

ChE352
Numerical Techniques for Chemical Engineers
Professor Stevenson

Lecture 16

Midterm rules

- You will have 90 minutes.
- You may use only these resources: the two course textbooks (F&B and PNM), my slides, your own notes, your group's HWs, and Colab
- You may use laptops and/or tablets, but not phones.
- The exam will be graded based on your blue book. Show all your work clearly in your blue book and draw a box around each answer.
- If you have a question, raise your hand.

Practice midterm

1. (15 points) You need to sum a vector of N positive floating-point numbers of many sizes.
- (5 points) What is the big-O time of this operation in terms of N ? **$O(N)$**
 - (5 points) What numerical problems might arise that would reduce the accuracy of the sum? **Swamping ($A+B \approx A$ for large A). Overflow also possible, but unlikely from just addition (biggest 32-bit float is $> 10^{38}$). No cancelation (all positive).**
 - (5 points) How could you improve the accuracy of the sum? (Assume that you cannot increase the precision of the floating-point numbers.) **Sum the numbers in pairs, then sum those pairs recursively, so that in each sum the numbers are likely to be similar in scale. Or, sort the numbers and sum them from smallest to largest.**

Common mistake: too many answers, some incorrect (you will get points off for wrong answers even if you also give the right answer). If you want me to be sure, box the right answer.

Practice midterm

Common mistake:
sig figs

2. (20 points) The square root of 2 is approximately 1.41421356237.
- A. (5 points) If you approximate $\text{sqrt}(2)$ as 1.4, what is the relative percent error in this approximation? $(1.4 - 1.414) / 1.414 = 0.014/1.414 = \underline{1.0\%}$
- B. (10 points) Estimate $\text{sqrt}(2)$ by bisection, starting with $x_{\min} = 1$ and $x_{\max} = 2$, until you can prove that your estimate is within 0.2 of the answer *without* knowing the answer in advance. Show each step.

Solve: $x^2 - 2 = 0$

Starting bounds: $1^2 - 2 = -1$, $2^2 - 2 = 2$ \rightarrow Yes, different signs.

$(1+2)/2 = 1.5 = x_1$, $f(x_1) = 1.5^2 - 2 = 0.25$, replace 2 with 1.5, bounds = 1, 1.5

(Could prove 1.5 is an answer, but not exactly bisection)

$(1+1.5)/2 = 1.25 = x_2$, $f(x_2) = 1.25^2 - 2 = -0.4375$, replace 1 with 1.25, bounds = 1.25, 1.5

(Could also prove this is an answer)

$(1.25+1.5)/2 = \underline{1.375}$, this is < 0.2 from both bounds, must be < 0.2 from the answer

Common mistake: checking $f(x)$ vs 0 instead of x vs bounds

Practice midterm

3. (15 points) You are modeling a distillation unit that takes in a mixture of components A & B. Assume that you have two functions that define the change in concentrations in liquid phase over time:

$$\frac{\partial A}{\partial t} = f_A(A, B, t) \quad \frac{\partial B}{\partial t} = f_B(A, B, t)$$

- A. (10 points) Write out the first step of Euler's method for this system in terms of the required initial conditions, a step size h , and the functions f_A , and f_B .

Initial conditions: t_0, A_0, B_0

$t_1 = t_0 + h$

$A_1 = A_0 + h * f_A(A_0, B_0, t_0)$

$B_1 = B_0 + h * f_B(A_0, B_0, t_0)$

Common mistake: saying $f(A, B, t)$
when meaning $f(A_0, B_0, t_0)$.

- B. (5 points) How would you estimate whether this IVP system was stiff? Would Euler's method or RK4 be better in that case? If you could pick any method, which would you use?

Try different step sizes h and see if the IVP was very sensitive to step size. Also try different solvers, such as RK4 vs BDF. Euler's method is better than RK4 for a very stiff IVP because RK4 is higher-order and higher-order derivatives are more sensitive to stiffness. An implicit solver such as BDF or Radau would be best for the stiff system (RK45 less so).

Practice midterm

4. (20 points) Your colleague has written a function `fast_sparse_eig(A)` which is supposed to give the eigenvectors x for a sparse symmetric matrix A . To make it more robust, you need to add some consistency checks. Write a few lines of Python pseudocode to check each of the following:

A. (10 points) Given a matrix A , is A sparse and symmetric?

def is_sparse_symmetric(A):

is_symmetric = np.allclose(A, A.T) # allclose, not ==

fraction_nonzero = np.sum(A != 0) / A.size

return is_symmetric and fraction_nonzero < 0.1 # could be up to 0.5

Practice midterm

4. (20 points) Your colleague has written a function `fast_sparse_eig(A)` which is supposed to give the eigenvectors x for a sparse symmetric matrix A . To make it more robust, you need to add some consistency checks. Write a few lines of Python pseudocode to check each of the following:
- A. (10 points) Given a matrix A , is A sparse and symmetric?
 - B. (10 points) Given a matrix A and a list of vectors v , is every vector really an eigenvector of A ? (Note that the function `fast_sparse_eig` does not give you the corresponding eigenvalue, and it would be too slow to run a full eigenvalue-finding function like `np.linalg.eigh` here.)

```
def test_all_eigenvectors(A, vectors):
```

```
    all_eigenvectors = True
```

```
    for vec in vectors:
```

```
        Av = A @ vec
```

```
        should_be_eigenvalue = Av / vec
```

```
        if not np.allclose(should_be_eigenvalue, np.mean(should_be_eigenvalue)):
```

```
            all_eigenvectors = False
```

```
            break
```

```
    return all_eigenvectors
```

