PHY604 Lecture 6

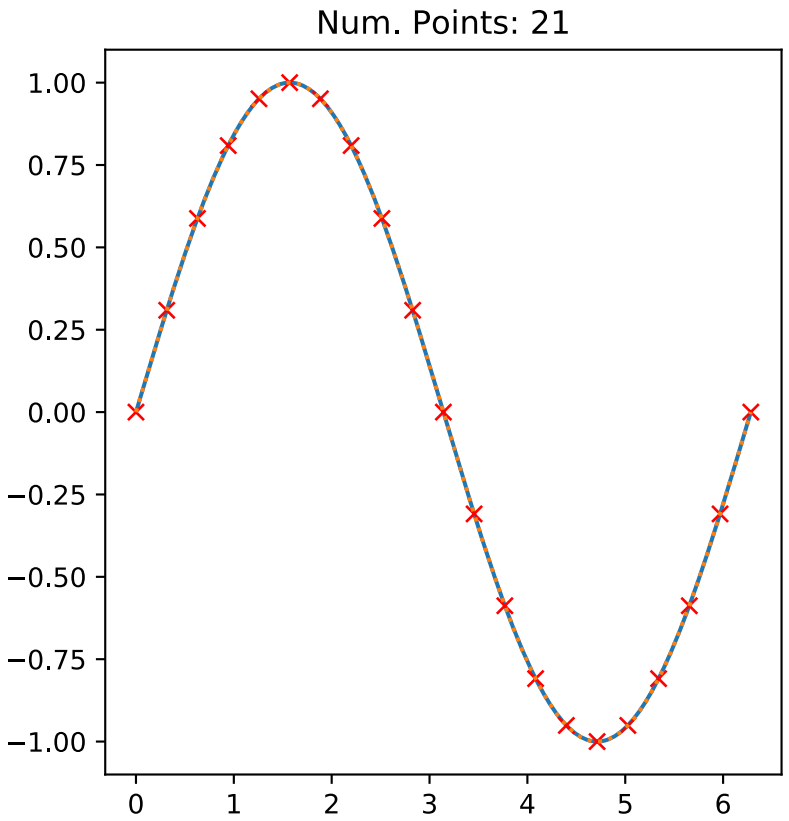
September 9, 2021

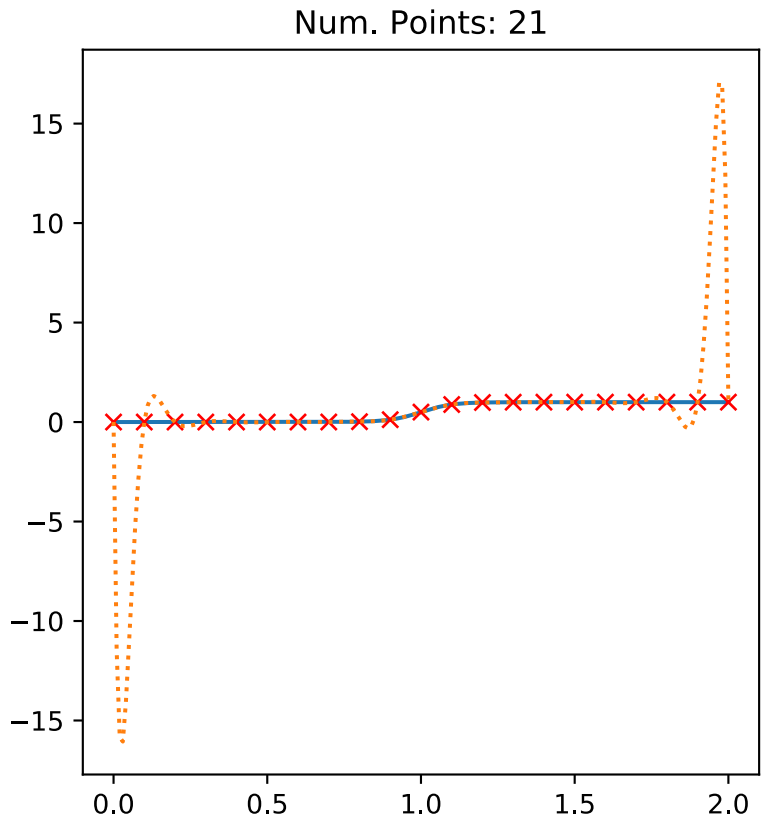
Review: Lagrange interpolation

- General method for building a single polynomial that goes through all the points (alternate formulations exist)

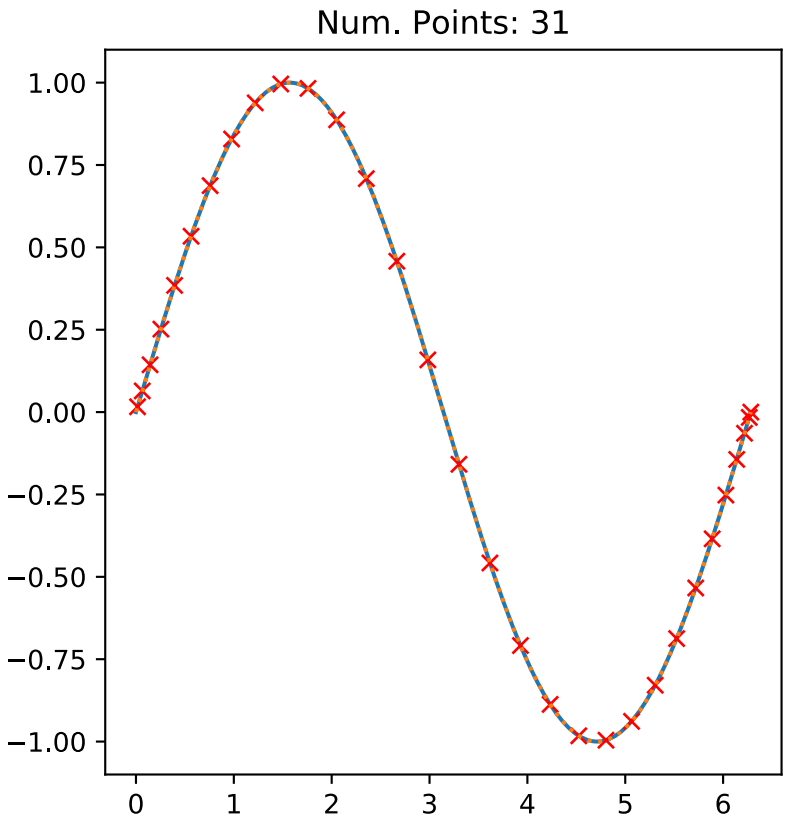
- Given n points: $x_0, x_1, \ldots, x_{n-1}$, with associated function values: $f_0, f_1, \ldots, f_{n-1}$
  - Construct basis functions: $$l_i(x) = \prod_{j=0, i \neq j}^{n-1} \frac{x - x_j}{x_i - x_j}$$
  - Note basis function l_i is 0 at all x_j except for x_i (where it is one)
  - Function value at x is: $$f(x) = \sum_{i=0}^{n-1} l_i(x) f_i$$

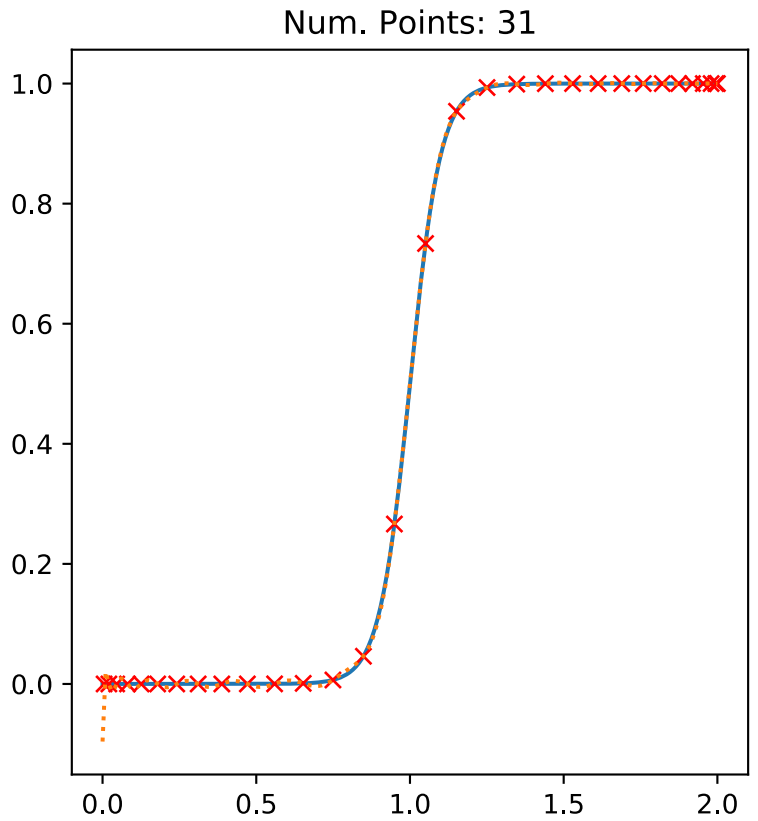
Review: Lagrange Interpolation of two functions on even grid



