Nonlinear Systems

Lecture 5, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 10/13/2022

Announcements

- ***HW2 is due TODAY***. Late submission within one week will receive ***75%*** of the credit
- Please follow TAs' instructions for submission of homework assignments and bonus credits
  - For homework assignments, please submit all solutions/plots/codes ***in a single PDF file***
  - For bonus credits, please submit it as a HW in the TA session and ***paste the link*** to your plots/codes in the box
- Schedule for the midterm presentation has been posted on eLearn. Please let me know ASAP if you have any question. Here's the link to the spreadsheet:

https://docs.google.com/spreadsheets/d/16zh4Xw3bOGANhSBlbKo6is6DRghPC3Nul8tmeOVIl8c/edit?usp=sharing

Schedule for midterm presentations (project proposal)

Date	Time	Name	Title
11/3/22	14:25-14:40	劉欣旻	
	14:40-14:55	李佳倫	
	14:55-15:10	石郡翰	
	15:30-15:45	凌志騰	
	15:45-16:00	張修瑜	
	16:00-16:15	簡嘉成	
	16:15-16:30	謝明學	
11/10/22	14:25-14:40	劉一璠	
	14:40-14:55	郭弈翔	
	14:55-15:10	胡英祈	
	15:30-15:45	周育如	
	15:45-16:00	吳耕緯	
	16:00-16:15	鄭文淇	
	16:30-16:45	龔一桓	
	16:45-17:00	考司圖巴	

Midterm oral presentation for the term project (20%)

- Term project: each student will apply the knowledge or numerical techniques learned in this class to their research projects or a topic of their interest
- The project could be related to one of the following:
  - Reproduce scientific numerical results in published journals
  - Design a new numerical technique (tool/library/application)
  - Tackle an astrophysical problem that involve numerical techniques covered in class
- ***Midterm oral presentation on the project proposal (10+5 mins/person):*** scientific motivations, literature review, descriptions of methods, feasibility

Midterm oral presentation for the term project (20%)

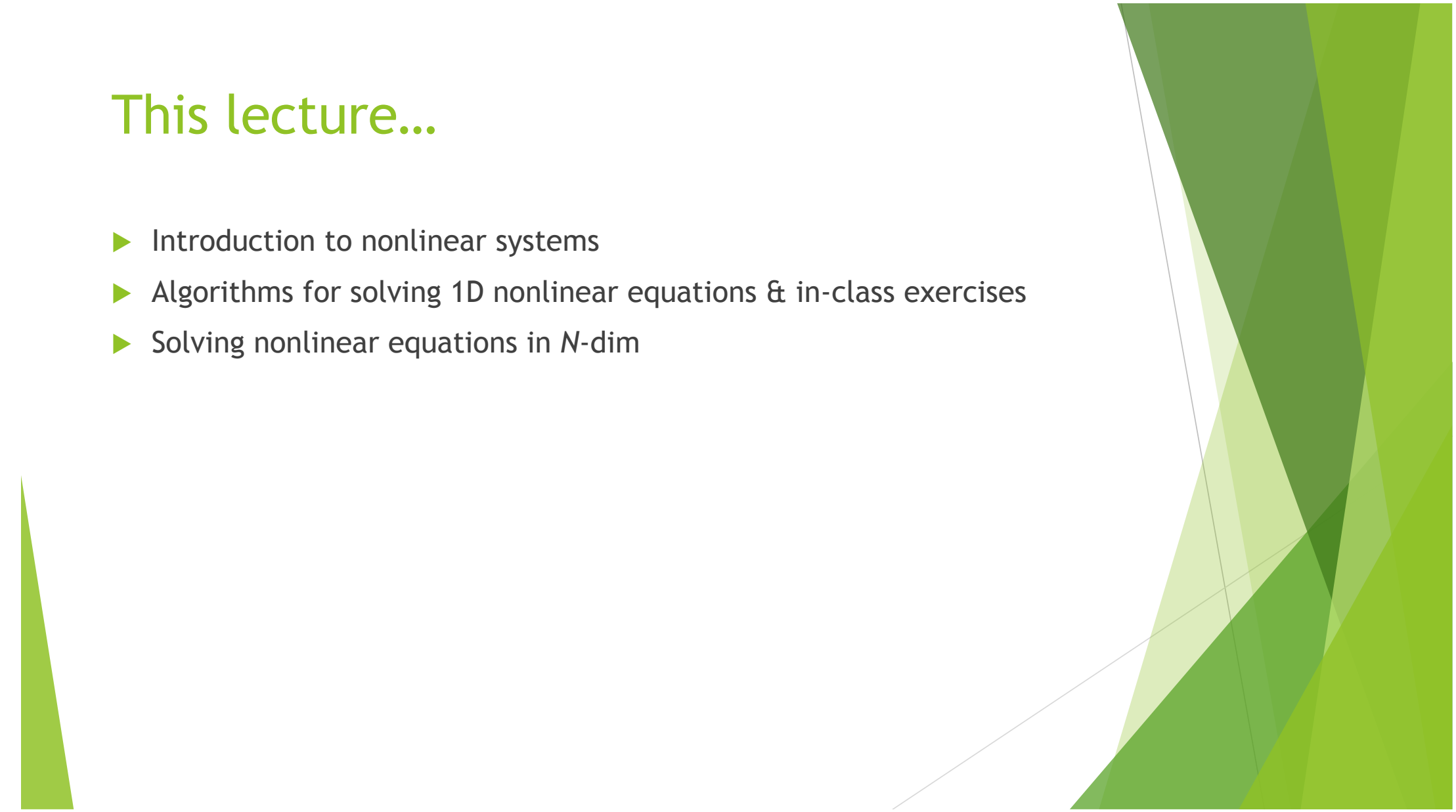
- How the project proposal will be evaluated
  - Based on evaluation from the instructor/TAs/peers
  - Score will depend on (1) ***scientific justification***, (2) ***justification of numerical methods and feasibility***, (3) ***presentation skills***
  - The goal for any proposal is to persuade the reviewers that this project is important, feasible, and worth funding!
- Other notes/reminders about the term project
  - You could either write a code from scratch, or apply existing, publicly available codes to the problem you'd like to address
  - Even if you choose to use existing codes, be sure to understand and describe how their numerical methods work in the presentation (don't treat it as a black box!)
  - Don't worry too much about setting the goal too high, because the final presentation/product of the project will be based on your ***efforts/accomplishments***

Previous lecture...