Practice midterm

5. (10 points) Write a few lines of Python which calculate the 1-norm of a vector x , without using `np.linalg.norm`. Small syntax errors are fine as long as the code is logically correct.

```
def one_norm(x):  
    norm = 0.0  
    for a in x:  
        norm += abs(a) # sum of absolute values of x  
    return norm
```

Common mistakes: forgetting `abs()`

Practice midterm

6. (25 points) You are designing a battery for use across a wide range of temperatures. You have access to the heat capacity data below for a candidate battery material:

Temperature (K)	260	290	340
C_p (J/mol/K)	42	48	65

Common mistake: "cubic spline" on 3 points

A. (10 points) Describe two ways to find C_p at 310 K. Which would you choose and why?

1. **Linear interpolation between the two nearest known points (at 290 K and 340 K)**
2. **Lagrange polynomial of order 2, aka fitting all three points to a quadratic and using it**

The Lagrange polynomial will be slightly more accurate because it takes into account any nonlinearity in the data, but more risky because it can exceed the data bounds.

B. (15 points) Estimate how much heat, in kJ/mol, is required to increase the temperature of this material from 250 K to 340 K. Don't use a method that will take forever, but accuracy counts.

Practice midterm

6. (25 points) You are designing a battery for use across a wide range of temperatures. You have access to the heat capacity data below for a candidate battery material:

Temperature (K)	260	290	340
C_p (J/mol/K)	42	48	65

Common mistake: "cubic spline" on 3 points

- A. (10 points) Describe two ways to find C_p at 310 K. Which would you choose and why?
- B. (15 points) Estimate how much heat, in kJ/mol, is required to increase the temperature of this material from 250 K to 340 K. Don't use a method that will take forever, but accuracy counts.
- **250 K is slightly below the available range, so I will have to extrapolate slightly. I will use linear extrapolation since simpler is better when extrapolating. Since I'm already modeling the function as a piecewise line for the extrapolation, I might as well continue to treat it that way by using trapezoidal integration.**
 - **Slope of line from 260 to 290 = $(48 - 42) / (290 - 260) = 6 / 30 = 0.2$ (J/mol/K) / K**
 - **Extrapolated C_p at 250 K = $42 + (250 - 260) * 0.2 = 40$ J/mol/K**
 - **Trapezoidal integration:**
 - **$(40 + 48) / 2 * (290 - 250) = 1760$ J/mol**
 - **$(48 + 65) / 2 * (340 - 290) = 2825$ J/mol**
 - **$1760 + 2825 = 4585$ J/mol = 4.6 kJ/mol**

Common mistakes: sig figs, using the wrong bounds (260 K vs 250 K)

Other common mistakes from HWs

- In general, you can't solve a higher-order IVP by integrating one part at a time (y'' , y' , y , etc)
- IVPs are subject to accumulating error, a problem which quadrature does not have
- The highest temperature is *not* usually when the temperature is rising the fastest
- Anything in an $\exp()$ has to be unitless
- The most **precise** method when everything is going well is sometimes **unreliable** otherwise

Know your references

Having textbooks, slides, and notes can give you a false sense of security, especially under time constraints!

Useless if you don't know what to look for

Review all your reference materials

Example exercises from F&B

Answers to odd-numbered exercises at back

Selected examples to consider (not exhaustive):

1.4 - 3

2.4 - 3c

3.2 - 13

4.2 - 1c

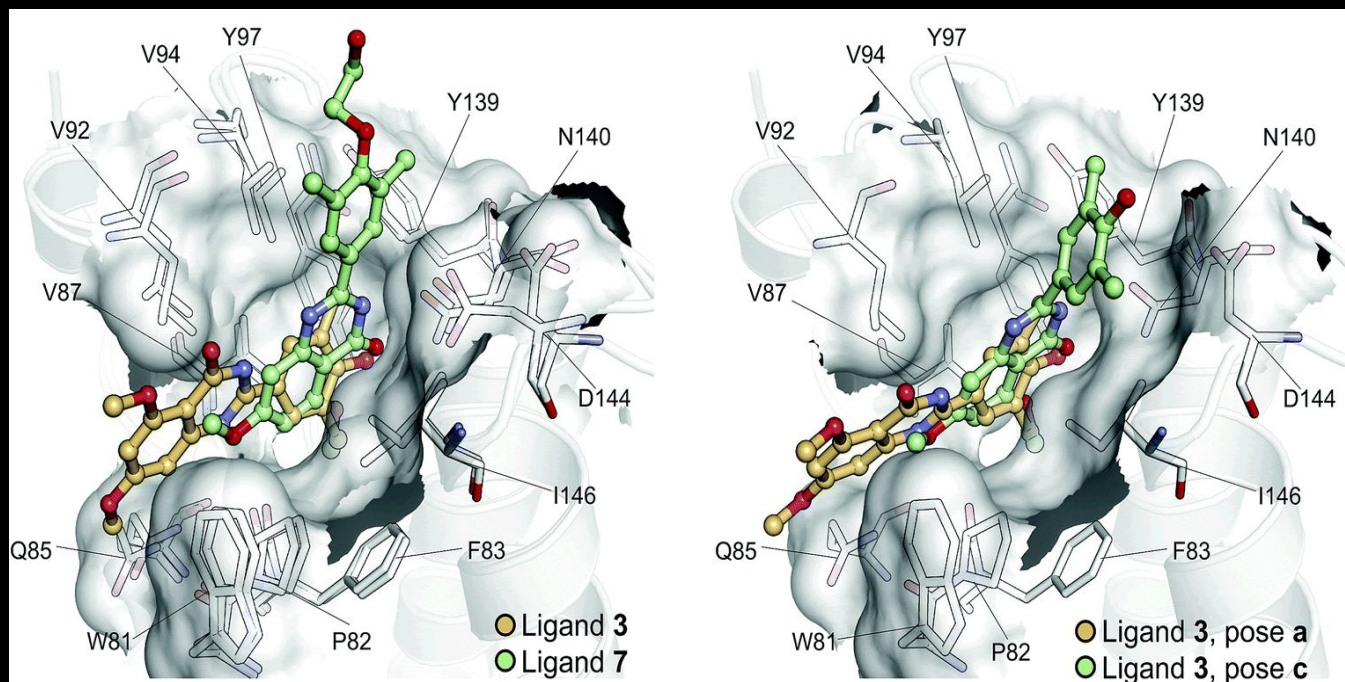
4.9 - 3a

5.2 - 11 (note, long)

