Topic #2

Numerical analysis and symbolic computation

Introduction

What is it?

Numerical analysis

Symbolic computation

Which tools can we use?

→ scipy

→ sympy

Background info – David's compendium reloaded!

https://davrot.github.io/pytutorial/

Topics:

- Sympy
- Numerical Integration, Differentiation, and Differential Equations

Which mathematical problems are we interested in?

- Solving equations (only symbolic)
- Integrals over functions
- Derivatives of functions
- Solving differential equations

Numerical solutions will (almost) always be approximations!

- Precision is limited
- Range is limited
- Algorithm is approximating
- Errors can accumulate dramatically (stability of algorithms)

Examples of errors:

- Multiplication, one decimal place: 2.5 * 2.5 = 6.25
- Addition, 8-bit unsigned int: 200+200 = 400
- Euler integration of ODE **(→ Whiteboard)**

Notation:

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} + O(\Delta x^k)$$

…means, if one reduces Δx by a factor of 2, error will reduce by factor 2k

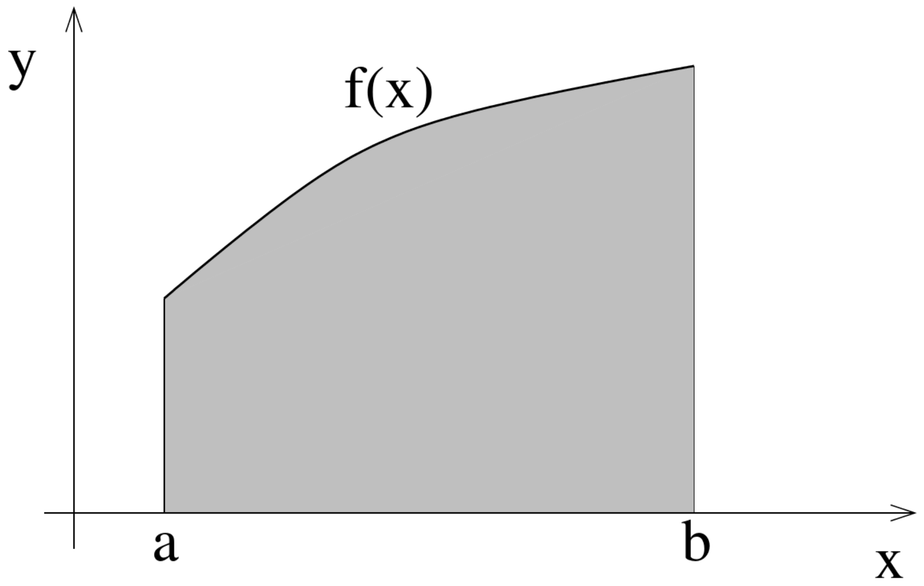
…$O(\Delta x^3)$: reduce Δx by 2, error will reduce by factor $2^3 = 8$

…above example: k=1, approximation is not very good or requires very small Δx

Integrals over functions (‚quadrature‘)

Numerical methods

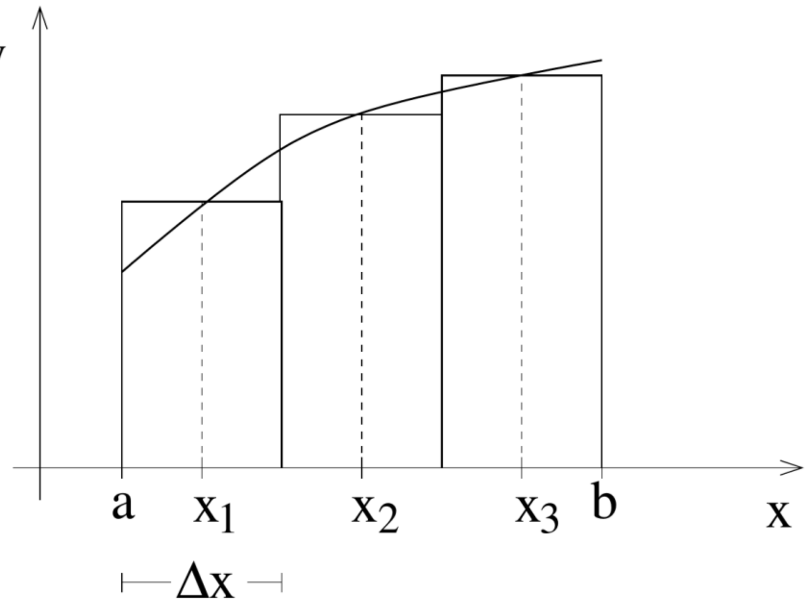
Integral = area under curve $F(a,b) = \int_a^b f(x)dx$