- Numerical differentiation
  - Forward difference: $f'(x) \equiv \frac{f(x+h)-f(x)}{h}$, *error is* $\propto \mathcal{O}(h)$ *(1st order)*
  - Central difference: $f'(x) \equiv \frac{f(x+h)-f(x-h)}{2h}$, *error is* $\propto \mathcal{O}(h^2)$ *(2nd order)*
  - 2nd derivatives (central difference): $f^{(2)}(x) \equiv \frac{[f(x+h)-f(x)]-[f(x)-f(x-h)]}{h^2}$
- ***Monte Carlo simulations*** uses repeated random sampling to obtain numerical results, requiring (i) known probability distributions, (ii) random numbers, (iii) large sample size
- ***Pseudo random numbers*** are sequence of numbers generated by computers that appear to be random but are deterministic and repeatable. Generators must be chosen carefully to suit your applications

This lecture…

- Introduction to nonlinear systems
- Algorithms for solving 1D nonlinear equations & in-class exercises
- Solving nonlinear equations in *N*-dim



Intro to nonlinear systems

Nonlinear equations

- In physics/astrophysics, we often deal with equations that are *nonlinear in the unknowns*
- **Example in 1D:**

$$x^2 - 4\sin(x) = 0$$

for which $x = 1.9$ is one approximate solution

- **Example in 2D:**

$$\begin{aligned} x_1^2 - x_2 + 0.25 &= 0 \\ -x_1 + x_2^2 + 0.25 &= 0 \end{aligned}$$

for which $\boldsymbol{x} = [0.5 \;\; 0.5]^T$ is the solution vector

- In general, we could write the nonlinear equations as:

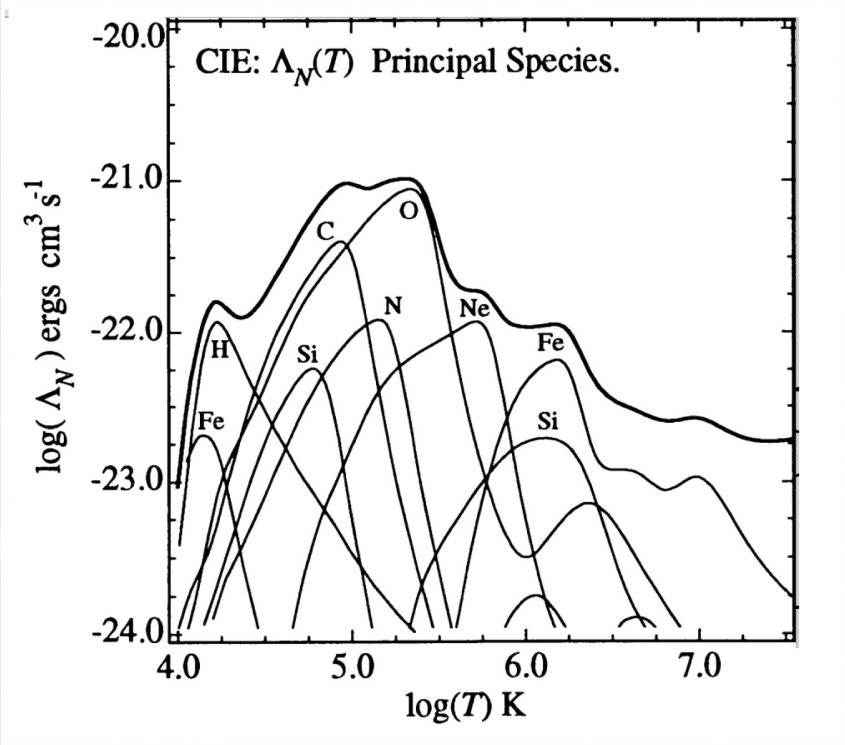
$$f(x) = 0 \quad \text{1D}$$

$$\boldsymbol{f}(\boldsymbol{x}) = 0 \quad \text{nD}$$

Example in astrophysics: temperature of the interstellar medium (ISM)

Example in astrophysics: temperature of the interstellar medium (ISM)

Radiative cooling rate (also called "cooling function") assuming collisional-ionization equilibrium (CIE)



Cooling:

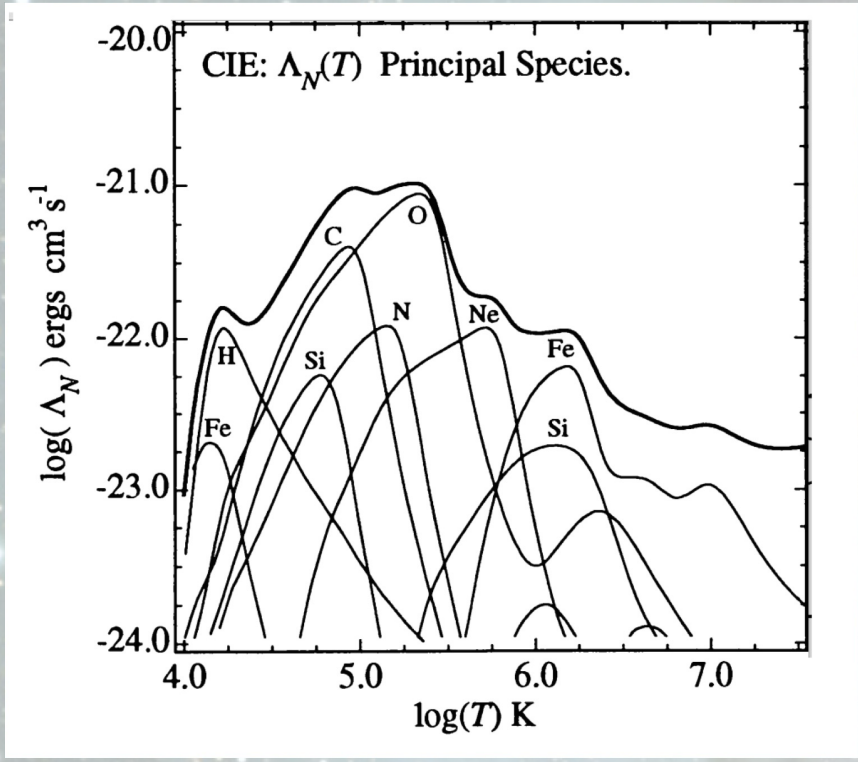
- Bremsstrahlung: collision between electrons and ions
- Radiative cooling: atom-electron collisions
- Thermal radiation from dust grains