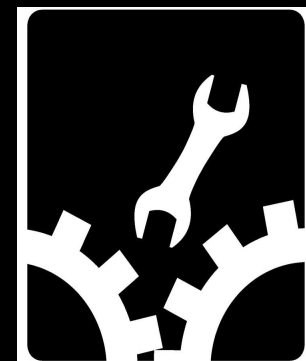
6.2 - 5

7.3 - 1b

The docking problem

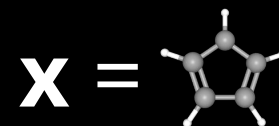


Most drugs work by binding to proteins
Can be predicted from molecule shape



Follow the forces \Rightarrow find \mathbf{x}_{\min}

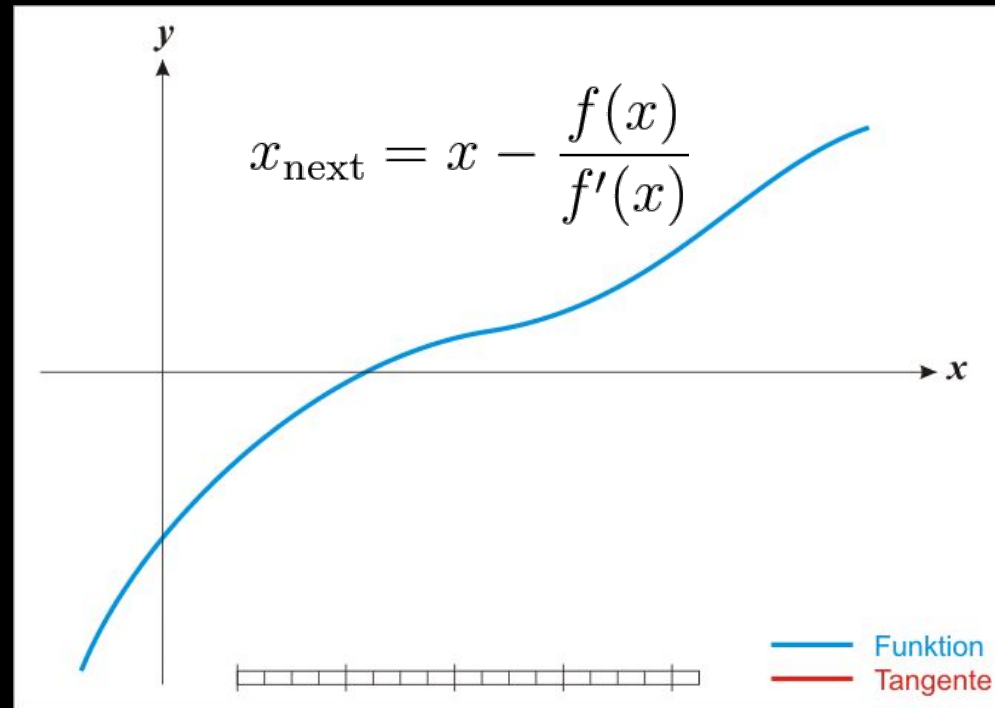
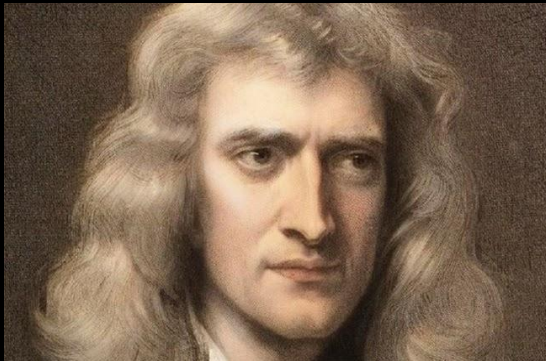
- Coords vector \mathbf{x} represents all atoms
- Potential energy $E(\mathbf{x})$ is a scalar
- $\nabla E(\mathbf{x})$ has what shape?
- Each component of $-\nabla E(\mathbf{x})$ is a force
- Follow the forces: gradient descent
- Faster ideas?



	x	y	z
C	2.80	0.45	0.01
C	2.00	-0.12	0.00
C	3.61	-0.12	0.03
C	2.30	-1.07	0.01
C	3.30	-1.07	0.00
H	2.80	1.07	0.02
H	1.41	0.06	0.00
H	4.20	0.06	0.01
H	1.94	-1.58	0.00
H	3.67	-1.58	0.01

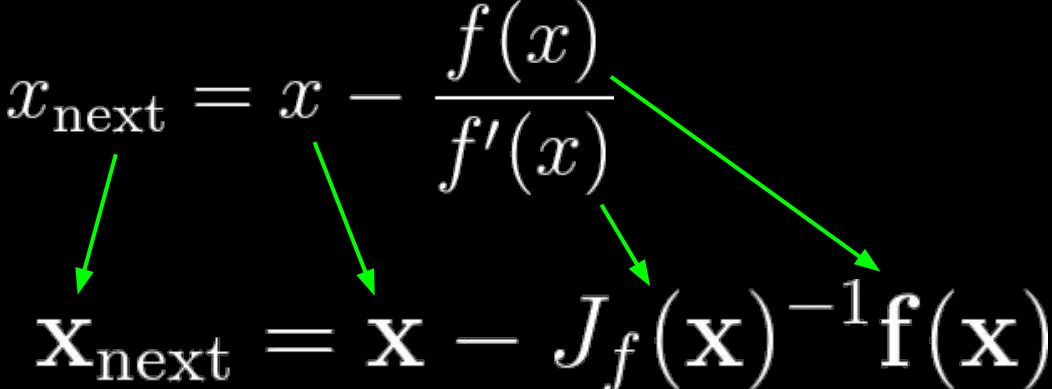
Use Newton's method?