Approximate area by many small boxes, e.g. by ***midpoint rule***:

$$F(a,b) \approx \sum_{i=1}^{N} f(x_i)\Delta x$$

→ Live coding!



Error: $-\frac{\Delta x}{24}\left(f'(b) - f'(a)\right) + O(\Delta x^4)$

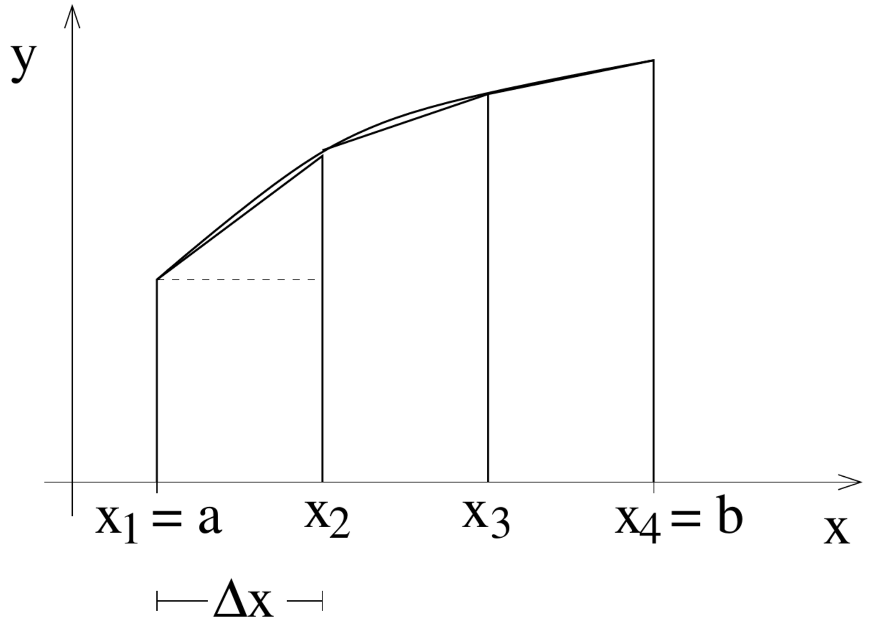
Other rules:

Trapezoidal rule:

$$F(a,b) \approx \frac{1}{2}\left(f(x_1) + f(x_N)\right)\Delta x + \sum_{i=2}^{N-1} f(x_i)\Delta x$$

Error: $-\frac{\Delta x}{12}\left(f'(b) - f'(a)\right) + O(\Delta x^4)$

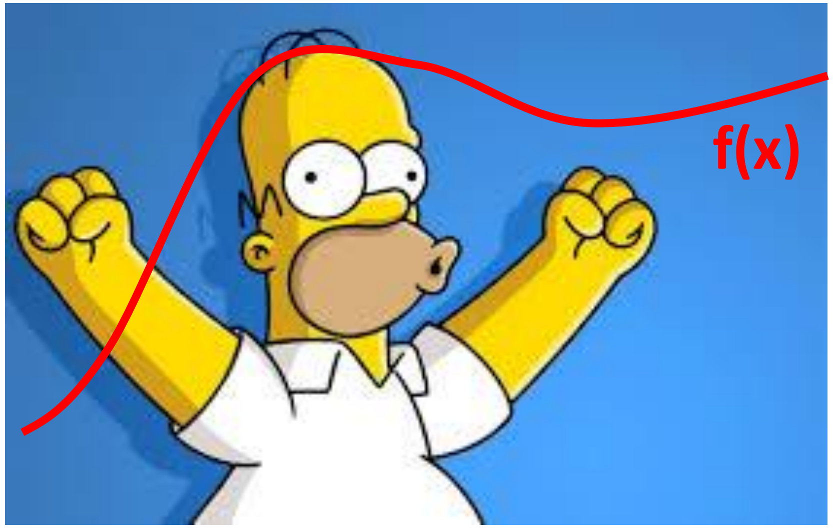
worse than midpoint!