Heating:

- Heating from nearby stars
- Heating from cosmic-rays

Credit: KC Pan

Example in astrophysics: temperature of the interstellar medium (ISM)



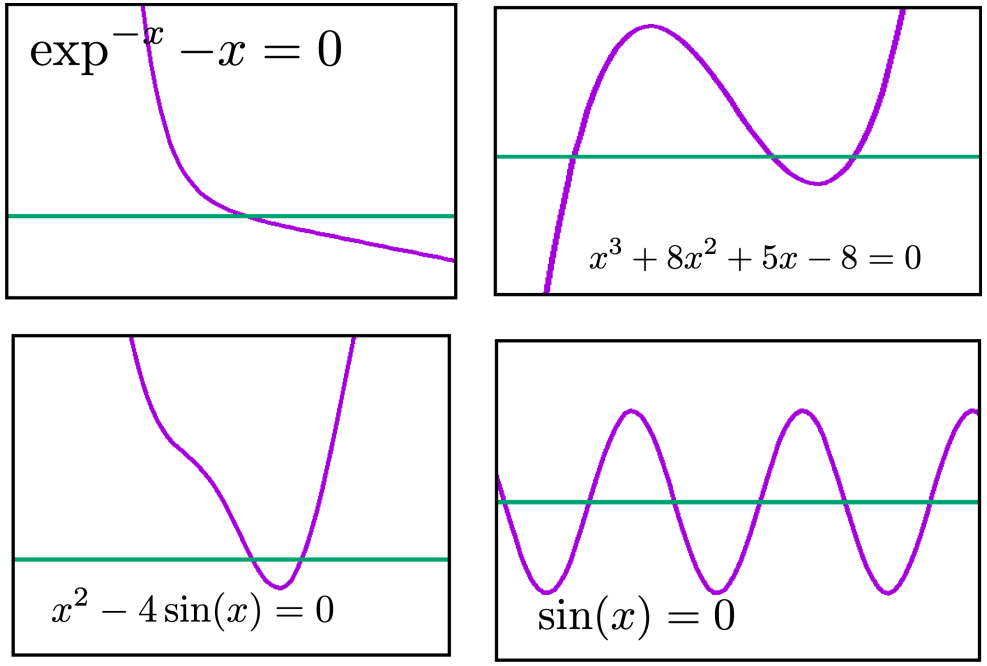
Question:

