $(\mathbf{X}^T \mathbf{X})\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{y})$.
- We would like to be able to test, in practical cases, whether the system of equations is **ill-conditioned**.
- Even better, we would like to calculate a **condition number** C : the ‘**worst case**’ ratio of the relative change in output to the relative change in input.

Example:

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

Is this system well-conditioned?

- The solution is $x = 2, y = 1$.
- Now change the RHS slightly, to $\begin{pmatrix} 4.001 \\ 7.998 \end{pmatrix}$, and recalculate.
- The new solution is $x = -3.999, y = 4$
- A small change in b produced a **large** change in the solution \Rightarrow **ill-conditioned** system.
- How could we see this **without** computing the solutions?

```
import numpy as np
A = np.matrix("1 2 ; 2 3.999")
b = np.matrix("4 ; 7.999")
db = np.matrix("0.001 ; -0.001") # a small perturbation
sol0 = np.linalg.solve(A,b)
sol1 = np.linalg.solve(A,b+db)
print("Original solution:\n", sol0)
print("Perturbed solution:\n", sol1)
```

Original solution:

```
[[ 2.]
```

```
[ 1.]]
```

Perturbed solution:

```
[[ -3.999]
```

```
[ 4.   ]]
```

- How could we see this **without** finding the solutions?

Definition :

- The **row sum norm** of an $m \times n$ matrix A is defined as

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

where a_{ij} is the element in the i th row and j th column of A .

- i.e. for each row, compute the sum of the absolute values of its elements; then take the maximum of these sums; this is the row-sum norm.

- Suppose we introduce some small change Δb in b , which produces a change Δx in the solution x , i.e.,

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

- We make use of an inequality for the row-sum norm, which holds that for any matrix or vector B, C :

$$\|BC\| \leq \|B\| \|C\|$$

- For example,

$$\begin{aligned}\|b\| &= \|Ax\| \\ \Rightarrow \|b\| &\leq \|A\| \|x\|\end{aligned}$$

- Similarly,

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

- Thus,

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

and so

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

where $C \equiv \|A\| \|A^{-1}\|$.

Definition :

The **condition number** for a matrix A is defined as

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

- We have shown that

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

- A matrix equation $Ax = b$ is said to be **ill-conditioned** if

$$C \gtrsim 10^n$$

where n is the number of equations.

Example:

Find the condition number of the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 3.999 \end{pmatrix}$$

Solution :

- Determinant: $\det A = 3.999 - 4 = -0.001$
- Matrix inverse:

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} = -1000 \times \begin{pmatrix} 3.999 & -2 \\ -2 & 1 \end{pmatrix}$$

- Row-sum norms:

$$\begin{aligned} \|A\| &= \max(1 + 2, 2 + 3.999) = 5.999 \\ \|A^{-1}\| &= 1000 \times \max(3.999 + 2, 2 + 1) = 5999 \end{aligned}$$

- Condition number:

$$\Rightarrow C = \|A\| \|A^{-1}\| = 35988$$

- $C \approx 3.6 \times 10^4 > 10^2 \Rightarrow Ax = b$ is **ill-conditioned**.

```
def norm(A):  
    """Returns the row sum norm of A."""  
    rowsums = [abs(A[k,:]).sum() for k in range(A.shape[0])]   
    return max(rowsums)  
  
def cond(A):  
    """Returns the condition number of A"""  
    return norm(A)*norm(np.linalg.inv(A))  
  
def wellcond(A):  
    condA = cond(A)  
    w = "well" if (condA < 10**(A.shape[0])) else "ill"  
    print("A is " + w + "-conditioned: C = %e" % condA)
```

```
>>> wellcond(A)  
A is ill-conditioned: C = 3.598800e+04
```

Iterative improvement

$$\mathbf{Ax} = \mathbf{b}$$

- Suppose $\mathbf{A} = \mathbf{A}_0 + \Delta\mathbf{A}$ where \mathbf{A}_0 is some matrix whose inverse is **known**, $\mathbf{B}_0 \equiv \mathbf{A}_0^{-1}$.
- Suppose we wanted to solve **iteratively**, i.e. **without** finding the inverse directly, or applying Gaussian elimination.
- Then, starting with an initial guess \mathbf{x}_0 , we may try an iterative approach:

$$\begin{aligned}\mathbf{A}_0\mathbf{x} &= \mathbf{b} - \Delta\mathbf{A}\mathbf{x} \\ \Rightarrow \mathbf{x}_{k+1} &= \mathbf{B}_0(\mathbf{b} - \Delta\mathbf{A}\mathbf{x}_k) \\ &= \mathbf{R}\mathbf{x}_k + \mathbf{c}\end{aligned}$$

where $\mathbf{R} \equiv -\mathbf{B}_0\Delta\mathbf{A}$ is the **residual matrix** and $\mathbf{c} = \mathbf{B}_0\mathbf{b}$.

