

# Computational Environments and Toolchains

## Topic 02 — Scientific Computing using Python

---

### Lecture 03 — scipy

Kieran Murphy and David Power

Department of Science,  
Waterford IT.  
(kmurphy@wit.ie,dpower@wit.ie)

Autumn Semester, 2021

#### Outline

- matplotlib (2D and 3D plotting library)
- numpy (high performance matrix library)
- scipy (scientific computation library)

# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# What is SciPy?

## SciPy

is a library of algorithms and mathematical tools built to work with NumPy arrays..

- Special functions ([scipy.special](#))
- Integration ([scipy.integrate](#))
- Optimization ([scipy.optimize](#))
- Interpolation ([scipy.interpolate](#))
- Fourier Transforms ([scipy.fftpack](#))
- Signal Processing ([scipy.signal](#))
- Linear Algebra ([scipy.linalg](#))
- Sparse Eigenvalue Problems with ARPACK
- Compressed Sparse Graph Routines ([scipy.sparse.csgraph](#))
- Spatial data structures and algorithms ([scipy.spatial](#))
- Statistics ([scipy.stats](#))
- Multidimensional image processing ([scipy.ndimage](#))
- File IO ([scipy.io](#))
- Weave ([scipy.weave](#))

# Importing

- The SciPy docs recommend loading the various modules as

scipy.py

```
5 import numpy as np
6 import scipy as sp
7 import matplotlib as mpl
8 import matplotlib.pyplot as plt
```

- Each subpackage must be imported separately

scipy.py

```
10 from scipy import linalg, optimize
```

- In addition to the `help()`, there is a `sp.info()` function that outputs things without a pager
- Python also has a `dir()` function that lists the names a module defines

# Health Warning

Every numerical method has its own  
strengths, weaknesses, and assumptions

- You should research a bit to understand what these methods are doing under the hood.
- Many of the SciPy methods are wrappers to well-tested implementations of algorithms that were developed many years (decades even) ago.
- But if your problem does not satisfy the assumptions that the algorithm or its implementation assumed then all bets are off regarding the reliability of the generated results.

# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# Numerical Integration

We want to compute:

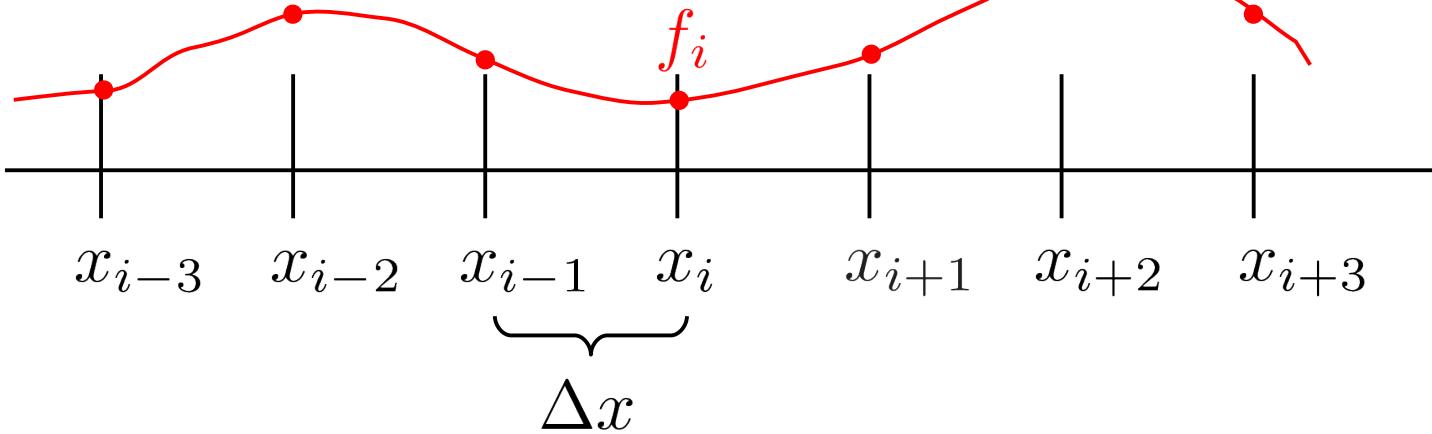
$$I = \int_a^b f(x) \, dx \quad (1)$$

We can imagine 2 situations

- ➊ Our function,  $f(x)$  is defined only at a set of (possibly regularly spaced) points.
  - Generally speaking, asking for greater accuracy involves using more of the discrete points in the approximation for  $I$ .
- ➋ We have an analytic expression for  $f(x)$ 
  - We have the freedom to pick our integration points, and this can allow us to optimise the calculation of  $I$
  - Any numerical integration method that represents the integral as a (weighted) sum at a discrete number of points is called a **quadrature rule**.
  - Fixed spacing between points: **Newton-Cotes quadrature**

# Gridded Data

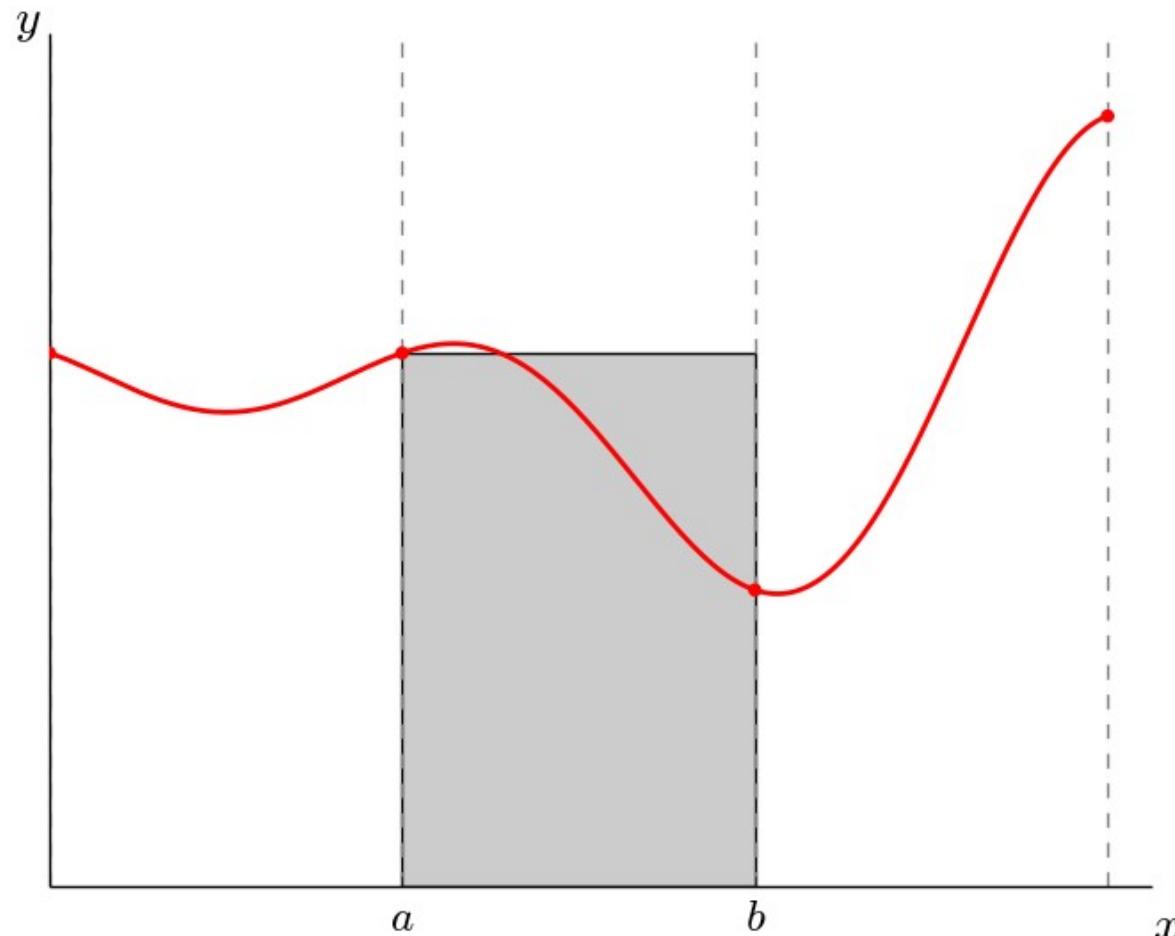
- Discretized data is represented at a finite number of locations
  - Integer subscripts are used to denote the position (index) on the grid
  - Structured/regular: spacing is constant



- Data is known only at the grid points:  $f_i = f(x_i)$

# Rectangular Rule

- Simplest case: piecewise constant interpolant ([rectangle rule](#))

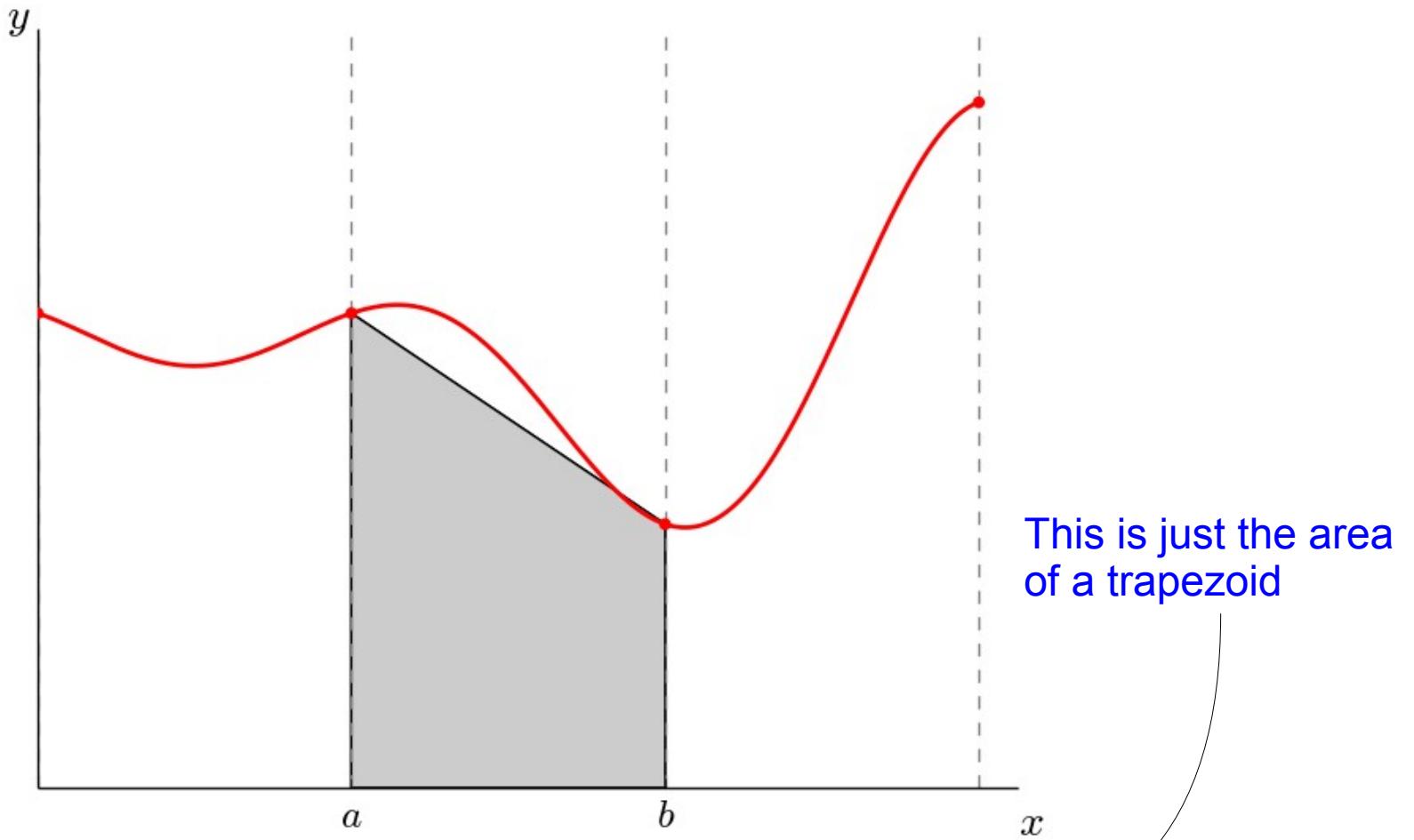


$$I \equiv \int_a^b f(x)dx \approx \Delta x f(a)$$

— © 2014 C. J. Ong — Numerical Methods for Engineers —

# Trapezoidal Rule

- One step up: piecewise linear interpolant (**trapezoid rule**)



$$I \equiv \int_a^b f(x)dx \approx \Delta x \frac{f(a) + f(b)}{2}$$

# Simpson's Rule

- Piecewise quadratic interpolant (**Simpson's rule**)

- 3 unknowns and 3 points

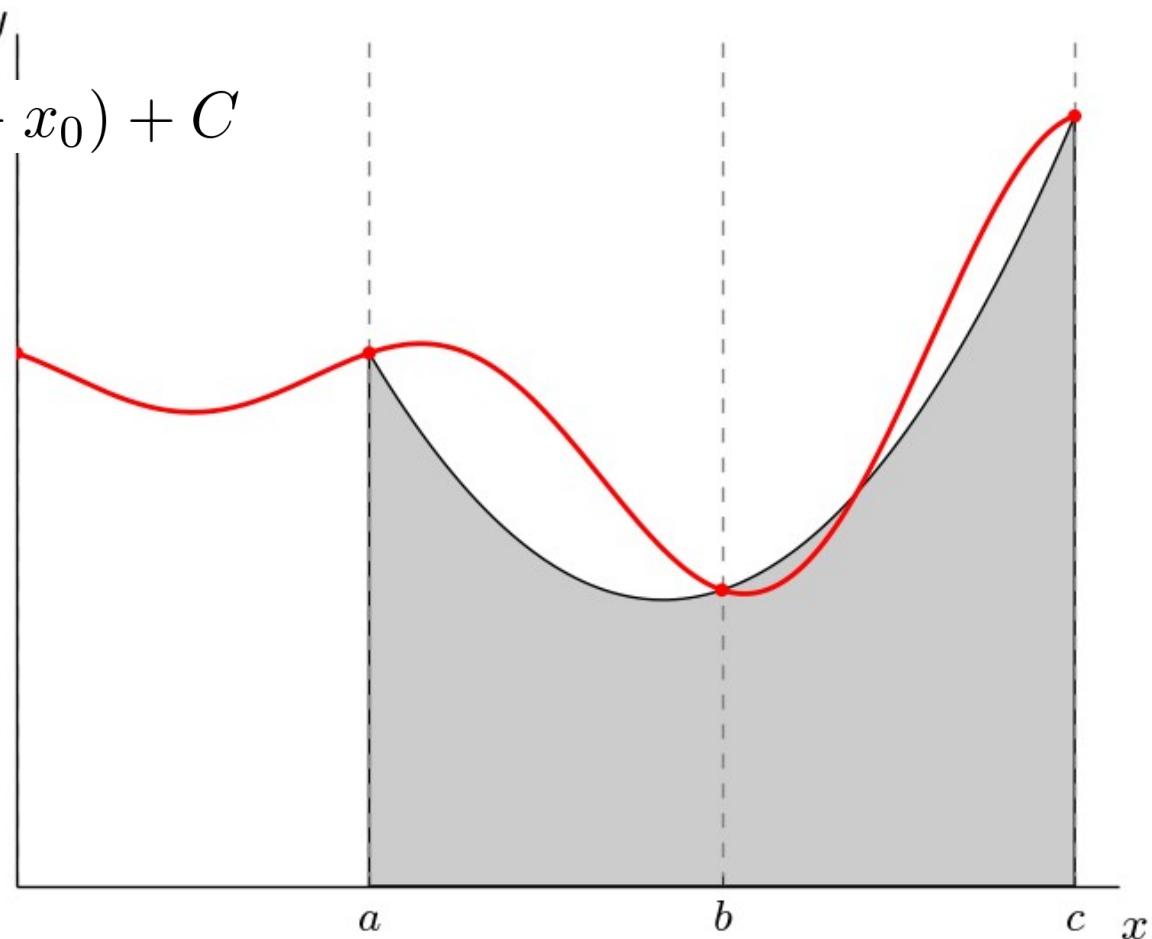
$$f(x) = A(x - x_0)^2 + B(x - x_0) + C$$

- Solving:**

$$A = \frac{f_a - 2f_b + f_c}{2\delta^2}$$

$$B = -\frac{f_c - 4f_b + 3f_a}{2\delta}$$

$$C = f_a$$

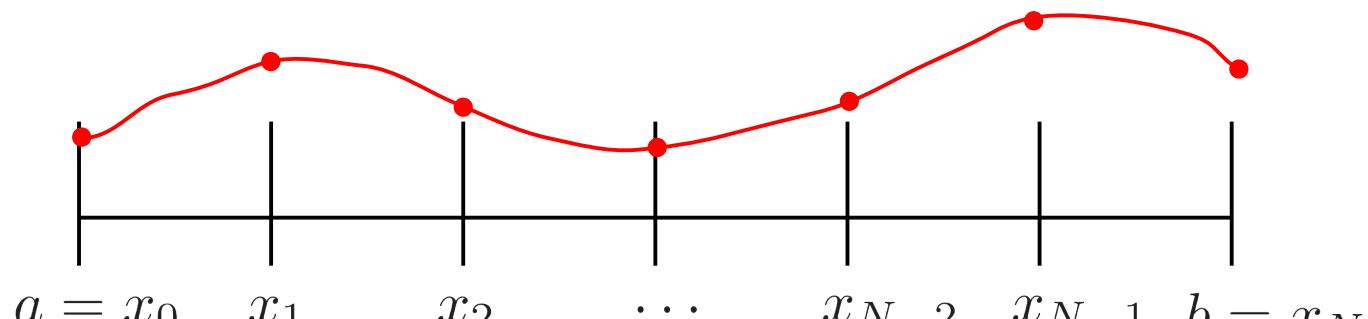


$$I = \int_{x_0}^{x_2} [A(x - x_0)^2 + B(x - x_0) + C] dx = \frac{c - a}{6} (f_0 + 4f_1 + f_2)$$