Find T, such that H - C(T) = 0

Credit: KC Pan

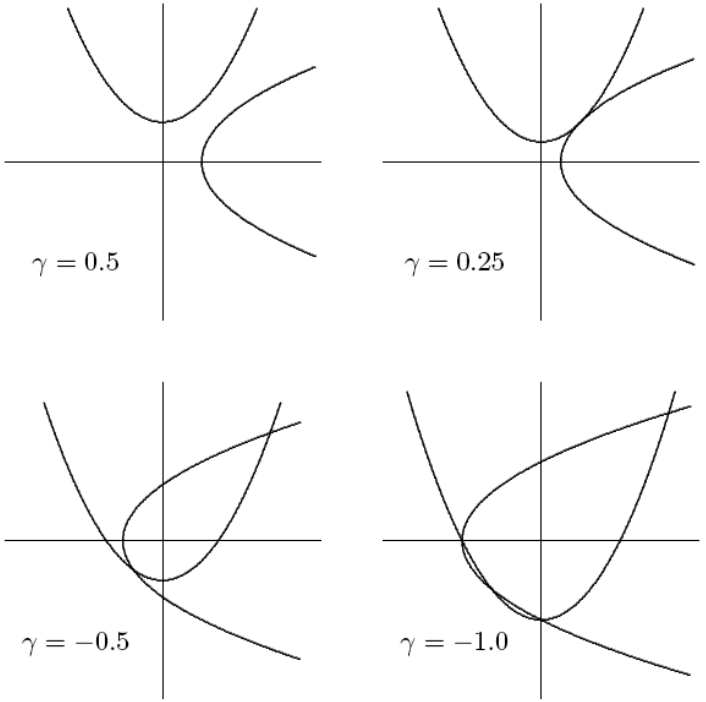
Properties of nonlinear equations

- Nonlinear equations can have any number of solutions
- **Example in 1D:**



- **Example in 2D:**

$$x_1^2 - x_2 + \gamma = 0$$
$$-x_1 + x_2^2 + \gamma = 0$$



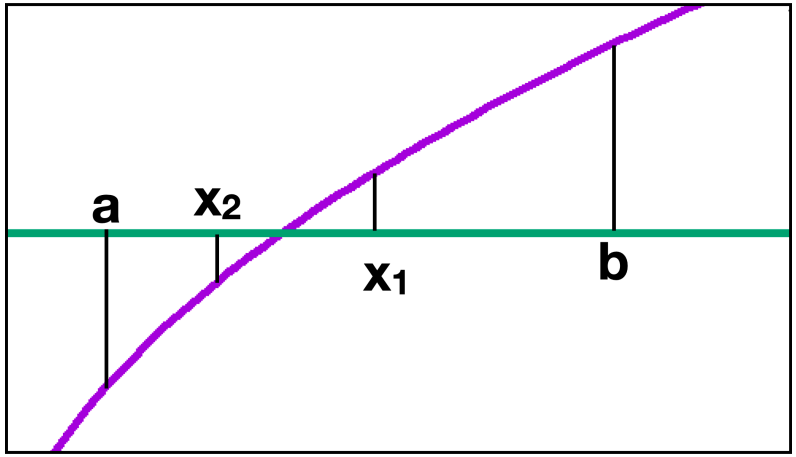
Properties of nonlinear equations

- In *1D*, the problem is reduced to solving for *roots* of the function *f*
  - If *f* is continuous and changes sign in interval [a,b], there must exist a solution between [a,b]
  - Easier to *bracket* the solution and hunt it down
  - Algorithms often involve *iteration* – improve from an initial guess until converged
- *Multi-D* nonlinear equations are more difficult to solve because
  - Determining existence/number of solutions is more complex
  - Choosing a good initial guess is not easy
  - Not guaranteed that solution could be bracketed and converge
  - High computational cost

Root-finding algorithms

Algorithms #1 – bisection method

- Bisection method begins with initial bracket (a and b) and repeatedly halves its length until solution is found

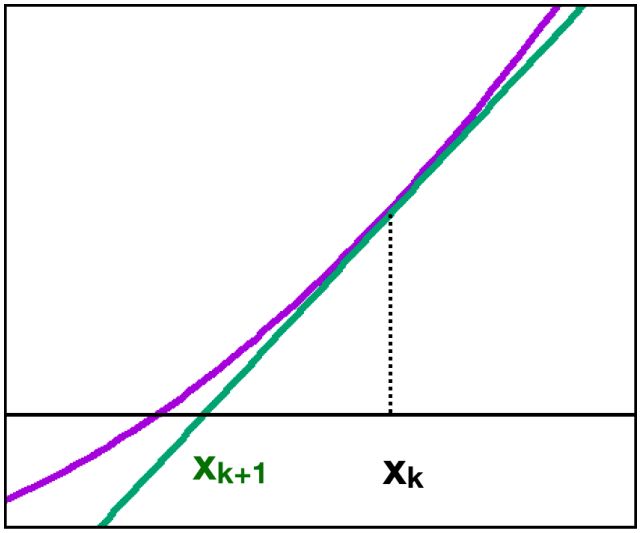


- x = (a+b)/2
- If f(a) and f(x) have different sign, update b
- If f(a) and f(x) have same sign, update a

- Bisection is certain to converge, but does so slowly

Algorithms #2 – Newton-Raphson method

- This algorithm draws a straight line tangent to the curve at *x*, then use the intercept of that line with the *x*-axis as an improved guess for the root



$$f(x) + f'(x)h = 0$$

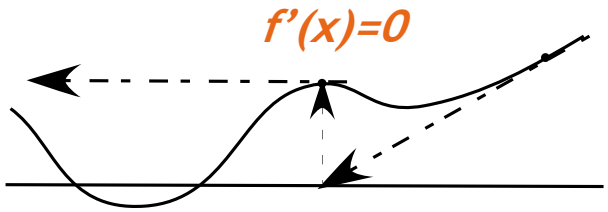
$$h = x_{k+1} - x_k$$

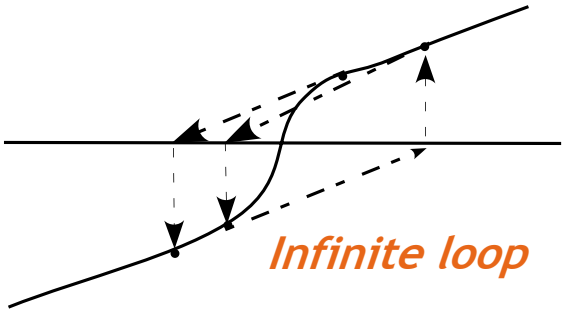
$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

- This method converges faster than the bisection method

Algorithms #2 – Newton-Raphson method

- However, this method may fail when the initial guess is not close enough to the region where *f(x)* is approximately linear
- Examples:

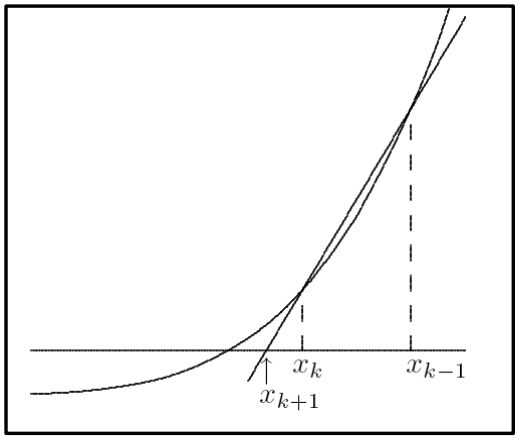




- Solution -- "***backtracking***": use a smaller step (e.g., $h/2$) if the original guess leads to $|f(x+h)|^2 \geq |f(x)|^2$

Algorithms #3 – Secant method

- Similar to Newton-Raphson method, but since computing $f'(x)$ is not easy, approximate $f'(x_k)$ using the slope between x_k and x_{k-1} instead
- This method requires two initial guesses, x_0 and x_1



$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

$$f'(x_k) \sim \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

- Effectively, secant method uses linear interpolation to approximate f(x)
- It's possible to use higher-order polynomial interpolation to achieve faster convergence (e.g., Muller's method uses quadratic interpolation)

In-class exercise

In-class exercise #1 – implementing the bisection method using *modules* in Fortran

- Connect to CICA and load the Fortran compiler:

  ```
  module load pgi
  ```

- Go to your exercise directory on CICA:

  ```
  cd astr660/exercise
  ```

- Copy the template Fortran files to the current directory:

  ```
  cp -r /data/hyang/shared/astro660CompAstro/L5exercise ./
  ```