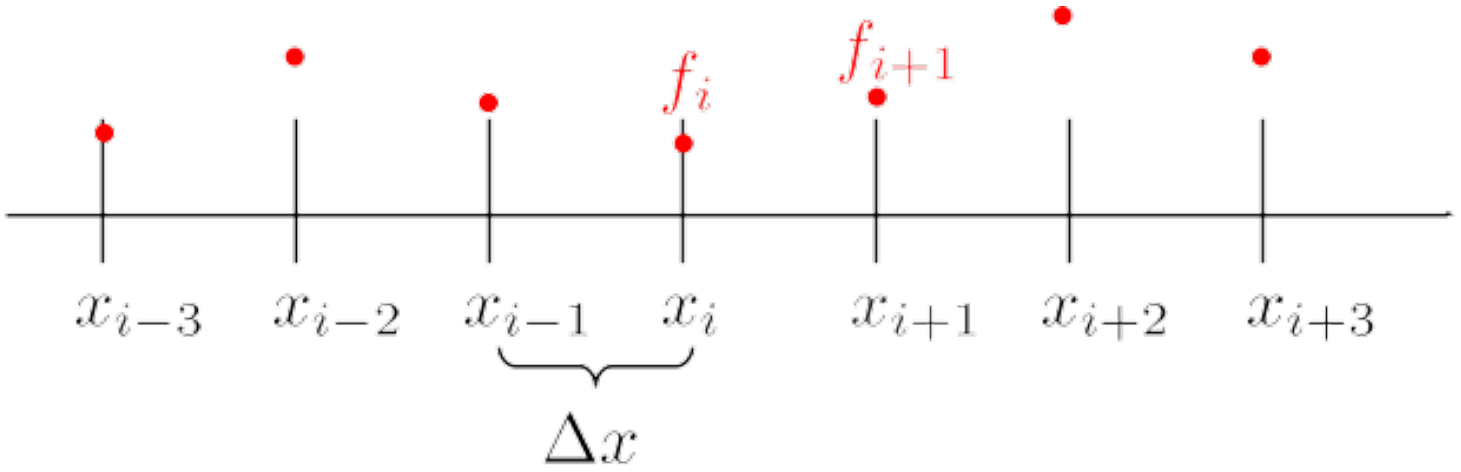


Review: Lagrange Interpolation of two functions with Chebyshev nodes





Review: Splines



- We have a set of regular-spaced discrete data: $f_i=x(x_i)$ at $x_0,x_1,x_2,\ldots,x_n$
- *m*-th order polynomial to approximate $f(x)$ for x in $[x_i,x_{i+1}]$:

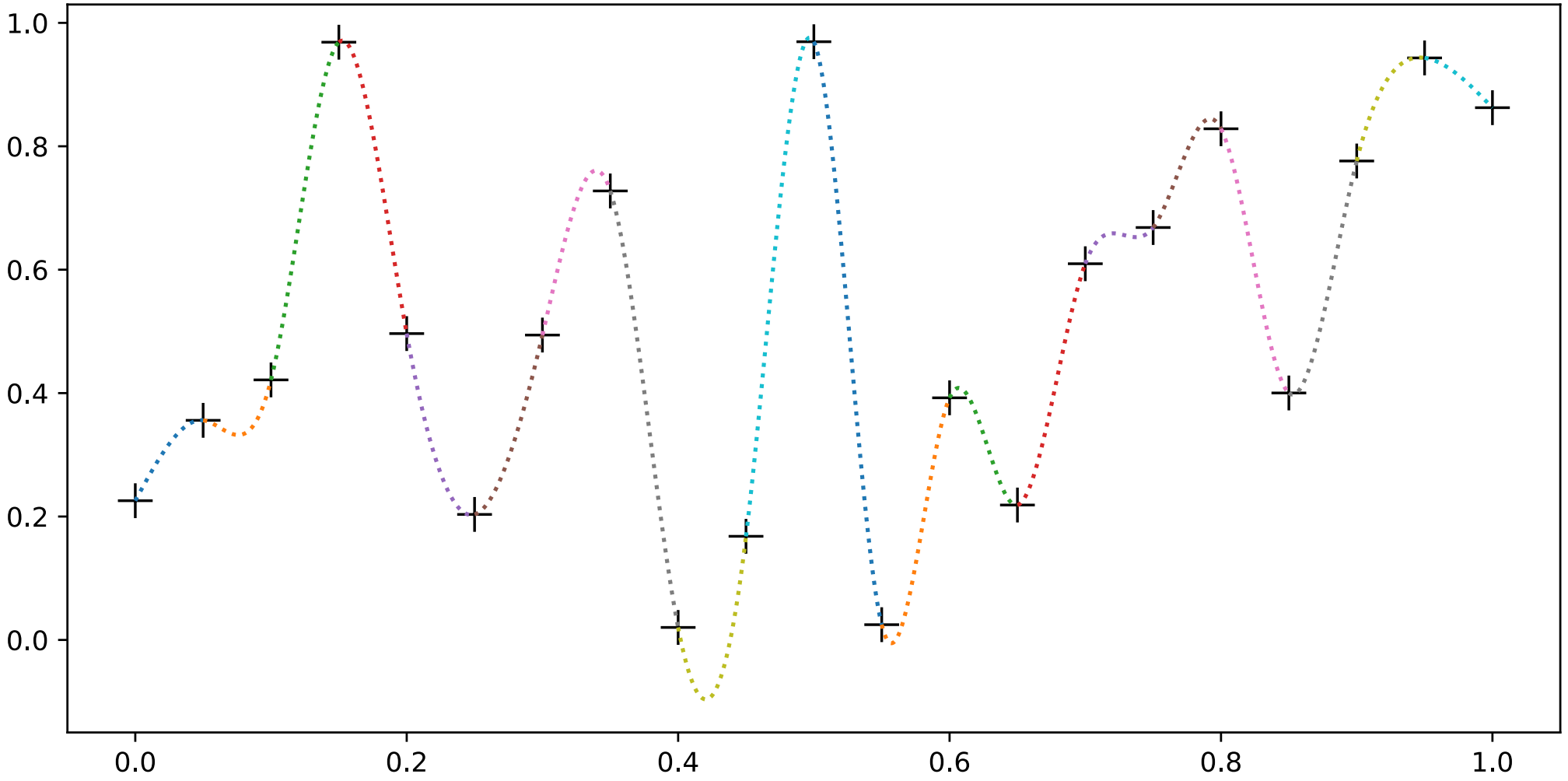
$$p_i(x) = \sum_{k=0}^{m} c_{ik} x^k$$

- Coefficients chosen so $p_i(x_i)=f_i$ and from smoothness condition: all derivatives (l) match at the endpoints

$$p_i^{(l)}(x_{i+1}) = p_{i+1}^{(l)}(x_{i+1}), \quad l = 0, 1, \ldots, m-1$$

- Except for points on the boundary of the curve

Review: Cubic spline for random numbers



Today’s lecture

- Finish discussing roots of functions:
  - Newton Raphson method
  - Secant method
- Begin discussing ordinary differential equations

Newton-Raphson method procedure

- 1. Make an **initial guess** for the root: x_0
- 2. Use the Taylor series expansion to **find a better estimate** of the root:

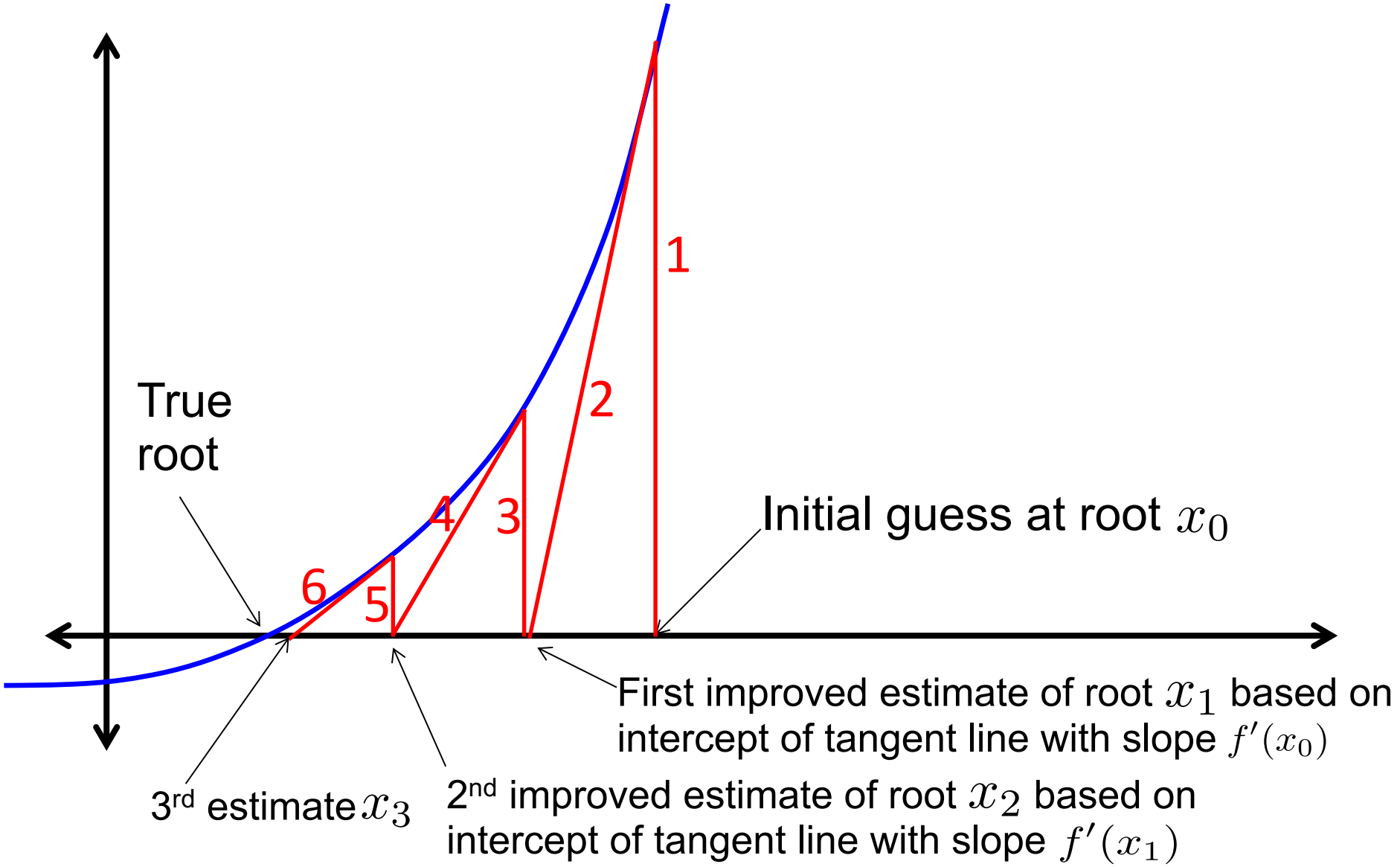
$$x_1 \simeq x_0 - \frac{f(x_0)}{f'(x_0)}$$

- 3. Use x_1 as an improved estimate at the root and employ the Taylor series expansion again to get a better estimate x_2

- Repeat process until the answer is accurate enough at the *n*th estimate:

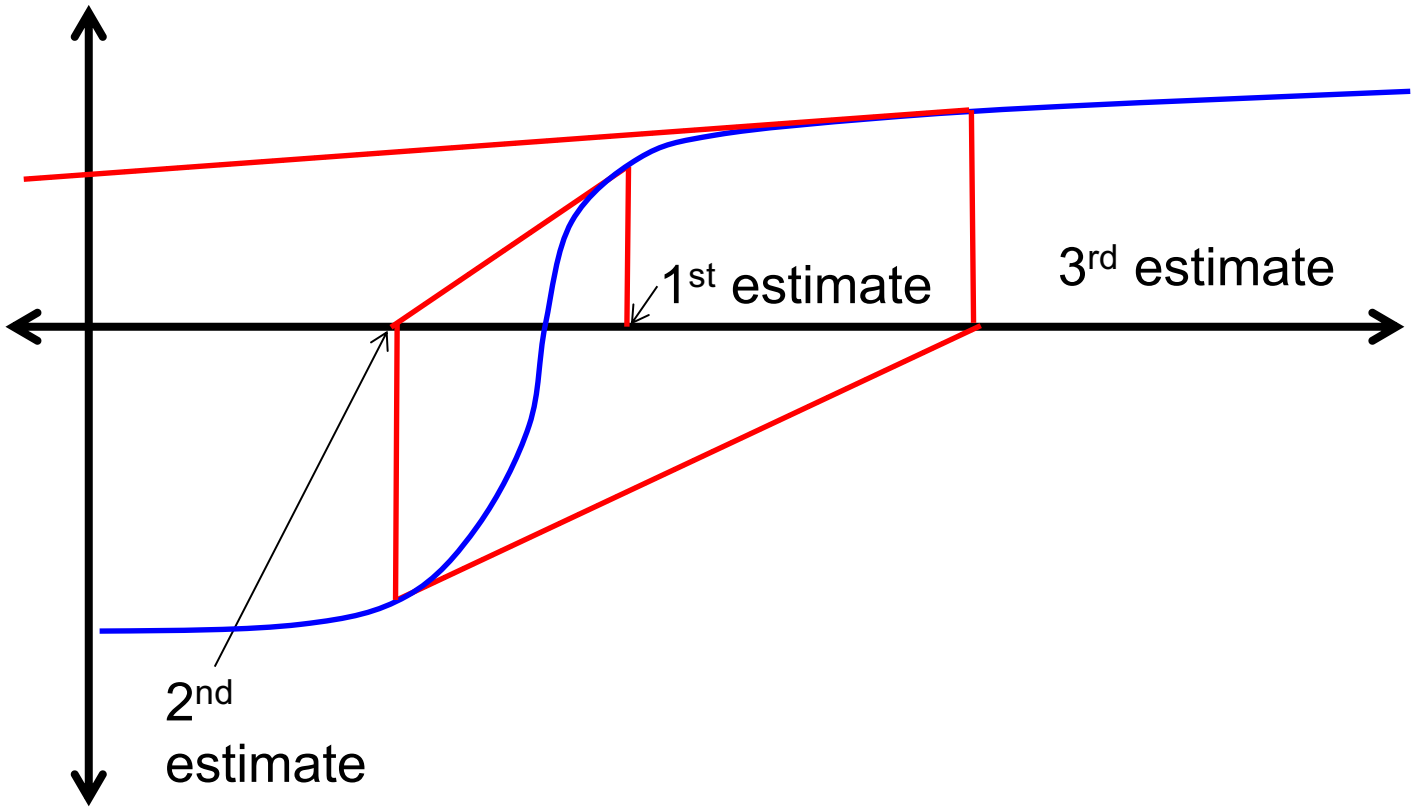
$$x_n \simeq x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

Geometrical Interpretation of Newton-Raphson Iteration



Failure of Newton-Raphson

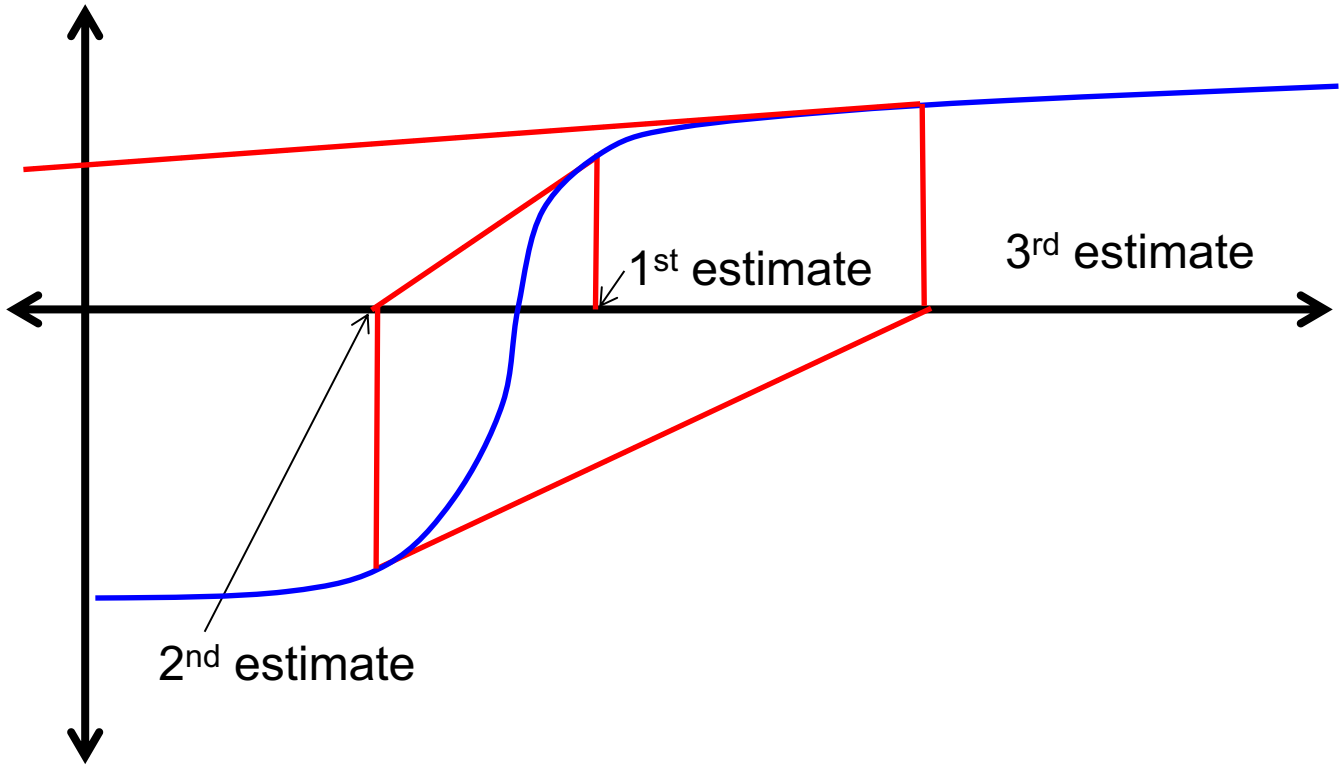
- Example of a simple function that will defeat Newton-Raphson Iteration:



- Each estimate gets further from the true root. Estimates are diverging not converging

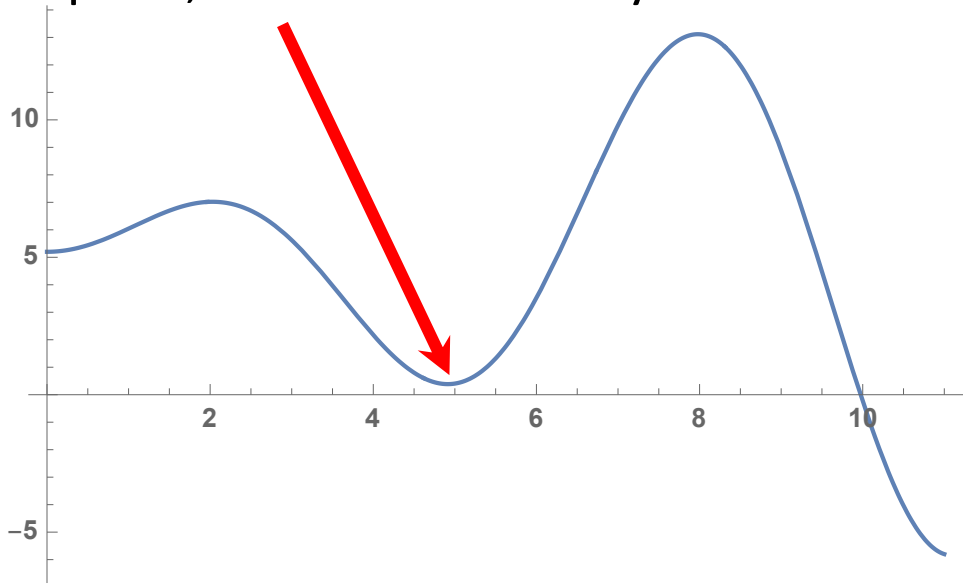
Stopping criteria for iterations must be chosen carefully

- Could stop when we reach some maximum number of iterations
  - Estimate may be no where near the root
  - We can consider this case a failure of the method and warn user about it.



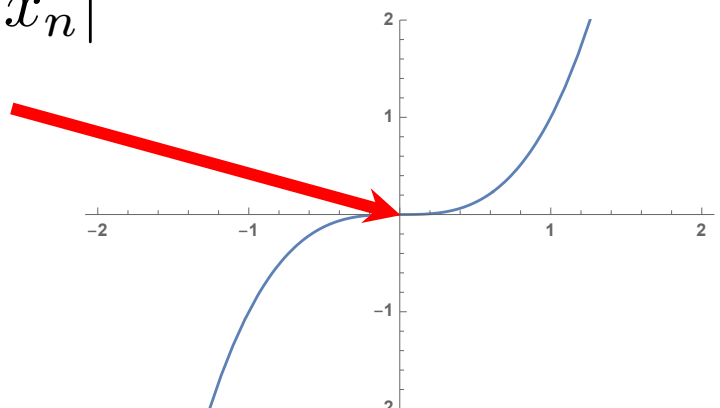
Stopping criteria for iterations must be chosen carefully

- Could stop when we reach some maximum number of iterations
  - Estimate may be no where near the root
  - We can consider this case a failure of the method and warn user about it.
- Could stop when value of the function evaluated at the *n*th estimate less than small number : $|f(x_n)| < \epsilon$
  - But this can be deceptive; final estimate may not be near the root, might just be close to zero



Stopping criteria for iterations must be chosen carefully

- Could stop when we reach some maximum number of iterations
  - Estimate may be no where near the root
  - We can consider this case a failure of the method and warn user about it.
- Could stop when value of the function evaluated at the *n*th estimate less than small number : $|f(x_n)| < \epsilon$
  - But this can be deceptive; final estimate may not be near the root, might just be close to zero
- Could stop when change between estimates becomes small relative to the current (*n*th) estimate: $|x_{n+1} - x_n| < \epsilon |x_n|$
  - Better, but still fails when root is located at zero



Stopping criteria for iterations must be chosen carefully

- Could stop when we reach some maximum number of iterations
  - Estimate may be no where near the root
  - We can consider this case a failure of the method and warn user about it.
- Could stop when value of the function evaluated at the *n*th estimate less than small number : $|f(x_n)| < \epsilon$
  - But this can be deceptive; final estimate may not be near the root, might just be close to zero
- Could stop when change between estimates becomes small relative to the current (*n*th) estimate: $|x_{n+1} - x_n| < \epsilon |x_n|$
  - Better, but still fails when root is located at zero
- So let's use:

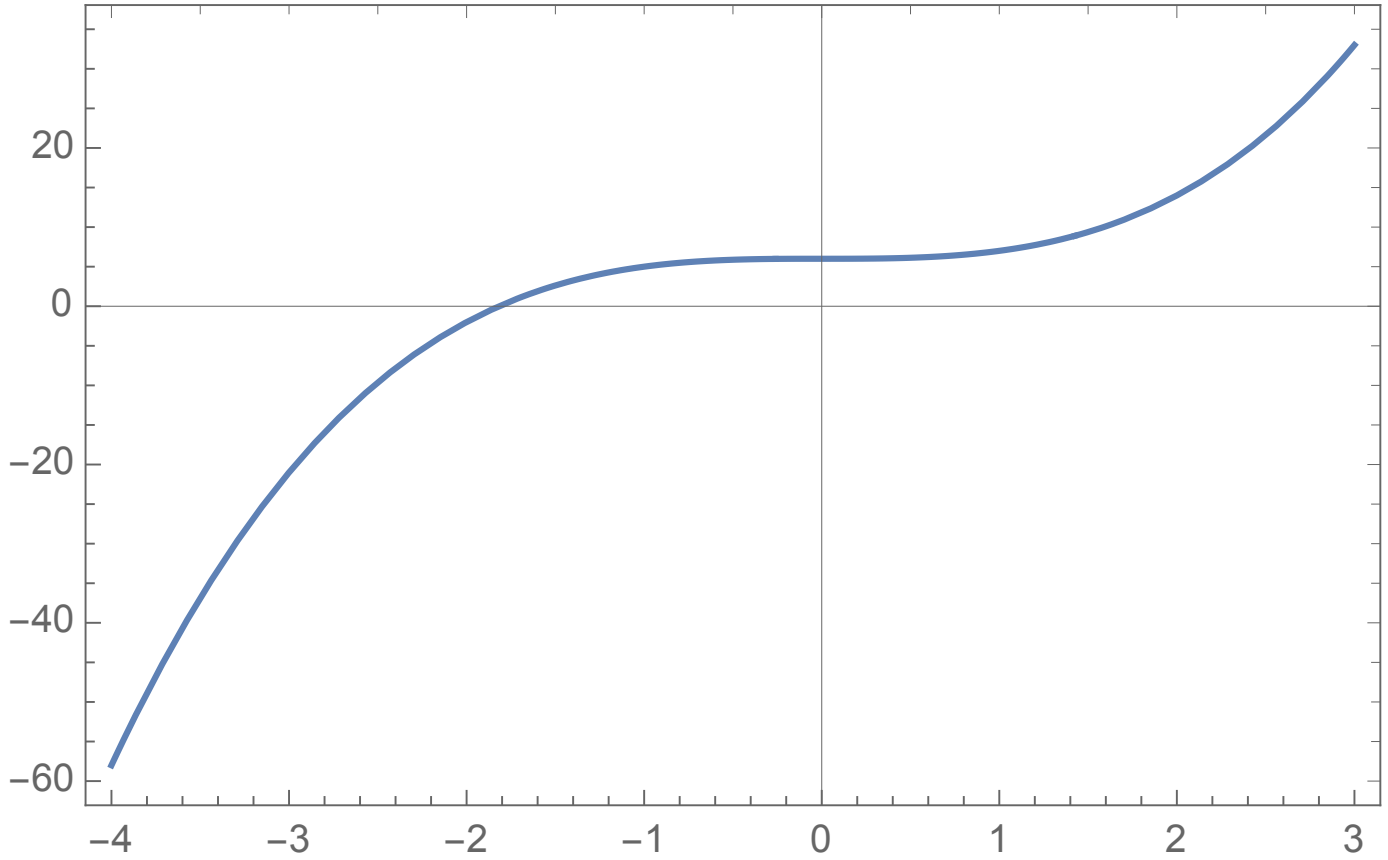
$$|x_{n+1} - x_n| < \begin{cases} \epsilon |x_n|, & \text{when } |x_n| \neq 0 \\ \epsilon, & \text{when } |x_n| = 0 \end{cases}$$

