

# **Scientific Computation**

**Spring, 2019**

**Lecture 10**

# Feedback

---

Please fill out the short online class feedback form here:

<https://goo.gl/forms/K5YEXbxZFTbUdfda2>

# Diffusion in networks

---

- With networks, we no longer have spatial derivatives
- Need to think about flux along a link between two nodes
- Then the rate of change at a node will be the sum of fluxes from its neighbors
- The diffusive flux from node  $j$  to node  $i$  is taken to be:  $J_{ij} = -\mathcal{D}(f_i - f_j)$   
which leads to:  $\frac{\partial f_i}{\partial t} = \sum_j J_{ij}$  and the sum is over all nodes with links to node  $i$
- We can rewrite the sum using the network's adjacency matrix:

$$\frac{\partial f_i}{\partial t} = \sum_{j=1}^N A_{ij} J_{ij}$$

$$\frac{\partial f_i}{\partial t} = - \sum_{j=1}^N A_{ij} \mathcal{D}(f_i - f_j)$$

- And this is a representative model for diffusion/spreading in networks

# Diffusion in networks

---

- Can we further simplify the equation below?

$$\frac{\partial f_i}{\partial t} = - \sum_{j=1}^N A_{ij} \mathcal{D} (f_i - f_j)$$

- There are two “sum invariants” and we rewrite the equation as:

$$\frac{\partial f_i}{\partial t} = \mathcal{D} \left( \sum_{j=1}^N A_{ij} f_j - f_i \sum_{j=1}^N A_{ij} \right)$$

- For undirected graphs, the second sum is simply the degree of node i,  $q_i$
- Define a diagonal matrix,  $Q$ , where the  $(k,k)$  element is the degree,  $q_k$
- Then, our diffusion equation can be re-written as:

$$\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right), \quad L = Q - A$$

where  $L$  is the *Laplacian* matrix

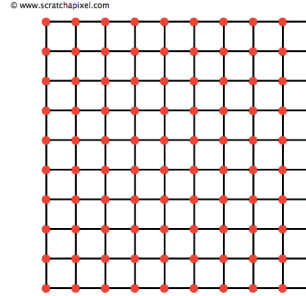
- Does the Laplacian matrix have any connection to the Laplacian operator?

# Laplacians

- Let's think about Laplacian operator on a 2D x-y grid
- Need to approximate  $\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$  on discrete grid with spacing  $\Delta$ :

$$x_i = i * \Delta, \quad i = 0, 1, 2, \dots, N + 1$$

$$y_j = j * \Delta, \quad j = 0, 1, 2, \dots, N + 1$$



- Use a 2<sup>nd</sup>-order finite-difference approximation:

$$\left( \frac{\partial^2 T}{\partial x^2} \right)_{i,j} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$\left( \frac{\partial^2 T}{\partial y^2} \right)_{i,j} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

$$\Delta x = \Delta y = \Delta$$

The approximation error is  $O(\Delta^2)$  – can use Taylor series expansions to show this

Rearranging, we have:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i+1,j} + T_{i,j+1} - 4T_{i,j} + T_{i-1,j} + T_{i,j-1}}{\Delta^2}$$

# Laplacians

---

- Our 2-D grid can be re-interpreted as a graph
- Each node has four links (degree=4), so:

$$L_{ij}T_j = 4T_{i,j} - T_{i-1,j} - T_{i+1,j} - T_{i,j+1} - T_{i,j-1}$$

- This is the negative of our approximate Laplacian operator with  $\Delta=1$
- So we can anticipate “diffusion-like” behavior with our model for spreading on graphs
- The Laplacian matrix also provides insight into the structure of graphs (e.g. connectivity)

# Diffusion in networks

---

$$\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right), \quad L = Q - A$$

- How do we find solutions?
- Need initial values to be specified for each node
- Then, this is a system of linear constant-coefficient ODEs
  - i.e. it's an eigenvalue problem
- For undirected networks, the Laplacian is real-valued and symmetric
  - So its eigenvalues are real (they are also non-negative)

# Diffusion in networks

---

$$\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right), \quad L = Q - A$$

- How do we find solutions?
- Need initial values to be specified for each node
- Then, this is a system of linear constant-coefficient ODEs
  - i.e. it's an eigenvalue problem
- For undirected networks, the Laplacian is real-valued and symmetric
  - So its eigenvalues are real (they are also non-negative)
- How do we compute eigenvalues in Python?
  - `np.linalg.eig`
  - `scipy.sparse.linalg.eig`
- What are the advantages/disadvantages of the two functions?



# Diffusion in networks

$$\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right), \quad L = Q - A$$

- **An example:**

```
In [3]: G = nx.erdos_renyi_graph(15,0.2)
```

```
In [6]: A = nx.adjacency_matrix(G)
```

```
In [7]: A.todense()
```

```
Out[7]:
```

```
matrix([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1],
        [0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]])
```

# Diffusion in networks

---

$$\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right), \quad L = Q - A$$

- **An example:**

```
In [22]: Q = A.toarray().sum(axis=1)
```

```
In [23]: L = np.diag(Q)-A.toarray()
```

```
In [27]: e,v = np.linalg.eig(L)
```

```
In [28]: e
```

```
Out[28]:
```

```
array([ 8.46780548e+00,  6.22084072e+00,  5.59130087e+00,  4.25530240e+00,  
        3.73535212e+00,  1.66002435e-15,  2.55436419e+00,  2.37762994e+00,  
        1.87790207e+00,  1.69700500e+00,  1.46423645e+00,  8.66087659e-01,  
        8.92173104e-01,  0.00000000e+00,  0.00000000e+00])
```

- **Unaddressed questions:**

- **There are three eigenvalues with value zero – are these important? What is their physical interpretation?**
- **What should be done with the eigenvectors?**