© 2014 Python for Scientific Computing, by Jake VanderPlas

# Compound Integration

- Break interval into chunks



$$I \equiv \int_a^b f(x)dx = \sum_{i=0}^{N-1} \underbrace{\int_{x_i}^{x_{i+1}} f(x)dx}_{\text{Integral over a single slab}}$$

Integral over a  
single slab

# Compound Integration

- Compound Trapezoidal

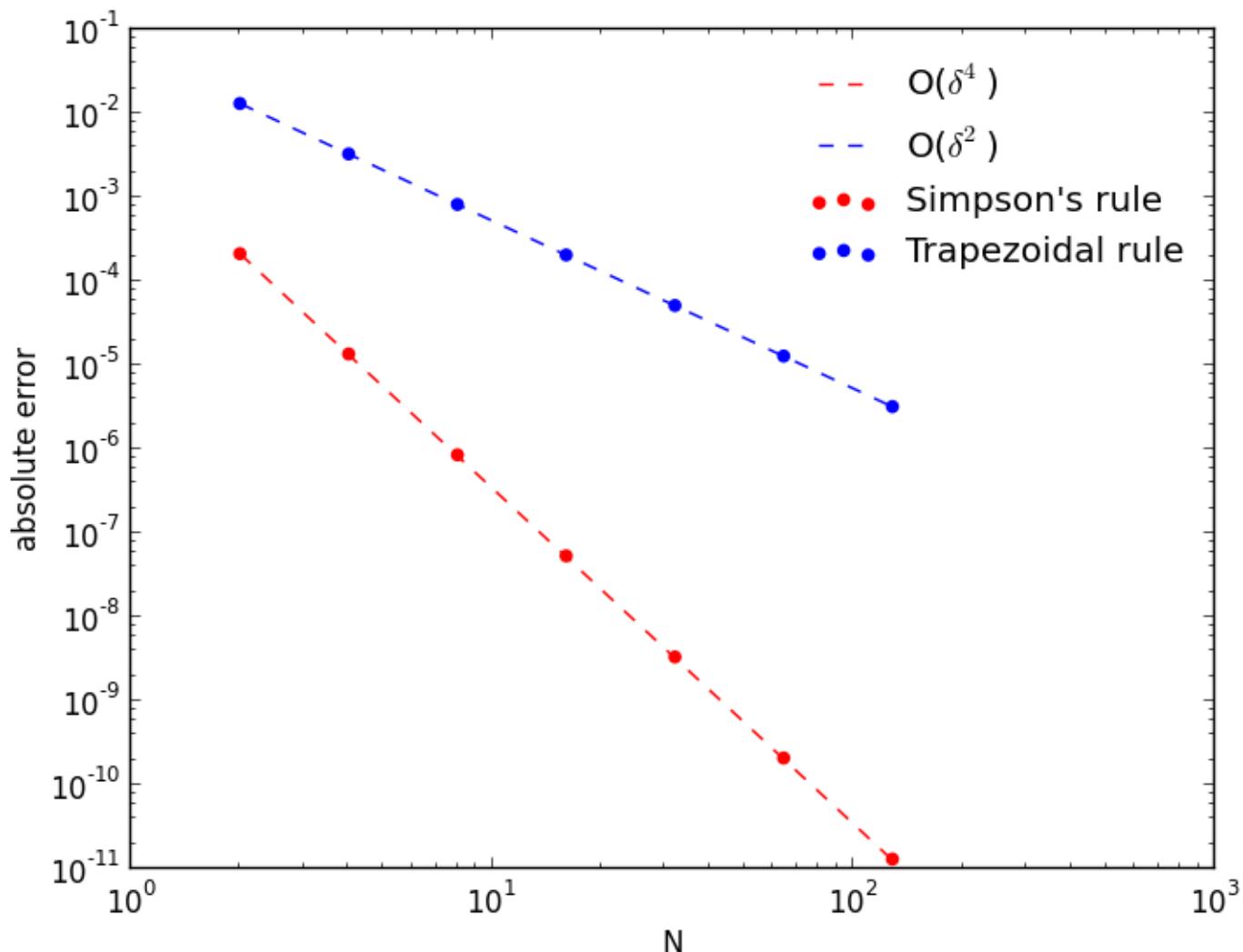
$$\int_a^b f(x)dx = \frac{\Delta x}{2} \sum_{i=0}^{N-1} (f_i + f_{i+1}) + \mathcal{O}(\Delta x^2)$$

- Compound Simpson's

- Integrate pairs of slabs together (requires even number of slabs)

$$\int_a^b f(x)dx = \frac{\Delta x}{3} \sum_{i=0}^{N/2-1} (f_{2i} + 4f_{2i+1} + f_{2i+2}) + \mathcal{O}(\Delta x^4)$$

# Compound Integration



$$\int_0^1 e^{-x} dx$$

Always a good idea to check the convergence rate!

# Gaussian Quadrature

---

- Instead of fixed spacing, what if we strategically pick the spacings?
  - We want to express

$$\int_a^b f(x)dx \approx w_1 f(x_1) + \dots w_N f(x_N)$$

- w's are weights. We will choose the location of points  $x_i$

# Gaussian Quadrature

- Gaussian quadrature: fundamental theorem
  - $q(x)$  is a polynomial of degree  $N$ , such that

$$\int_a^b q(x)\rho(x)x^k dx = 0$$

- $k = 0, \dots, N-1$  and  $\rho(x)$  is a specified weight function.
- Choose  $x_1, x_2, \dots, x_N$  as the roots of the polynomial  $q(x)$
- We can write

$$\int_a^b f(x)\rho(x)dx \approx w_1 f(x_1) + \dots w_N f(x_N)$$

and there will be a set of  $w$ 's for which the integral is exact if  $f(x)$  is a polynomial of degree  $< 2N$ !

# Gaussian Quadrature

- Many quadratures exist:

Interval	$\omega(x)$	Orthogonal polynomials	A & S	For more information, see ...
$[-1, 1]$	1	Legendre polynomials	25.4.29	Section <a href="#">Gauss–Legendre quadrature</a> , above
$(-1, 1)$	$(1 - x)^\alpha(1 + x)^\beta, \quad \alpha, \beta > -1$	Jacobi polynomials	25.4.33 ( $\beta = 0$ )	<a href="#">Gauss–Jacobi quadrature</a>
$(-1, 1)$	$\frac{1}{\sqrt{1 - x^2}}$	Chebyshev polynomials (first kind)	25.4.38	<a href="#">Chebyshev–Gauss quadrature</a>
$[-1, 1]$	$\sqrt{1 - x^2}$	Chebyshev polynomials (second kind)	25.4.40	<a href="#">Chebyshev–Gauss quadrature</a>
$[0, \infty)$	$e^{-x}$	Laguerre polynomials	25.4.45	<a href="#">Gauss–Laguerre quadrature</a>
$[0, \infty)$	$x^\alpha e^{-x}$	Generalized Laguerre polynomials		<a href="#">Gauss–Laguerre quadrature</a>
$(-\infty, \infty)$	$e^{-x^2}$	Hermite polynomials	25.4.46	<a href="#">Gauss–Hermite quadrature</a>

(Wikipedia)

- In practice, the roots and weights are tabulated for these out to many numbers of points, so there is no need to compute them.

# Gaussian Quadrature

- Example:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2} dy$$

erf(1) (exact):	0.84270079295
3-point trapezoidal:	0.825262955597 -0.017437837353
3-point Simpson's:	0.843102830043 0.000402037093266
3-point Gauss-Legendre:	0.842690018485 -1.0774465204e-05

Notice how well the Gauss-Legendre does for this integral.

# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

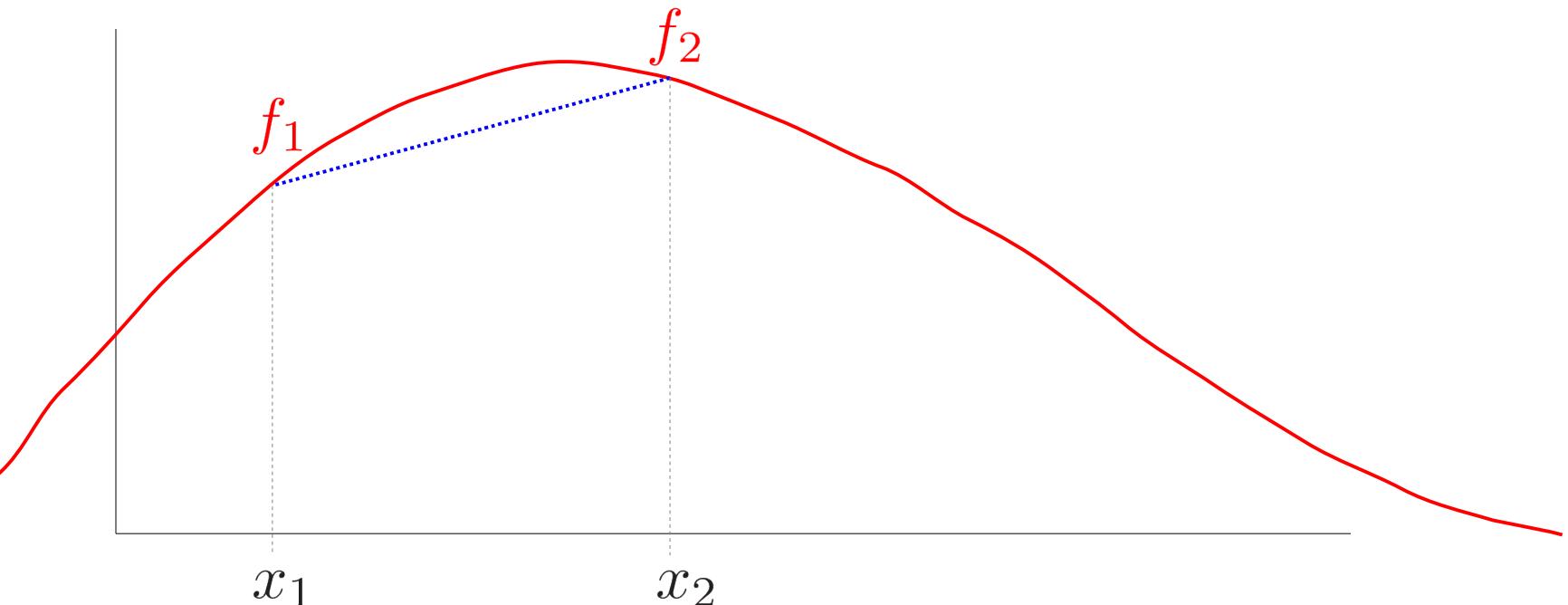
# Interpolation

---

- We frequently have data only at a discrete number of points
  - Interpolation fills in the gaps by making an assumption about the behavior of the functional form of the data
- Many different types of interpolation exist
  - Some ensure no new extrema are introduced
  - Some match derivatives at end points
  - ...
- Generally speaking: larger number of points used to build the interpolant, the higher the accuracy in a local region
  - Pathological cases exist
  - You may want to enforce some other property on the form of the interpolant

# Linear Interpolation

- Simplest idea—draw a line between two points



$$f(x) = \frac{f_2 - f_1}{x_2 - x_1}(x - x_1) + f_1$$

- Exactly recovers the function values at the end points

# Lagrange Interpolation

- General method for building a single polynomial that goes through all the points (alternate formulations exist)
- Given  $n$  points:  $x_0, x_1, \dots, x_{n-1}$ , with associated function values:  $f_0, f_1, \dots, f_{n-1}$ 
  - construct basis functions:

$$l_i(x) = \prod_{j=0, i \neq j}^{n-1} \frac{x - x_j}{x_i - x_j}$$

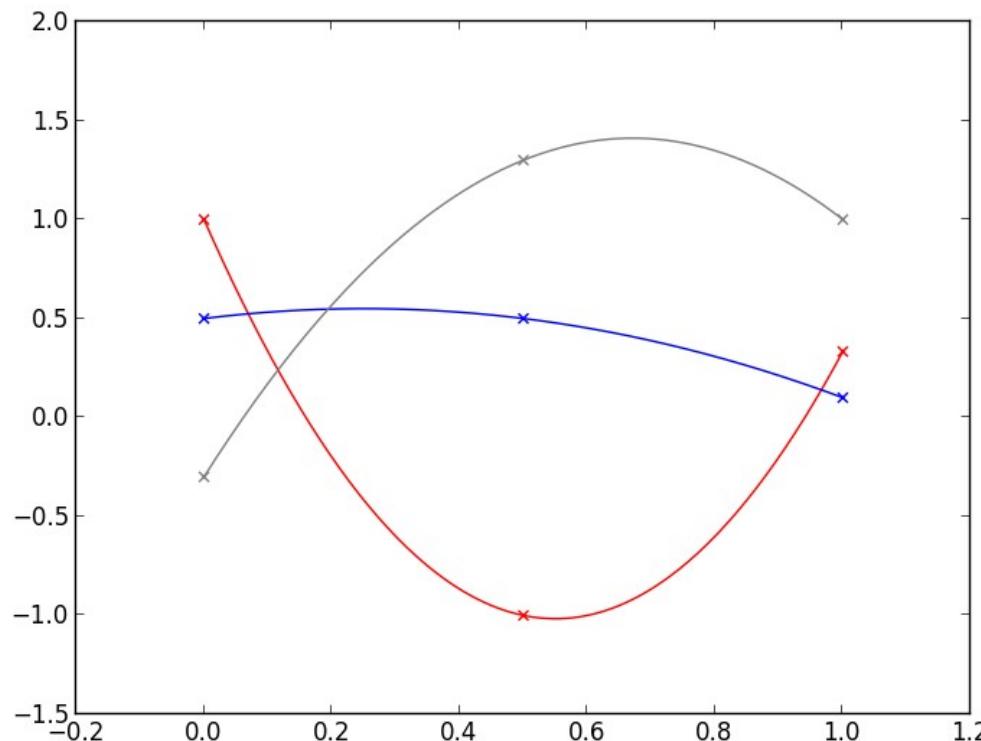
- Basis function  $l_i$  is 0 at all  $x_j$  except for  $x_i$  (where it is 1)
- Function value at  $x$  is:

$$f(x) = \sum_{i=0}^{n-1} l_i f_i$$

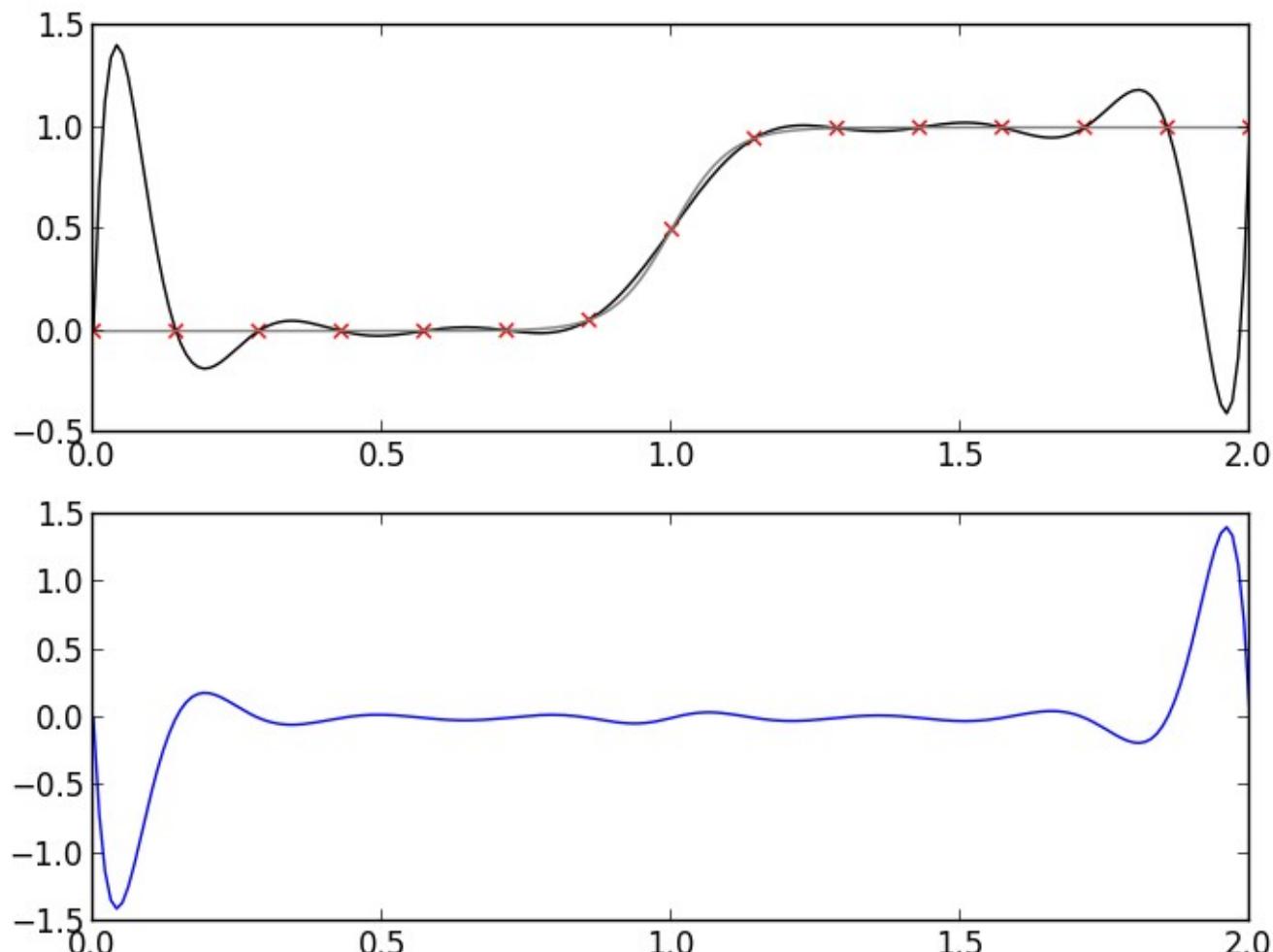
# Lagrange Interpolation

- Quadratic Lagrange polynomial:

$$f(x) = \frac{(x - x_1)(x - x_2)}{2\Delta x^2} f_0 - \frac{(x - x_0)(x - x_2)}{\Delta x^2} f_1 + \frac{(x - x_0)(x - x_1)}{2\Delta x^2} f_2$$



