Numerical Integration (in Chemistry)

Guido Falk von Rudorff

Goal

Evaluate a proper integral

$$\int_a^b f(x)dx \simeq \sum_i \alpha_i f(x_i)$$

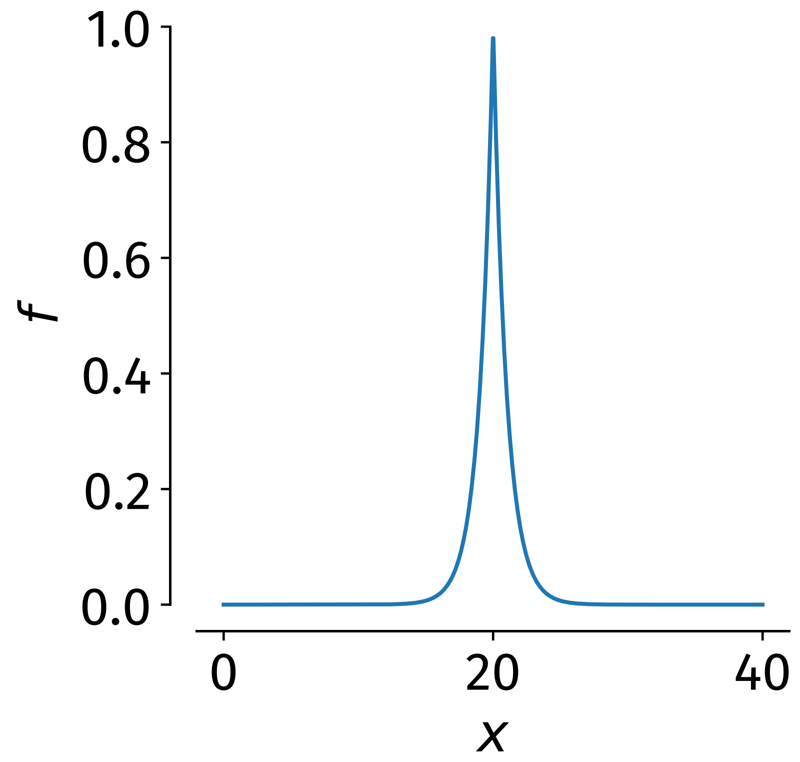
How

- Replace integral by weighted sum
- Be clever about weights and positions

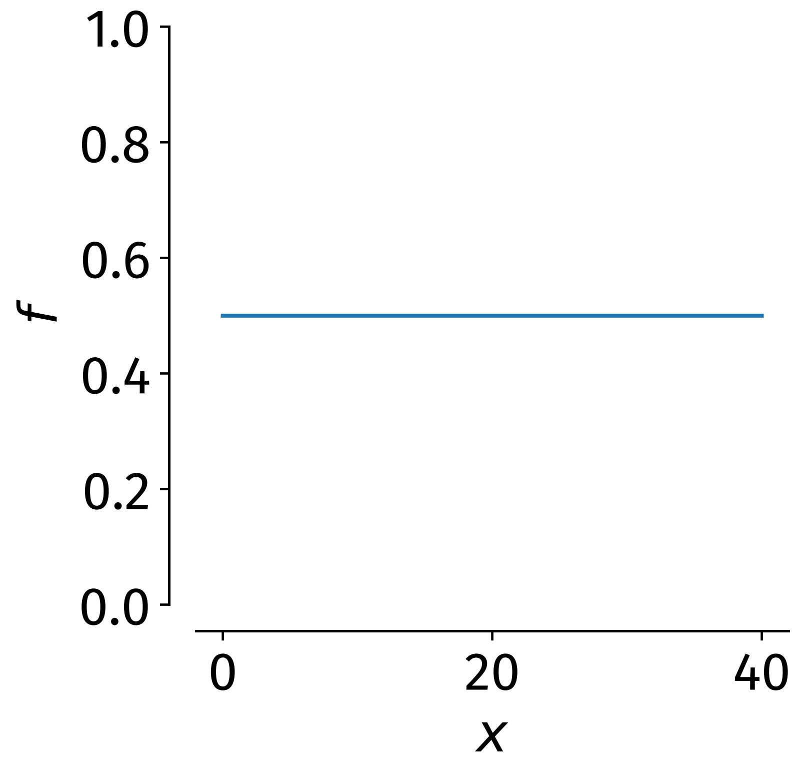
Why *numerical* integration?

