Scientific Computation

Spring, 2019

Lecture 17

Imperial College
London

Wavenumber analysis

- Consider 'wave,' $f(x) = e^{ikx}$
- and its derivative, $f' = ike^{ikx}$

- Finite difference approximation of f':

$$f_{i+1} = e^{ik(x+h)}$$

$$f_{i-1} = e^{ik(x-h)}$$

$$\frac{(f_{i+1} - f_{i-1})}{2h} = \frac{e^{ikh} - e^{-ikh}}{2h} e^{ikx}$$

$$\frac{(f_{i+1} - f_{i-1})}{2h} = i \left[\frac{\sin(kh)}{h} \right] e^{ikx}$$

Wavenumber analysis

- Consider 'wave,' $f(x) = e^{ikx}$
- and its derivative, $f' = ike^{ikx}$
- Finite difference approximation of f':

$$f_{i+1} = e^{ik(x+h)}$$

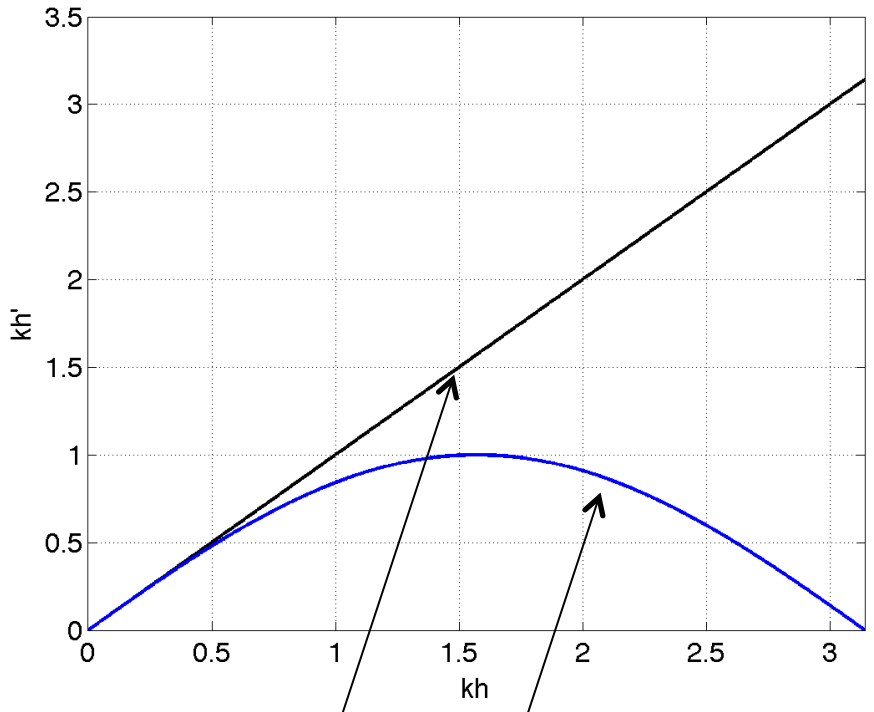
$$f_{i-1} = e^{ik(x-h)}$$

$$\frac{(f_{i+1} - f_{i-1})}{2h} = \frac{e^{ikh} - e^{-ikh}}{2h} e^{ikx}$$

$$\frac{(f_{i+1} - f_{i-1})}{2h} = i \left[\frac{\sin(kh)}{h} \right] e^{ikx}$$

Wavenumber analysis

- **Compare 'modified wavenumber,' $\sin(kh)$ to kh**



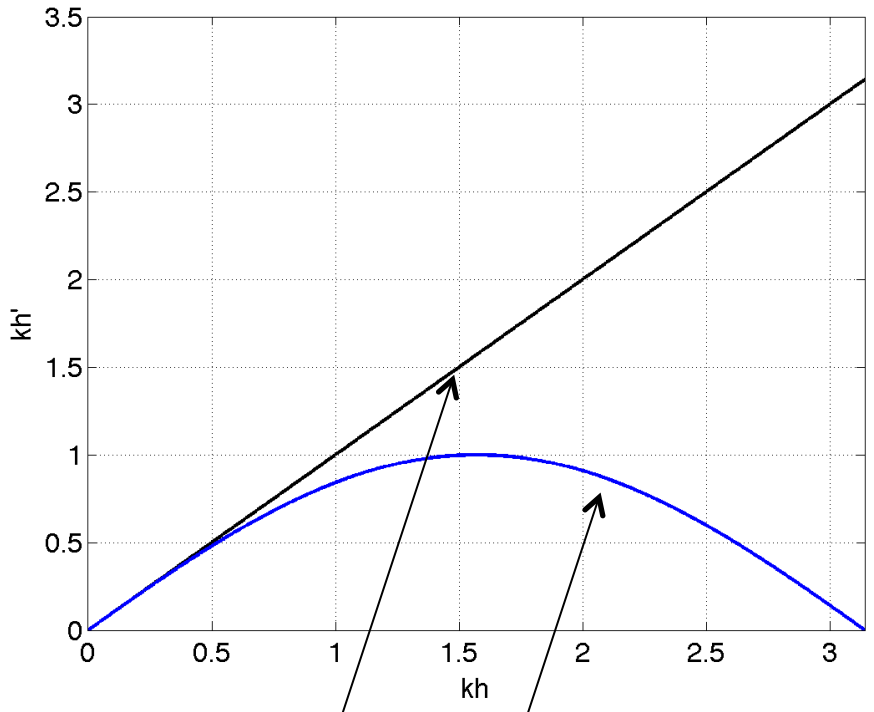
- **FD only accurate for low wavenumbers (long waves)**