# Lagrange Interpolation



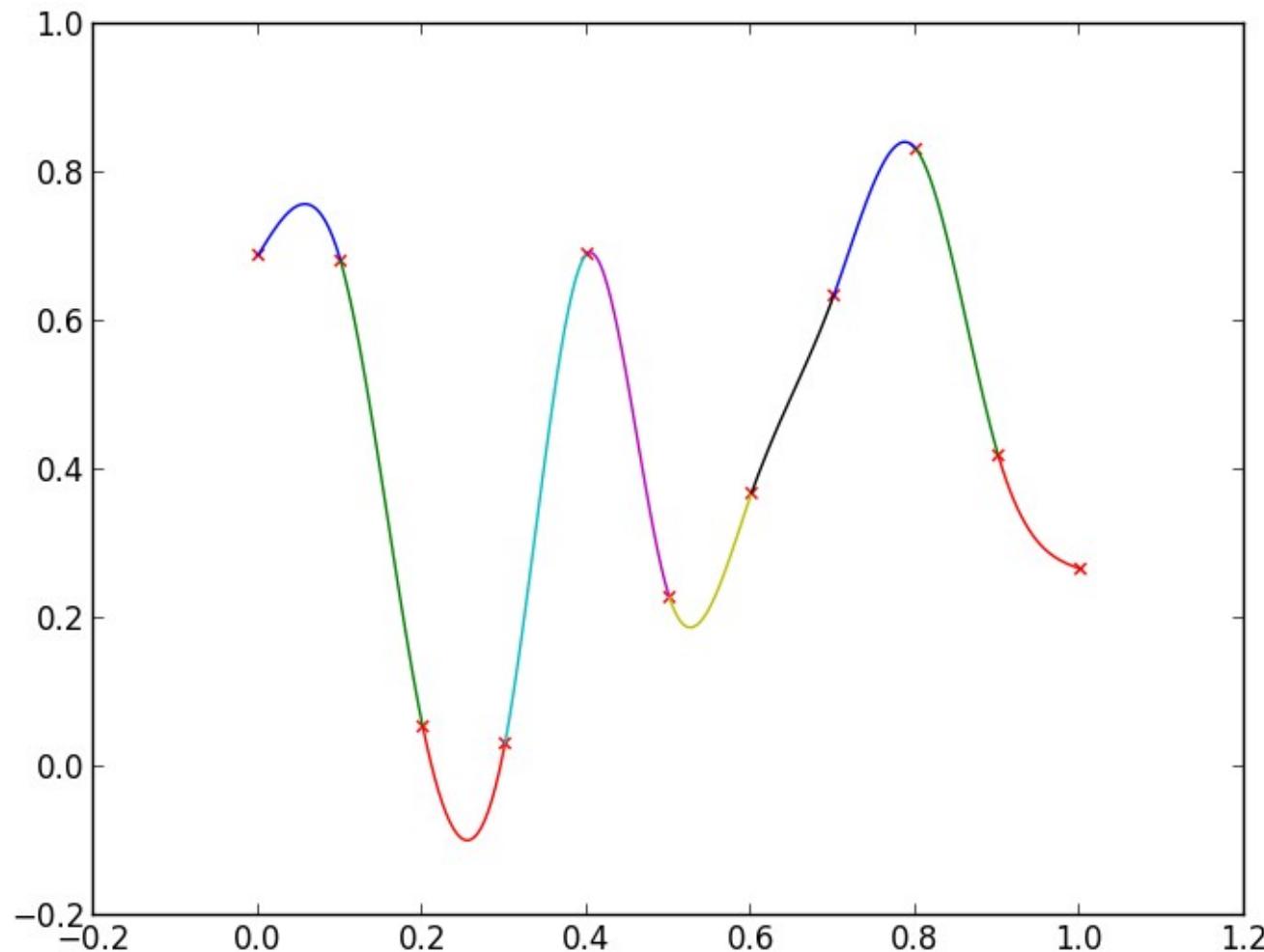
High-order is not always better: Interpolation through 15 points sampled uniformly from  $\tanh()$ . The error is shown below.

# Splines

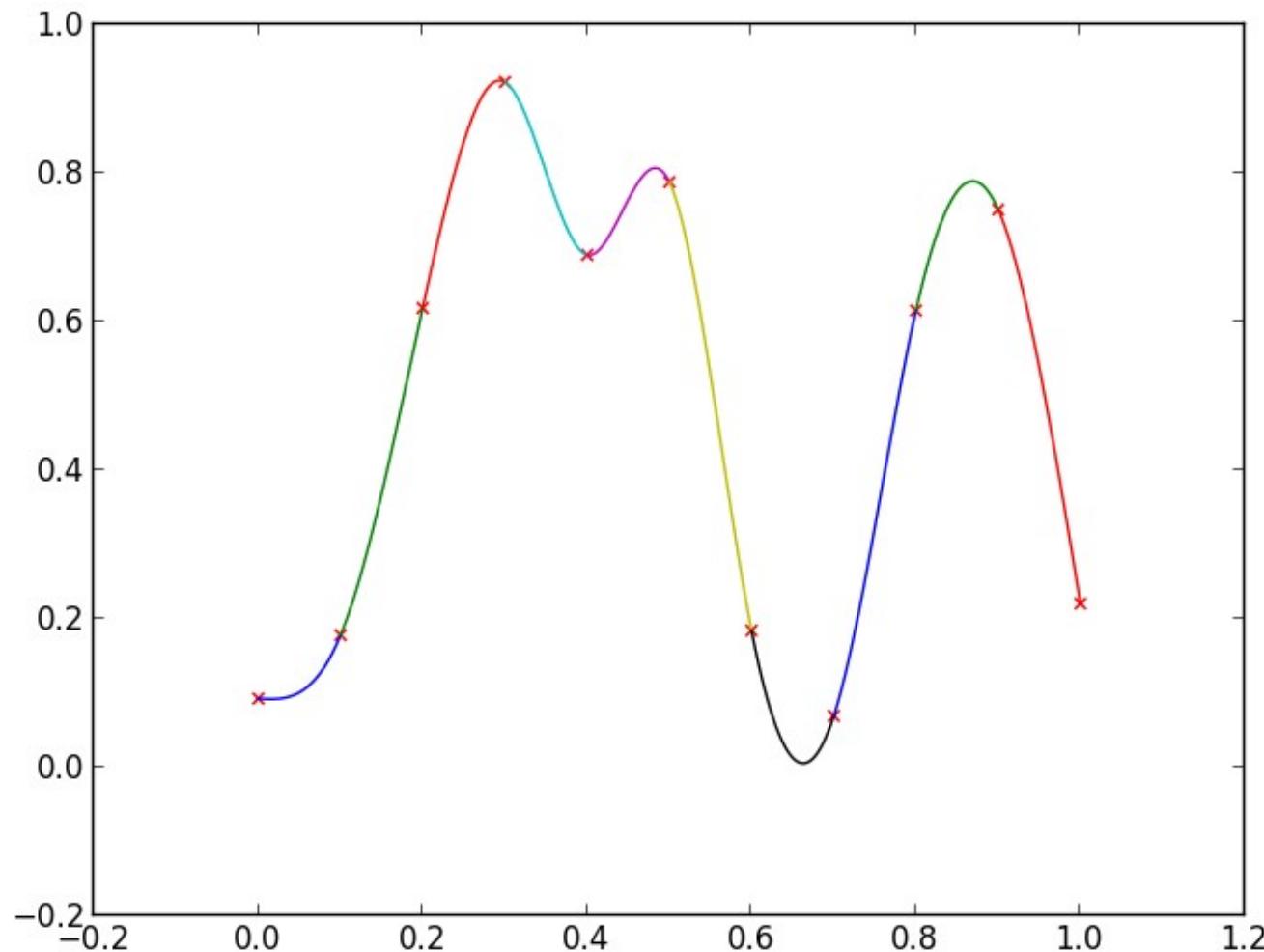
---

- So far, we've only worried about going through the specified points
- Large number of points → two distinct options:
  - Use a single high-order polynomial that passes through them all
  - Fit a (somewhat) high order polynomial to *each interval* and match all derivatives at each point—this is a spline
- Splines match the derivatives at end points of intervals
  - Piecewise splines can give a high-degree of accuracy
- Cubic spline is the most popular
  - Matches first and second derivative at each data point
  - Results in a smooth appearance
  - Avoids severe oscillations of higher-order polynomial

# Cubic Splines



# Cubic Splines



Note that the splines can overshoot the original data values

# Cubic Splines

---

- Note: cubic splines are not necessarily the most accurate interpolation scheme (and sometimes far from...)
- But, for plotting/graphics applications, they look right

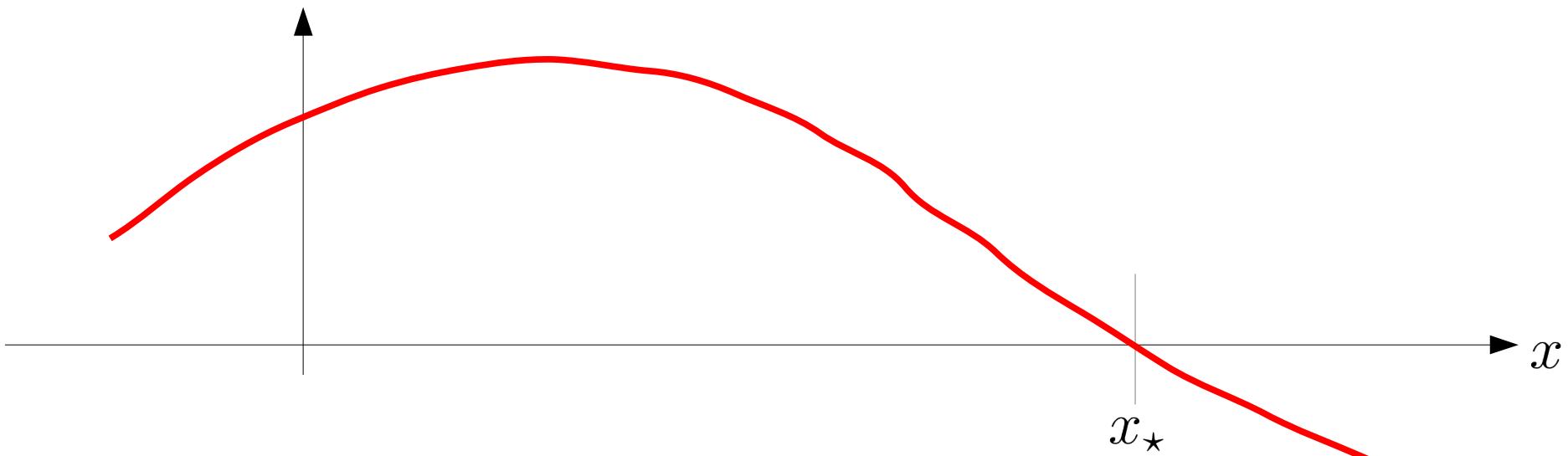
# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# Root Finding

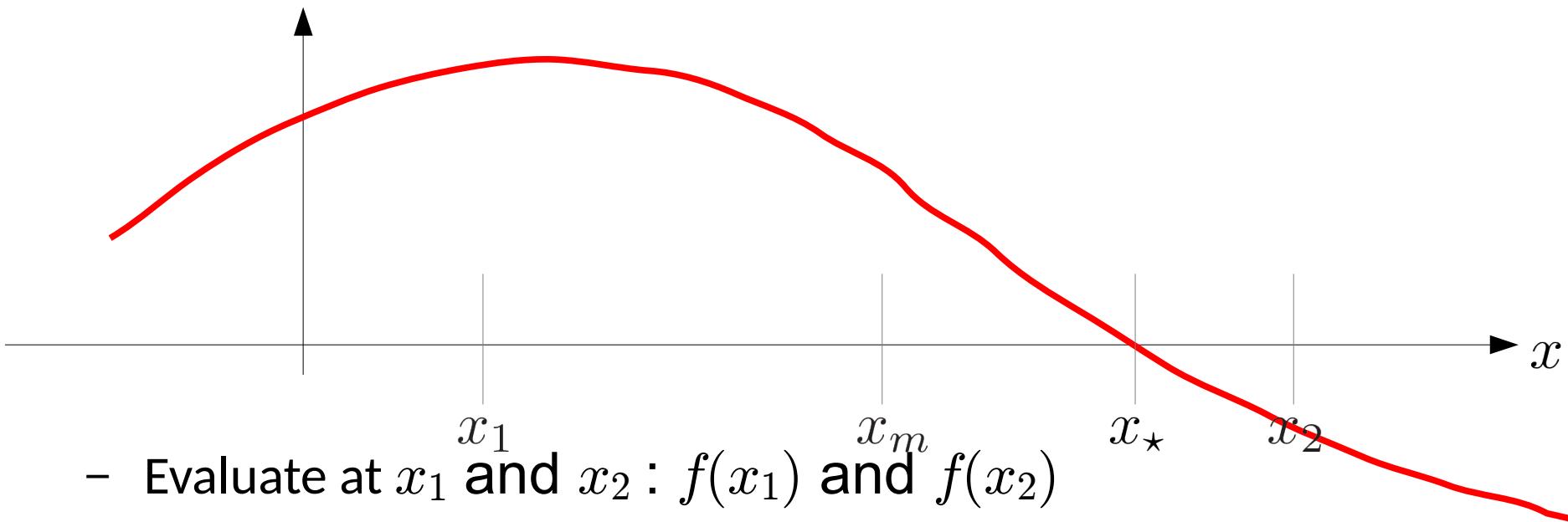
- Basic methods can be understood by looking at the function graphically



- Function  $f(x)$  has a zero at  $x_*$
- Note the sign of  $f(x)$  changes at the root

# Bisection

- Simplest method: **bisection**



- Evaluate at  $x_1$  and  $x_2$ :  $f(x_1)$  and  $f(x_2)$
- If these are different signs, then the root lies between them
- Evaluate at the midpoint:  $x_m = (x_1 + x_2)/2$  getting  $f(x_m)$
- The root lies in one of the two intervals—repeat the process

# Newton-Raphson

- If we know  $df/dx$  we can do better
  - Start with an initial guess,  $x_0$ , that is “close” to the root
  - Taylor expansion:

$$f(x_0 + \delta) \approx f(x_0) + f'(x_0)\delta + \dots$$

- If we are close, then
$$f(x_0 + \delta) \approx 0 \longrightarrow \delta = -\frac{f(x_0)}{f'(x_0)}$$
  - Update
$$x_1 = x_0 + \delta$$
  - We can continue, iterating again and again, until the change in the root  $< \varepsilon$
- Converges fast: usually only a few iterations are needed