Iterative improvement

$$\mathbf{x}_{k+1} = \mathbf{R}\mathbf{x}_k + \mathbf{c}$$

- When do iterative methods of this type **converge**?
- Repeated application leads to

$$\mathbf{x}_n = \mathbf{R}^n \mathbf{x}_0 + (\mathbf{I} + \mathbf{R} + \mathbf{R}^2 + \dots + \mathbf{R}^{n-1}) \mathbf{c}$$

- For convergence, we require $\mathbf{R}^n \rightarrow 0$ as $n \rightarrow \infty$.

Spectral radius and convergence

Definition :

The **spectral radius** $\rho(\mathbf{R})$ of an $n \times n$ matrix \mathbf{R} is given by the maximum magnitude of its eigenvalues λ_i :

$$\rho(\mathbf{R}) = \max_{i=1\dots n} |\lambda_i|.$$

Theorem :

- $\lim_{k \rightarrow \infty} \mathbf{R}^k = 0$ **if and only if** $\rho(\mathbf{R}) < 1$
- Thus an iterative method can be used iff \mathbf{R} is '**small enough**' that all of its eigenvalues have a magnitude of less than unity.
- Iterative methods are used (e.g.) to efficiently solve the linear equations arising in *implicit methods*.

Example:

Let

$$\mathbf{A} = \begin{pmatrix} 1.1 & 0.2 \\ -0.3 & 1.9 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Solve the matrix equation $\mathbf{Ax} = \mathbf{b}$ iteratively with

$$\mathbf{A}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

Example

- Let $\mathbf{B}_0 = \mathbf{A}_0^{-1}$

$$\begin{aligned}(\mathbf{A}_0 + \Delta\mathbf{A})\mathbf{x} &= \mathbf{b} \\ \Rightarrow (\mathbf{I} + \mathbf{B}_0\Delta\mathbf{A})\mathbf{x} &= \mathbf{B}_0\mathbf{b} \\ \Rightarrow \mathbf{x} &= -(\mathbf{B}_0\Delta\mathbf{A})\mathbf{x} + \mathbf{B}_0\mathbf{b}\end{aligned}$$

- Turn into an iterative equation:

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

where $\mathbf{R} = -\mathbf{B}_0\Delta\mathbf{A}$ and $\mathbf{c} = \mathbf{B}_0\mathbf{b}$

- In this case, $\mathbf{A}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \Rightarrow \mathbf{B}_0 = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$ and

$$\Delta\mathbf{A} = \begin{pmatrix} 0.1 & 0.2 \\ -0.3 & -0.1 \end{pmatrix}.$$

Example

```
B0 = np.matrix("1 0 ; 0 0.5")
dA = np.matrix("0.1 0.2 ; -0.3 -0.1")
b = np.matrix("1; 2")
R = -np.dot(B0, dA)
c = np.dot(B0, b)
x0 = np.matrix("0 ; 1")
for k in range(13):
    print(k, x0.transpose())
    x0 = R*x0 + c
```

Example

```
0 [[0 1]]
1 [[ 0.8  1.05]]
2 [[ 0.71  1.1725]]
3 [[ 0.6945  1.165125]]
4 [[ 0.697525  1.16243125]]
5 [[ 0.69776125  1.16275031]]
6 [[ 0.69767381  1.1628017 ]]
7 [[ 0.69767228  1.16279116]]
8 [[ 0.69767454  1.1627904 ]]
9 [[ 0.69767447  1.1627907 ]]
10 [[ 0.69767441  1.1627907 ]]
11 [[ 0.69767442  1.1627907 ]]
12 [[ 0.69767442  1.1627907 ]]
```