Black: exact

Blue: 2nd-order finite difference

Wavenumber analysis

- **Compare 'modified wavenumber,' $\sin(kh)$ to kh**



Black: exact

Blue: 2nd-order finite difference

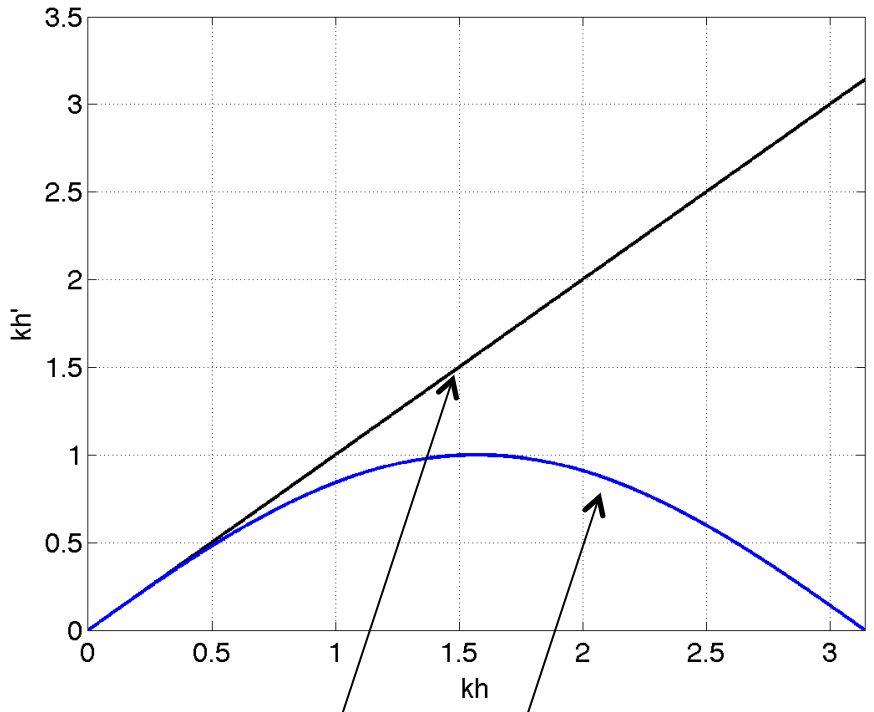
- **FD only accurate for low wavenumbers (long waves)**
- **1% error when** $kh \approx 0.39$

$$kh = \frac{2\pi h}{\lambda}$$

$$\lambda/h \approx 16$$

Wavenumber analysis

- **Compare 'modified wavenumber,' $\sin(kh)$ to kh**



Black: exact

Blue: 2nd-order finite difference

- **FD only accurate for low wavenumbers (long waves)**
- **1% error when** $kh \approx 0.39$

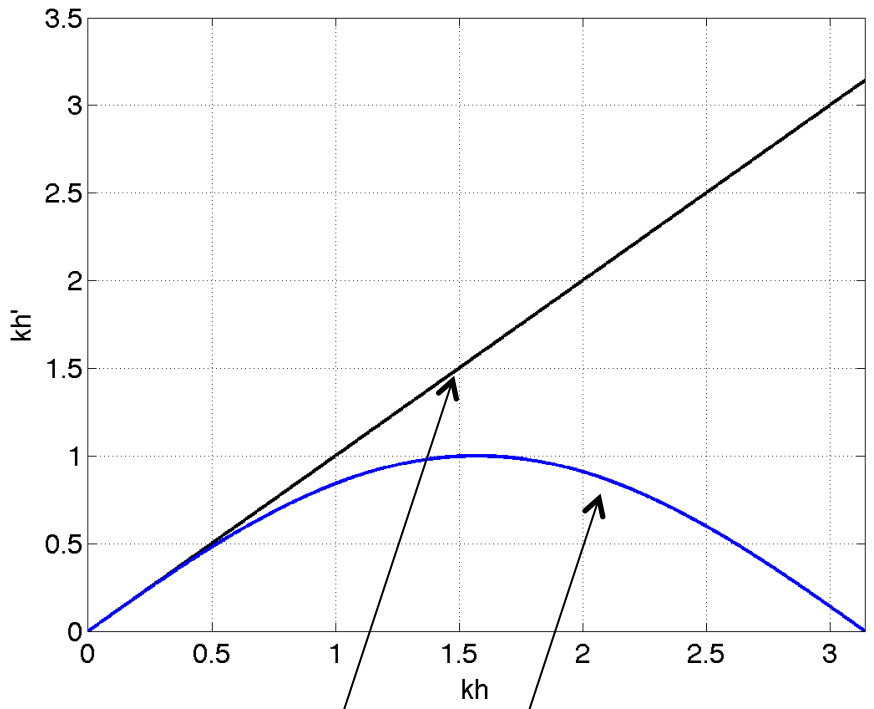
$$kh = \frac{2\pi h}{\lambda}$$

$$\lambda/h \approx 16$$

- **So, need ~16 points/wavelength for ~1 % error**
- **Spectral (Fourier): >2 points/ wavelength, exact**

Wavenumber analysis

- **Compare 'modified wavenumber,' $\sin(kh)$ to kh**



Black: exact

Blue: 2nd-order finite difference

- **FD only accurate for low wavenumbers (long waves)**
- **1% error when** $kh \approx 0.39$

$$kh = \frac{2\pi h}{\lambda}$$

$$\lambda/h \approx 16$$

- **So, need ~16 points/wavelength for ~1 % error**
- **Spectral (Fourier): >2 points/ wavelength, exact**

Exercise: What is modified wavenumber for 2nd derivative?:

$$f_i'' = \frac{(f_{i+1} - 2f_i + f_{i-1})}{h^2}$$

FD schemes with better resolution?

- **Think about time marching:**
  - **Explicit Euler has poor stability properties**
  - **Implicit Euler: much better stability, but requires matrix inversion → more expensive**

- **Explicit finite difference:** $f'_i = \frac{(f_{i+1} - f_{i-1})}{2h}$

- **Or 'implicit' finite difference stencils:**

$$\beta f'_{i-2} + \alpha f'_{i-1} + f'_i + \alpha f'_{i+1} + \beta f'_{i+2} =$$

$$c\frac{f_{i+3} - f_{i-3}}{6h} + b\frac{f_{i+2} - f_{i-2}}{4h} + a\frac{f_{i+1} - f_{i-1}}{2h}$$

- **Now have a system of equations (a *banded* matrix)**
- **Lots of coefficients to play with!**

$$\begin{bmatrix} f_1 & g_1 & h_1 & & & & \\ e_2 & f_2 & g_2 & h_2 & & & \\ d_3 & e_3 & f_3 & g_3 & h_3 & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & d_{n-1} & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & & & d_n & e_n & f_n \end{bmatrix}$$