Simpson's rule: approximate by parabolas

→ **Whiteboard**

Error: $O(\Delta x^4)$



Python tools

Numerical methods:

```
scipy.integrate.quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08, limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50, complex_func=False)[source]
```

→ **Live coding!**

Symbolic Methods

We will use module **sympy**.

For symbolic operations (i.e., without concrete numbers), we have to **declare variables/symbols** (and later functions…).

For **mathematical functions such as cos(…)**, use the sympy equivalents (not from math or numpy modules!)

```
import sympy

x, y = sympy.symbols("x y")

y = sympy.integrate(sympy.cos(x), x)
print(y)   # -> sin(x)
```

For **definite integrals**, we can specify boundaries a and b by **creating a tuple (x, a, b)** for the second argument.

The solution can be **evaluated** by using the methods **.subs(variable, value)** to substitute a value for a variable and **.evalf()** to get a numerical output.

→ **Live coding!**

„Genug für heute?“

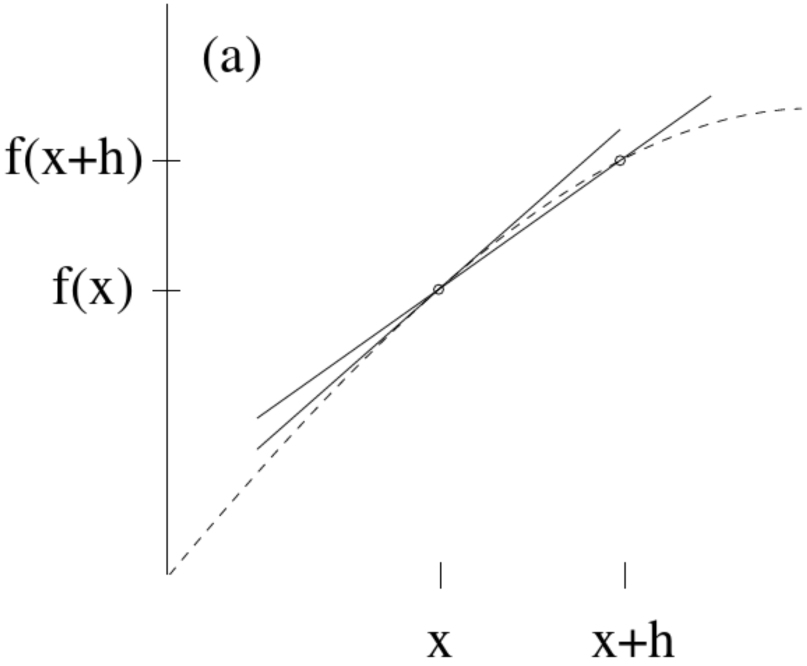
https://davrot.github.io/pytutorial/sympy/intro/
https://davrot.github.io/pytutorial/numpy/7/
https://davrot.github.io/pytutorial/numpy/8/

Differentiation of functions

Example live-coding:
integration and
differentiation,
stability and instability

Numerical methods:

right-sided differentiation



$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Symbolic methods:

For differentiation, the corresponding command is **diff**:

```
import sympy

x, y = sympy.symbols("x y")

y = sympy.diff(sympy.sin(x) * sympy.exp(x), x)
print(y) # -> exp(x)*sin(x) + exp(x)*cos(x)
```

Integration of differential equations

Differential quotient approximated by finite difference, like in previous example.
Solution constructed by considering the following aspects: → **Whiteboard!**

- What do we want to know, what is known?
- Where do we start? → **Initial value problem…**
- How far do we step? → Smaller than fastest timescale implies **maximum step size**

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t) \quad \text{with} \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0$$

→ Euler: $\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \Delta t \, \boldsymbol{f}(\boldsymbol{x}(t), t) + O(\Delta t^2)$

Warning:

- differentiation/integration of functions can be performed in parallel,
- differential equations require an iterative solution which can not be parallelized!

What about systems of differential equations?