Pseudocode of Newton-Raphson Algorithm

- 1. Choose initial guess at the root (x_0), and the convergence tolerance (ε).
- 2. Loop through *n* up to a maximum number N_{max} (exit and tell the user that the root finding has failed if it reaches N_{max})
- 3. Make sure f'(x) ≠ 0
- 4. Compute new estimate of root: $x_n \simeq x_{n-1} - \dfrac{f(x_{n-1})}{f'(x_{n-1})}$
- 5. Check convergence criteria:

$$|x_{n+1} - x_n| < \begin{cases} \epsilon|x_n|, & \text{when } |x_n| \neq 0 \\ \epsilon, & \text{when } |x_n| = 0 \end{cases}$$

Example: f(x) = x^3 + 6



- See **`NR_root.f08`**

Secant method

- Similar to the Newton-Raphson method, but does not require calculating the derivative of the function
- Start with two initial guesses, x_{i-1} and x_i
- Use finite difference derivative to get a new guess x_{i+1}

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

- Proceed in the same way as the Newton-Raphson method

Summary of root-finding methods

- Bisection:
  - Robust (with appropriate initial guesses)
  - Slow, each iteration reduces error by a factor of two
  - Need to make sure root is within initial guesses

- Newton-Raphson:
  - Fast: often only takes a few iterations
  - Need to know derivative of function, and they must exist
  - Can diverge, e.g., in cases with small second derivatives

- Secant method
  - Similar convergence speed as NR method
  - Don’t need analytical derivatives
  - Same divergence properties as NR method
  - Numerical derivatives may be noisy

Today’s lecture

- Finish discussing roots of functions:
  - Newton Raphson method
  - Secant method
- Begin discussing ordinary differential equations

Differential equations (Newman Ch. 8)

- One of the major applications of computation to science and engineering is solving differential equations
  - Even for very simple-looking equations if they are "nonlinear," they are difficult or imposible to solve analytically
- Classifications:
  - Initial value problems
  - Boundary value problems
  - Eigenvalue problems
- Often problems are described by systems of coupled differential equations
- As with the other topics, there are many different methods
  - We just want to see the basic ideas and popular methods

Example of system of differential equations: Equations of motion

- We know that the equations of motion for a point particle with mass are given by:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}(t), \qquad \frac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{x}, \mathbf{v}, t)$$

- In order to fully describe the trajectory of this particle, we need to specify initial conditions, i.e., the position and velocity, of the particle at the initial time $t = 0$:

$$\mathbf{x}(0) = \mathbf{x}_0, \qquad \mathbf{v}(0) = \mathbf{v}_0$$

Approximating the Equations of Motion

- If we consider a time interval that is sufficiently short, we can approximate the differential by

$$dt \simeq \Delta t$$

- We can then approximate the time derivative of the position by:

$$\frac{d\mathbf{x}}{dt} \simeq \frac{\mathbf{x}(t+\Delta t) - \mathbf{x}(t)}{\Delta t}$$

- Similarly, the time derivative of the velocity can be approximated by

$$\frac{d\mathbf{v}}{dt} \simeq \frac{\mathbf{v}(t+\Delta t) - \mathbf{v}(t)}{\Delta t}$$

Euler's method for integrating the equations of motion

- We can then substitute the approximate derivatives into the equations of motion to obtain:

$$\frac{\mathbf{x}(t+\Delta t)-\mathbf{x}(t)}{\Delta t} \simeq \mathbf{v}(t), \quad \frac{\mathbf{v}(t+\Delta t)-\mathbf{v}(t)}{\Delta t} \simeq \mathbf{a}(\mathbf{x},\mathbf{v},t)$$

- We can then solve for the new values of the position and velocity

$$\mathbf{v}(t+\Delta t) \simeq \mathbf{v}(t) + \mathbf{a}(\mathbf{x},\mathbf{v},t)\Delta t$$

$$\mathbf{x}(t+\Delta t) \simeq \mathbf{x}(t) + \mathbf{v}(t)\Delta t$$

- This algorithm for "integrating" the equations of motion forward in time in known as Euler's method

Example: A body orbiting the sun

- We consider the Sun's location to be at the origin and the plane of the orbit to be the x-y plane

- In this case we have: $\mathbf{a}(\mathbf{x}) = \dfrac{-GM_{\text{sun}}}{r^2}\hat{\mathbf{x}}$

- Where: $\hat{\mathbf{x}} = \dfrac{\mathbf{x}}{r} = \dfrac{\mathbf{x}}{x^2 + y^2}$

- The components of the acceleration are then given by:

$$a_x(x, y) = \frac{-GM_{\text{sun}}x}{r^3}, \quad a_y(x, y) = \frac{-GM_{\text{sun}}y}{r^3}$$

Euler's method for body orbiting the sun

- Now we discretize in time and apply Euler's method:

$$v_x(t+\Delta t) = v_x(t) - \frac{GM_{\text{sun}}x(t)\Delta t}{\left(x(t)^2 + y(t)^2\right)^{3/2}}$$

$$v_y(t+\Delta t) = v_y(t) - \frac{GM_{\text{sun}}y(t)\Delta t}{\left(x(t)^2 + y(t)^2\right)^{3/2}}$$

$$x(t+\Delta t) = x(t) + v_x(t)\Delta t$$

$$y(t+\Delta t) = y(t) + v_y(t)\Delta t$$

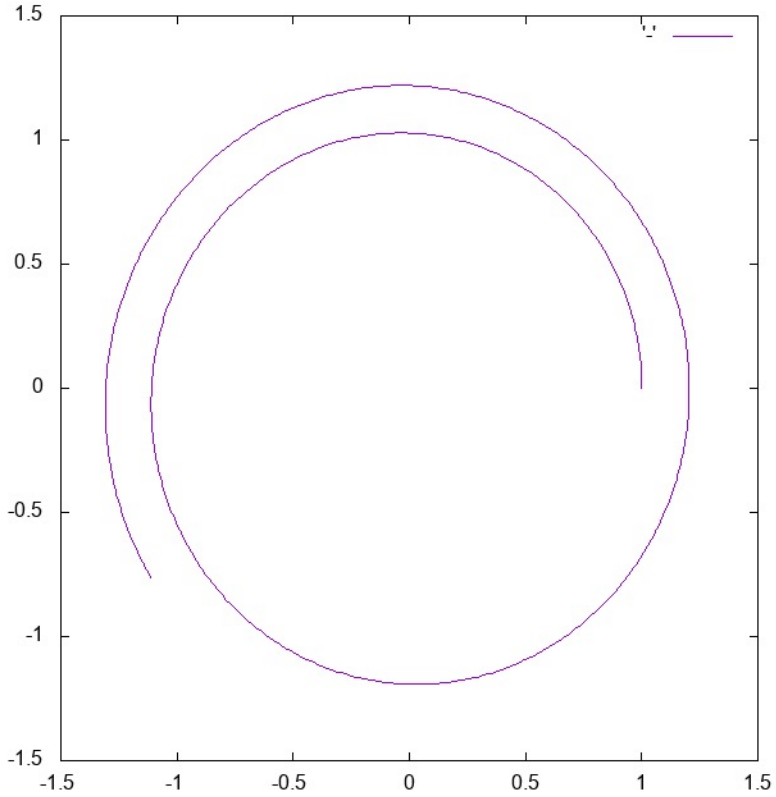
Parameters for orbit problem

- We'll use units of solar masses, and Astronomical Units (AU) for distance
  - In these units, M_{sun}=1 and G = 39.47 $AU^3 M_{sun}^{-1} yr^{-2}$

- Initial conditions:
  - At t = 0 we'll place the body along the x-axis at a distance of 1 AU from the sun and give it the Earth's velocity in the y-direction:
    - $x(0)$ = 1, $y(0)$ =0
    - v_y (0) = 6.283185 AU/yr
  - We will try a time step of 1 day: Δt = 1/365 yr