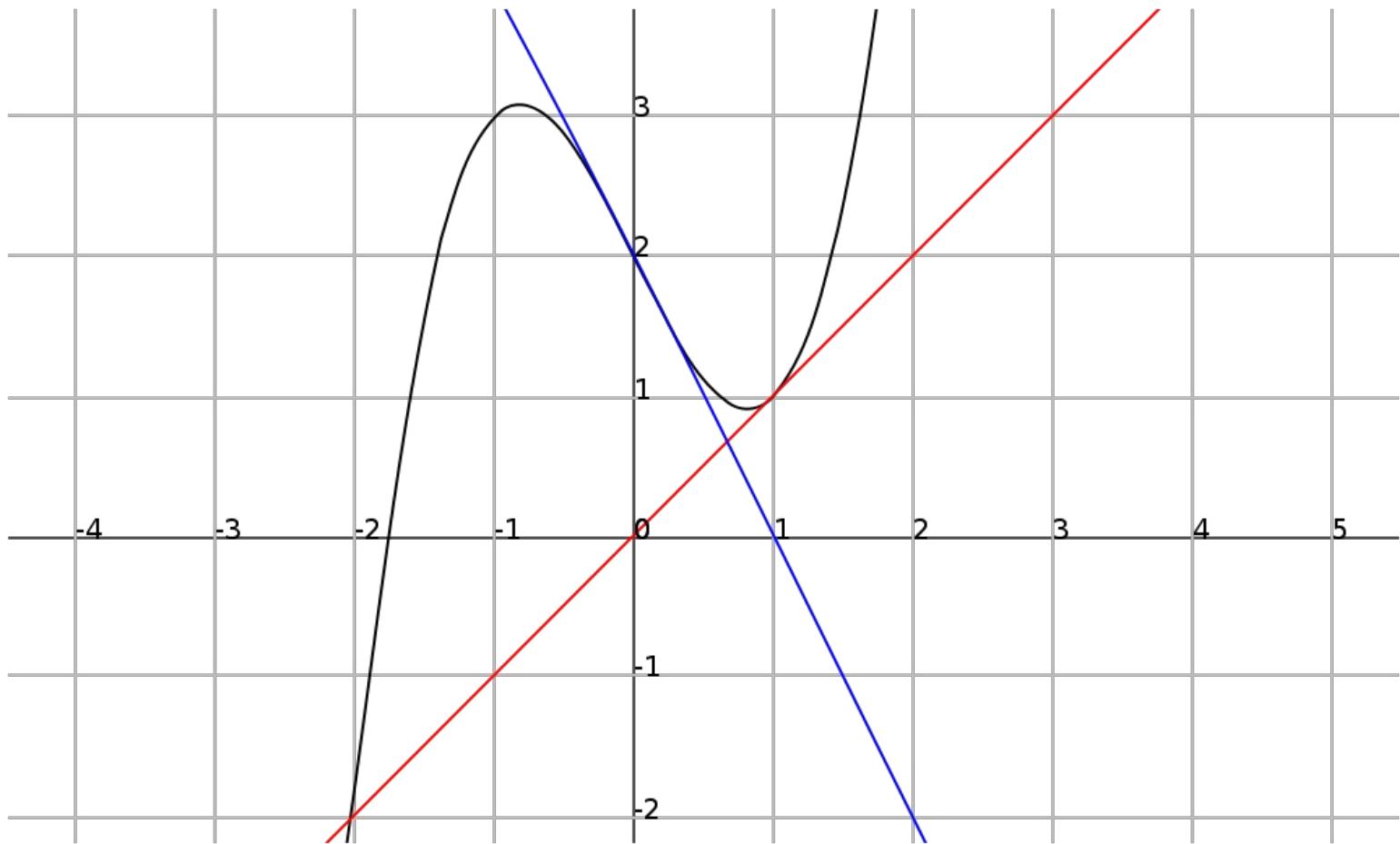
# Newton-Raphson

- Requirements for good convergence:
  - Derivative must exists and be non-zero in the interval near the root
  - Second derivative must be finite
  - $x_0$  must be close to the root
- Can be used with systems (we'll see this later)
- Multiple roots?
  - Generally: try to start with a good estimate\*

\*not a guarantee

# Newton-Raphson

- Consider  $f(x) = x^3 - 2x + 2$ 
  - Start with  $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \dots$
  - Cycle



# Secant Method

- If we don't know  $df/dx$ , we can still use the same ideas
  - We need to initial guesses:  $x_{-1}$  and  $x_0$
  - Use approximate derivative

$$x_1 = x_0 - \frac{f(x_0)}{[f(x_0) - f(x_{-1})]/(x_0 - x_{-1})}$$

- Used when an analytic derivative is unavailable, or too expensive to compute (e.g. EOS)
- Brent's method combines bisection, secant, and other methods to provide a very reliable method

# Multivariate Newton's Method

- Imagine a vector function:  $\mathbf{f}(\mathbf{x})$ 
  - $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}) \ f_2(\mathbf{x}) \ f_3(\mathbf{x}) \ \dots \ f_N(\mathbf{x}))^T$
  - Column vector of unknowns:  $\mathbf{x} = (x_1 \ x_2 \ x_3 \ \dots \ x_N)^T$
- We want to find the zeros:
  - Initial guess:  $\mathbf{x}^{(0)}$
  - Taylor expansion:
 
$$f_i(\mathbf{x}^{(0)} + \delta\mathbf{x}) \approx 0 = f_i(\mathbf{x}^{(0)}) + \sum_{j=1}^N \frac{\partial f_i}{\partial x_j} \delta x_j + \dots$$



This is the Jacobian
- Update to initial guess is:  $\delta\mathbf{x} = -\mathbf{J}^{-1}\mathbf{f}(\mathbf{x}^{(0)})$ 
  - Cheaper to solve the linear system than to invert the Jacobian
  - Iterate:  $\mathbf{J}\delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$ ,  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta\mathbf{x}^{(k)}$

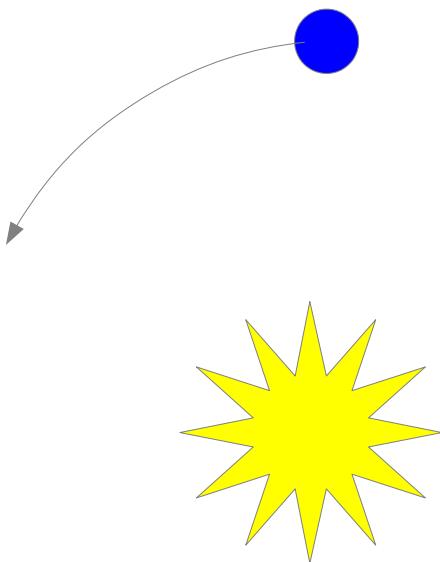
# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# ODEs

- Consider orbits around the Sun
  - Another simple system that allows us to explore the properties of ODE integrators



$$\dot{\mathbf{x}} = \mathbf{v} \quad \dot{\mathbf{v}} = -\frac{GM\mathbf{r}}{r^3}$$

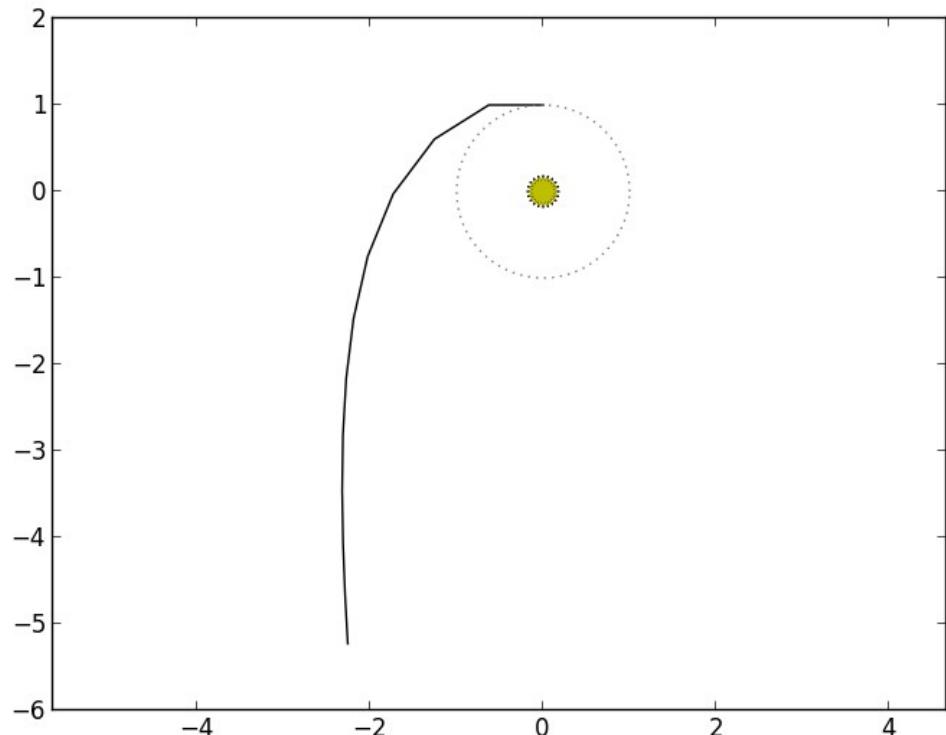
- Kepler's law (neglecting orbiting object mass):
$$4\pi^2 a^3 = GM_\star P^2$$
- Work in units of AU, solar masses, and years
  - $GM = 4\pi^2$

# Orbits: Euler Method

- Simplest case: Euler's method

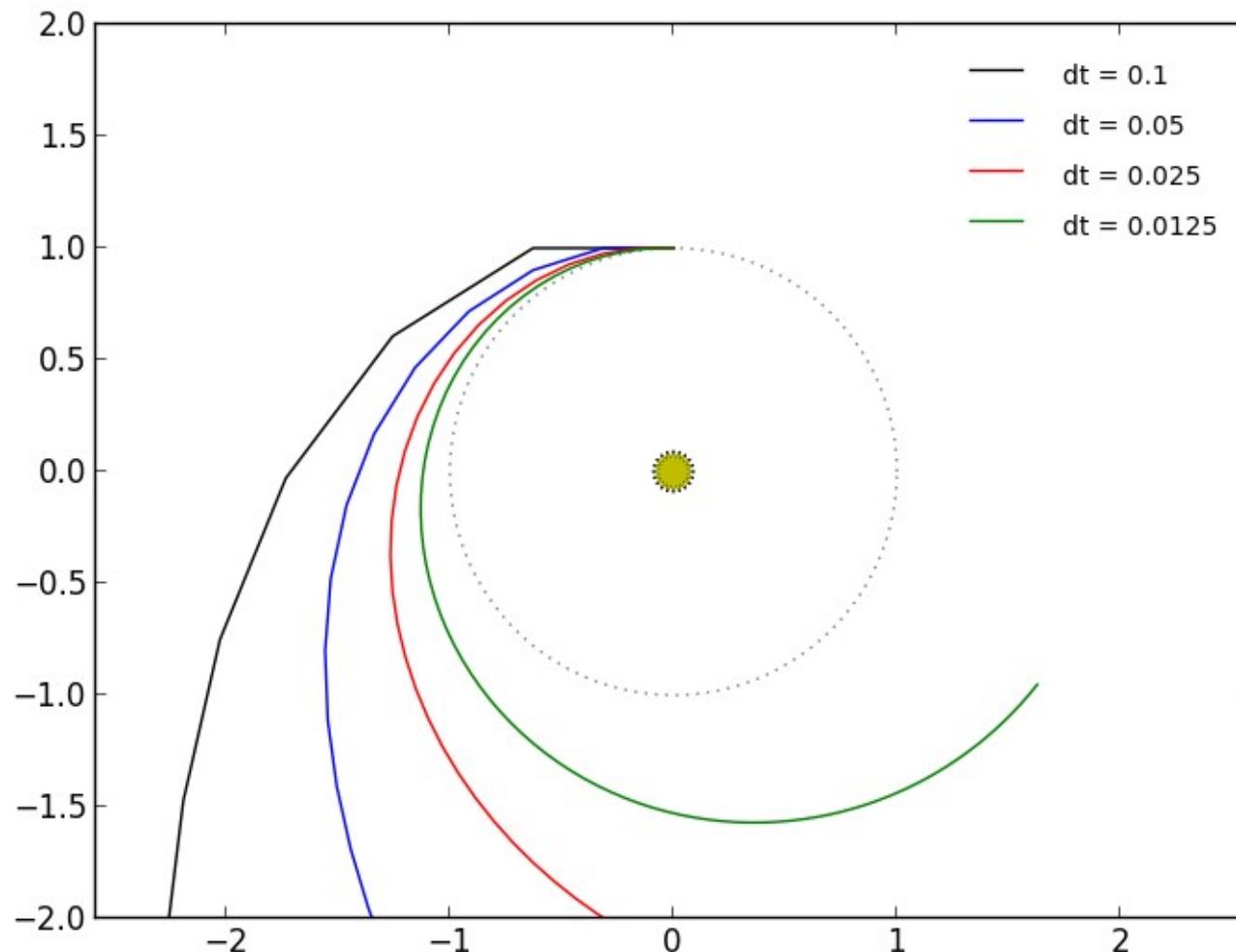
$$\mathbf{x}^{n+1} = \mathbf{x}^n + \tau \mathbf{v}^n \quad \mathbf{v}^{n+1} = \mathbf{v}^n + \tau \mathbf{a}^n$$

- Need to specify a semi-major axis and eccentricity
- Initial conditions:
  - $x = 0, y = a(1 - e)$
  - $u = -\sqrt{\frac{GM}{a}} \frac{1+e}{1-e}, v = 0$
  - This is counter-clockwise orbiting



Our planet escapes—clearly energy is not conserved here!

# Orbits: Euler Method



Things get better with a smaller timestep, but this is still first-order

Let's look at the code and see how small we need to get a closed circle

# Higher Order Methods

---

- Midpoint or 2<sup>nd</sup> order Runge-Kutta:

$$\frac{\mathbf{r}^{n+1} - \mathbf{r}^n}{\tau} = \mathbf{v}^{n+1/2} + \mathcal{O}(\tau^2) \quad \frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\tau} = \mathbf{a}^{n+1/2} + \mathcal{O}(\tau^2)$$

- The updates are then:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \tau \mathbf{v}^{n+1/2} + \mathcal{O}(\tau^3)$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \tau \mathbf{a}^{n+1/2} + \mathcal{O}(\tau^3)$$

- This is third-order accurate (locally), to start things off, we do:

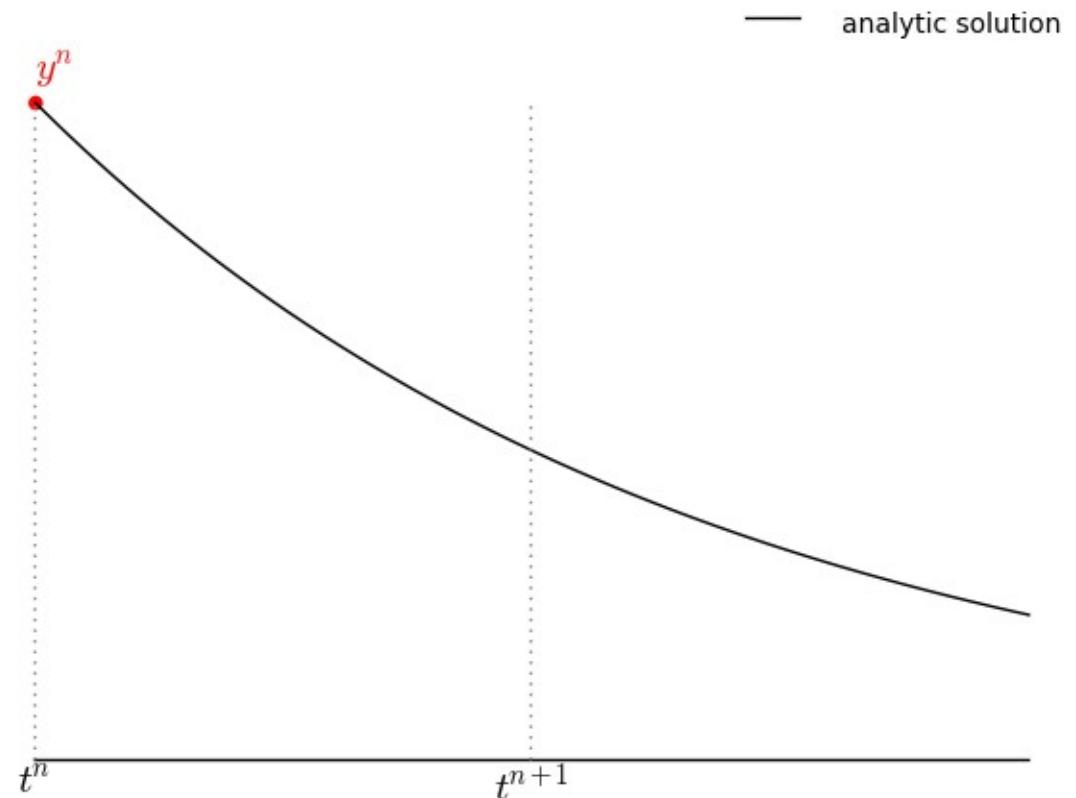
$$\mathbf{r}^\star = \mathbf{r}^n + (\tau/2)\mathbf{v}^n$$

$$\mathbf{v}^\star = \mathbf{v}^n + (\tau/2)\mathbf{a}^n$$

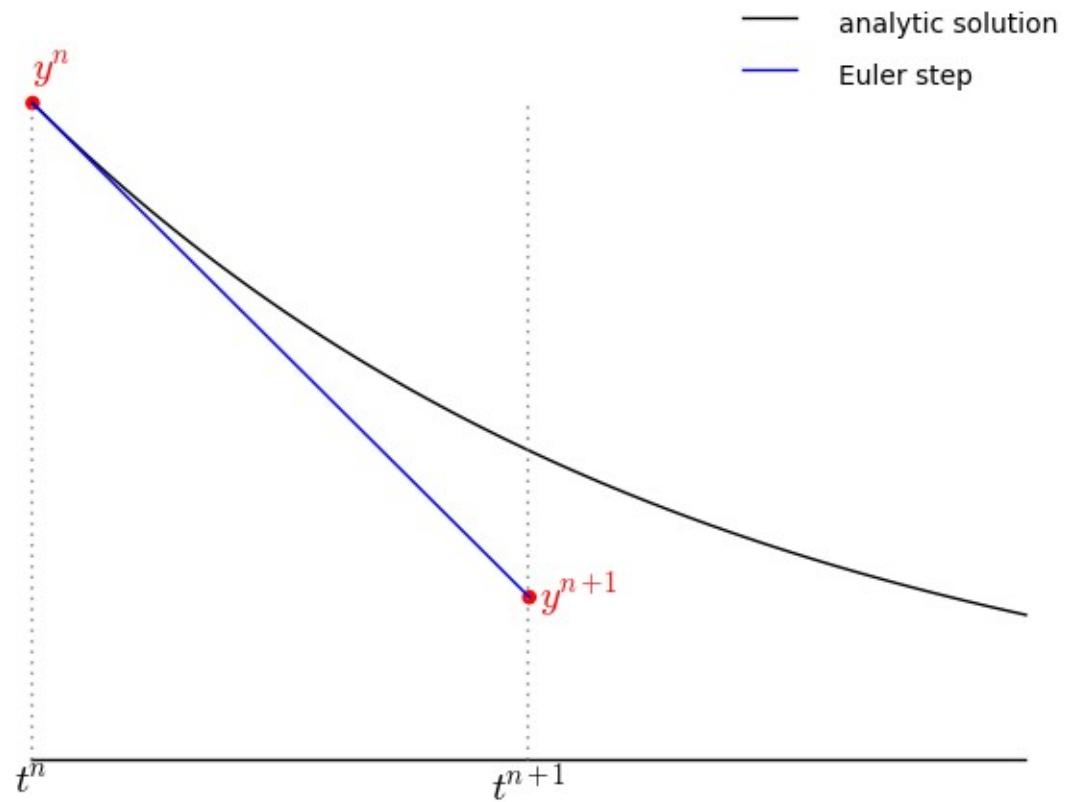
$$\mathbf{r}^{n+1} = \mathbf{r}^n + \tau \mathbf{v}^\star$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \tau \mathbf{a}(\mathbf{r}^\star)$$