- Enter the directory and see what files are there:

  ```
  cd L5exercise
  ls
  ```

solver.f90 – where a *module* is defined

Tell the module that "func" will be declared in other files

Use "save" so that the values of a and b are stored every time this subroutine is called

```
module solver

    implicit none
    contains

        subroutine bisection(func, xs, err)
        implicit none
        real, external    :: func    ! the function to solve
        real, intent(out) :: xs      ! solution
        real, intent(out) :: err     ! error
        real, save :: a = 1.0        ! bracking interval [a,b]
        real, save :: b = 3.0        ! bracking interval [a,b]
        real  :: fa, fx              ! f(a) and f(x)

        end subroutine bisection

        subroutine newton(func, dfunc, xs, err)
        implicit none
        real, external    :: func  ! the function to solve
        real, external    :: dfunc ! the first derivative of the function to solve
        real, intent(out) :: xs    ! solution
        real, intent(out) :: err   ! error
        real, save :: x = 2.0      ! trial value
        real :: fx
        real :: dfdx

        end subroutine newton

        subroutine secant(func, xs, err)
        implicit none
        real, external    :: func  ! the function to solve
        real, intent(out) :: xs    ! solution
        real, intent(out) :: err   ! error

        real,save :: x0 = 1.0  ! initial guess
        real,save :: x1 = 3.0  ! initial guess

        real :: fx0, fx1       ! f(x0) and f(x1)

        end subroutine secant

end module solver
```

nonlinear.f90 – where the main program and f(x) are defined

```
program nonlinear

   use solver
   implicit none

   ! variable declarations

   ! output file name
   fname = "bisection.txt"
   open(unit=1,file=trim(fname))
   write(1,11) "#", "N", "solution","Error"

   N     = 1
   error = 1e99

   do while (error > eps)
       call bisection(my_func, xs, error)
       print *, "N = ",N, " solution is ", xs, " error = ", error
       write(1,12) N, xs, error
       N = N+1
       ! check if we have reached the maximum iteration number
       if (N .gt. NMAX) then
           print *, "The problem is not converged within ", N, " iterations."
           stop
       endif
   enddo

   close(1)

11  format(a2,a4,2a24)
12  format(2x,i4,2e24.14)

end program nonlinear

!-----------------------------------------------------
! my_function : The function to solve
!        f(x) = x^2 - 4 sin(x) = 0
!-----------------------------------------------------
real function my_func(x)
        implicit none
        real, intent(in)  :: x

        ! TODO:

        return
end function my_func
```

use the module defined in solver.f90

How shall we compile these files?

- **Step 1**: compile solver.f90 into solver.o and solver.mod

```
gfortran –fdefault-real-8 –fdefault-integer-8 –c solver.f90
```

- **Step 2**: compile nonlinear.f90 into nonlinear.o

```
gfortran –fdefault-real-8 –fdefault-integer-8 –c nonlinear.f90 –o nonlinear.o
```

- **Step 3**: compile *.o into the executable nonlinear (and link required libraries)

```
gfortran –fdefault-real-8 –fdefault-integer-8 solver.o nonlinear.o –o nonlinear
```

Alternatively, we could generate a ***Makefile*** to compile all the codes simply by typing `make`

Using Makefiles

Format of a simple Makefile

```
variable definitions

target : prerequisites...
[ tab ] recipe
```

**See supplemental information for more details about Makefiles

see Makefile_simple

```
F90=gfortran
F90FLAGS= -O2 -fdefault-real-8 -fdefault-integer-8

nonlinear: solver.o nonlinear.o
        $(F90) $(F90FLAGS) solver.o nonlinear.o -o nonlinear

nonlinear.o: nonlinear.f90
        $(F90) $(F90FLAGS) -c nonlinear.f90 -o nonlinear.o

solver.o: solver.f90
        $(F90) $(F90FLAGS) -c solver.f90
```

Using Makefiles

Format of a simple Makefile

```
variable definitions

target : prerequisites...
[ tab ] recipe
```

**See supplemental information for more details about Makefiles

see Makefile

```
F90=gfortran
F90FLAGS= -fdefault-real-8 -fdefault-integer-8

SOURCES=solver.f90 nonlinear.f90

CLEANSTUFF=rm -rf *.o *.mod *.dat *~ nonlinear

OBJECTS=$(SOURCES:.f90=.o)

all: mod main

main: $(OBJECTS)
	$(F90) $(F90FLAGS) $(OBJECTS) -o nonlinear

mod: solver.f90
	$(F90) $(F90FLAGS) -c $^

$(OBJECTS): %.o: %.f90
	$(F90) $(F90FLAGS) -c $< -o $@

clean:
	$(CLEANSTUFF)
```

*convert all *.f90 to *.o*

execute all targets

*compile *.o into executable*

names of all prerequisities

name of the target

name of the 1st prerequisities

remove all compiled files if typing "make clean"

In-class exercise #1 – implementing the bisection method using *modules* in Fortran

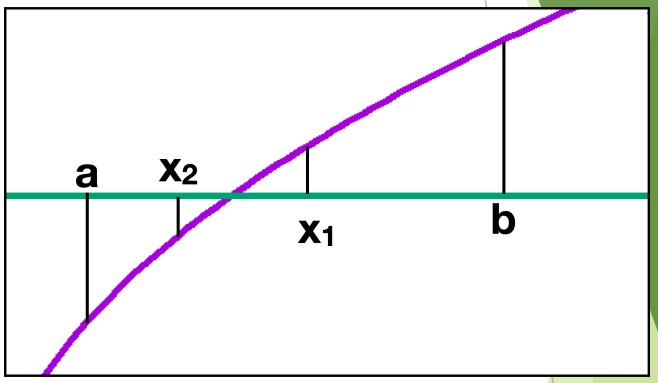
- Let's find the root for:

$$f(x) = x^2 - 4\sin(x) = 0$$

- Fill in the subroutine `bisection` in `solver.f90` (algorithm shown on the right)
- Compile and run the codes by

```
make
./nonlinear
```

- Take a look at `bisection.txt` and verify that your code has found the root



- x = (a+b)/2
- If f(a) and f(x) have different sign, update b
- If f(a) and f(x) have same sign, update a