- No analytical expression available (most calculations don't have one)
- Expensive evaluations of f

Challenge



Find relevant integration points…



…but don’t use too many of them

Challenge in Chemistry

Electron densities in molecules

- Highly peaked (Kato's cusp condition)

$$Z_I \propto \left. \frac{d\rho(\mathbf{r})}{dr} \right|_{r \to \mathbf{R_I}}$$

- Integrals relate to dipole moments, ionic forces, ...:

$$\mu = \int d\mathbf{r} \rho \mathbf{r}$$

$$\mathbf{F}_I = Z_I \int d\mathbf{r} \rho \frac{\mathbf{r} - \mathbf{R}_I}{|\mathbf{r} - \mathbf{R}_I|^3}$$

$$Q_{ij} = \int d\mathbf{r} \rho \left(3 r_i r_j - |\mathbf{r}|^2 \delta_{ij} \right)$$

- Obtaining the density at one point is expensive

Line shapes in experiments

- Extremely narrow functions

Basic concept

$$\int_a^b f(x)dx \simeq \sum_i \alpha_i f(x_i)$$

Take a function f

- Bounded
- Smooth
- Defined over interval [*a*, *b*]

Approximate f with polynomials

- Typically *not* Taylor expansions
- Different kinds of polynomials are used
- Main reason: can be integrated exactly
- Commonly: approximate subintervals separately (“composite integration”)

Integrate polynomials

- Take their values at different points x_i
- Calculate weights from the points

Derivation

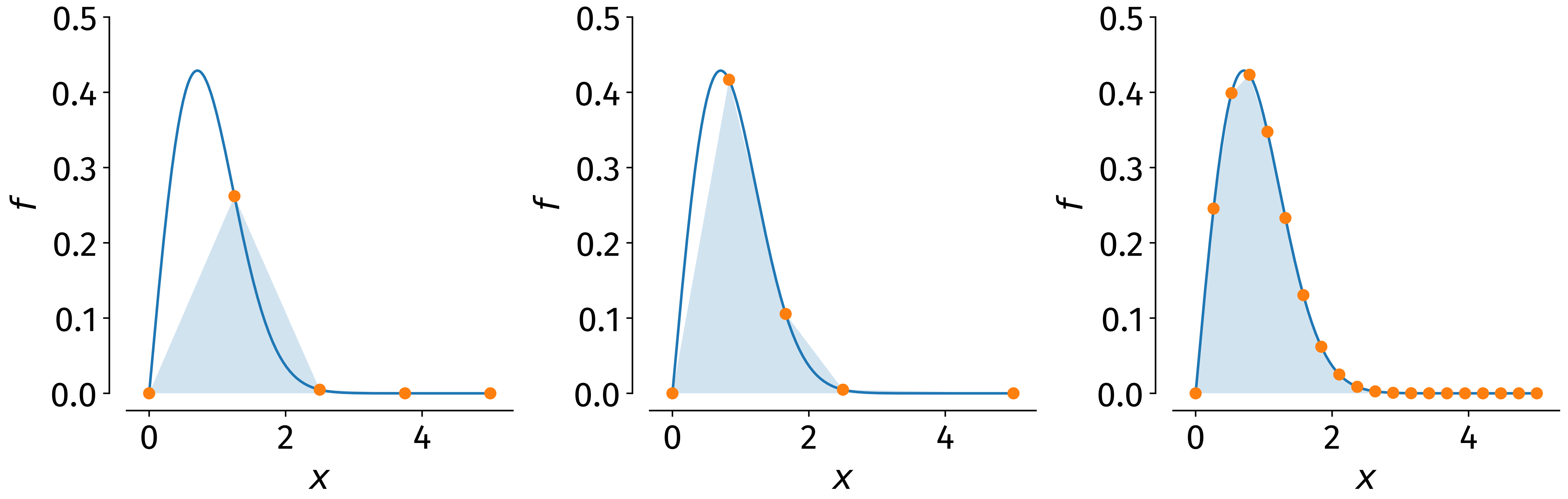
- Polynomial approximation, points -> weights