# Dynamical process on networks

---

- Previously, we considered *linear* diffusion:  $\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right)$ ,  $L = Q - A$
- Often, the most interesting/important problems involve *nonlinear* dynamics:

$$\frac{\partial f_i}{\partial t} = \mathcal{F}(t, f_j, A_{ij}), \quad i, j \in 0, 1, \dots, N - 1$$

- How can we solve problems of this form?
  - Generally, no analytical solution, need to find approximated numerical solution

# Dynamical process on networks

---

- Previously, we considered *linear* diffusion:  $\frac{\partial f_i}{\partial t} = -\mathcal{D} \left( \sum_{j=1}^N L_{ij} f_j \right)$ ,  $L = Q - A$
- Often, the most interesting/important problems involve *nonlinear* dynamics:

$$\frac{\partial f_i}{\partial t} = \mathcal{F}(t, f_j, A_{ij}), \quad i, j \in 0, 1, \dots, N - 1$$

- How can we solve problems of this form?
  - Generally, no analytical solution, need to find approximated numerical solution
- Basic idea: discretize time,  $t = 0, \Delta t, \dots, N \Delta t$ , and starting from  $f(0)$  march forward in time and compute  $f(\Delta t), \dots, f(N \Delta t)$

- Explicit Euler method:  $f_i(t_0 + \Delta t) = f_i(t_0) + \Delta t \frac{df_i}{dt} \Big|_{t_0} + O(\Delta t^2)$

$$f_i(t_0 + \Delta t) \approx f_i(t_0) + \Delta t \mathcal{F}(t_0, f_j(t_0), A_{ij})$$

- Accuracy: global error  $\sim \Delta t \rightarrow$  smaller time step, more accurate solution
- Stability: for s: unconditionally unstable  $\rightarrow$  for large enough N, solution becomes unbounded (blows up)

# Time-marching methods

---

- **Basic idea:** discretize time,  $t = 0, \Delta t, \dots, N\Delta t$ , and starting from  $f(0)$  march forward in time and compute  $f(\Delta t), \dots f(N\Delta t)$
- **Explicit Euler method:** 
$$f_i(t_0 + \Delta t) = f_i(t_0) + \Delta t \left. \frac{df_i}{dt} \right|_{t_0} + O(\Delta t^2)$$
$$f_i(t_0 + \Delta t) \approx f_i(t_0) + \Delta t \mathcal{F}(t_0, f_j(t_0), A_{ij})$$
- **Accuracy:** global error  $\sim \Delta t \rightarrow$  smaller time step, more accurate solution
- **Stability:** for  $dy/dt = y$ : unconditionally unstable  $\rightarrow$  for large enough  $N$ , solution becomes unbounded (blows up)
- **Runge-Kutta methods provide better accuracy and stability**
  - Evaluate function at sub-steps and construct approximation using this “extra information”

# Time-marching methods

---

Runge-Kutta methods provide better accuracy and stability

- Evaluate function at sub-steps and construct approximation using this “extra information”

- **4<sup>th</sup>-order RK** for  $dy/dt = f(t,y)$ :  $y(t_0 + \Delta t) = y(t_0) + \frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4$

$$k_1 = \Delta t f(t_0, y(t_0))$$

$$k_2 = \Delta t f(t_0 + \Delta t/2, y(t_0) + k_1/2)$$

$$k_3 = \Delta t f(t_0 + \Delta t/2, y(t_0) + k_2/2)$$

$$k_4 = \Delta t f(t_0 + \Delta t, y(t_0) + k_3)$$

- **Error**  $\sim \Delta t^4$
- **Conditionally** stable for  $dy/dt=y$  (bounded solutions for sufficiently small  $\Delta t$ )
- Requires 4 function evaluations per time step
- Was once *very* widely used

# Time-marching methods

---

**Variable time-step Runge-Kutta methods are very popular**

- **Similar to RK4, however time-step is automatically adjusted to ensure a specified accuracy**
- **In Matlab: ODE23 and ODE45 (also available in `scipy.integrate`)**

# Time-marching methods

---

Variable time-step Runge-Kutta methods are very popular

- Similar to RK4, however time-step is automatically adjusted to ensure a specified accuracy
- In Matlab: ODE23 and ODE45 (also available in scipy.integrate)
- Explicit time-marching methods (like R-K methods) can struggle with systems of nonlinear ODEs
- These systems often contain rapidly varying components which affect stability much more than accuracy
- Then, should look to *implicit* methods:
- **Implicit Euler method:**  $f_i(t_0 + \Delta t) \approx f_i(t_0) + \Delta t \mathcal{F}(t_0 + \Delta t, f_j(t_0 + \Delta t), A_{ij})$
- **Accuracy:** global error  $\sim \Delta t \rightarrow$  smaller time step, more accurate solution
- **Stability:** for  $dy/dt = ay$ : *unconditionally stable*  $\rightarrow$  bounded solution for any time-step
- Requires solution of system of equations each time step



# Time-marching methods

---

Variable time-step implicit methods are also widely available

- Again, time-step is automatically adjusted to ensure a specified accuracy
- In Matlab: ODE23s and ODE15s (also available in `scipy.integrate`)

# Time-marching methods

---

Variable time-step implicit methods are also widely available

- Again, time-step is automatically adjusted to ensure a specified accuracy
- In Matlab: ODE23s and ODE15s (also available in `scipy.integrate`)
- We will use `scipy.integrate.odeint` (in this week's lab)
  - Have to specify: 1) initial conditions  
2) time span  
3) function which evaluates RHS of ODEs
  - Can specify: 1) error tolerance  
2) times at which to return solution  
3) many other parameters

Look at online documentation!