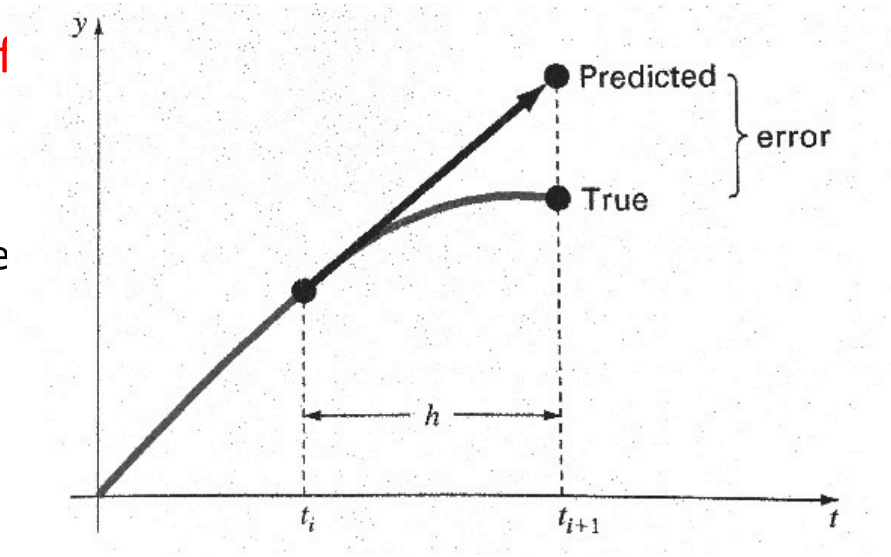
Example program for Euler orbit problem

- See **`euler_orbit.f08`**

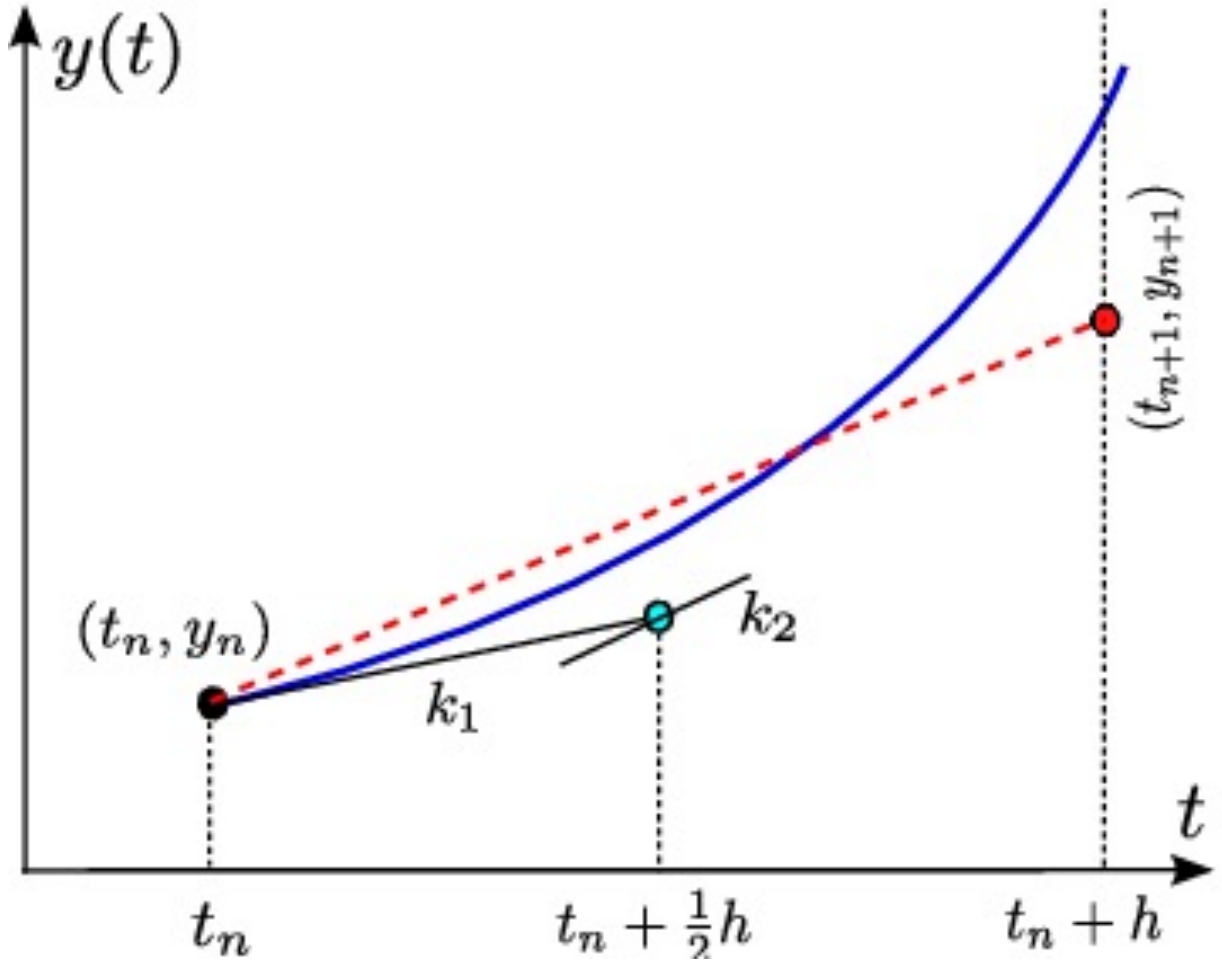


More accurate ODE numerical methods

- The problem with Euler's method is that the right-hand-side of the equations is evaluated at the beginning of the timestep
- The right-hand-side usually changes over the course of each timestep and we may be getting an inaccurate answer as a result
  - It would be better if we could evaluate the right-hand-side in the middle of the timestep.
  - However, we can't do that unless we know the solution in advance
- We could use higher-order finite differences, however this is not a common approach
- **Strategy:** Use Euler's method to estimate the solution at the midpoint of the timestep. And then use this estimate to evaluate the right-hand-side
- This is called a second order Runge-Kutta method



Second-order Runge-Kutta method



Fadlisyah, Muhammad thesis (2014)

Aside: Notation for coupled systems of ordinary differential equations

- The equations we were solving with Euler's method were of the form:

$$\frac{dy_1}{dt} = f_1(y_1, y_2, \ldots, y_N, t)$$
$$\frac{dy_2}{dt} = f_2(y_1, y_2, \ldots, y_N, t)$$
$$\vdots$$
$$\frac{dy_N}{dt} = f_N(y_1, y_2, \ldots, y_N, t)$$

- This is a set of coupled first-order ordinary differential equations (ODEs)

Aside: Euler's Method for Coupled Systems of ODEs

- Use shorthand notation for the time at the *n*th step: t^n, and denote $y_i(t^n)$ as y_i^n
- Then approximate the derivatives are written:

$$\frac{dy_i}{dt} \simeq \frac{y_i^{n+1} - y_i^n}{\Delta t}$$

- And Euler's method for a set of coupled ODEs is:

$$\begin{aligned}
y_1^{n+1} &= y_1^n + \Delta t f_1(y_1, y_2, \ldots, y_N, t) \\
y_2^{n+1} &= y_2^n + \Delta t f_2(y_1, y_2, \ldots, y_N, t) \\
&\vdots \\
y_N^{n+1} &= y_N^n + \Delta t f_N(y_1, y_2, \ldots, y_N, t)
\end{aligned}$$

Aside: Coupled systems of ODEs in vector notation

- In order to simplify the description of the second order Runge-Kutta algorithm we use the following vector notation to simplify the equations:

$$\mathbf{y} \equiv (y_1, y_2, y_3, \ldots, y_N)$$
$$\mathbf{f} \equiv (f_1, f_2, f_3, \ldots, f_N)$$

- Using this notation, the original set of ODEs is:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t)$$

- In this notation Euler's method is:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t \mathbf{f}(\mathbf{y}^n, t^n)$$

Second-order Runge-Kutta method

- Taylor expand around $t + 1/2\,\Delta t$:

$$y(t+\Delta t) = y(t+\frac{1}{2}\Delta t) + \frac{1}{2}\Delta t \left.\frac{dy}{dt}\right|_{t+\frac{1}{2}\Delta t} + \frac{1}{8}\Delta t^2 \left.\frac{d^2y}{dt^2}\right|_{t+\frac{1}{2}\Delta t} + \mathcal{O}(\Delta t^3)$$

$$y(t) = y(t+\frac{1}{2}\Delta t) - \frac{1}{2}\Delta t \left.\frac{dy}{dt}\right|_{t+\frac{1}{2}\Delta t} + \frac{1}{8}\Delta t^2 \left.\frac{d^2y}{dt^2}\right|_{t+\frac{1}{2}\Delta t} - \mathcal{O}(\Delta t^3)$$

- Subtract the two expressions

$$y(t+\Delta t) = y(t) + \Delta t \left.\frac{dy}{dt}\right|_{t+\frac{1}{2}\Delta t} + \mathcal{O}(\Delta t^3)$$

Need *f* evaluated at midpoint

$$= y(t) + \Delta t f(y(t+\frac{1}{2}\Delta t), t+\frac{1}{2}\Delta t) + \mathcal{O}(\Delta t^3)$$

Second-order Runge-Kutta method

- **Step 1:** Estimate change due of the right-hand side using Euler's method:

$$\mathbf{k}_1 = \Delta t \mathbf{f}(\mathbf{y}^n, t^n)$$

- **Step 2:** Use estimate to predict value of solution at midpoint of the timestep. Evaluate right hand side at midpoint:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t \mathbf{f}(\mathbf{y}^n + \frac{1}{2}\mathbf{k}_1, t^n + \frac{1}{2}\Delta t)$$