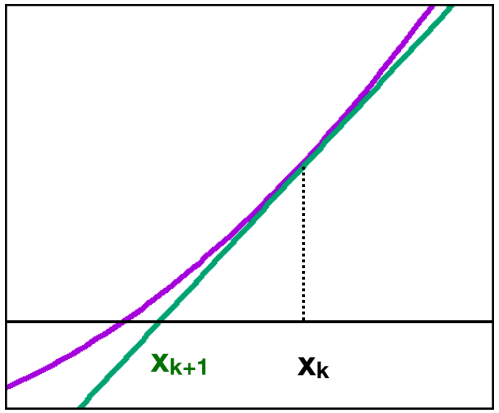
In-class exercise #2 – implement the Newton-Raphson method

- Fill in the subroutine newton in solver.f90 (algorithm shown on the right)
- Modify nonlinear.f90
  - make the code call the newton subroutine and output to newton.txt
  - Implement f'(x)
- Compile and run the codes by

  ```
  make
  ./nonlinear
  ```

- Take a look at newton.txt and verify that your code has found the root

$$f(x) = x^2 - 4\sin(x) = 0$$



$$f(x) + f'(x)h = 0$$

$$h = x_{k+1} - x_k$$

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

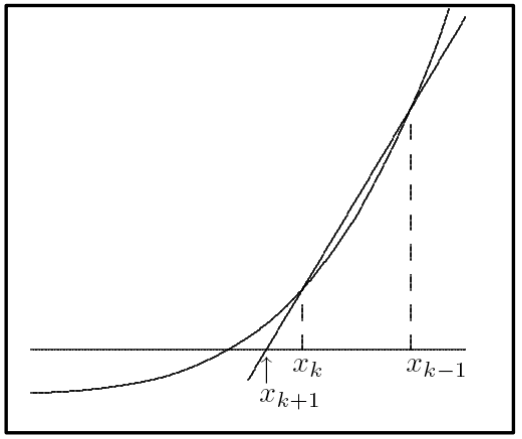
In-class exercise #3 – implement the secant method

- Fill in the subroutine secant in solver.f90 (algorithm shown on the right)
- Modify nonlinear.f90 -- make the code call the secant subroutine and output to secant.txt
- Compile and run the codes by

  ```
  make
  ./nonlinear
  ```

- Take a look at secant.txt and verify that your code has found the root

$$f(x) = x^2 - 4\sin(x) = 0$$



$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

$$f'(x_k) \sim \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

In-class exercise #4 – compare the convergence of different methods

- Now the solutions and errors from the three methods are stored in bisection.txt, newton.txt, and secant.txt
- Use your favorite plotting program, read in the data from the files (e.g., using numpy.loadtxt function in Python)
- ***Plot $\log_{10}$(error) vs. N*** for the three methods (using different line colors)
- To get the bonus credit, please submit both the codes (nonlinear.f90 and solver.f90) and the plot to the TAs by the end of today (10/13/2022)

Q: Which of the three methods converges the fastest?

Convergence rate of iterative methods

- For iterative methods, one can define error at iteration k by

$$\boldsymbol{e}_k = \boldsymbol{x}_k - \boldsymbol{x}^*$$

  where x_k is approximate solution and x^* is the true solution

- Sequence converges with rate r if

$$\lim_{k \to \infty} \frac{\|\boldsymbol{e}_{k+1}\|}{\|\boldsymbol{e}_k\|^r} = C$$

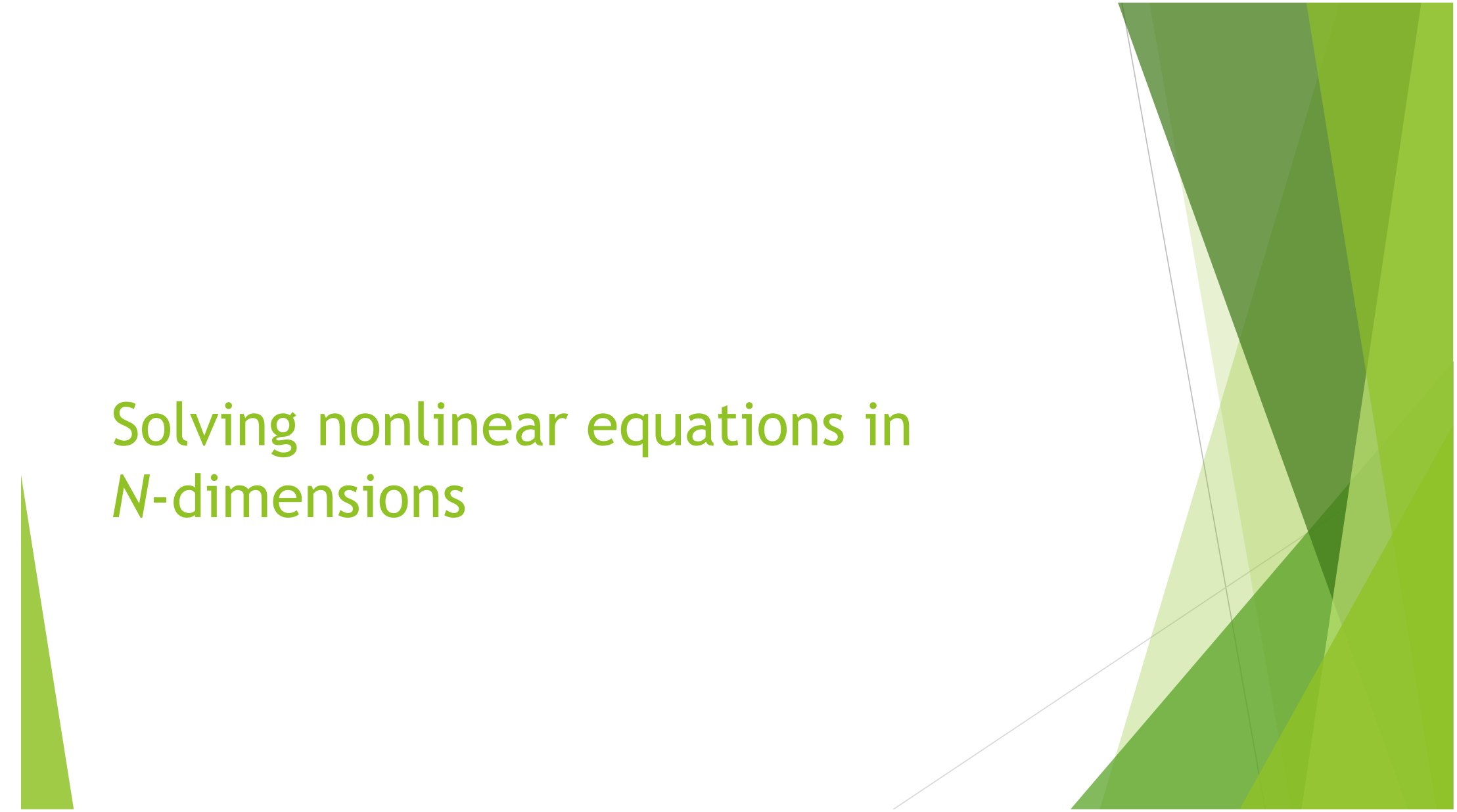
  where C is some finite nonzero constant