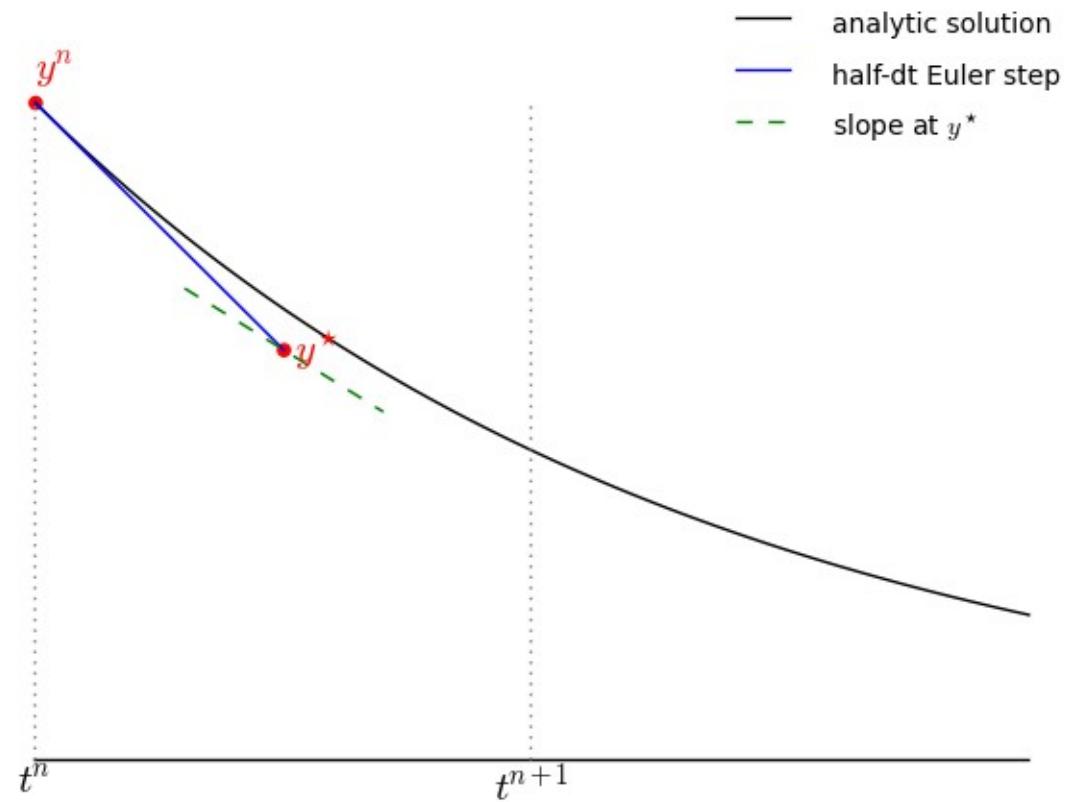
# 2<sup>nd</sup> Runge-Kutta Methods



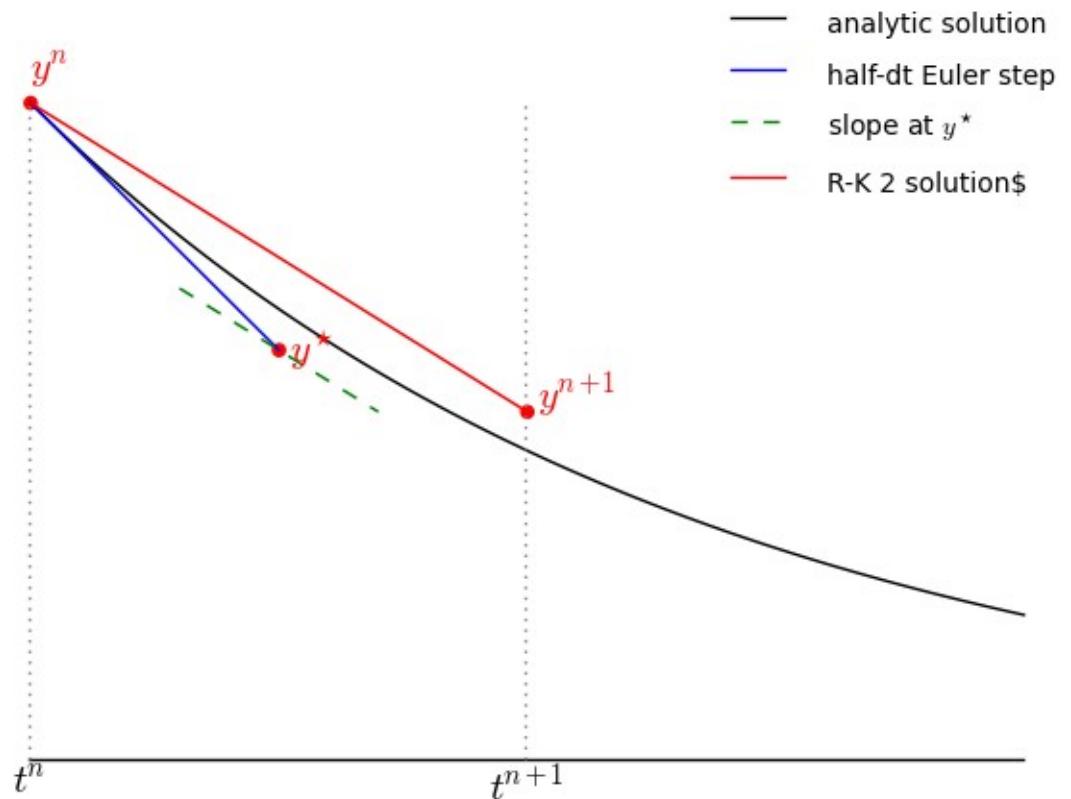
# 2<sup>nd</sup> Runge-Kutta Methods



# 2<sup>nd</sup> Runge-Kutta Methods



# 2<sup>nd</sup> Runge-Kutta Methods



# 4<sup>th</sup> Runge-Kutta Methods

- One of the most popular methods is 4<sup>th</sup>-order Runge-Kutta
  - Consider system:  $\dot{\mathbf{y}} = \mathbf{g}(t, \mathbf{y})$
  - Update through  $\tau$ :

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{\tau}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) + \mathcal{O}(\tau^5)$$

$$\mathbf{k}_1 = \mathbf{g}(t, \mathbf{y})$$

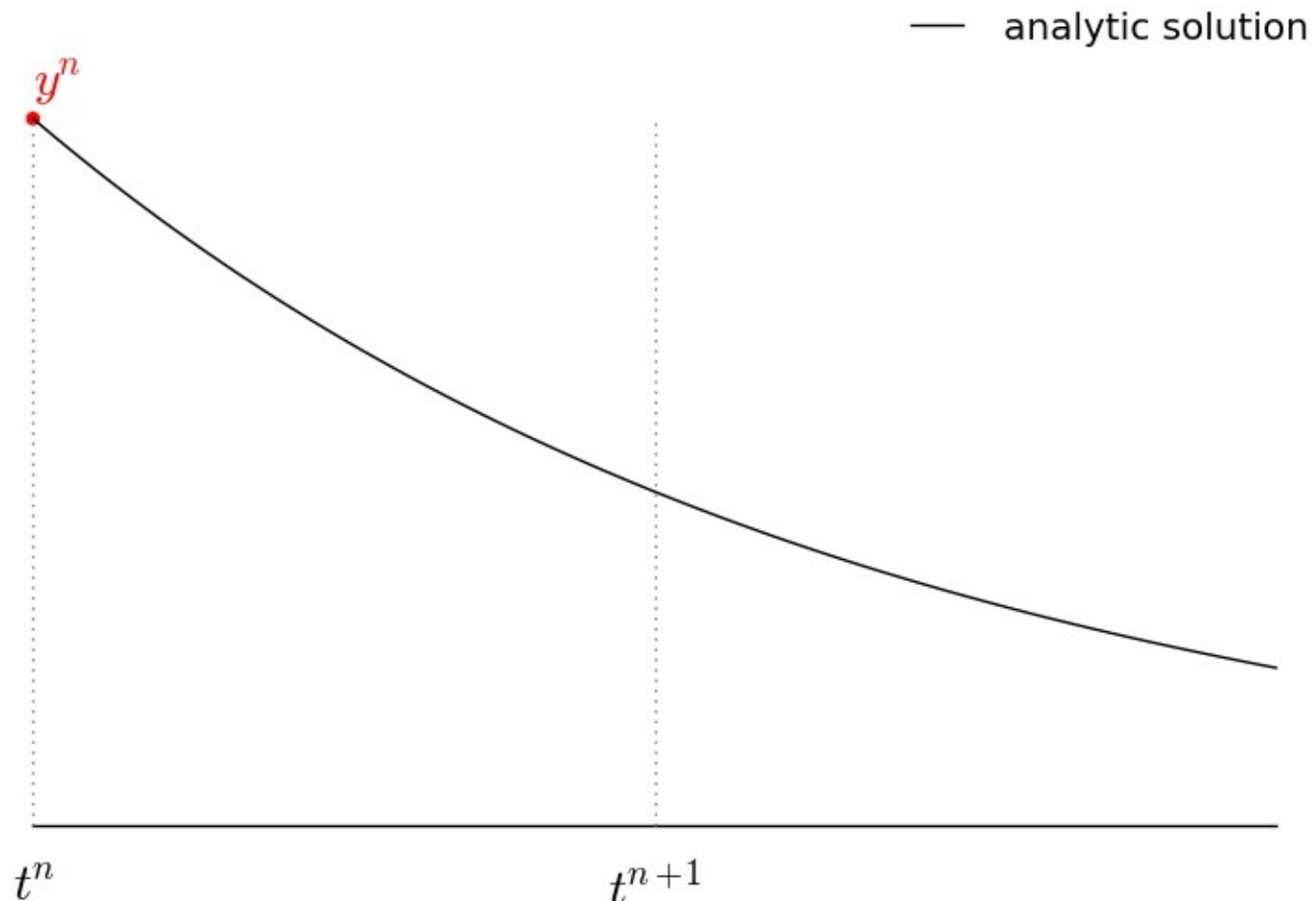
$$\mathbf{k}_2 = \mathbf{g}\left(t + \frac{\tau}{2}, \mathbf{y} + \frac{\tau}{2}\mathbf{k}_1\right)$$

$$\mathbf{k}_3 = \mathbf{g}\left(t + \frac{\tau}{2}, \mathbf{y} + \frac{\tau}{2}\mathbf{k}_2\right)$$

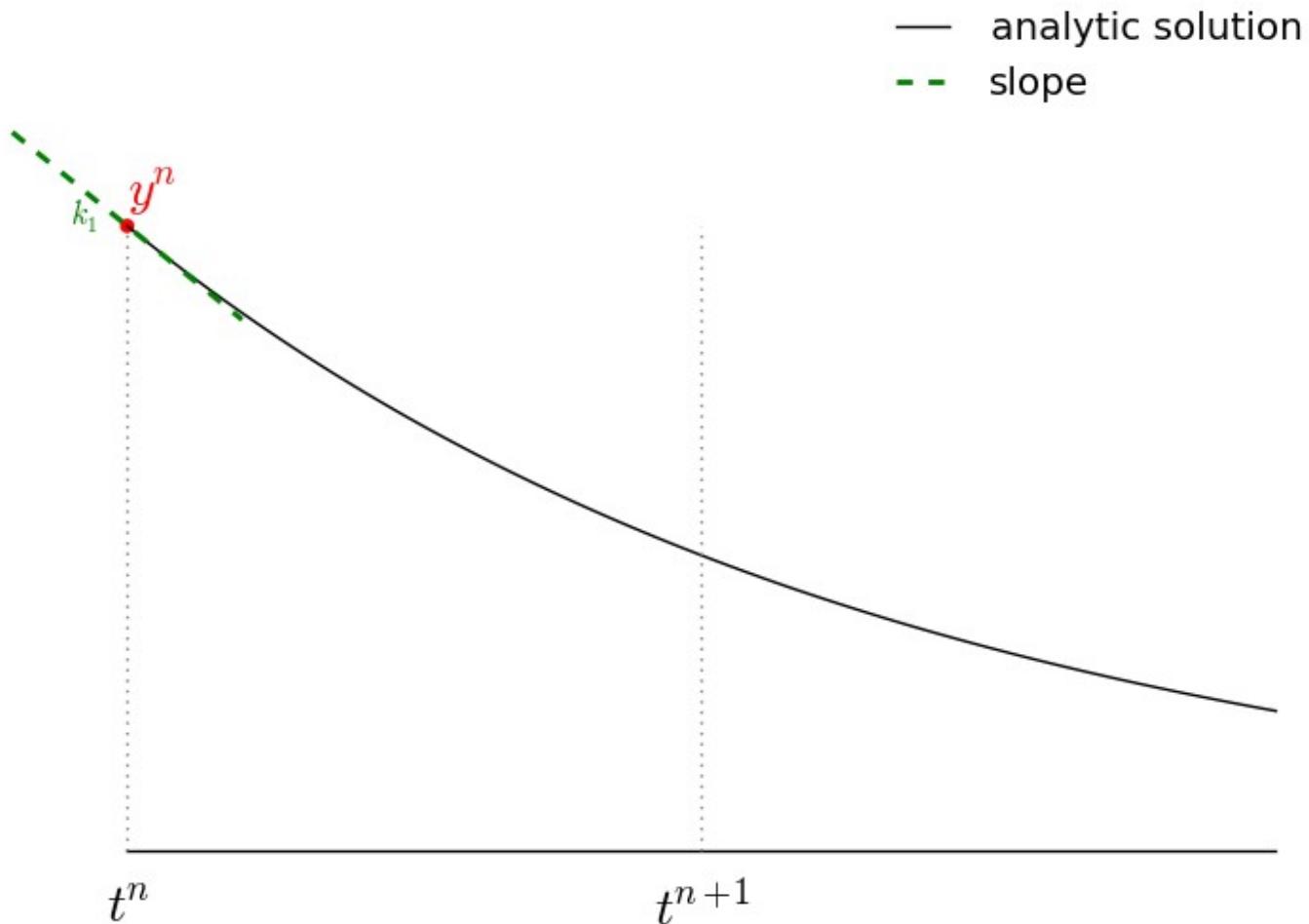
$$\mathbf{k}_4 = \mathbf{g}(t + \tau, \mathbf{y} + \tau\mathbf{k}_3)$$

- Notice the similarity to Simpson's integration
- Derivation found in many analysis texts