Designing your FD stencil

$$\beta f'_{i-2} + \alpha f'_{i-1} + f'_i + \alpha f'_{i+1} + \beta f'_{i+2} =$$

$$c\frac{f_{i+3} - f_{i-3}}{6h} + b\frac{f_{i+2} - f_{i-2}}{4h} + a\frac{f_{i+1} - f_{i-1}}{2h}$$

- **First think about truncation error:**
  - **Taylor series + lots of algebra →**

2nd order: $a + b + c = 1 + 2\alpha + 2\beta$

4th order: $a + 2^2 b + 3^2 = \frac{4!}{2!}(\alpha + 2^2\beta)$

6th order: $a + 2^4 b + 3^4 c = \frac{6!}{4!}(\alpha + 2^4\beta)$

Designing your FD stencil

2nd order: $a + b + c = 1 + 2\alpha + 2\beta$

4th order: $a + 2^2 b + 3^2 = \frac{4!}{2!}(\alpha + 2^2\beta)$

6th order: $a + 2^4 b + 3^4 c = \frac{6!}{4!}(\alpha + 2^4\beta)$

- **Highest possible accuracy: 10th order:**

$$\alpha = \frac{1}{2}, \beta = \frac{1}{20}, a = \frac{17}{12}, b = \frac{101}{150}, c = \frac{1}{100}$$

- **But is this the best scheme?**

Wavenumber analysis

$$\beta f'_{i-2} + \alpha f'_{i-1} + f'_i + \alpha f'_{i+1} + \beta f'_{i+2} =$$

$$c\frac{f_{i+3}-f_{i-3}}{6h} + b\frac{f_{i+2}-f_{i-2}}{4h} + a\frac{f_{i+1}-f_{i-1}}{2h}$$

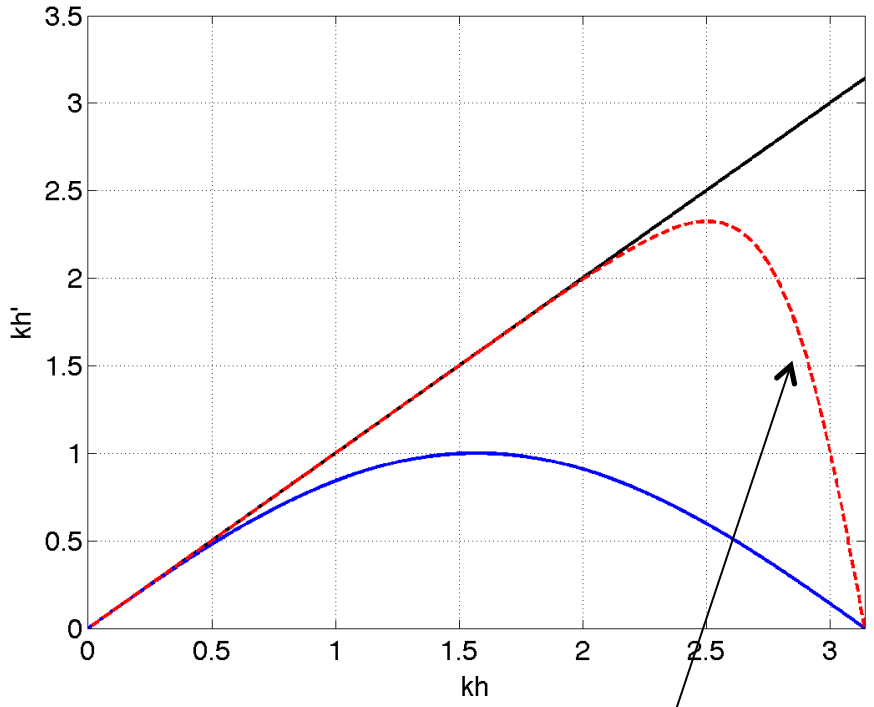
- **Same approach as before,** $f(x) = e^{ikx}$

$$f_{i+1} = e^{ik(x+h)}$$

gives the general modified wavenumber:

$$kh' = \frac{a\sin(kh)+(b/2)\sin(2kh)+(c/3)\sin(3kh)}{1+2\alpha\cos(kh)+2\beta\cos(2kh)}$$

Wavenumber analysis



- Previous result for 2nd order FD
- 10th order ~ 3 points/wavelength for 1% error
- But can we do better?

Black: exact

Blue: 2nd-order finite difference

Red: 10th-order finite difference

Wavenumber analysis

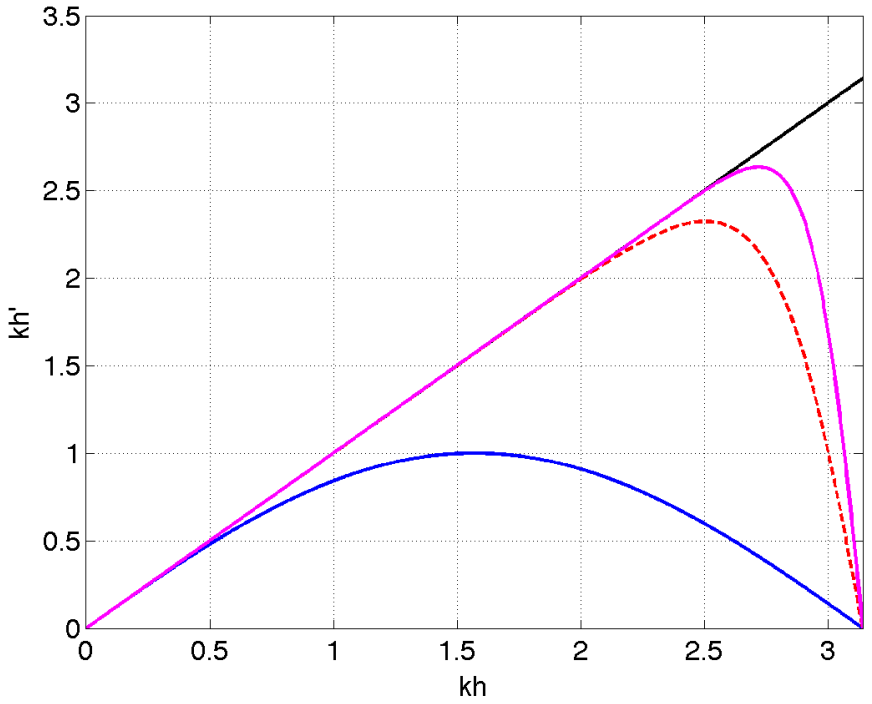
- 4th order schemes have three free constants
- Can impose constraints on modified wavenumber:
  - *e.g.:* $kh'(2.2) = 2.2$