- Convergence is faster if r is greater
  - If r = 1: linear (e.g., bisection)
  - If 2 > r > 1: superlinear (e.g., secant)
  - If r = 2: quadratic (e.g., Newton-Raphson)

Convergence -- fast or safe?

- Rapidly convergent methods (e.g., Newton-Raphson) may not converge unless initial guess is close to solution
- However, safe methods (e.g., bisection) are slow
- Solution – ***hybrid*** methods
  - Use rapidly convergent method but maintain ***bracket/safeguard*** around solution
  - If next approximate solution falls outside bracketing interval, perform one iteration of safe method
  - Being closer to solution now, the fast method could have greater chance of success
- In this way, hybrid methods are faster than safe methods and more robust than fast methods

Solving nonlinear equations in *N*-dimensions

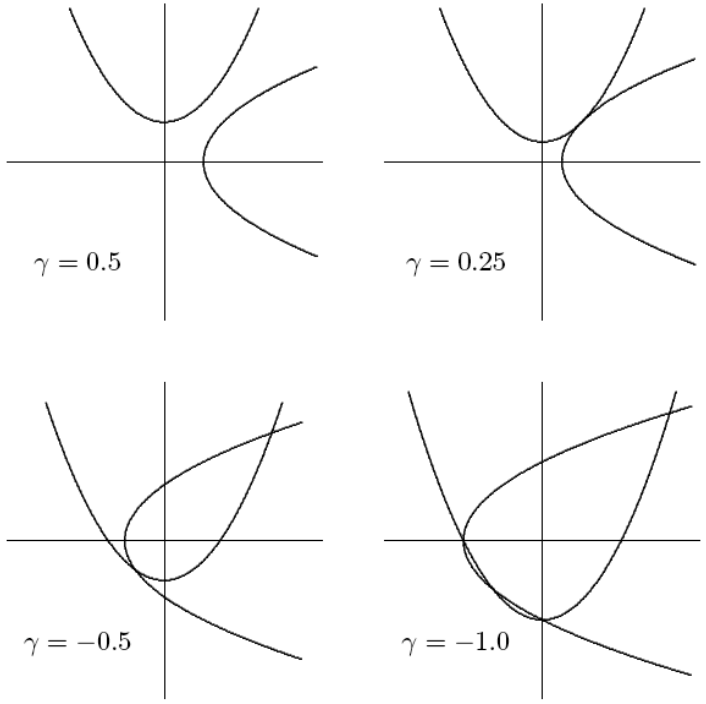


Multi-D nonlinear equations

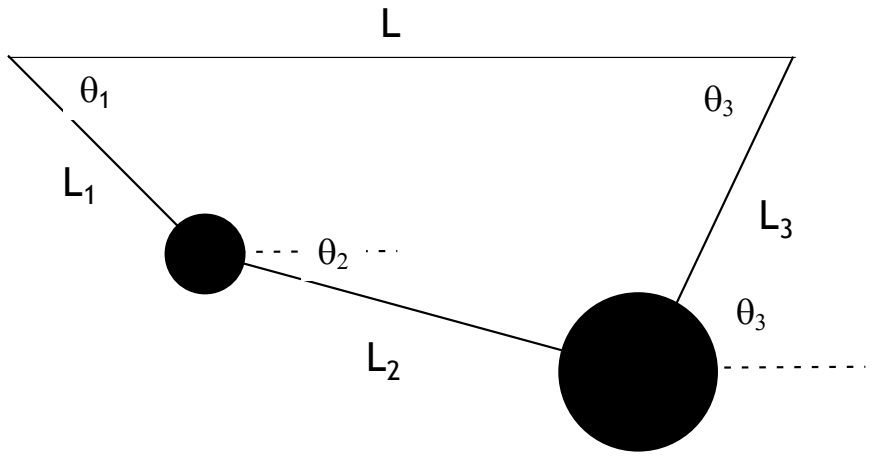
- Are more difficult to solve because
  - Determining existence/number of solutions is more complex
  - Choosing a good initial guess is not easy
  - Not guaranteed that solution could be bracketed and converge
  - High computational cost

- **Example in 2D:**

$$\begin{aligned} x_1^2 - x_2 + \gamma &= 0 \\ -x_1 + x_2^2 + \gamma &= 0 \end{aligned}$$



Multi-D nonlinear equations -- example



- Equations can be written down using geometric constraints and force balance
- Turns out to be a ***9-D*** nonlinear equation!