- See **`rk2_orbit.f08`**

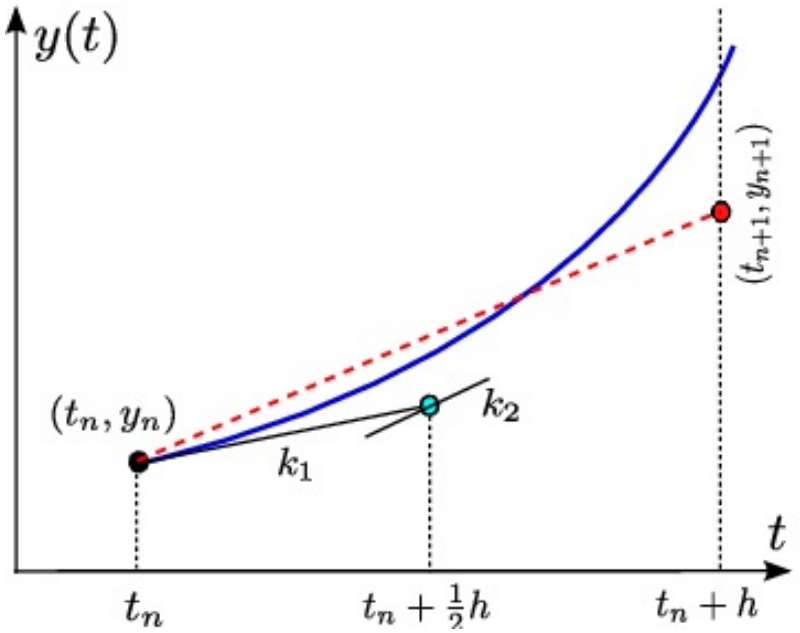
The fourth-order Runge-Kutta method

- In practice, the workhorse algorithm for first-order sets of ODEs is the fourth-order Runge-Kutta algorithm which (we state here without derivation)

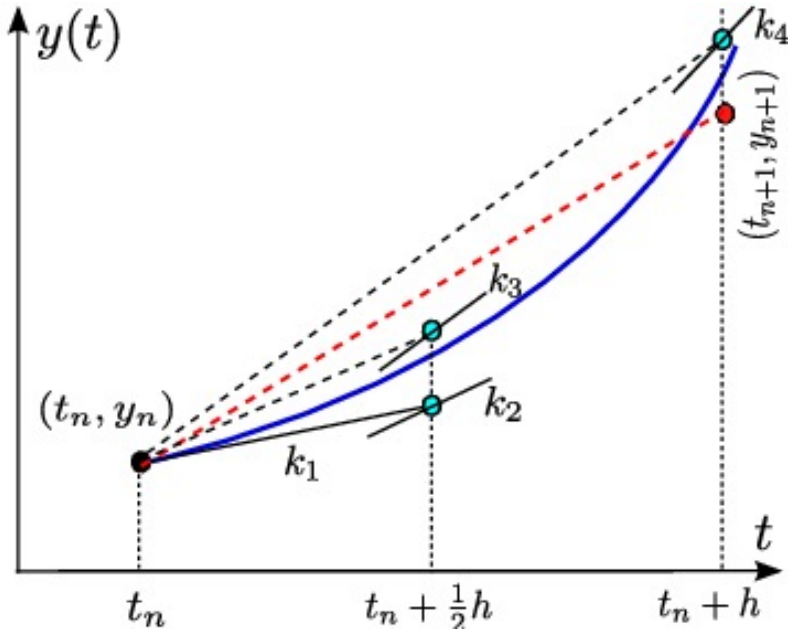
- Step 1: $$\mathbf{k}_1 = \Delta t\ \mathbf{f}(\mathbf{y}^n, t^n)$$
- Step 2: $$\mathbf{k}_2 = \Delta t\ \mathbf{f}(\mathbf{y}^n + \frac{1}{2}\mathbf{k}_1, t^n + \frac{1}{2}\Delta t)$$
- Step 3: $$\mathbf{k}_3 = \Delta t\ \mathbf{f}(\mathbf{y}^n + \frac{1}{2}\mathbf{k}_2, t^n + \frac{1}{2}\Delta t)$$
- Step 4: $$\mathbf{k}_4 = \Delta t\ \mathbf{f}(\mathbf{y}^n + \mathbf{k}_3, t^n + \Delta t)$$
- Step 5: $$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)$$

Second and fourth-order Runge-Kutta methods

2nd order



4th order



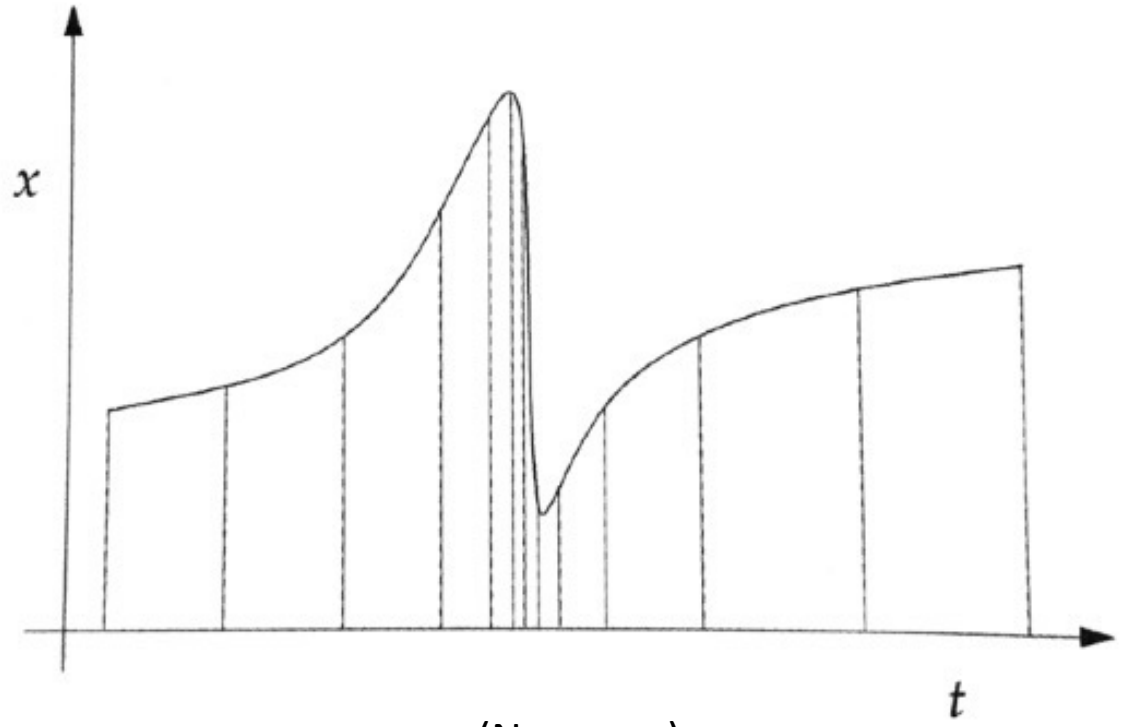
Fadlisyah, Muhammad thesis (2014)

Runge-Kutta methods

- Euler method can be though of as the first-order RK method
  - Accurate to first order in Δt, i.e., error is order Δt^2
- Second-order RK method accurate to Δt^2, so error Δt^3
- Fourth-order RK method accurate to Δt^4, so error Δt^5
  - By far the most common method for the numerical solution of ODEs
  - Balances accuracy and complexity
- Quoted accuracies are for one step, errors accumulate over the number of steps needed in the calculation, usually loose an order of accuracy (see Newman)