    $kh'(2.3) = 2.3$

    $kh'(2.4) = 2.4$
- Then,

  $\alpha = 0.5771439, \beta = 0.0896406, a = 1.3025166, b = 0.99335500, c = 0.03750245$

Wavenumber analysis



- **10th order ~ 3 points/wavelength for 1% error**
- **4th order ~ 2.5 points/wavelength for 1% error**
- **Order of accuracy isn't everything!**

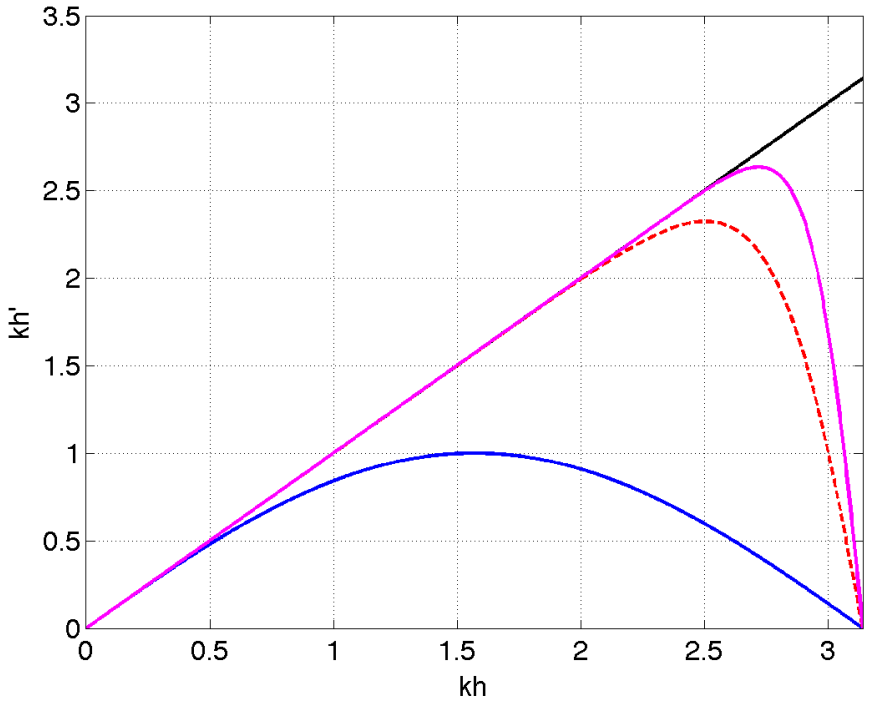
Black: exact

Blue: 2nd-order

Red: 10th-order

Magenta: Optimised, 4th order

Wavenumber analysis



Black: exact

Blue: 2nd-order

Red: 10th-order

Magenta: Optimised, 4th order

- 10th order ~ 3 points/wavelength for 1% error
- 4th order ~ 2.5 points/wavelength for 1% error
- Order of accuracy isn't everything!

- Notes on cost: operation counts→
  - 2nd order FD: N mult + N add
  - 4th order implicit FD: pentadiagonal linear system,
  - 7N mult + 7N add
  - Spectral (Fourier): ~N$\log_2$N

Key question: which method requires least time for desired accuracy?

Python implementation

- **Spectral-like scheme requires solution of** $\mathbf{Ax} = \mathbf{b}$ **and A is a pentadiagonal matrix:**

$$\begin{bmatrix} f_1 & g_1 & h_1 & & & & & \\ e_2 & f_2 & g_2 & h_2 & & & & \\ d_3 & e_3 & f_3 & g_3 & h_3 & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \cdot & \\ & & & & d_{n-1} & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & & & d_n & e_n & f_n \end{bmatrix}$$

- **In general,** np.linalg.solve(A,b)

- **But this isn't very efficient**
  - **Requires excessive memory (storing the zeros in** A**)**
  - **Requires excessive operations (again due to the zeros)**

Python implementation

- **Can build sparse banded matrix using** scipy.diags
- **First construct diagonals:**

```
#RHS
a = 1.3025166
b = 0.9935500
c = 0.03750245

#LHS
ag = 0.5771439
bg = 0.0896406

#Construct A
zv = np.ones(N)
agv = ag*zv[1:]
bgv = bg*zv[2:]
```

Python implementation

- **Then construct sparse matrix,** A:

```
A = sp.diags([bgv,agv,zv,agv,bgv],[-2,-1,0,1,2])
A.toarray()
```

```
Out[72]:
array([[1.       , 0.5771439, 0.0896406, 0.       , 0.       , 0.       ],
       [0.5771439, 1.       , 0.5771439, 0.0896406, 0.       , 0.       ],
       [0.0896406, 0.5771439, 1.       , 0.5771439, 0.0896406, 0.       ],
       [0.       , 0.0896406, 0.5771439, 1.       , 0.5771439, 0.0896406],
       [0.       , 0.       , 0.0896406, 0.5771439, 1.       , 0.5771439],
       [0.       , 0.       , 0.       , 0.0896406, 0.5771439, 1.       ]])
```

and use scipy.sparse.linalg.spsolve

- **This would solve the memory issue**
- **But it is still possible to do better with efficiency**

Python implementation

- scipy.linalg **has a solver specifically for banded matrices:** scipy.linalg.solve_banded **(actually a Fortran routine from Lapack)**

- **This is a little tricky to use**
  - **Need to provide matrix in "matrix diagonal ordered form"**

    $A_b[u + i - j, j] == A[i,j]$

  - A_b **is a 2-D array required as input to** solve_banded

  - u**, is the number of non-zero diagonals *above* the main diagonal (2 for our pentadiagonal matrix)**

- **Now we (nearly) have the "best" approach for our problem**
  - **Need to modify our method for the top two and bottom two rows in the matrix (i=0,1,N-2,N-1)**