- $E(\mathbf{x})$ is minimized where $\nabla E(\mathbf{x}) = 0$
- Can solve by root finding
- Find $\nabla E(\mathbf{x}) = 0$ using Newton's method?
 - Substitute $f(x) = \nabla E(\mathbf{x})$
 - Fast convergence
- But, $\nabla E(\mathbf{x})$ is not a scalar!



Newton's method in N dimensions

- What is $f'(x)$ for a function with multiple inputs & outputs?
- Multiple in, one out: gradient $\nabla f(\mathbf{x})$
- Multiple in, multiple out: Jacobian $J_f(\mathbf{x})$
- $J_f(\mathbf{x})$ = derivative of each output w.r.t. each input (a matrix)
- $f(x) / f'(x) \Rightarrow$ inverse Jacobian matrix * $f(x)$

$$x_{\text{next}} = x - \frac{f(x)}{f'(x)}$$


$$\mathbf{x}_{\text{next}} = \mathbf{x} - J_f(\mathbf{x})^{-1} \mathbf{f}(\mathbf{x})$$

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Newton's method for roots of a gradient: the Newton optimizer



$$\mathbf{x}_{\text{next}} = \mathbf{x} - J_f(\mathbf{x})^{-1} \mathbf{f}(\mathbf{x})$$

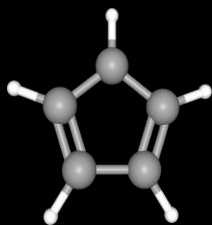
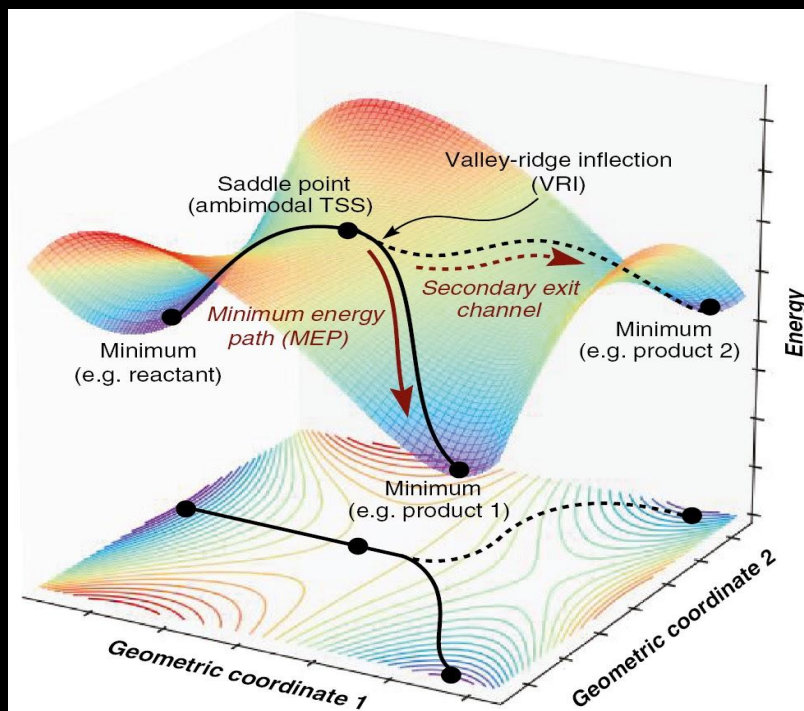
Substitute $\nabla E(\mathbf{x})$ for
 $f(x)$:

Jacobian of a gradient
is a **Hessian**

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \quad \nabla^2 E = \begin{bmatrix} \frac{\partial^2 E}{\partial x_1^2} & \cdots & \frac{\partial^2 E}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 E}{\partial x_n^2} \end{bmatrix}$$

Newton optimizer

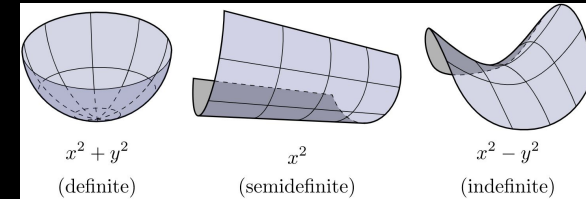
$$\mathbf{x}_{\text{next}} = \mathbf{x} - \nabla^2 E(\mathbf{x})^{-1} \nabla E(\mathbf{x})$$



- Pro: great convergence
 - Given local curvature, Newton step is the best step by definition
- Con: need Hessian $\nabla^2 E(\mathbf{x})$
- More common to have $\nabla E(\mathbf{x})$ than $\nabla^2 E(\mathbf{x})$
- Can always get Hessian by finite difference of $\nabla E(\mathbf{x})$, but takes time

Fake Hessians: *Quasi-Newton*

$$\mathbf{x}_{\text{next}} = \mathbf{x} - \nabla^2 E(\mathbf{x})^{-1} \nabla E(\mathbf{x})$$