…just solve them in parallel (see previous slide) → **Whiteboard!**

Higher-order methods

Idea: approximate differential quotient more precisely…

Solution (Runge-Kutta 2nd order): → **Whiteboard**

- Go ahead with Euler by half of the stepsize…

$$\boldsymbol{x}_{mid}(t + \Delta t/2) = \boldsymbol{x}(t) + \Delta t/2\ \boldsymbol{f}(\boldsymbol{x}(t), t)$$

- …use slope at that position for an Euler with the full stepsize.

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \Delta t\ \boldsymbol{f}(\boldsymbol{x}_{mid}(t + \Delta t/2), t + \Delta t/2) + O(\Delta t^3)$$

This idea can be extended, for example to obtain the **Runge-Kutta scheme of order 4**…

In addition, the **stepsize $\Delta \boldsymbol{t}$ can be adapted** by comparing errors made by a scheme of order N and scheme of order N+1 (e.g. „**Runge-Kutta 45**“)

Python Tools

Numerical methods:

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False,
events=None, vectorized=False, args=None, **options)
```

→ **Live coding**

Solvers:

'RK45' (default)	Explicit Runge-Kutta method of order 5(4). The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula (local extrapolation is done). A quartic interpolation polynomial is used for the dense output. Can be applied in the complex domain.
'RK23'	Explicit Runge-Kutta method of order 3(2). The error is controlled assuming accuracy of the second-order method, but steps are taken using the third-order accurate formula (local extrapolation is done). A cubic Hermite polynomial is used for the dense output. Can be applied in the complex domain.
'DOP853'	Explicit Runge-Kutta method of order 8. Python implementation of the "DOP853" algorithm originally written in Fortran. A 7-th order interpolation polynomial accurate to 7-th order is used for the dense output. Can be applied in the complex domain.
'Radau'	Implicit Runge-Kutta method of the Radau IIA family of order 5. The error is controlled with a third-order accurate embedded formula. A cubic polynomial which satisfies the collocation conditions is used for the dense output.
'BDF'	Implicit multi-step variable-order (1 to 5) method based on a backward differentiation formula for the derivative approximation. A quasi-constant step scheme is used and accuracy is enhanced using the NDF modification. Can be applied in the complex domain.
'LSODA'	Adams/BDF method with automatic stiffness detection and switching. This is a wrapper of the Fortran solver from ODEPACK.

Symbolic methods:

In addition to declaring variables, you need…

…to **declare functions** (for the solution we are looking for)

…to **define the (differential) equation**

…and the **command** dsolve for (trying to) solve the DEQ:

```
import sympy

# Undefined functions
f = sympy.symbols("f", cls=sympy.Function)

x = sympy.symbols("x")


diffeq = sympy.Eq(f(x).diff(x, x) - 2 * f(x).diff(x) + f(x), sympy.sin(x))

print(diffeq)  # -> Eq(f(x) - 2*Derivative(f(x), x) + Derivative(f(x), (x, 2)), sin(x))

result = sympy.dsolve(diffeq, f(x))
print(result)  # -> Eq(f(x), (C1 + C2*x)*exp(x) + cos(x)/2)
```

Symbolic methods, cont'd…

- For including initial conditions, **dsolve** has the **optional argument ics**.
- With **lambdify**, You can **convert the RHS of the solution to a normal numpy function**:
- Query the new function as to **which arguments it takes**, and in which order (**import inspect** for that purpose)

→ **Live coding**

```
result = sympy.dsolve(diffeq, f(x))

symbols = list(result.rhs.free_symbols)

f = sympy.lambdify(symbols, result.rhs, "numpy")


print("The arguments of the result:")
print(inspect.getfullargspec(f).args)
print("The source code behind f:")
print(inspect.getsource(f))
```

What about partial differential equations?

For example, the cable equation: → **Whiteboard**

$$\frac{\partial V(t,x)}{\partial t} = a\frac{\partial^2 V(t,x)}{\partial x^2} + bV(t,x) + I_{ext}(t,x)$$

More information:

https://davrot.github.io/pytutorial/sympy/intro/
https://davrot.github.io/pytutorial/numpy/7/
https://davrot.github.io/pytutorial/numpy/8/