Boundary modifications

At i=1 and i=N-2, we switch to a (8th-order) tridiagonal scheme:

$$\alpha = \tfrac{3}{8}. \qquad \beta = 0, \quad a = \tfrac{1}{6}(\alpha + 9),$$
$$b = \tfrac{1}{15}(32\alpha - 9), \quad c = \tfrac{1}{10}(-3\alpha + 1).$$

And at i=0,N-1, we switch to "one-sided" schemes:

$$f'_0 + \alpha f'_1 = \tfrac{1}{h}(af_0 + bf_1 + cf_2 + df_3)$$

$$f'_{N-1} + \alpha f'_{N-2} = -\tfrac{1}{h}(af_{N-1} + bf_{N-2} + cf_{N-3} + df_{N-4})$$

$$\alpha = 3, \quad a = -\tfrac{17}{6}, \quad b = \tfrac{3}{2},$$
$$c = \tfrac{3}{2}, \quad d = -\tfrac{1}{6}. \quad \text{(fourth order)}$$

This works for *periodic* functions where the RHS can be evaluated using these schemes. For general functions, we would have to modify the scheme at i=3,N-2 as well

Final notes on finite-difference methods

- **Advantages of quasi-spectral FD schemes**
  - **‘Competitive’ efficiency: depends on desired accuracy, aim for 1e-6 error or smaller for scientific applications**
  - **Low memory usage (relative to explicit FD and Fourier) – second key question: how much memory is available? Is there a performance gain from using less memory? (low space complexity)**

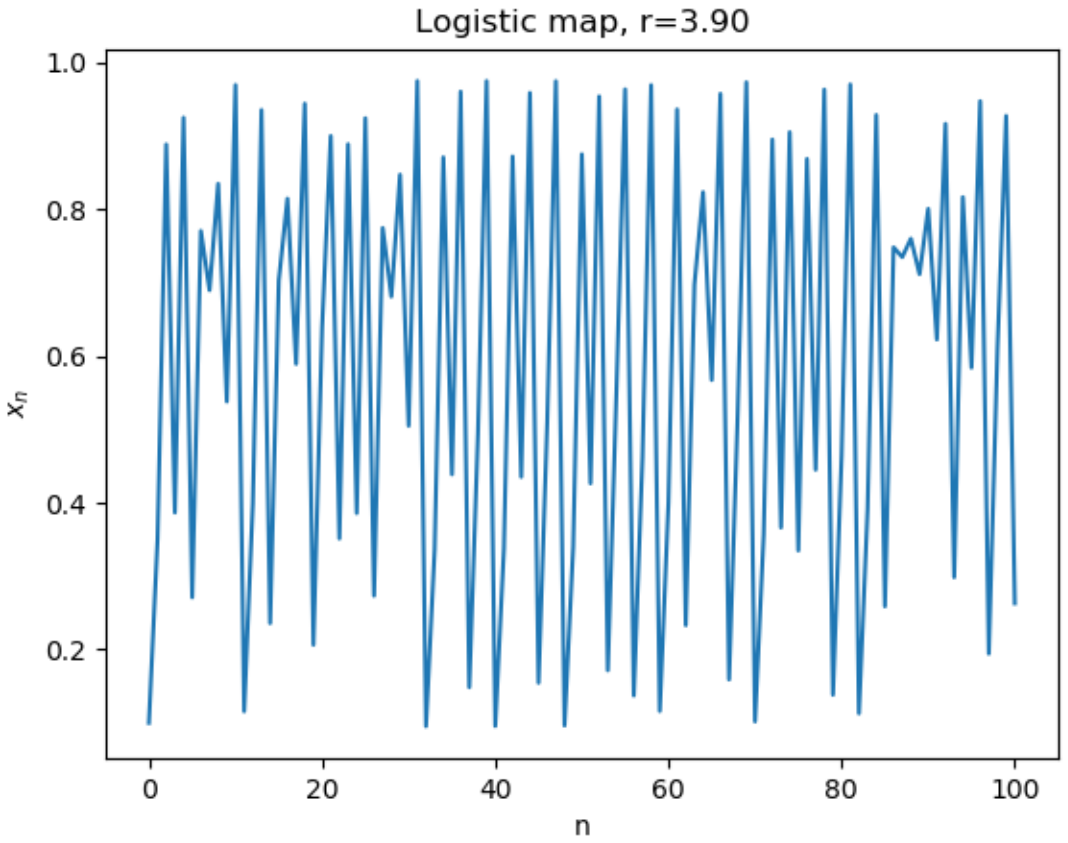
- **Disadvantages**
  - **Global (like spectral)**
  - **Difficult to apply to complex geometries**

(More) data analysis

- Fourier (energy or power) spectra are a good place to start with stationary data
  - If the length of the data is sufficiently large
  - And the sampling rate (i.e. Δx or Δt) is sufficiently small