- With identity matrix as the "Hessian" this is gradient descent $\mathbf{x}_{\text{next}} = \mathbf{x} - \nabla E(\mathbf{x})$
- Any approx Hessian matrix better than identity will help
- Approx Hessian \mathbf{B}_k from $\nabla E(\mathbf{x})$

$$B_{k+1} = B_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{B_k \mathbf{s}_k \mathbf{s}_k^T B_k^T}{\mathbf{s}_k^T B_k \mathbf{s}_k}$$

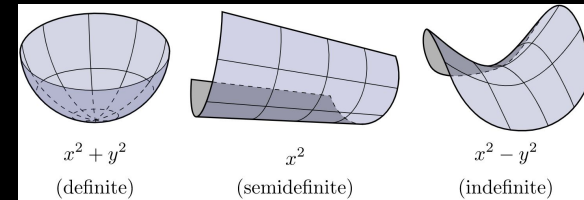
- Like a finite-difference Hessian, but only using $\nabla E(\mathbf{x})$ step history
- Must keep fake Hessian positive definite (upward curvature) - **why?**

$$\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$$

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$$

L-BFGS quasi-Newton optimizer

$$\mathbf{x}_{\text{next}} = \mathbf{x} - \nabla^2 E(\mathbf{x})^{-1} \nabla E(\mathbf{x})$$



- Store last k steps (say 10) of gradient $\nabla E(\mathbf{x})$ step history
- Apply inverse Hessian as an operator: don't even need to represent whole $N \times N$ matrix
- Reduce cost from N^2 to kN

$$\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$$

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$$

$$\rho_k = \frac{1}{\mathbf{y}_k^\top \mathbf{s}_k}$$

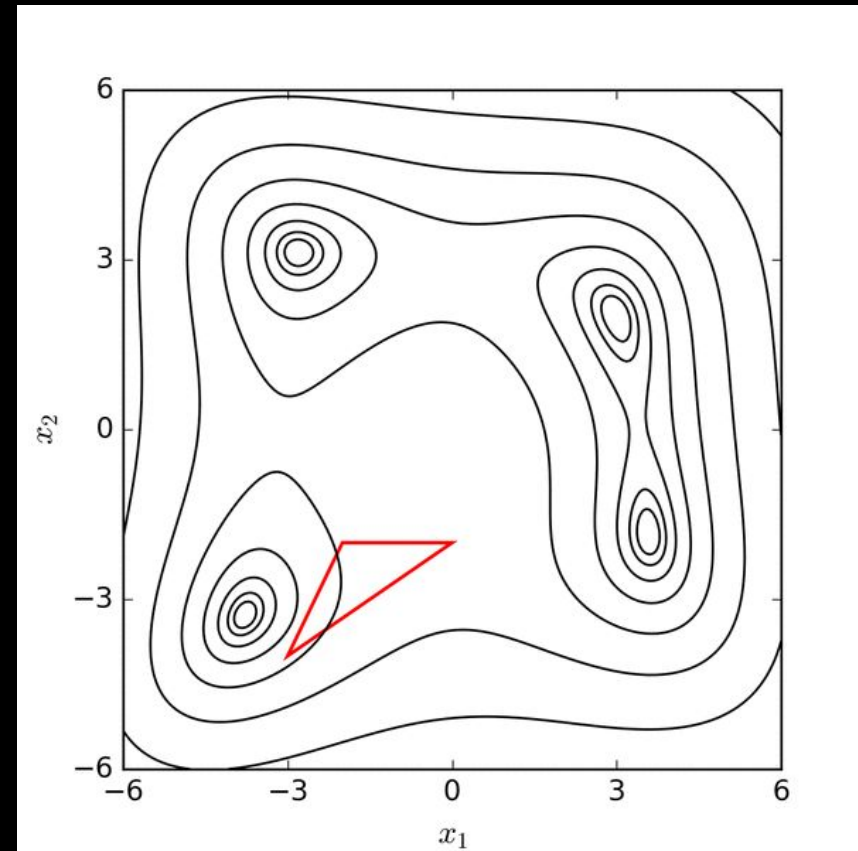
$$H_{k+1} = (I - \rho_k \mathbf{s}_k \mathbf{y}_k^\top) H_k (I - \rho_k \mathbf{y}_k \mathbf{s}_k^\top) + \rho_k \mathbf{s}_k \mathbf{s}_k^\top$$

Gradient-free optimization

- If at all possible, get the gradient and use a gradient-based method
 - Remember autodiff, PyTorch, etc
- But sometimes there is no practical gradient
 - Examples: numerical IVP solver, physical experiment
- Must *represent* the function somehow:
 - Use a set of known points
 - Optional: fit a model, optimize that instead