Errors

Points x_i

- More points, smaller errors (improvement varies with method though)
- Better coverage, smaller errors

$$f(x) = x\exp(-x^2)$$

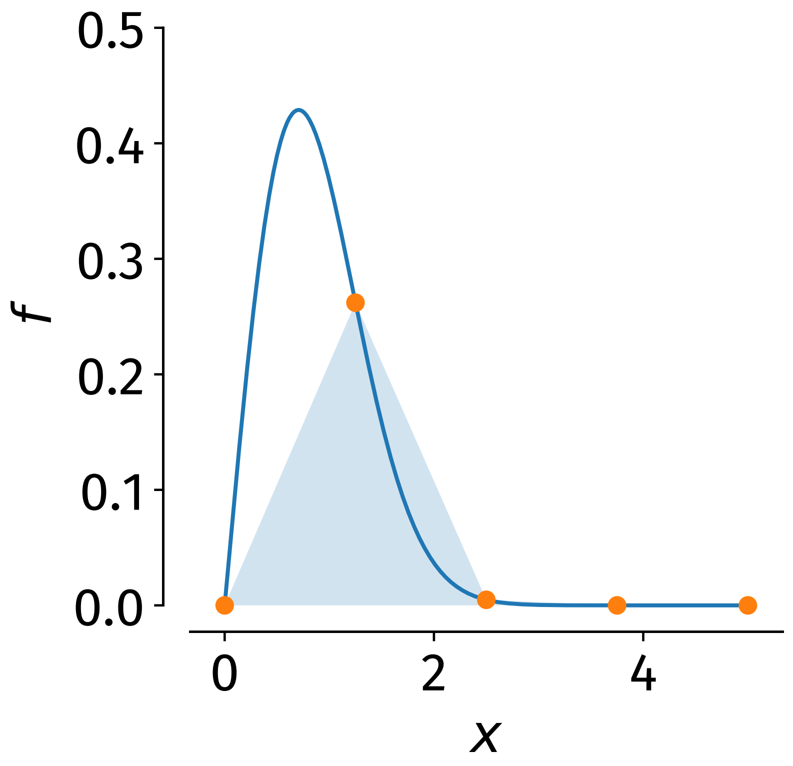


Errors

Polynomial approximation

- Approximation up to *n*-th order does not include higher order contributions
- In leading order: maximal *n+1*-th derivative in [*a*, *b*] gives error bound
- Smaller intervals always reduce errors

Can be bounded



Machine precision

Finite precision of data types

- Exists in all programming languages
- Hardware limit for performance reasons
- Math is done in base 2
  (so 0.1 is inexact for computers)
- Only floating-point calculations
- Most problematic: summation and multiplication
- Another reason to use libraries

Workarounds

- Rational numbers (as integers are exact)
- Group summations
- Sort before summation

In Python

- *math.fsum()* for better summation
- Library *mpmath* for arbitrary precision (as long as you have memory and patience)

Python

```python
a = 1.
b = a + 1e-10
print (1 - b)
```

```
-1.000000082740371e-10
```

```python
a = 1.
b = a + 1e-20
print (1 - b)
```

```
0.0
```

C

```c
#include <stdio.h>

int main() {
        double a = 1.;
        double b = a + 1e-10;
        printf("%.15e\n", 1-b);
        b = a + 1e-20;
        printf("%.15e\n", 1-b);
}
```

```
-1.000000082740371e-10
0.000000000000000e+00
```