$$
\begin{aligned}
L_1 \cos\theta_1 + L_2 \cos\theta_2 + L_3 \cos\theta_3 &= L \\
L_1 \sin\theta_1 + L_2 \sin\theta_2 - L_3 \sin\theta_3 &= 0 \\
\sin^2\theta_1 + \cos^2\theta_1 &= 1 \\
\sin^2\theta_2 + \cos^2\theta_2 &= 1 \\
\sin^2\theta_3 + \cos^2\theta_3 &= 1 \\
T_1 \sin\theta_1 - T_2 \sin\theta_2 - W_1 &= 0 \\
T_1 \cos\theta_1 - T_2 \cos\theta_2 &= 0 \\
T_2 \sin\theta_2 + T_3 \sin\theta_3 - W_2 &= 0 \\
T_2 \cos\theta_2 - T_3 \cos\theta_3 &= 0
\end{aligned}
$$

Solving multi-D nonlinear equations

- Fortunately, some of the 1D methods (e.g., Newton-Raphson, secant) are readily generalizable to multi-D problems
- Recall Newton-Raphson method in 1D: $x_{k+1} = x_k - f(x_k)/f'(x_k)$
- In *N*-dim, it becomes

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{J}(\boldsymbol{x}_k)^{-1}\boldsymbol{f}(\boldsymbol{x}_k)$$

where ***J(x)*** is ***Jacobian matrix*** of ***f***:

$$\{\boldsymbol{J}(\boldsymbol{x})\}_{ij} = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$$

- In practice, the Jacobian matrix is not explicitly solved, but instead this *linear* system is solved for the step $\boldsymbol{s}_k$:

$$\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$$

Then the next guess is:

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

*****This also means that multi-D nonlinear equations can be reduced to multi-D linear questions (next lecture)!!***

Example – using Newton-Raphson method for solving 2D nonlinear question

- Let's solve the nonlinear system: $\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{bmatrix} = \boldsymbol{0}$
- Jacobian matrix is $\boldsymbol{J}_f(\boldsymbol{x}) = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}$
- Take an initial guess $\boldsymbol{x}_0 = \begin{bmatrix} 1 & 2 \end{bmatrix}^T$, then

$$\boldsymbol{f}(\boldsymbol{x}_0) = \begin{bmatrix} 3 \\ 13 \end{bmatrix}, \quad \boldsymbol{J}_f(\boldsymbol{x}_0) = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix}$$

- Solving the linear system $\begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix} \boldsymbol{s}_0 = \begin{bmatrix} -3 \\ -13 \end{bmatrix}$ gives $\boldsymbol{s}_0 = \begin{bmatrix} -1.83 \\ -0.58 \end{bmatrix}$
- So, $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{s}_0 = \begin{bmatrix} -0.83 & 1.42 \end{bmatrix}^T$

$$\{\boldsymbol{J}(\boldsymbol{x})\}_{ij} = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$$

$$\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$$

Example – using Newton-Raphson method for solving 2D nonlinear question (cont'd)

- 2nd iteration: $\boldsymbol{f}(\boldsymbol{x}_1) = \begin{bmatrix} 0 \\ 4.72 \end{bmatrix}, \quad \boldsymbol{J}_f(\boldsymbol{x}_1) = \begin{bmatrix} 1 & 2 \\ -1.67 & 11.3 \end{bmatrix}$
- Solving the linear system $\begin{bmatrix} 1 & 2 \\ -1.67 & 11.3 \end{bmatrix} \boldsymbol{s}_1 = \begin{bmatrix} 0 \\ -4.72 \end{bmatrix}$ gives $\boldsymbol{s}_1 = \begin{bmatrix} 0.64 & -0.32 \end{bmatrix}^T$
- So, $\boldsymbol{x}_2 = \boldsymbol{x}_1 + \boldsymbol{s}_1 = \begin{bmatrix} -0.19 & 1.10 \end{bmatrix}^T$
- ....
- Iterations eventually converge to solution $x^* = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$

Remarks about *N*-dim Newton-Raphson method

- Convergence is typically quadratic, as long as the Jacobian matrix is nonsingular
- It must start close enough to solution to converge
  - Needs ***backtracking*** or ***safeguards*** to ensure convergence
- Computational cost per iteration in *N*-dim is substantial!
  - Computing Jacobian matrix costs N^2 function evaluations
  - Solving linear system cost $\mathcal{O}(N^3)$ operations
- To save costs for computing the Jacobian matrix, analog of the ***secant method*** in *N*-dim is available
  - It is called ***Broyden's method***
  - It uses previous guesses to evaluate the Jacobian matrix
  - Convergence is superlinear

Nonlinear systems -- summary

- In physics/astrophysics systems we often encounter equations that are nonlinear in the unknown variables.
- 1D nonlinear equations are easier to solve => ***root-finding*** problem
- Commonly used 1D root-finding algorithms are based on iterations:
  - ***Bisection***: bracket the solution and halve the search region
  - ***Newton-Raphson***: use intercept of tangent line as next guess
  - ***Secant***: similar to Newton-Raphson but use previous guesses to evaluate f'(x)
- Rapidly convergent methods (e.g., Newton-Raphson) only works when initial guess is close enough to solution
  - Need to combine with ***backtracking/safeguard/hybrid*** methods
- For *N*-dim nonlinear systems, Newton-Raphson & secant analogs are available

$f(x) = 0$ 1D

$\boldsymbol{f}(\boldsymbol{x}) = 0$ nD

Supplemental information

Resources for Makefiles

- A complete introduction to GNU make:

https://www.gnu.org/software/make/manual/make.html - Introduction

- Learn Makefiles:

https://makefiletutorial.com

- 簡單學makefile (in Mandarin):

https://mropengate.blogspot.com/2018/01/makefile.html

References & acknowledgements

- Course materials of Computational Astrophysics from Prof. Kuo-Chuan Pan (NTHU)
- Course materials of Computational Astrophysics from Prof. Hsi-Yu Schive (NTU)
- Course materials of Computational Astrophysics and Cosmology from Prof. Paul Ricker (UIUC)
- “Computational Physics” by Rubin H. Landau, Manuel Jose Paez and Cristian C. Bordeianu
- “Scientific Computing – An Introductory Survey” by Michael T. Heath