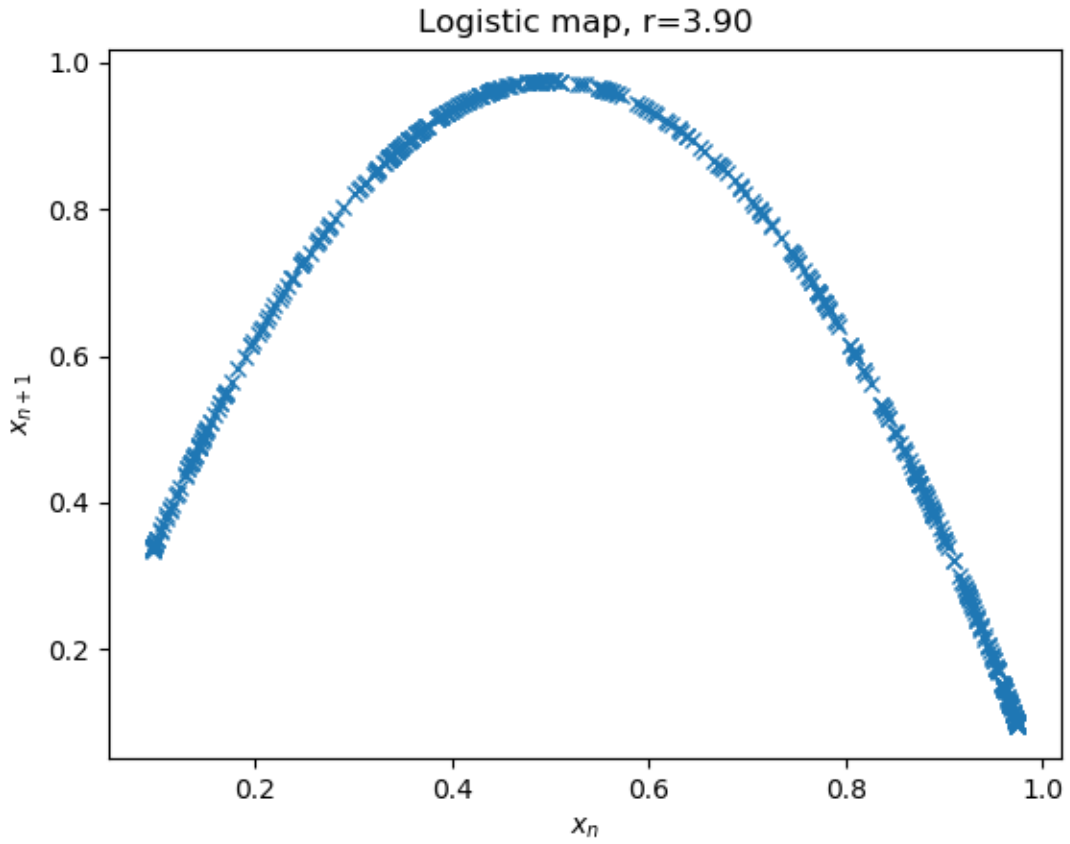
(More) data analysis

- Fourier (energy or power) spectra are a good place to start with stationary data
  - If the length of the data is sufficiently large
  - And the sampling rate sufficiently large (Δx or Δt sufficiently small)



- What can we do with this data?
- FFT won't work – data is not smooth
- Three basic options:
  - Compare peaks to peaks
  - Or troughs to troughs
  - Or peaks to troughs
- Let's try the 3rd option

Logistic map

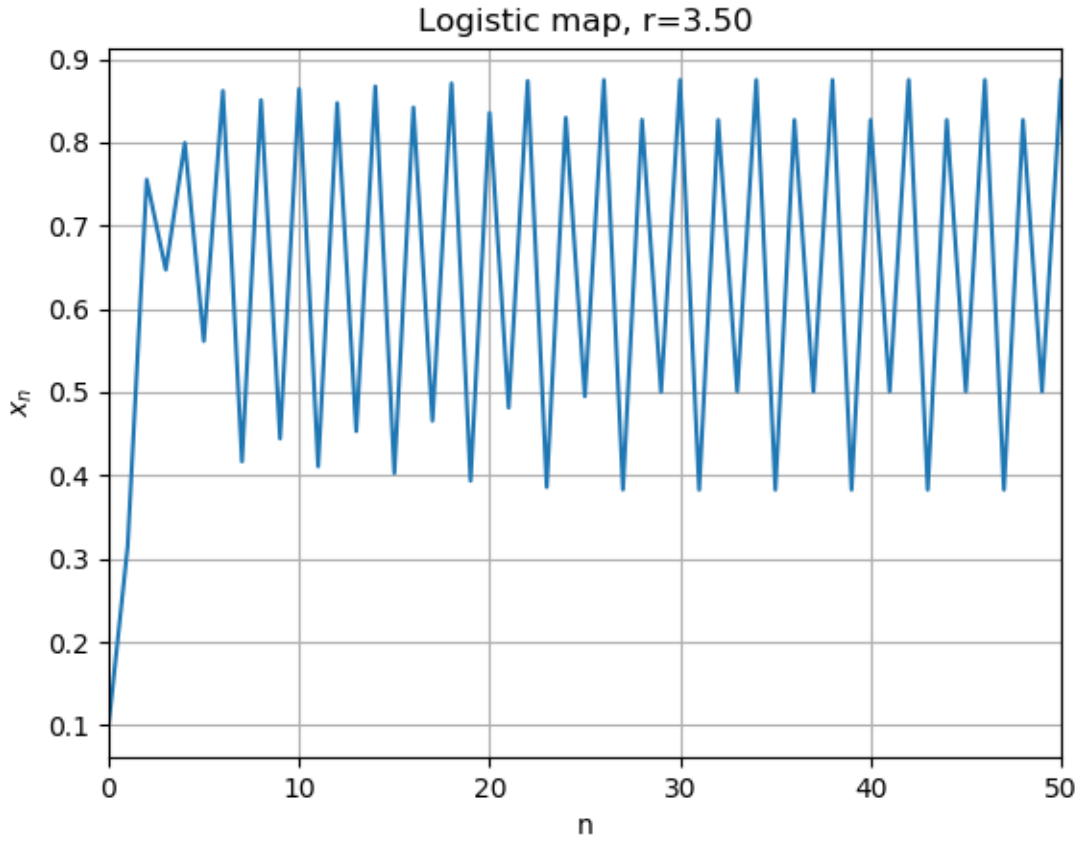


- There is clear "structure" within the data!
- The *logistic map* is:

$$x_{n+1} = r x_n (1 - x_n)$$

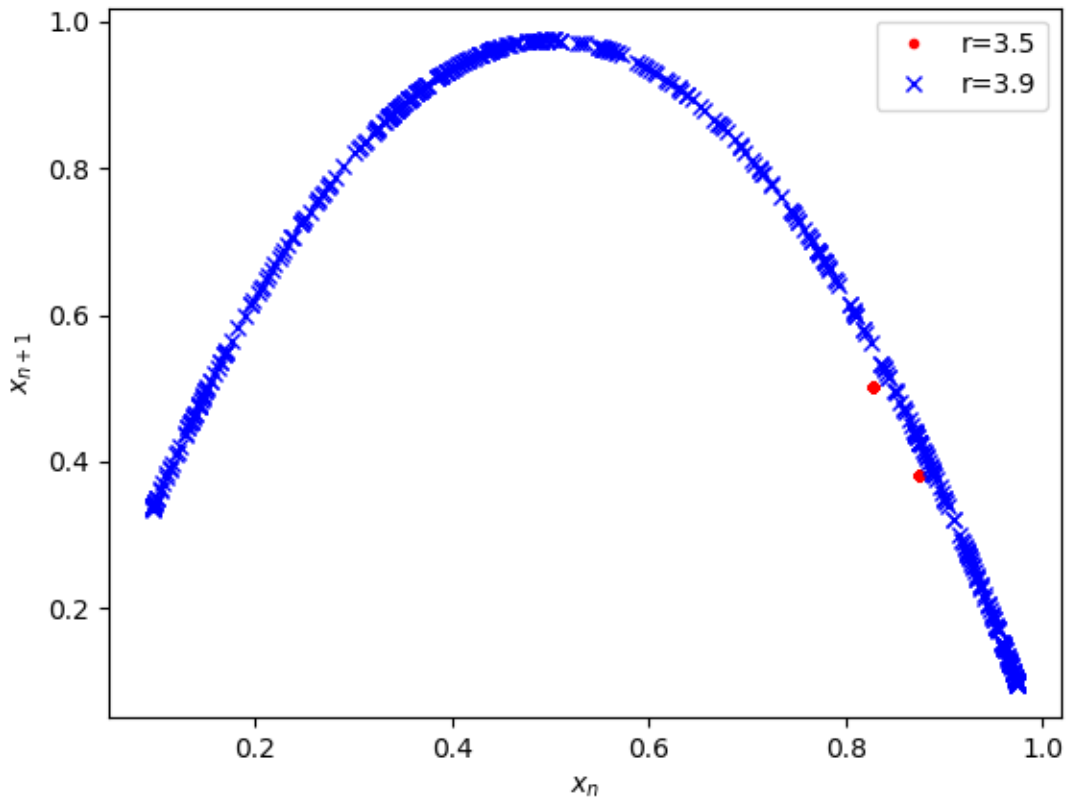
- The parameter, r, controls the dynamics.
  - For smaller r, simple periodic behavior
  - For r=3.9, chaotic behavior
  - As time increases, more and more points on the parabola will be visited in an irregular order

Logistic map



- At r=3.5, there are *period-2* oscillations
- $x_{n+4} = x_n$
- When characterizing a solution, we need to think about the set of points visited (after discarding the initial transient)
- And in particular, the *dimension* of this set

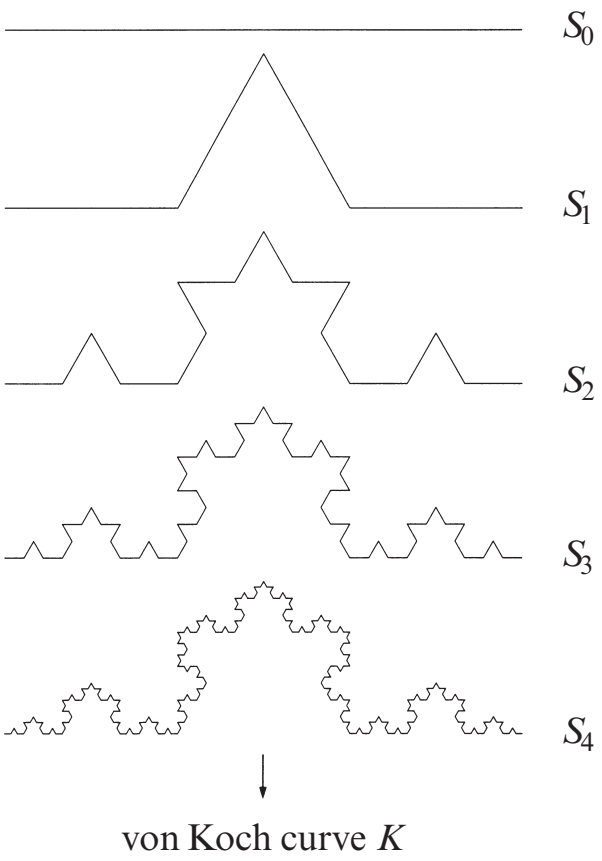
Logistic map



- At r=3.5, there are *period-2* oscillations
- $x_{n+4} = x_n$
- When characterizing a solution, we need to think about the set of points visited (after discarding the initial transient)
- And in particular, the *dimension* of this set

Fractal dimension

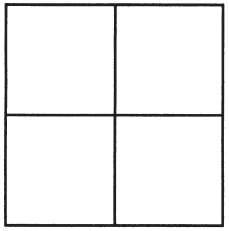
- We borrow from the study of fractals where there is a *similarity* or *fractal* dimension



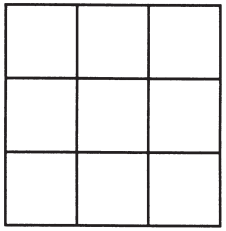
- An example: the Koch snowflake
  - Each “iteration”, the “curve” is broken up into 4 smaller copies of itself. The length of each copy is 1/3 of the original
  - What is the dimension of this curve?

Fractal dimension

- **We borrow from the study of fractals where there is a *similarity* or *fractal* dimension**



$m = 4$
$r = 2$



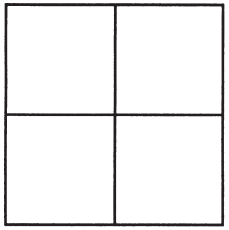
$m = 9$
$r = 3$

$m =$ number of copies
$r =$ scale factor

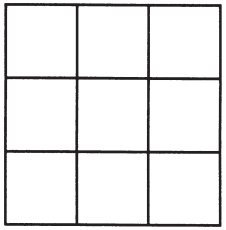
- **A simpler example: A square.**
  - **Break a square into 4 smaller copies (m=4) – the sides of each copy have been scaled down by a factor of 2 (r=2)**
  - **With m=9, we have r=3**
  - **And the dimension is** d = log(m)/log(r)=2
  - **For the Koch snowflake,** m=4, r=3, d=log(4)/log(3)=1.261…

Fractal dimension

- **We borrow from the study of fractals where there is a *similarity* or *fractal* dimension**



$m = 4$
$r = 2$

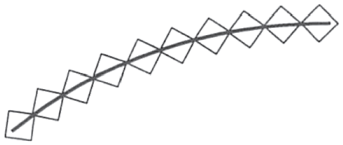


$m = 9$
$r = 3$

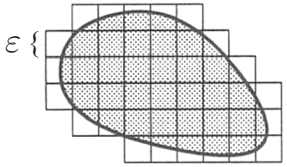
m = number of copies
r = scale factor

How do we compute the fractal dimension given data?