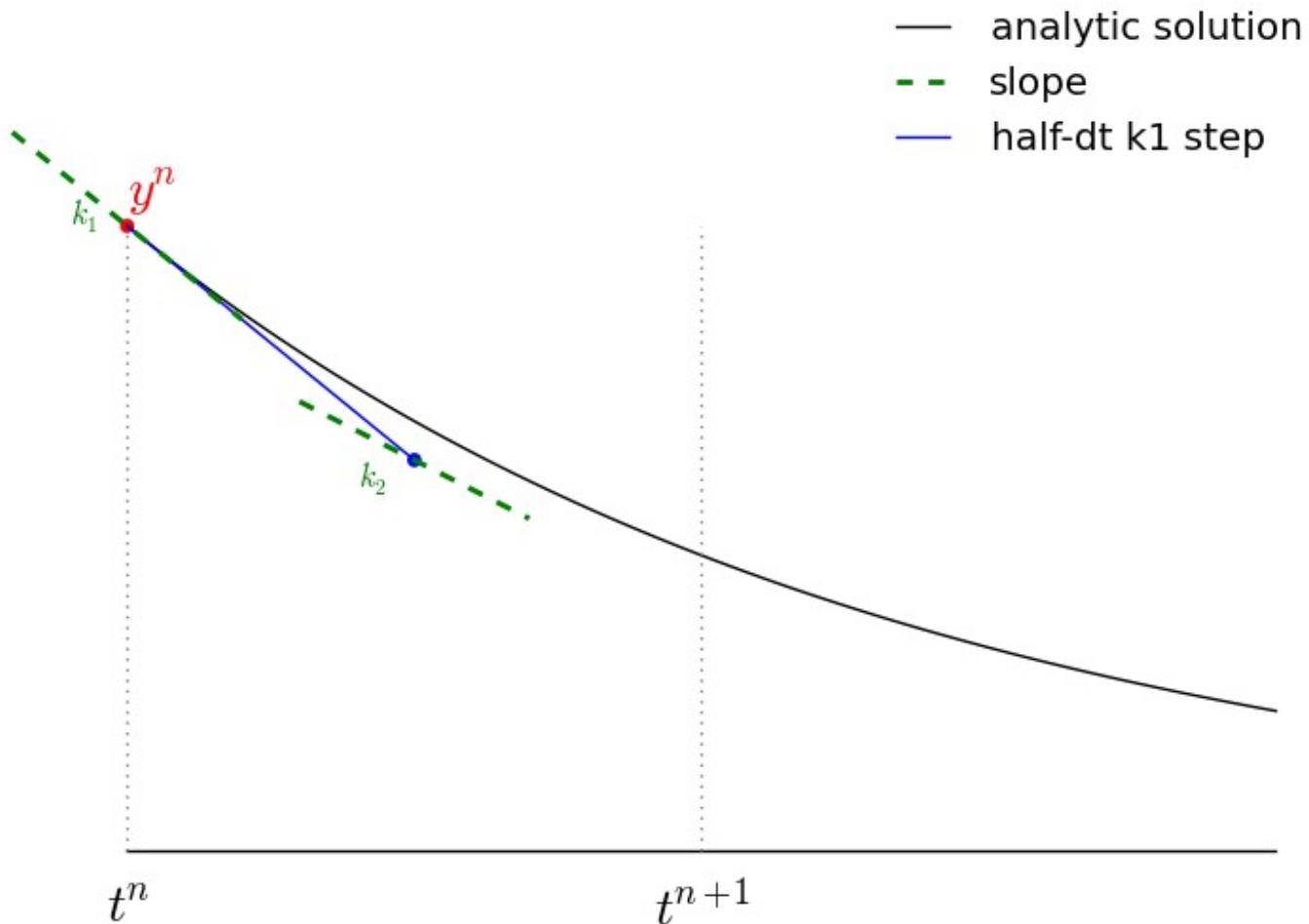
# 4<sup>th</sup> Runge-Kutta Methods



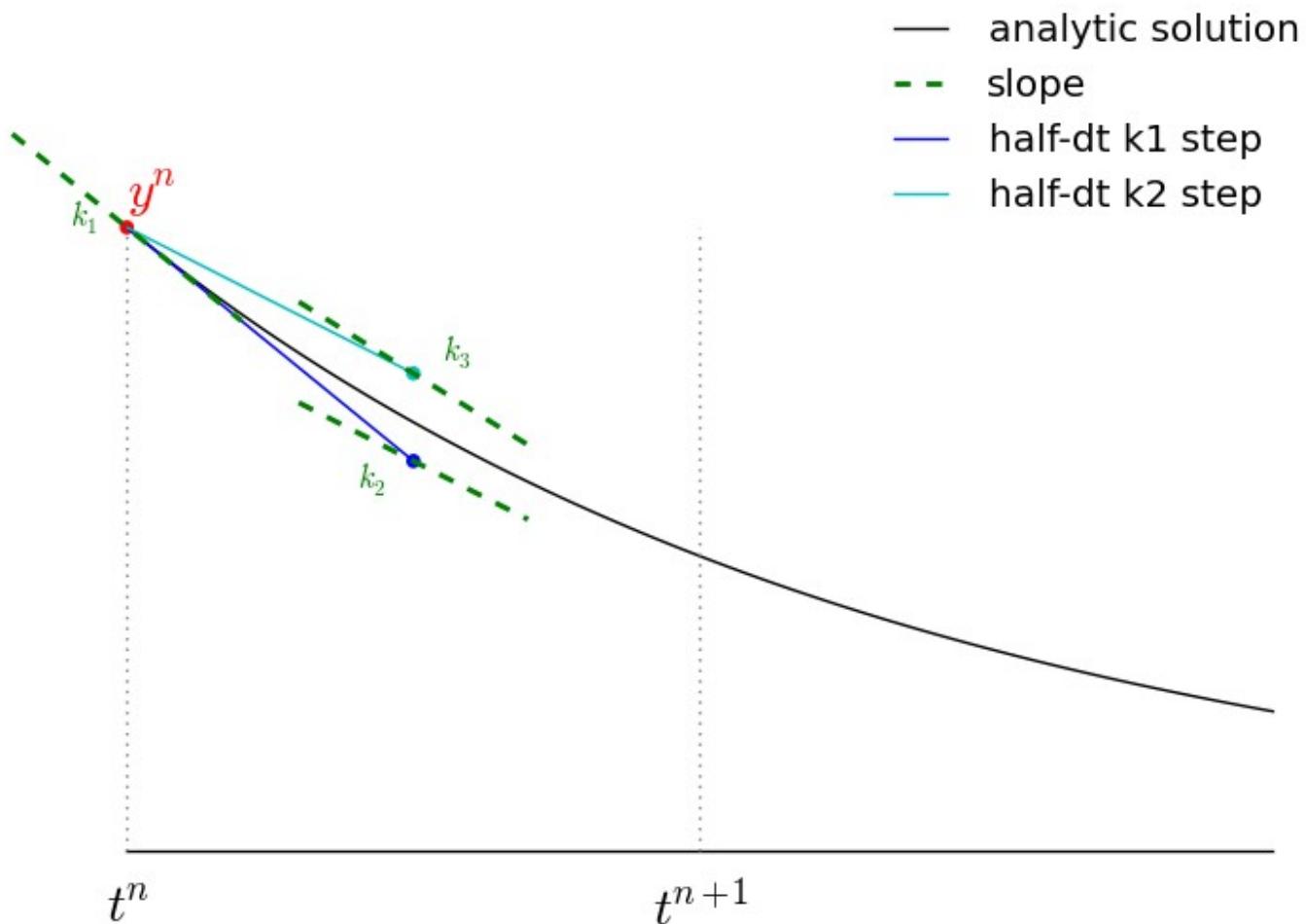
# 4<sup>th</sup> Runge-Kutta Methods



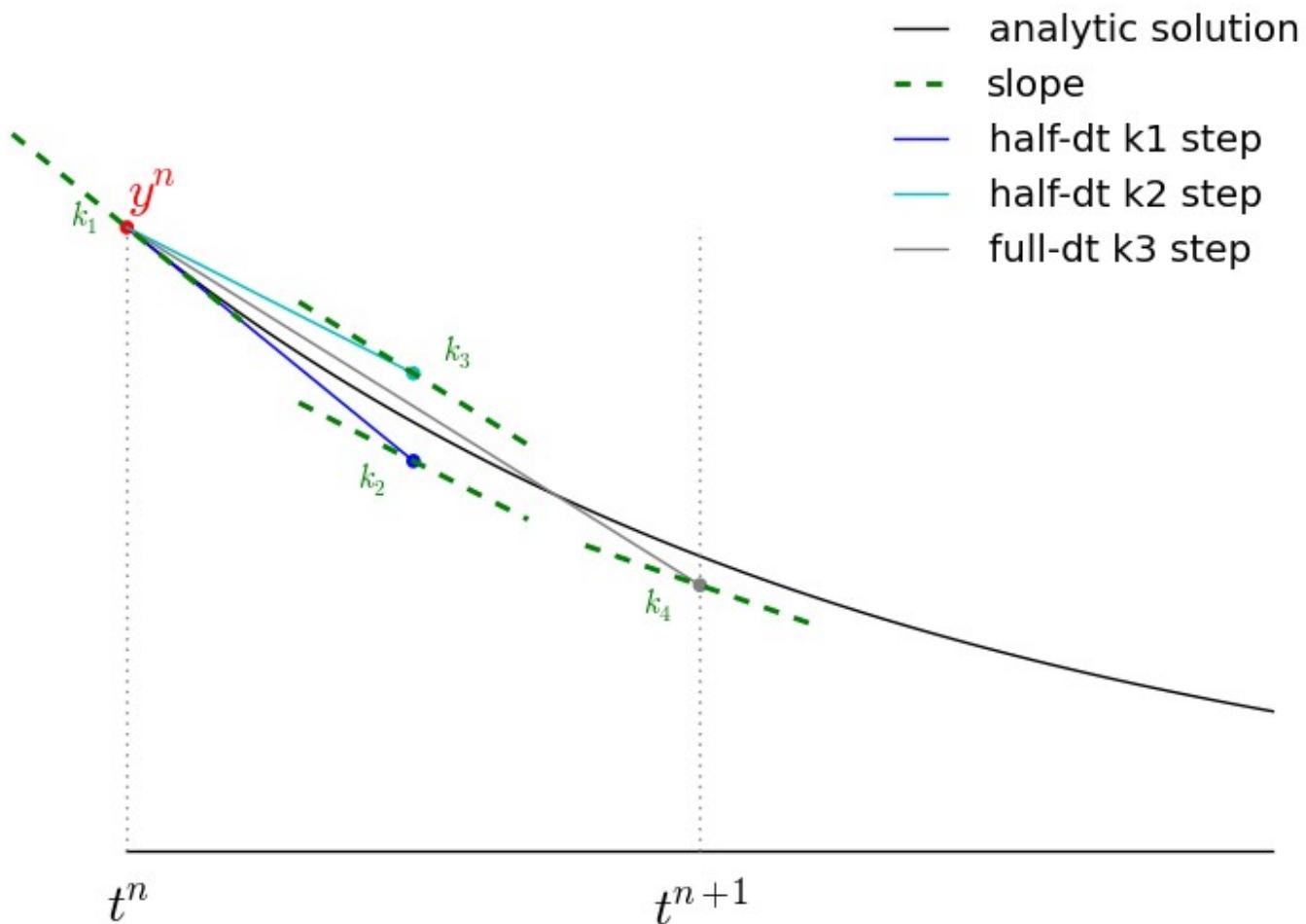
# 4<sup>th</sup> Runge-Kutta Methods



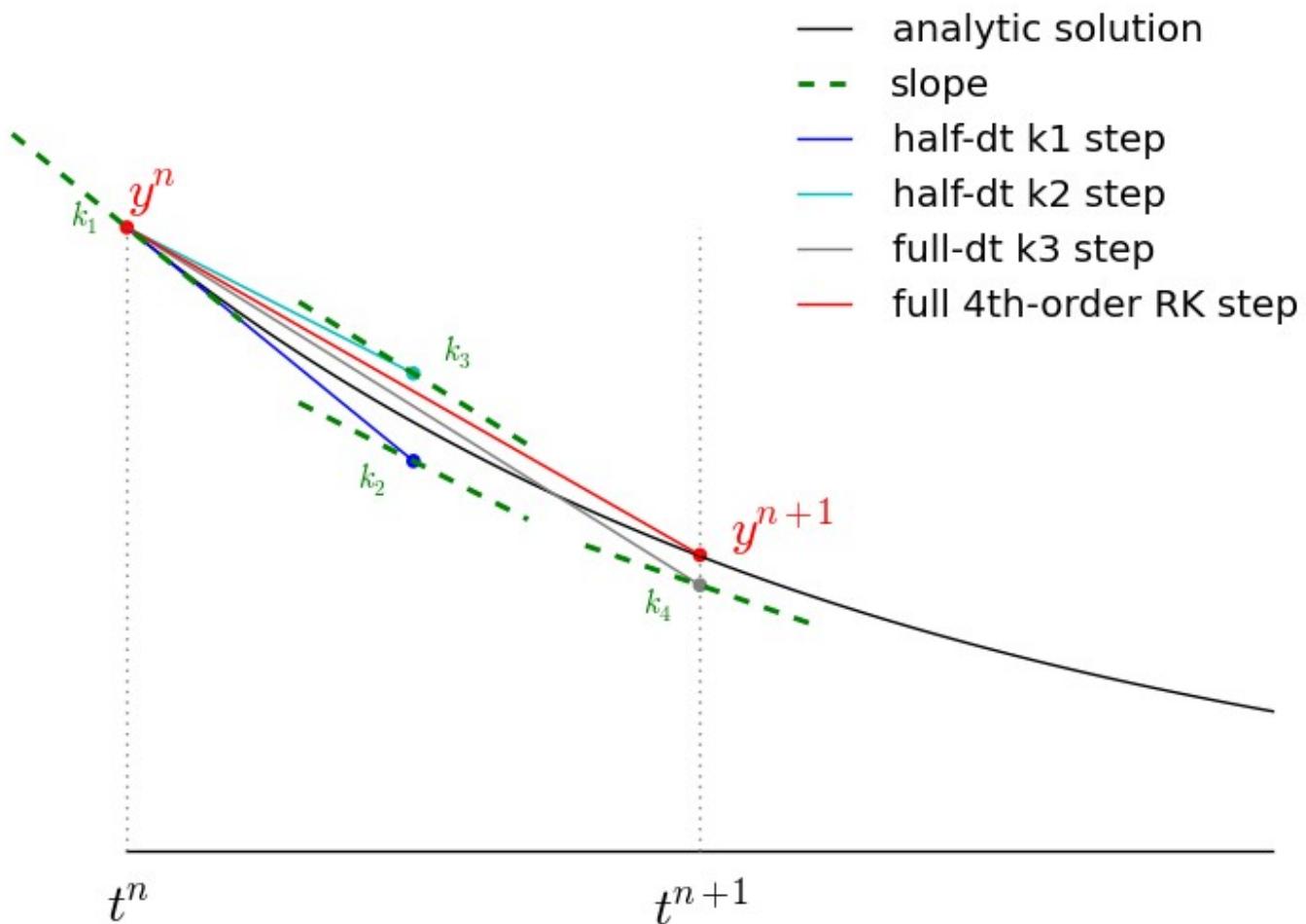
# 4<sup>th</sup> Runge-Kutta Methods



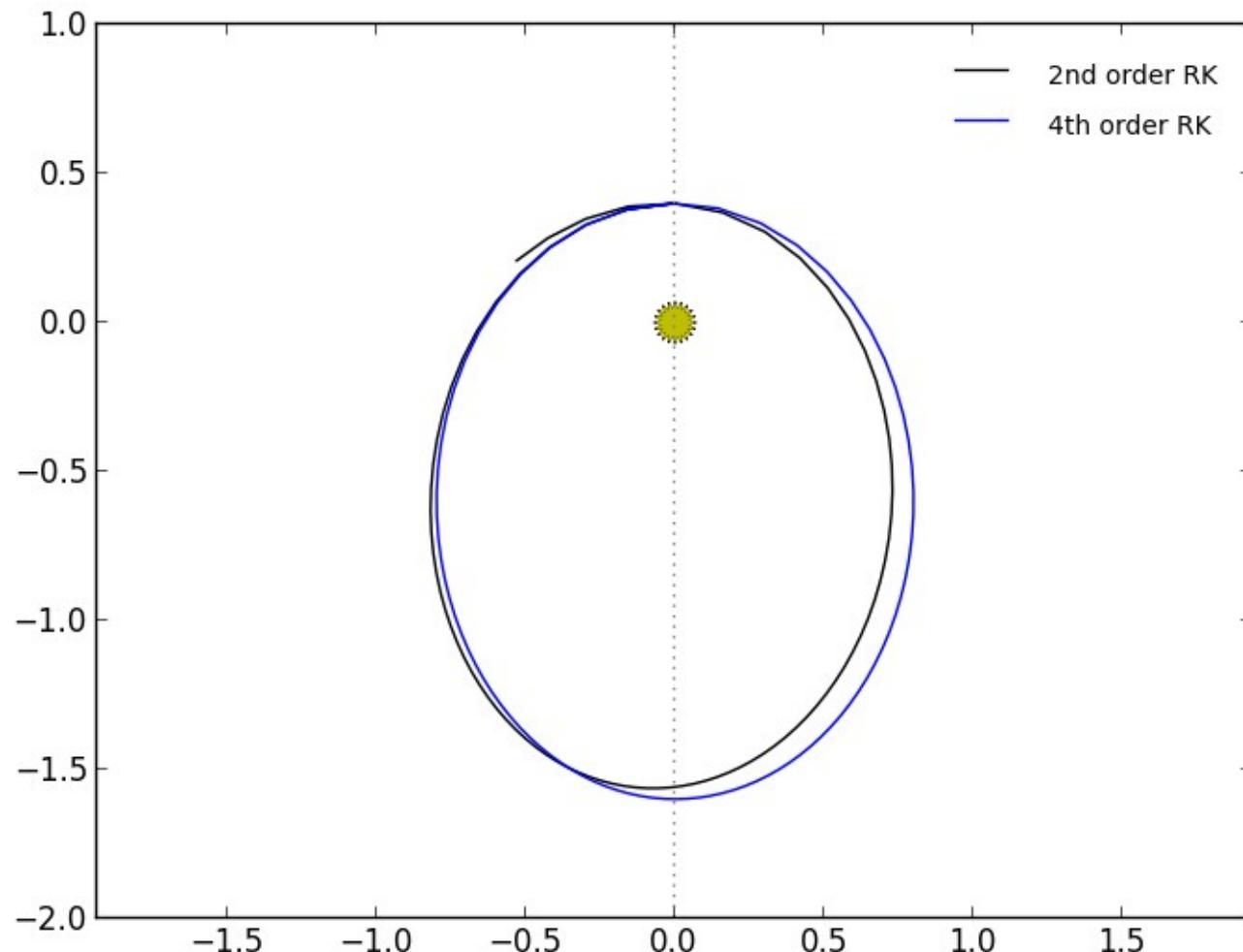
# 4<sup>th</sup> Runge-Kutta Methods



# 4<sup>th</sup> Runge-Kutta Methods

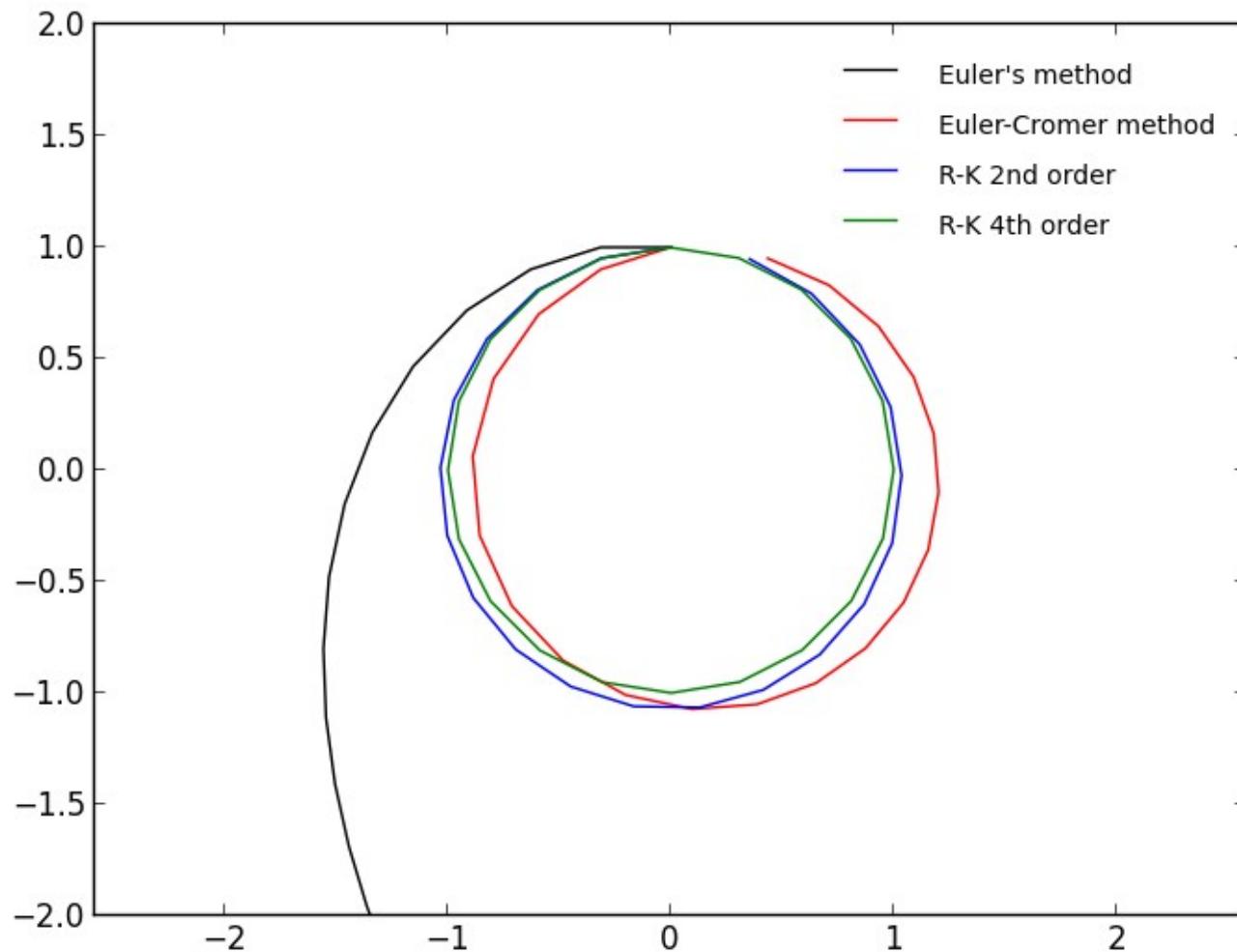


# 4<sup>th</sup> Runge-Kutta Methods



This looks great!

# 4<sup>th</sup> Runge-Kutta Methods



Even with a coarse timestep, RK4 does very well.

# Adaptive Stepping

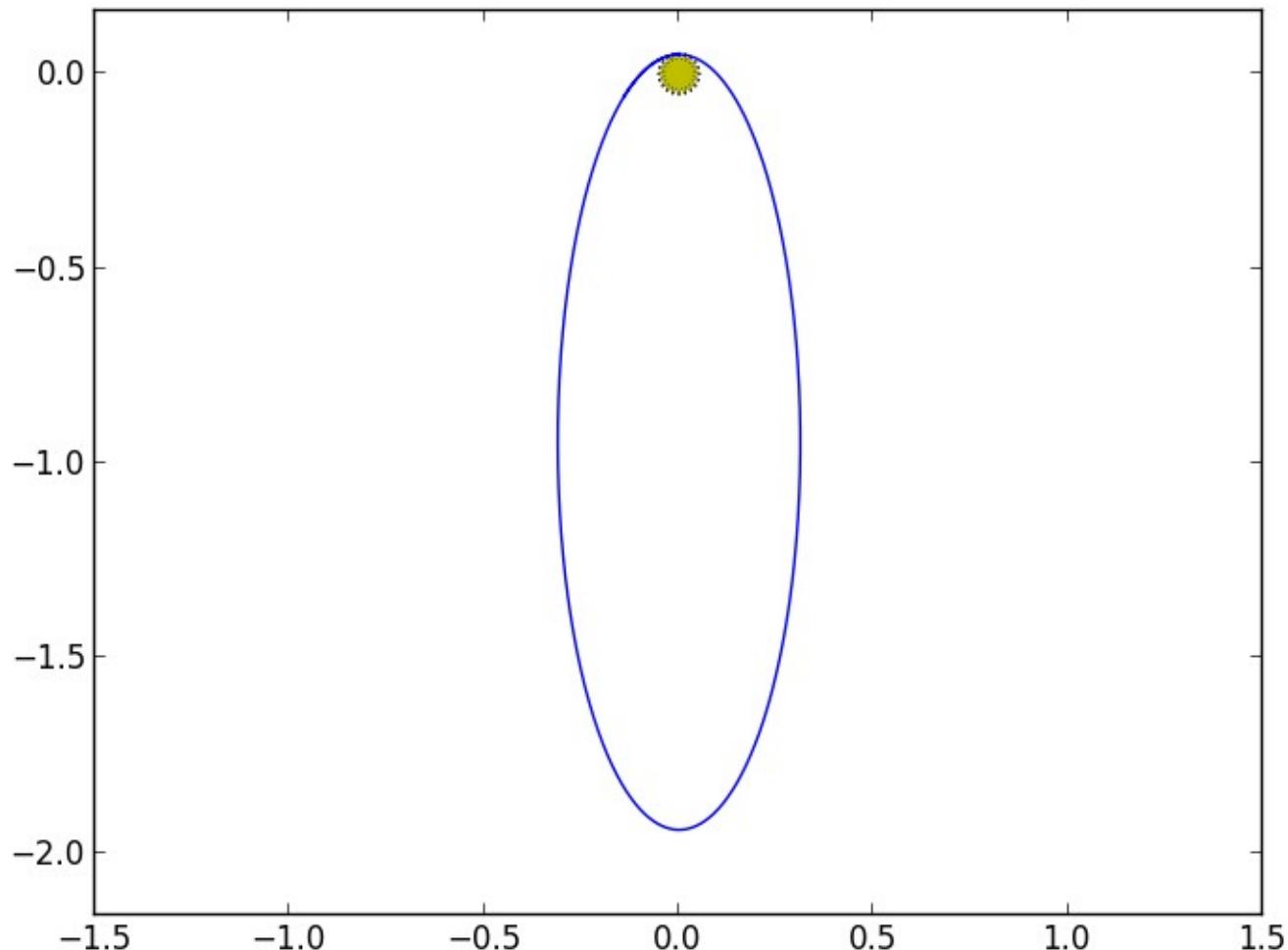