Nelder-Mead simplex

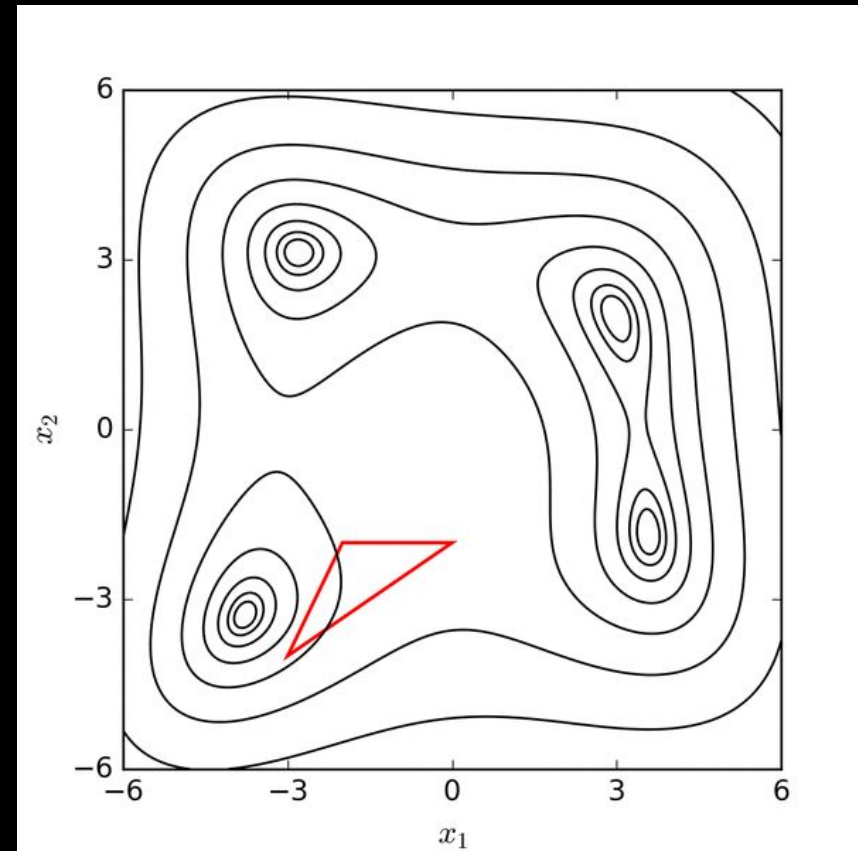
- Simplex = generalized triangle, a shape of $N+1$ vertices in a space of dimension N
- Reflect worst point across center of simplex to get next point
- Shrink steps as confidence improves



Nelder-Mead in Python

```
'''  
Python script to minimize f()  
'''  
  
import numpy as np  
from scipy.optimize import minimize  
x0 = np.array([your guess here])  
res = minimize(f,  
               x0,  
               method='nelder-mead',  
               options={'xatol': 1e-6,  
                        'disp': True})  
'''
```

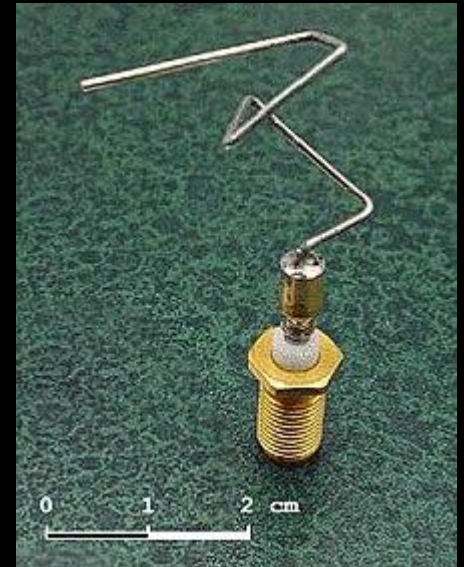
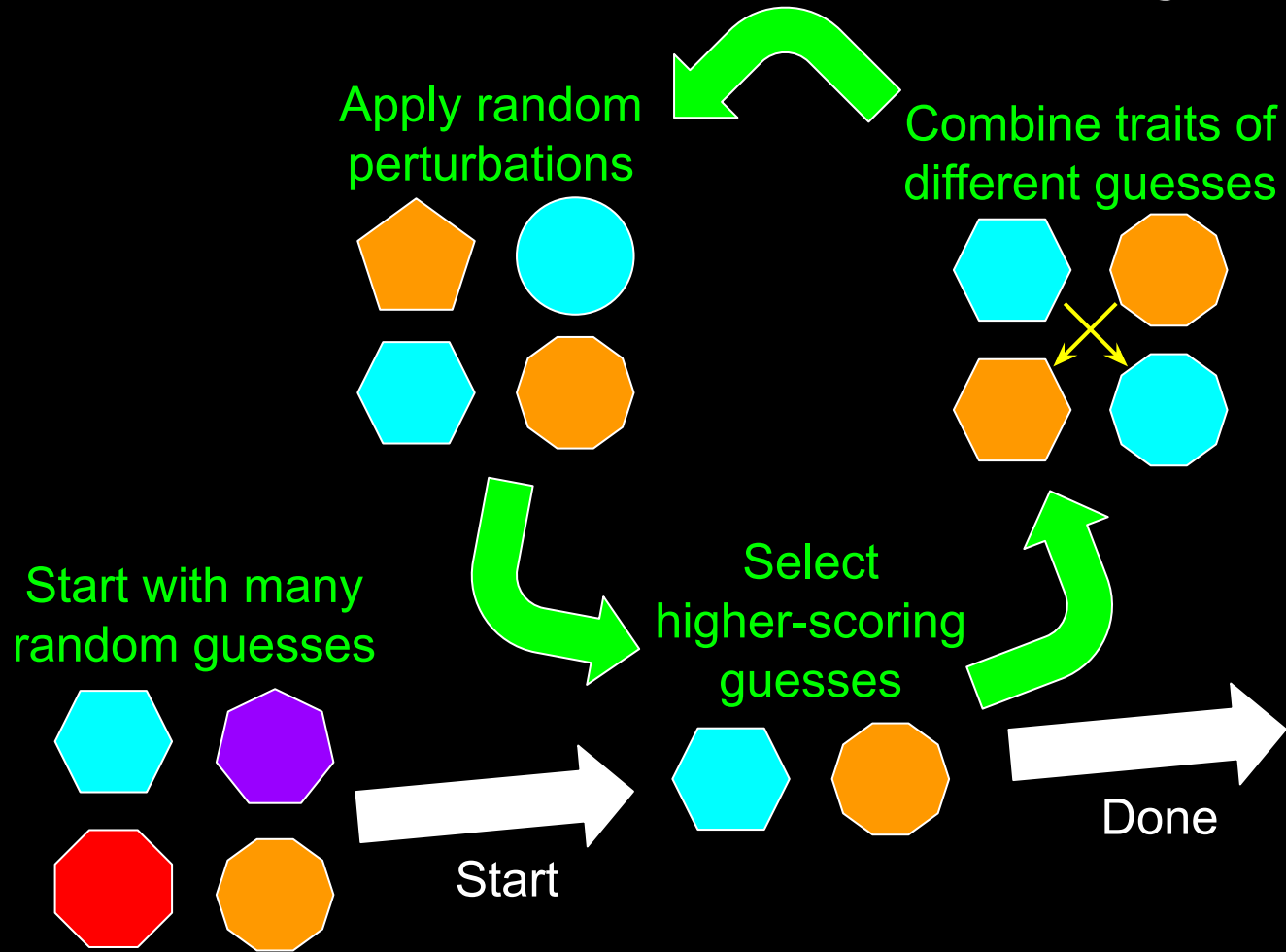
Output:
Optimization terminated successfully.
Current function value: 0.0
Iterations: 339
Function evaluations: 571
'''