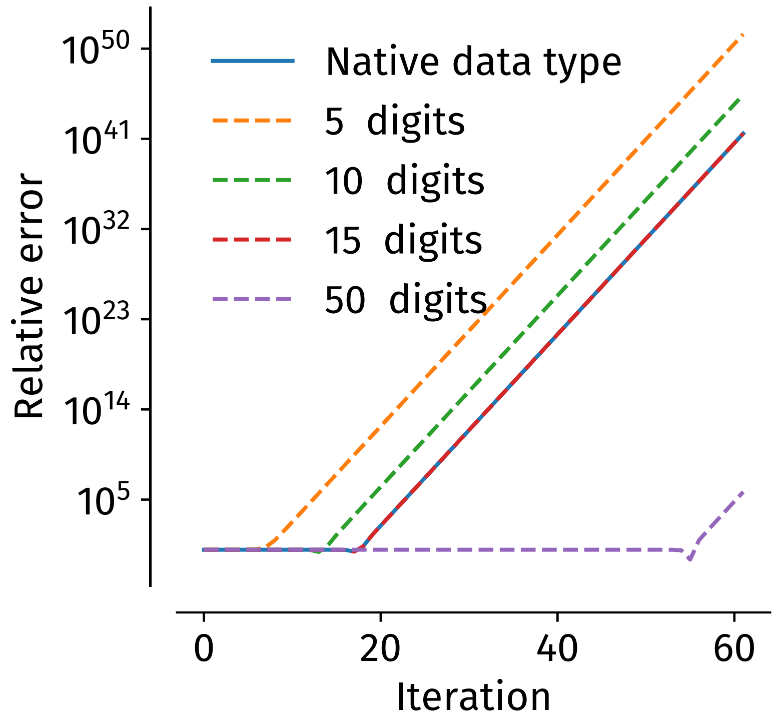
Machine precision

Example

```python
import mpmath as mp

def iteration_native_types(count):
    a_n = [1, 1/3]

    for i in range(count):
        a_n.append(10*a_n[-1]/3 - a_n[-2])

    return a_n

def iteration_mpmath(count, precision):
    mp.mp.dps = precision
    a_n = [mp.mpf('1'), mp.mpf('1')/mp.mpf('3')]

    for i in range(count):
        a_n.append(mp.mpf('10')*a_n[-1]/mp.mpf('3') - a_n[-2])

    return a_n

def exact(count):
    mp.mp.dps = 100
    a_n = [mp.mpf('1'), mp.mpf('1')/mp.mpf('3')]

    for i in range(count):
        a_n.append(a_n[-1] *  mp.mpf('1')/mp.mpf('3'))

    return a_n
```

$$a_n \equiv \frac{10}{3} a_{n-1} - a_{n-2}$$

$$a_0 \equiv 1, a_1 \equiv \frac{1}{3} \Rightarrow a_n = \frac{1}{3^n}$$



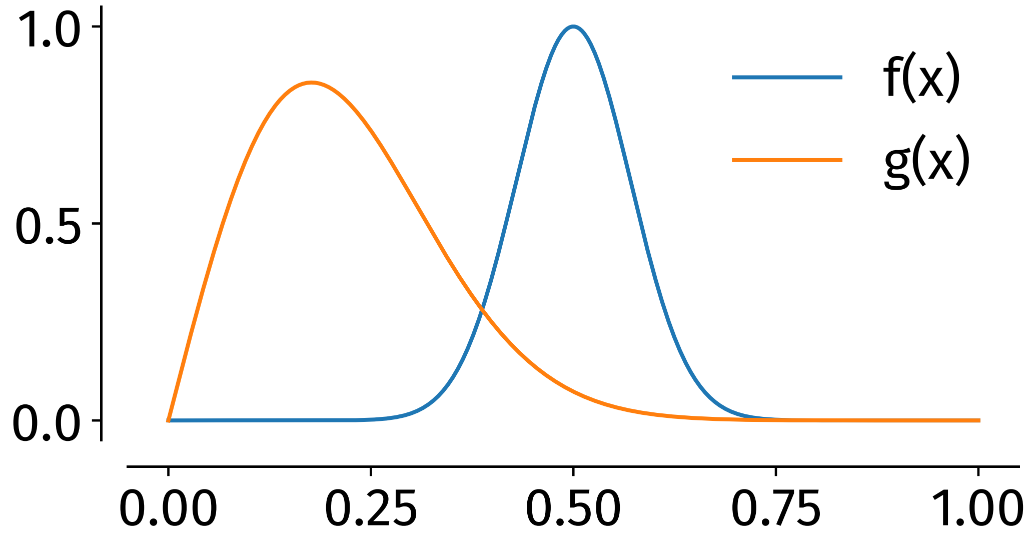
Method Overview

Methods for regular functions

- Newton-Cotes	equidistant points
- Gauss	non-equidistant but predefined points
- Trapezoidal rule	arbitrary points
- Monte Carlo	random points