- We need a smaller timestep where the solution changes most rapidly
  - We can get away with large timesteps in regions of slow evolution
- Monitoring the error can allow us to estimate the optimal timestep to reach some desired accuracy
- Lot's of different techniques for this in the literature
  - Take two half steps and compare to one full state
  - Compare higher and lower order methods

# Example: Highly Elliptical Orbit

- Consider a highly elliptical orbit:  $a = 1.0$ ,  $e = 0.95$ 
  - Sun-grazing comet

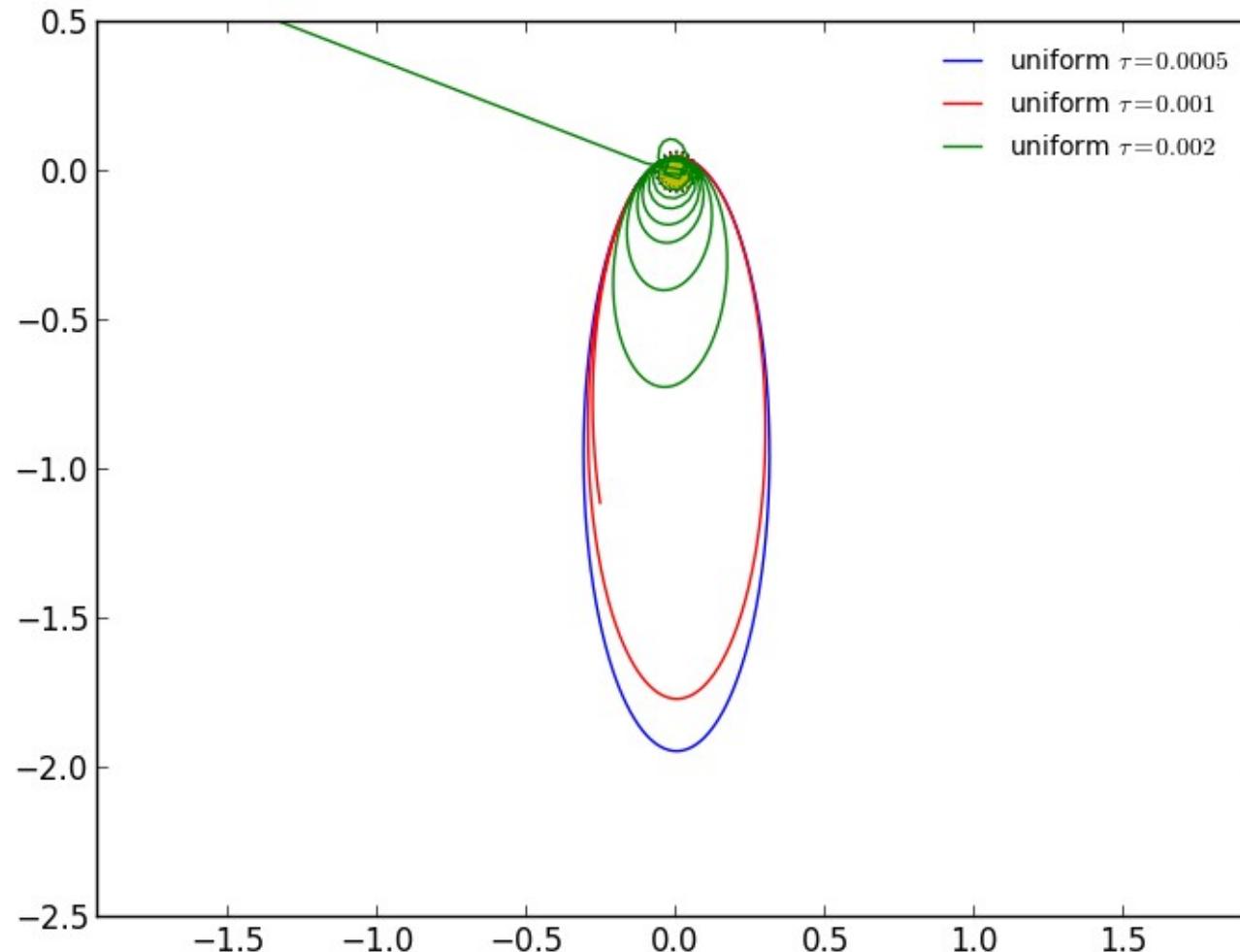
Just to get a reasonable-looking solution, we needed to use  $\tau = 0.0005$