Adaptive step size

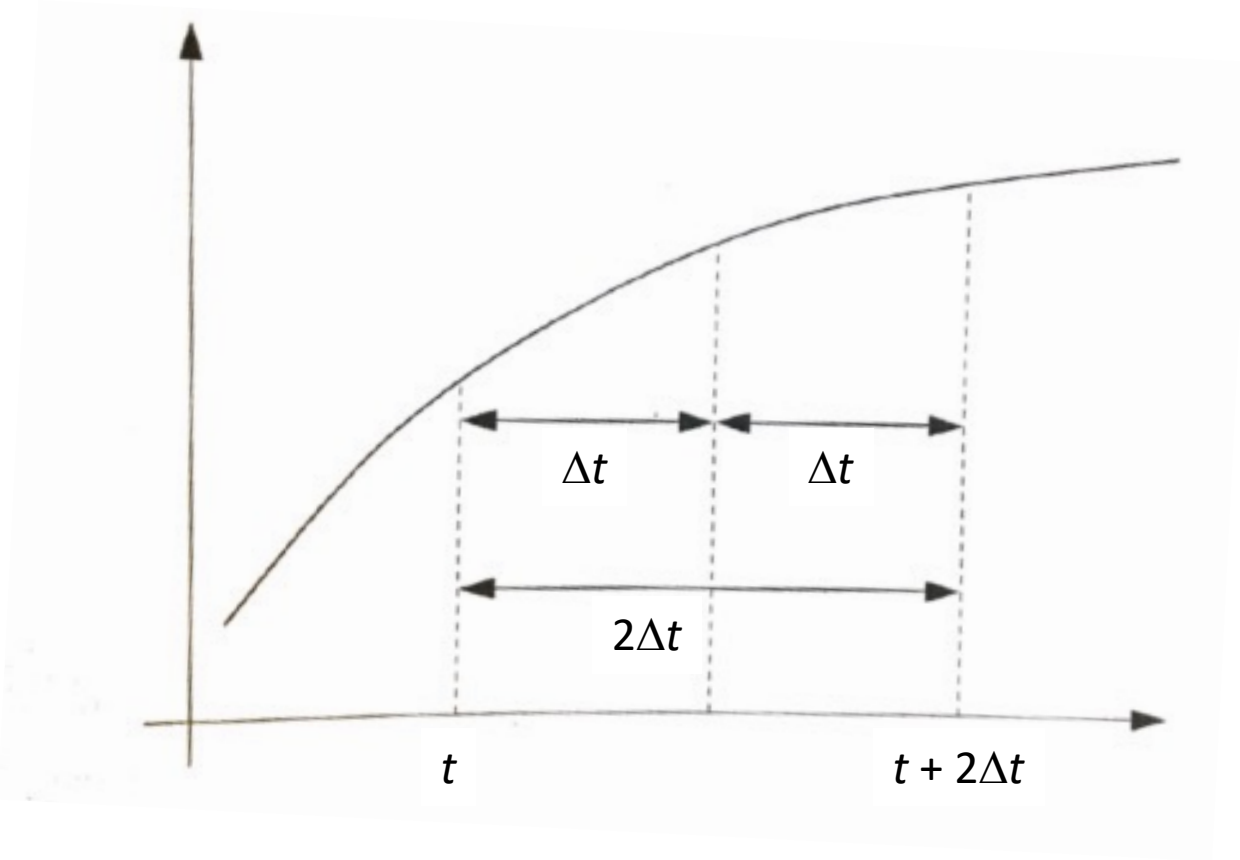
- So far, we have set by hand a constan step size Δt
- Often, we can get better results by varying the step size
  - Increase in regions where function varies rapidly, decrease where it varies slowly
- Approach: vary Δt so the error introduced per unit interval is roughly constant
  - First we need to estimate the error in the steps



(Newman)

Adaptive step size: Estimating the error

- 1. Choose initial (small) Δt
- 2. Use RK method to do two Δt steps of the solution
- 3. Go back to initial t and do an RK step with $2\Delta t$
- 4. Compare the results to estimate the error



Adaptive step size: Estimating the error

- True value of function related to estimate $y_{\Delta t}$:

$$y(t + 2\Delta t) = y_{\Delta t} + 2c\Delta t^5$$

- For doubled step size $y_{2\Delta t}$:

$$y(t + 2\Delta t) = y_{2\Delta t} + 32c\Delta t^5$$

- So per step error is:

$$\epsilon = c\Delta t^5 = \frac{1}{30}(y_{\Delta t} - y_{2\Delta t})$$

- Take δ to be the target accuracy per step. Then the step size necessary to get that accuracy is:

$$\Delta t' = \Delta t \sqrt[4]{\frac{30\Delta t\delta}{|y_{\Delta t} - y_{2\Delta t}|}}$$

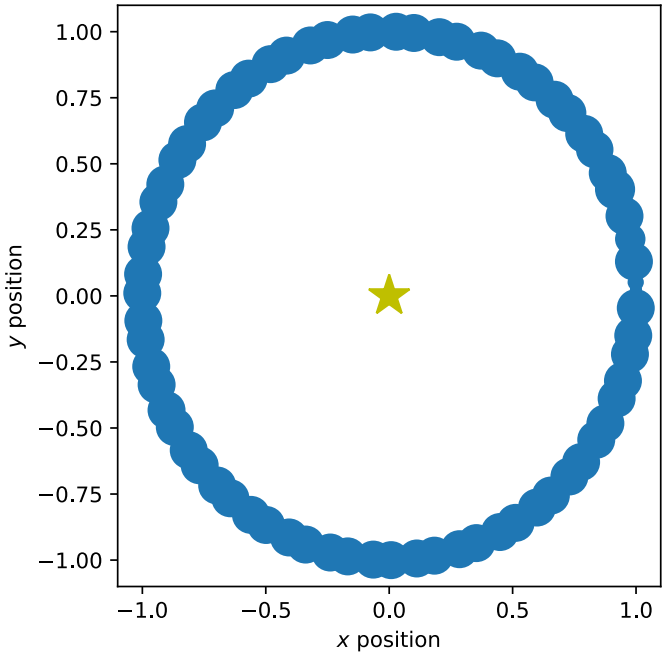
Adaptive step size: Complete approach

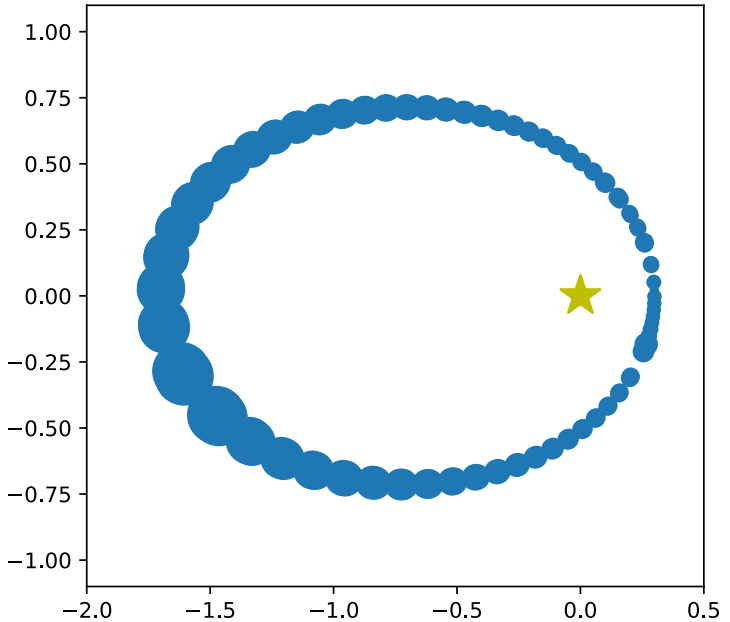
- 1. Choose initial (small) Δt
- 2. Use RK method to do two Δt steps of the solution
- 3. Go back to initial t and do an RK step with $2\Delta t$
- 4. Compare the results to estimate the error
- 5. Calculate ideal step size $\Delta t'$
  - If $\varepsilon > \delta$, then redo the calculation with $\Delta t'$
  - If $\varepsilon < \delta$, take the results obtained using Δt and move on to time t + Δt. In the next iteration use $\Delta t'$ as the timestep

- Requires at least 3 RK steps for every two actually used, but usually results in an overall speedup for a given accuracy
- Usually limit how much $\Delta t'$ can differ from Δt (e.g., by less than a factor of two) in case the denominator happens to diverge

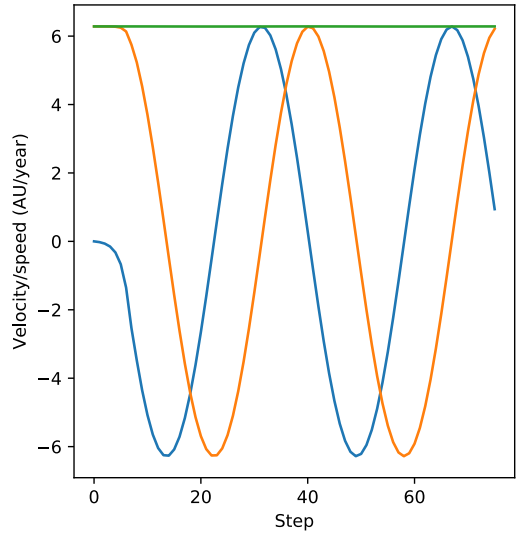
Example: Elliptical orbit with adaptive 4th-order RK

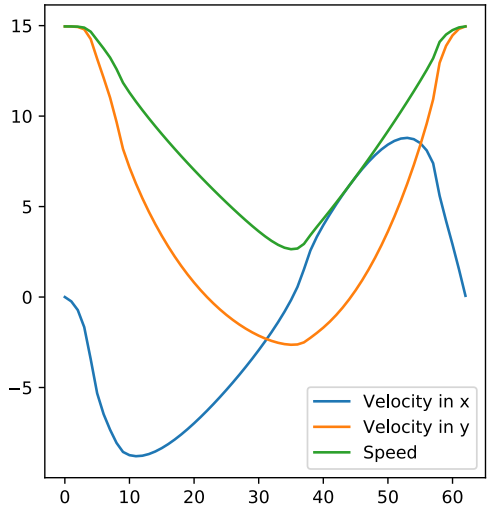
Circular:
x_0= 1 AU
v_{y0} =6.283185 AU/year





Elliptical:
x_0= 0.3 AU
v_{y0} =14.955378 AU/year





After class tasks

- Homework 1 due Sept. 16 by 11pm
  - Let me know if you have HW questions or questions/issues on github classroom
  - Office hours: Mondays, 3:00pm to 4:00pm; Thursdays, 11:05am to 1:00pm
    - Feel free to send me an email, and remember, if you push your changes, I should be able to see them
- Readings:
  - Newman Ch. 8
  - Wikipedia page on root finding