Genetic algorithms

A heuristic based on *biological evolution*

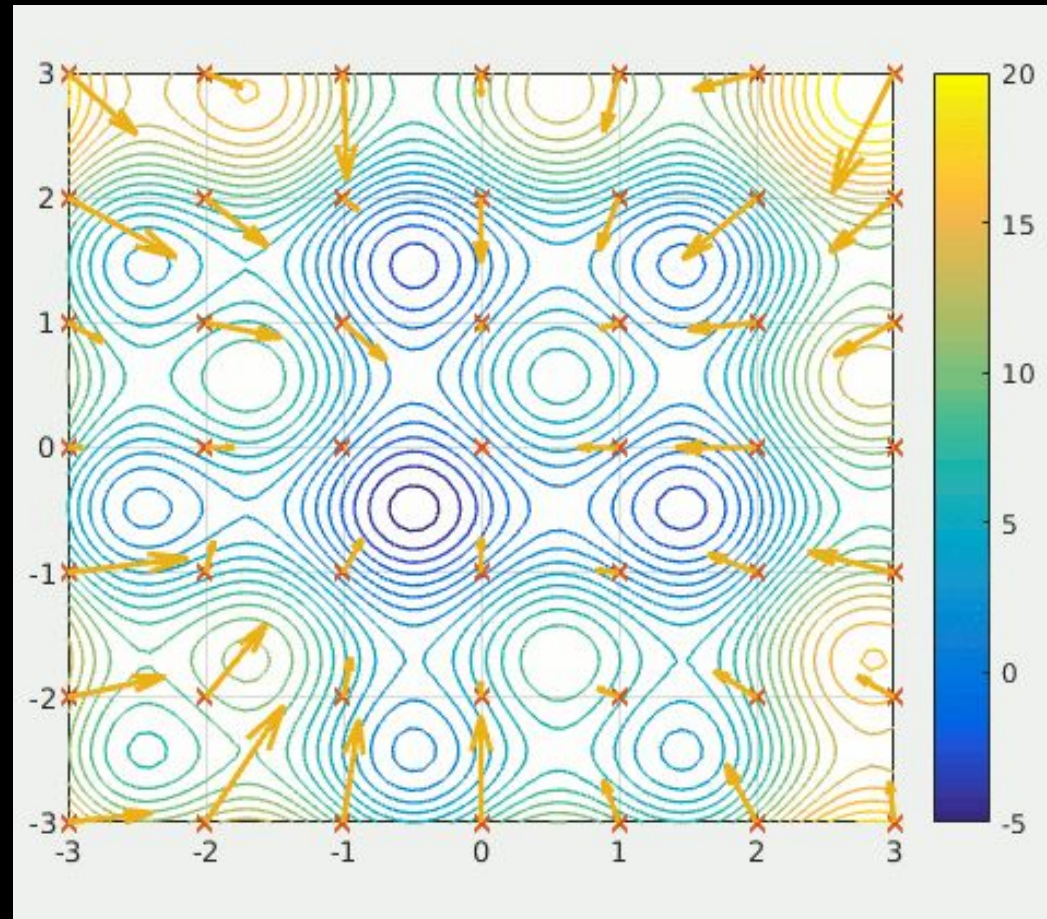
Slow, but *thorough*



Example output:
Microsatellite antenna
optimized by genetic
algorithm

Genetic algorithms in Python

```
'''  
Python script to minimize f()  
'''  
  
import numpy as np  
from scipy.optimize import \\  
    differential_evolution as ga  
bounds = [(-3,3), (-3, 3)]  
res = ga(f, bounds, workers=-1)  
print('Best x:', result.x)  
print('Best f(x):', result.fun)  
'''  
  
Best x: array([-0.5, -0.5])  
Best f(x): -5.0  
'''
```