This takes 2001 steps



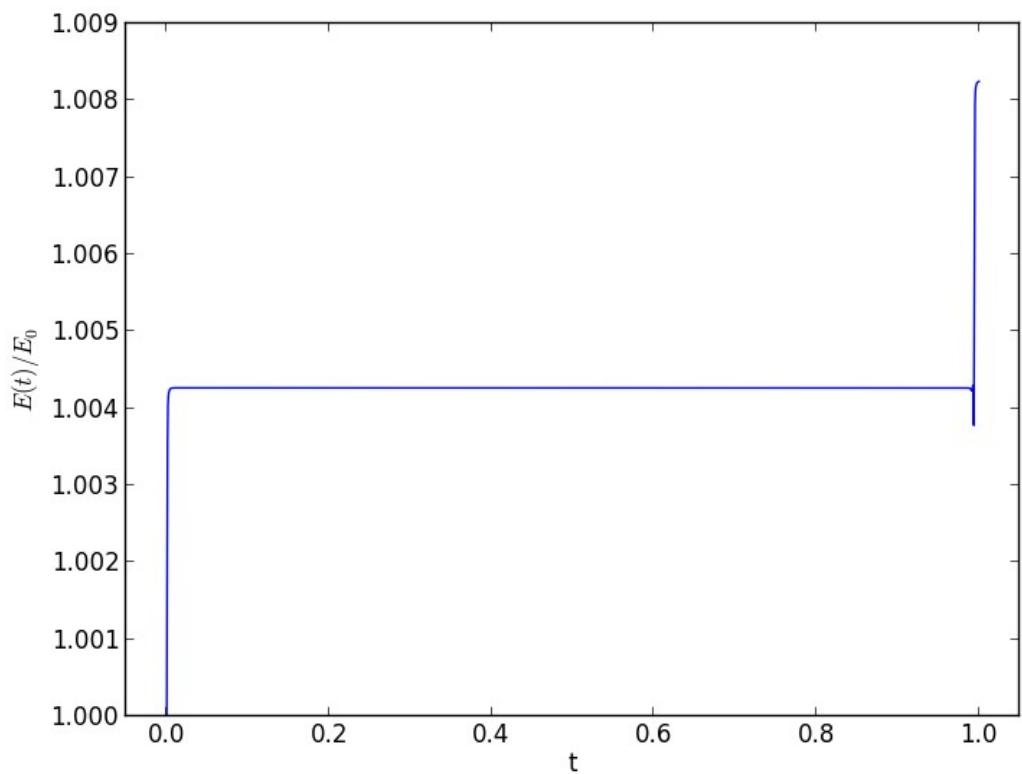
# Example: Highly Elliptical Orbit

- Solutions with various uniform timesteps



# Example: Highly Elliptical Orbit

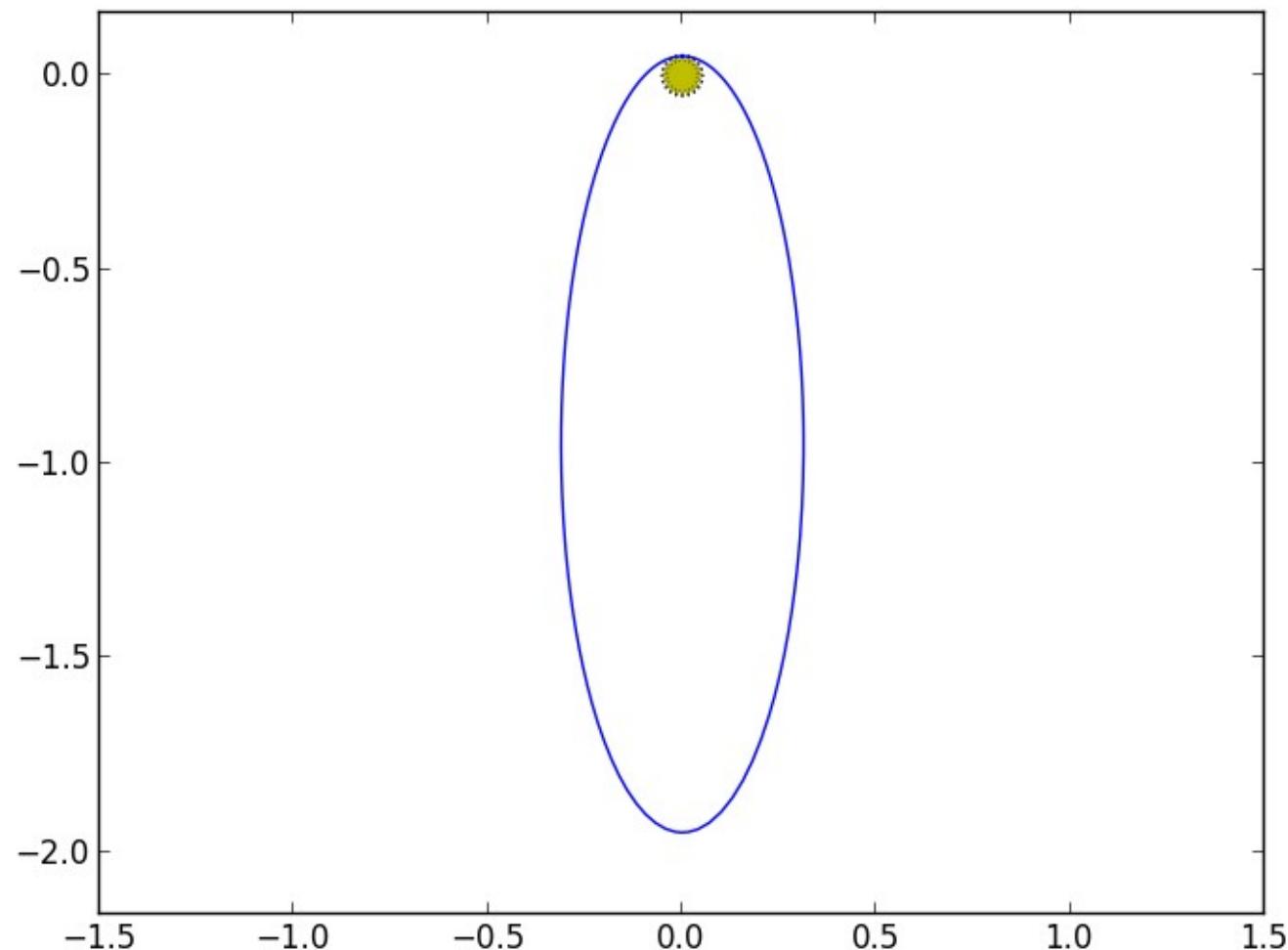
- Look at the total energy
  - At perihelion, conservation is the worse
  - Perihelion is where the velocity is greatest, and therefore the solution changes the fastest
- We can take a larger timestep at aphelion than perihelion



# Example: Highly Elliptical Orbit

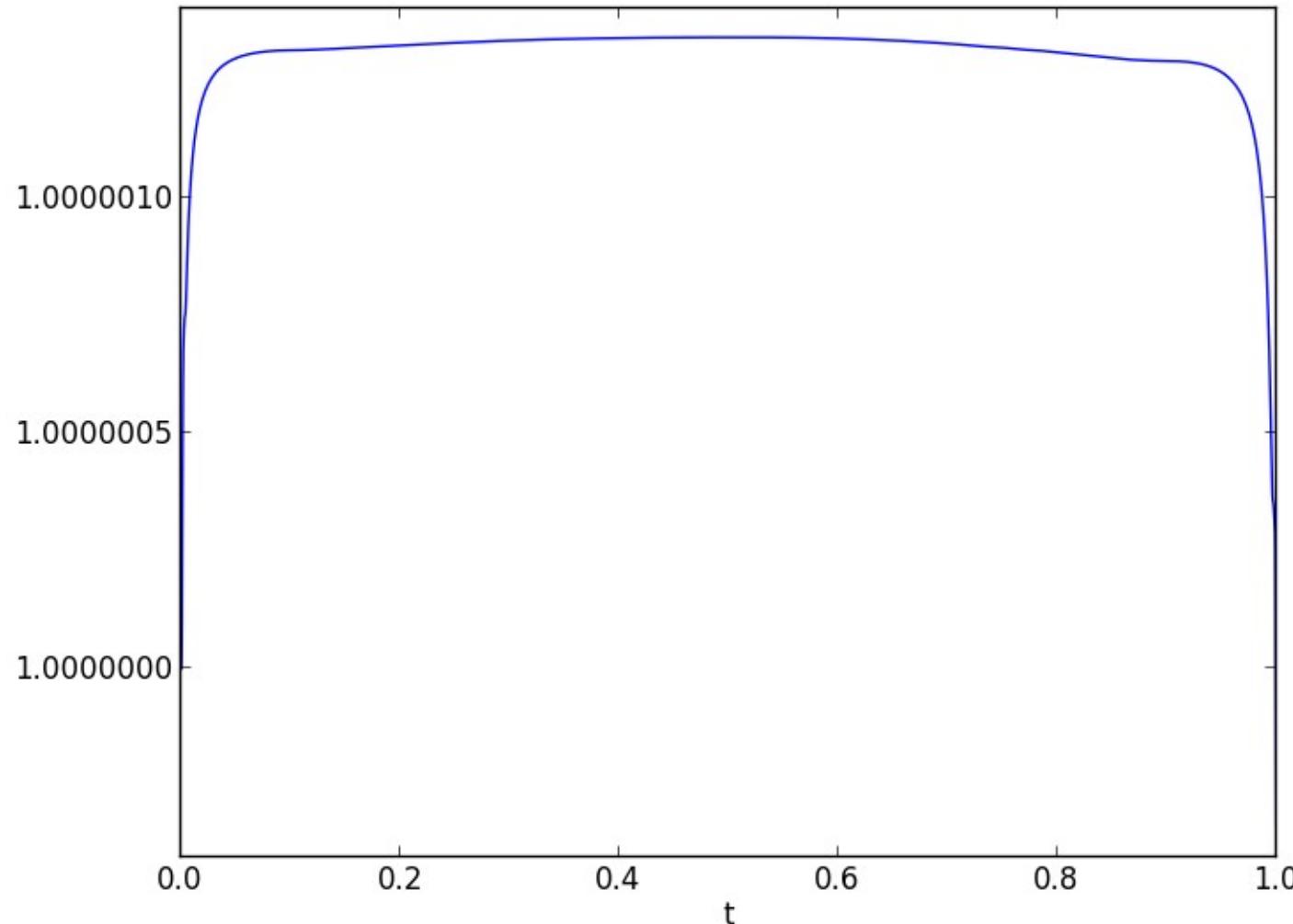
- Adaptive stepping, asking for  $\epsilon = 10^{-7}$ , with initial timestep the same as the non-adaptive case ( $\tau = 0.005$ )

This takes only 215 steps



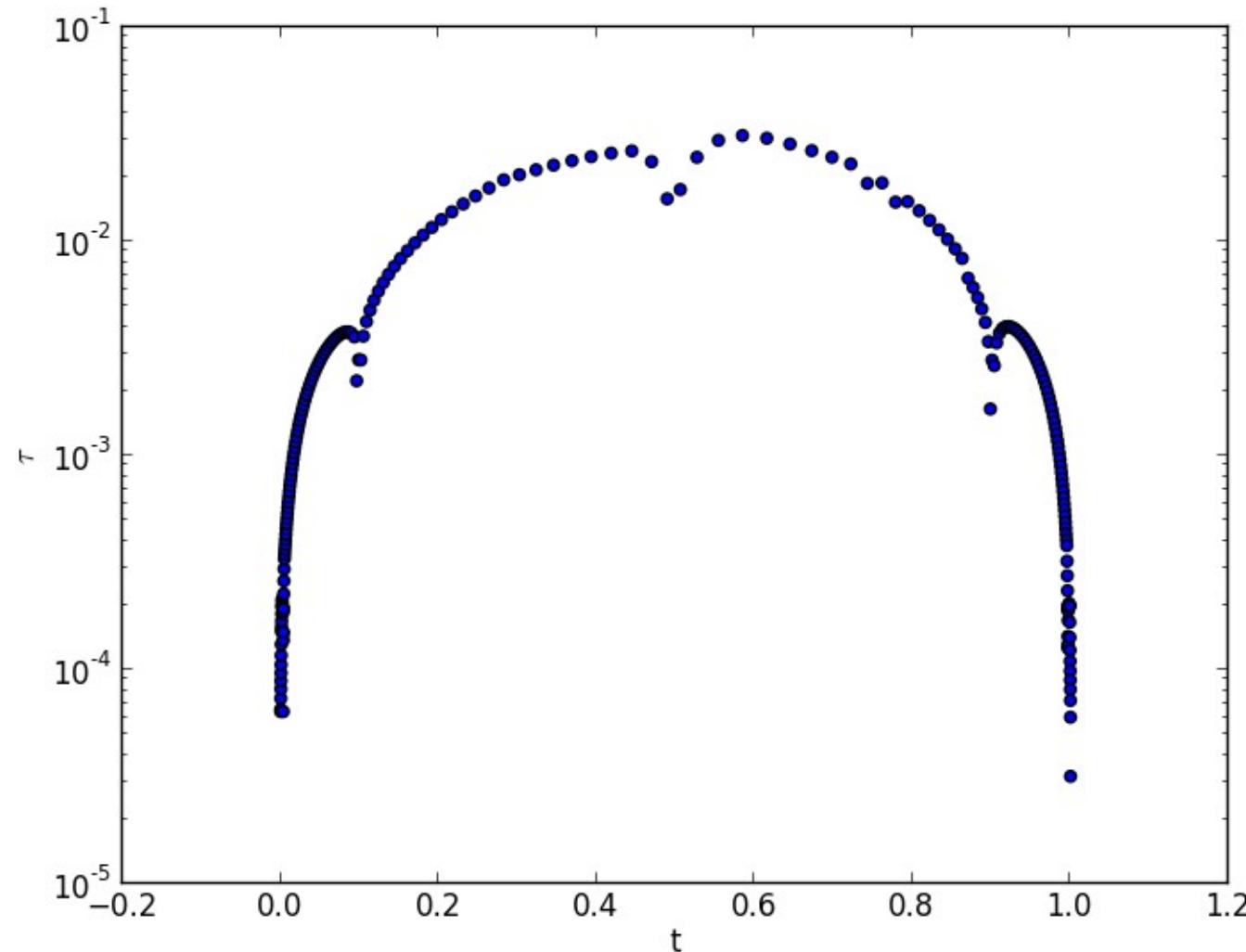
# Adaptive Stepping

- Energy conservation is now far superior



# Adaptive Stepping

- Timestep varies significantly over the evolution



# Adaptive Stepping / Error Estimation

---

- You should always perform some sort of error estimation
- Specifying absolute and relative errors in the state variables ensures you know about the quality of the solution
- The SciPy ODE packages can control all of this for you

# Stiff Equations / Implicit Methods

- Consider the ODE (example from Byrne & Hindmarsh 1986):

$$\dot{y} = -10^3(y - e^{-t}) - e^{-t}$$

$$y(0) = 0$$

- This has the exact solution:

$$y(t) = e^{-t} - e^{-10^3 t}$$

- Looking at this, we see that there are two characteristic timescales for change,  $\tau_1 = 1$  and  $\tau_2 = 10^{-3}$
- A problem with dramatically different timescales for change is called **stiff**
- Stiff ODEs can be hard for the methods we discussed so far
  - Stability requires that we evolve on the shortest timescale

# Outline

---

1. SciPy	2
1.1 Introduction	3
1.2 Integration	7
1.3 Interpolation	20
1.4 Root Finding	30
1.5 ODEs	38
1.6 Curve Fitting	65

# Fitting Data

- We get experimental/observational data as a sequence of times (or positions) and associate values
  - $N$  points:  $(x_i, y_i)$
  - Often we have errors in our measurements at each of these values:  $\sigma_i$  for each  $y_i$
- To understand the trends represented in our data, we want to find a simple functional form that best represents the data—this is the fitting problem
  - The `scipy.optimize` module offers routines to do fitting
- We'll look at least squares fitting
  - Two cases: general linear and nonlinear

# Fitting Data

- We want to fit our data to a function:  $Y(x, \{a_j\})$ 
  - Here, the  $a_j$  are a set of parameters that we can adjust
  - We want to find the optimal set of  $a_j$  that make  $Y$  best represent our data
- The distance between a point and the representative curve is

$$\Delta_i = Y(x_i, \{a_j\}) - y_i$$

- Least squares fit minimizes the sum of the squares of all these errors
- With error bars, we weight each distance error by the uncertainty in that measurement, giving:

$$\chi^2(\{a_j\}) = \sum_{i=1}^N \left( \frac{\Delta_i}{\sigma_i} \right)^2$$

This is what we minimize

# Example: Linear Regression (Least Squares)

---

- Minimization: derivative of  $\chi^2$  with respect to all parameters is zero:

$$\frac{\partial \chi^2}{\partial a_1} = 2 \sum_{i=1}^N \frac{a_1 + a_2 x_i - y_i}{\sigma_i^2} = 0 \quad \frac{\partial \chi^2}{\partial a_2} = 2 \sum_{i=1}^N \frac{a_1 + a_2 x_i - y_i}{\sigma_i^2} x_i = 0$$

- Define:

$$S = \sum_{i=1}^N \frac{1}{\sigma_i^2} \quad \xi_1 = \sum_{i=1}^N \frac{x_i}{\sigma_i^2} \quad \xi_2 = \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2}$$

$$\eta = \sum_{i=1}^N \frac{y_i}{\sigma_i^2} \quad \mu = \sum_{i=1}^N \frac{x_i y_i}{\sigma_i^2}$$

- Result: simple linear system to solve:

$$a_1 S + a_2 \xi_1 - \eta = 0$$

$$a_1 \xi_1 + a_2 \xi_2 - \mu = 0$$

# Goodness of the Fit

- Typically, if  $M$  is the number of parameters (2 for linear), then  $N \gg M$ 
  - Average pointwise error should be  $|y_i - Y(x_i)| \sim \sigma_i$
  - Number of degrees of freedom is  $N - M$ 
    - i.e. larger  $M$  makes it easier to fit all the points
    - See discussion in Numerical Recipes for more details and limitations
  - Putting these ideas into the  $\chi^2$  expression suggests that we consider

$$\frac{\chi^2}{N - M}$$

- If this is  $< 1$ , then the fit is good
- But watch out,  $\ll 1$  may also mean our errors were too large to begin with, we used too many parameters, ...

# General Linear Least Squares

- Garcia and Numerical Recipes provide a good discussion here
- We want to fit to

$$Y(x; \{a_j\}) = \sum_{j=1}^M a_j Y_j(x)$$

- Note that the  $Y$ s may be nonlinear but we are still linear in the  $a$ s
- Here,  $Y_j$  are our basis set—they can be  $x^j$  in which case we fit to a general polynomial
- Minimize:

$$\frac{\partial \chi^2}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_{i=1}^N \frac{1}{\sigma_i^2} \left\{ \sum_{k=1}^M a_k Y_k(x_i) - y_i \right\}^2 = 0 \longrightarrow$$

$$\sum_{i=1}^N \sum_{k=1}^M \frac{Y_j(x_i)}{\sigma_i} \frac{Y_k(x_i)}{\sigma_i} a_k = \sum_{i=1}^N \frac{Y_j(x_i)}{\sigma_i} \frac{y_i}{\sigma_i}$$

Linear system

# Error bars in Both $x$ and $y$

---

- Depending on the experiment, you may have errors in the dependent variable
  - For linear regression, our function to minimize becomes:

$$\chi^2(a_1, a_2) = \sum_{i=1}^N \frac{(a_1 + a_2 x_i - y_i)^2}{\sigma_{y,i}^2 + a_2^2 \sigma_{x,i}^2}$$

- Denominator is the total variance of the linear combination we are minimizing:

$$\begin{aligned} \text{Var}(a_1 + a_2 x_i - y_i) &= \text{Var}(a_2 x_i - y_i) \\ &= a_2^2 \text{Var}(x_i) + \text{Var}(y_i) = a_2^2 \sigma_{x,i}^2 + \sigma_{y,i}^2 \end{aligned}$$

(think about propagation of errors)

- We cannot solve analytically for the parameters, but we can use our root finding techniques on this.
  - See NR and references therein for more details

# General Non-linear Fitting

---

- Consider fitting directly to a function where the parameters enter nonlinearly:

$$f(a_0, a_1) = a_0 e^{a_1 x}$$

- We want to minimize

$$Q \equiv \sum_{i=1}^N (y_i - a_0 e^{a_1 x_i})^2$$

- Set the derivatives to zero:

$$f_0 \equiv \frac{\partial Q}{\partial a_0} = \sum_{i=1}^N e^{a_1 x_i} (a_0 e^{a_1 x_i} - y_i) = 0$$

$$f_1 \equiv \frac{\partial Q}{\partial a_1} = \sum_{i=1}^N x_i e^{a_1 x_i} (a_0 e^{a_1 x_i} - y_i) = 0$$

- This is a nonlinear system—we use something like the multivariate root find