Methods for molecules

- Becke-Lebedev grids	Follows electron density distribution



$$f(x) = \exp(-100(x-0.5)^2)$$

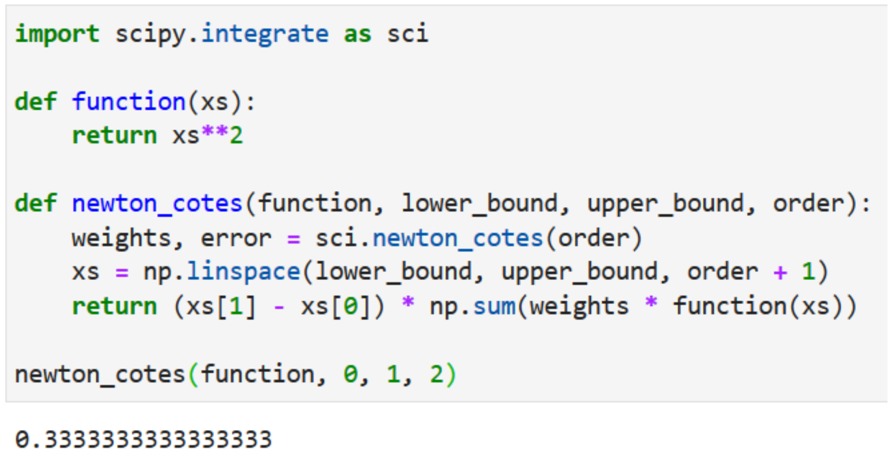
$$g(x) = 8x\exp(-16x^2)$$

Newton-Cotes

Approximation
- Based on Lagrange polynomials
- `scipy.integrate.newton_cotes`

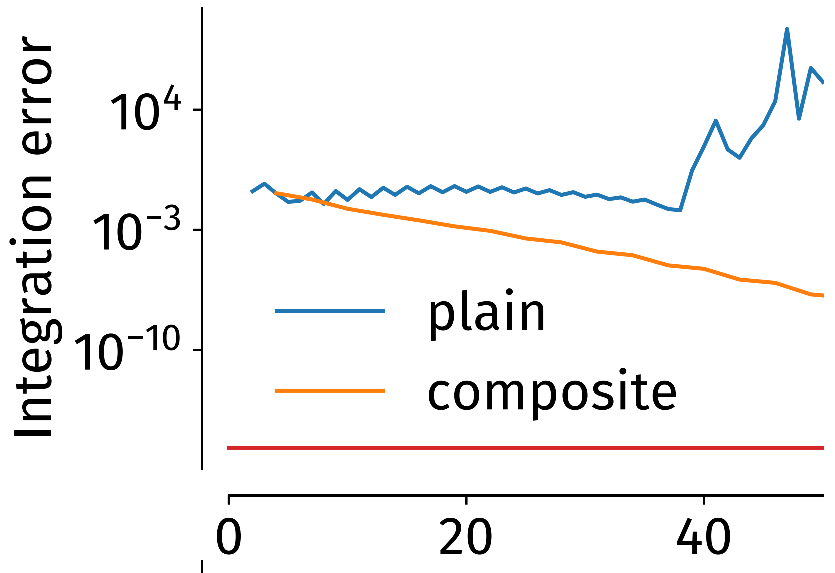
When to use
- Well-behaved curves
- Small integration domains

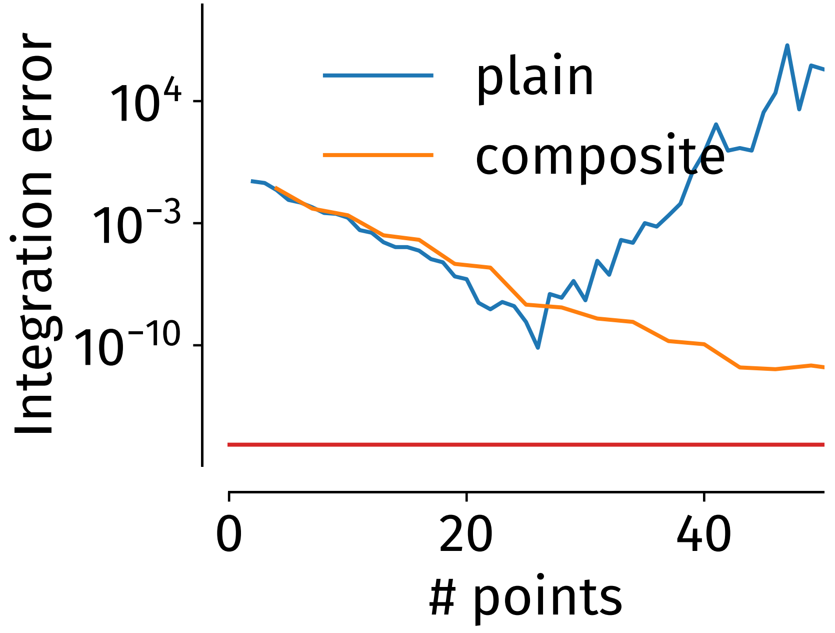
Caveats
- High orders unstable: prefer smaller domains



```python
import scipy.integrate as sci

def function(xs):
    return xs**2

def newton_cotes(function, lower_bound, upper_bound, order):
    weights, error = sci.newton_cotes(order)
    xs = np.linspace(lower_bound, upper_bound, order + 1)
    return (xs[1] - xs[0]) * np.sum(weights * function(xs))

newton_cotes(function, 0, 1, 2)
```

```
0.3333333333333333
```