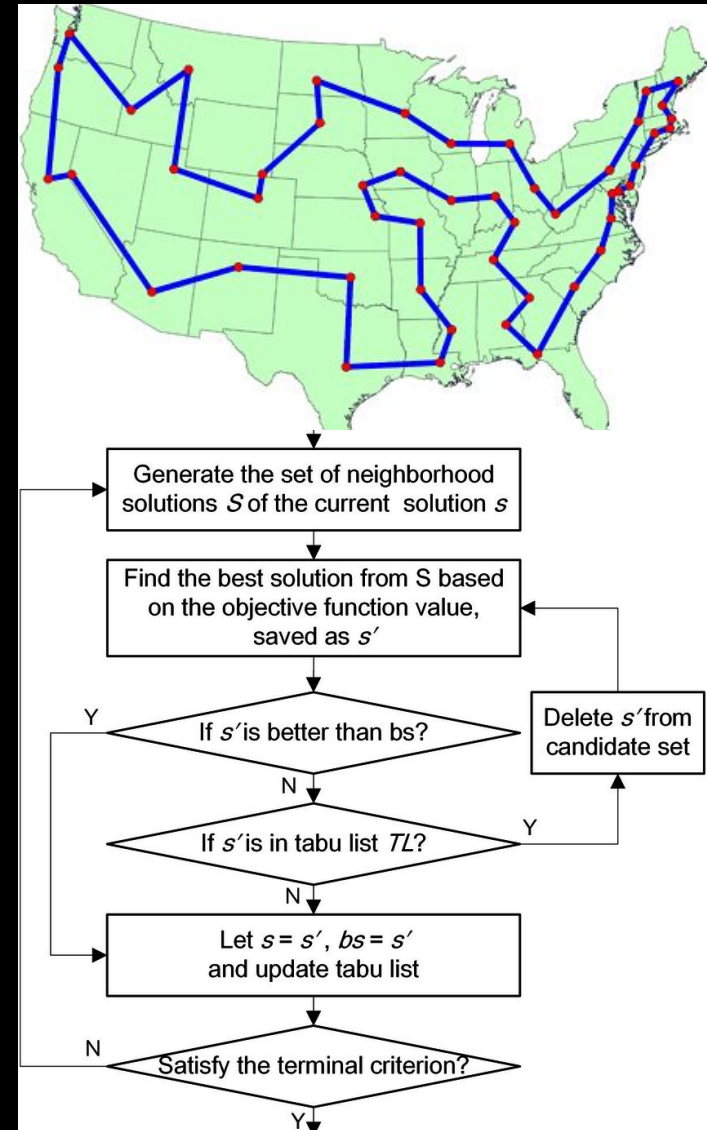


Discrete optimization

- Sometimes you want to optimize over a discrete (not continuous) variable
- Some optimization problems mix continuous & discrete variables (Example?)
- Good heuristic: Tabu Search
 - Start with a random solution
 - Try random improvements
 - Successful improvements are marked as "tabu" (not subject to change) for a few steps, focusing the problem space

Tabu search: traveling salesman

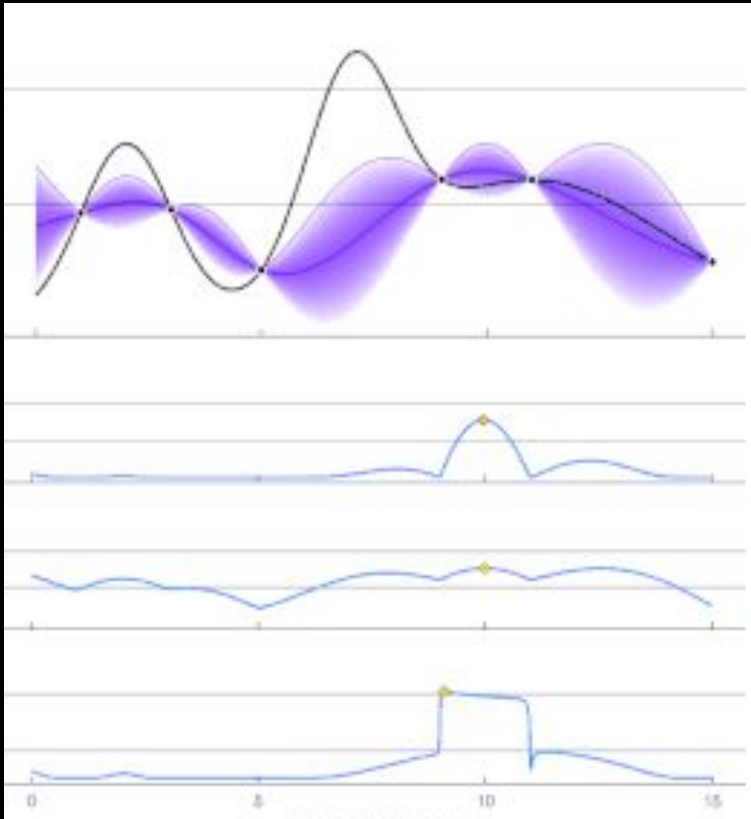
- Optimize length of a complete route
- Start simple (such as nearest neighbors)
- Choose a set S of "moves", pick best
- Recent moves cannot be switched back for N iterations



Model-based optimization

- Make a model with easy gradients, optimize that instead, then go check your result
 - Guess what your model says is best
 - Update your model with the new datapoint and repeat
- Downside: you have to fit the model at every step. Save this for costly functions.
- Neural networks are pretty good for this, so are Gaussian processes: weighted sum of Gaussian basis functions $\sum w_j \exp(-k_j(x - x_j)^2)$

Bayesian optimization



Bayesian optimization with different acquisition functions

- Special case of model-based optimization
- Takes into account uncertainty
- Example: pick x such that $f(x) - \text{stddev}(f(x))$ (purple area in plot) is minimized
 - Not just $f(x)$

Bayesian optimization in Python

```
import numpy as np
from scipy.optimize import minimize, rosen
from sklearn.gaussian_process import GaussianProcessRegressor
Ndims = 2
X = list(np.random.random((3, Ndims))) # initial data x
y = [rosen(x) for x in X] # initial data y
model = GaussianProcessRegressor() # model

def acquisition_function(x, explore_weight=1.0):
    pred, std = model.predict([x], return_std=True)
    return pred - std * explore_weight

for attempt_i in range(10):
    model.fit(X, y) # refit model with all known data
    result = minimize(acquisition_function, np.random.random(Ndims))
    X.append(result.x); y.append(rosen(result.x)) # add new data

print(np.min(y))
```