One approach is to compute the *box dimension*:



$$N(\varepsilon) \propto \frac{L}{\varepsilon}$$



$$N(\varepsilon) \propto \frac{A}{\varepsilon^2}$$

Construct m-dimensional cubes with edge-length, ε, find the number of cubes needed to cover the set of points in the solution, N(ε)

We expect: $N(\varepsilon) \propto 1/\varepsilon^d$. **where d is the box dimension**

Fractal dimension

- In practice, the box dimension is not used – it's computation is too expensive for large high-dimensional sets
- The *correlation dimension* is often used instead
- We collect n m-dimensional points "visited" during a process after discarding the effect of the initial condition: $\{\mathbf{x}_i, i = 1, \ldots, n\}$
- The *correlation sum*, C(ε) is: (total number of pairs of points within distance ε)/(Total number of distinct pairs)

$$C(\epsilon) = \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} H\left(\epsilon - ||\mathbf{x}_i - \mathbf{x}_j||\right)$$

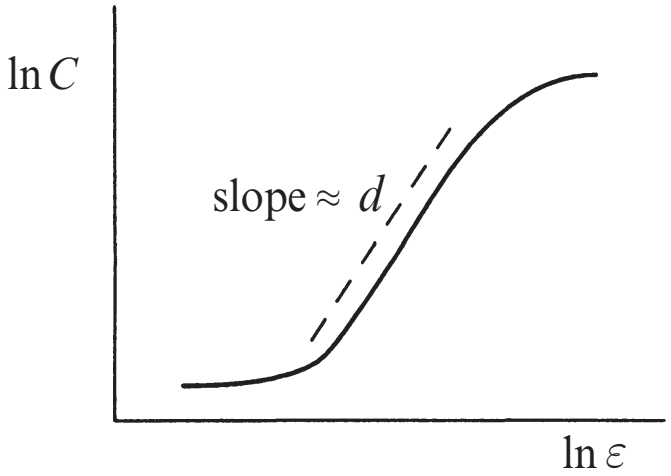
- Here, H, is the Heaviside function, H(z)=0 if z<=0, H(z)=1 if z>0
- $||\mathbf{x}_i - \mathbf{x}_j||$ is the Euclidian distance (for m-dimensional x)
- For points falling on a fractal-like structure, we expect: $C(\epsilon) \sim \epsilon^D$

Fractal dimension

- **The *correlation sum*, C(ε) is: (total number of pairs of points within distance ε)/(Total number of distinct pairs)**

$$C(\epsilon) = \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} H\left(\epsilon - ||\mathbf{x}_i - \mathbf{x}_j||\right)$$

Generically, we expect:



- At small ε, few if any pairs within ε of each other

- At large ε, almost all pairs within ε

- Often some trial-and-error is needed to choose a good range of ε

- **Straightforward to compute for logistic map using** scipy.spatial.distance.pdist

Fractal dimension

1. **Compute solution:**

```
for i in range(N):
    x[i+1] = r*x[i]*(1-x[i])
```

2. **Discard influence of initial condition:**

```
y = x[N//2:]
n = y.size
```

3. **Split into two vectors** (m=2) **and collect in** n x m **matrix**

```
y1 = y[:-1:2]
y2 = y[1::2]
A = np.vstack([y1,y2]).T
```

4. pdist **will then compute all** n(n-1)/2 **distances:**

```
D = pdist(A)
```

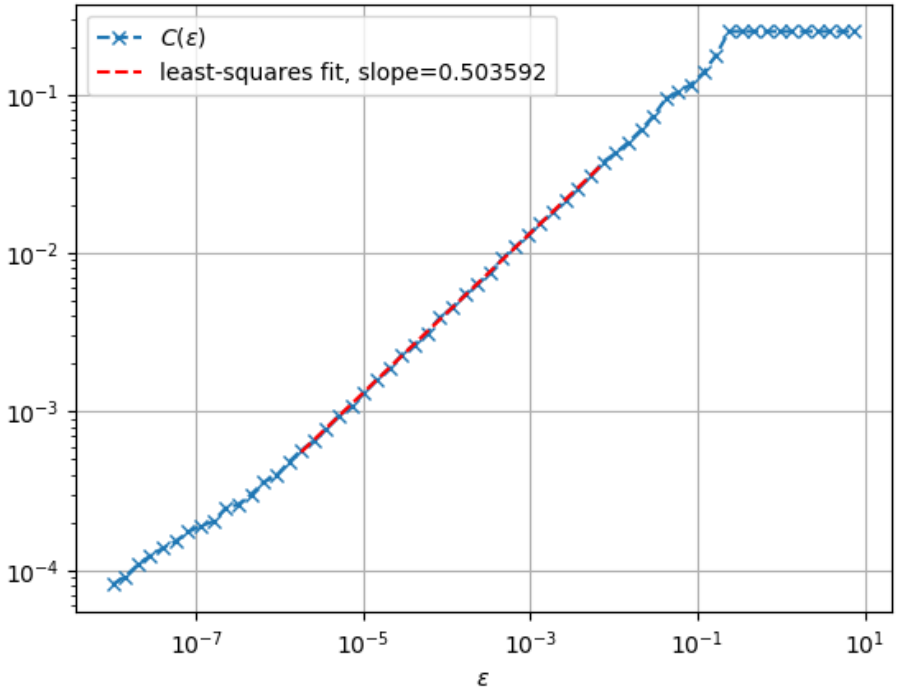
5. **Now just need to pass these distances through Heaviside function for a range of** ε…

Fractal dimension

5. Now just need to pass these distances through Heaviside function for a range of ε…

```
D = D[D<eps[i]] #Discard distances larger than eps. Assumes eps[i+1]<eps[i]

C[i] = D.size #Size of new D is Correlation sum (without m*(m-1)/2 scaling)
```



- Results for r=3.5699456 are shown, m=8000
- 1st 15 and last 20 points have been discarded for best fit calculation, fractal dimension is estimated as 0.5 (for this value of r)
- Estimate computed using np.polyfit