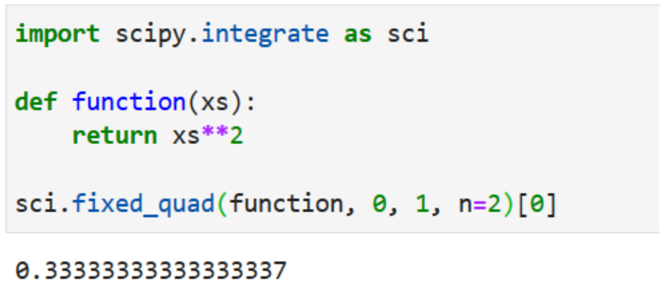
Gauss

Approximation

- Based on Legendre polynomials
- `scipy.integrate.fixed_quad`

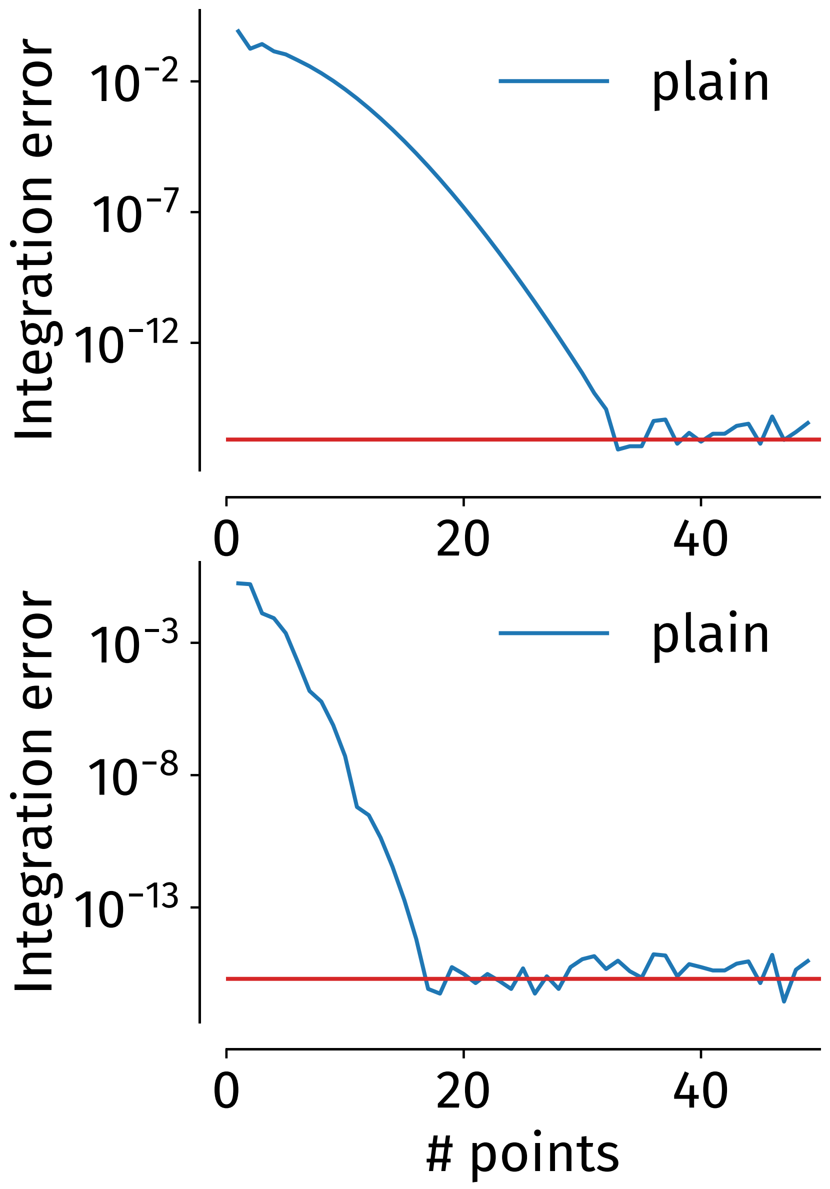
When to use

- Endpoints exist



```
import scipy.integrate as sci

def function(xs):
    return xs**2

sci.fixed_quad(function, 0, 1, n=2)[0]
```

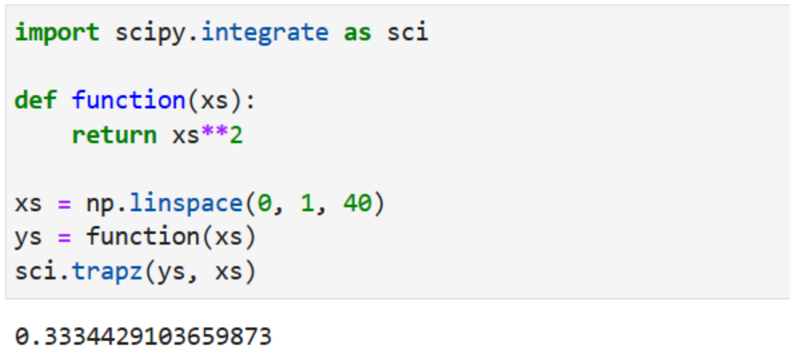
```
0.33333333333333337
```



Trapezoidal Rule

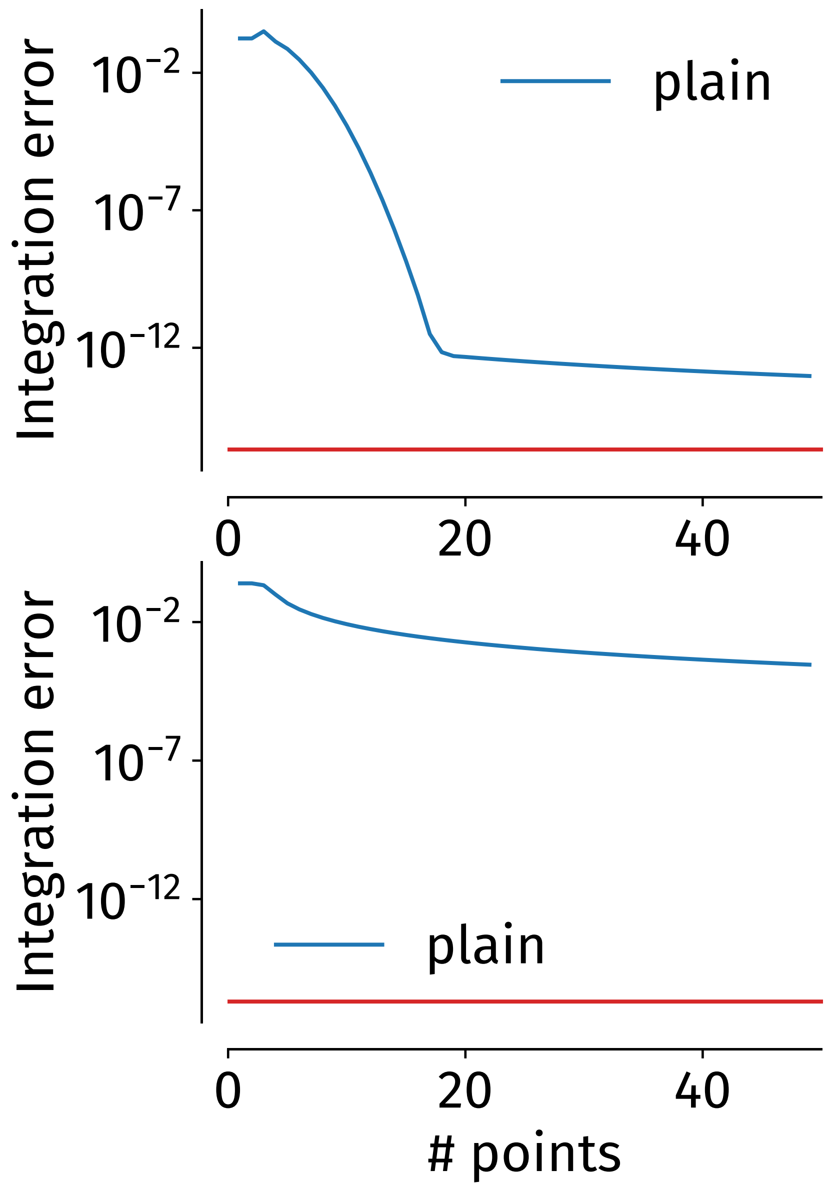
Approximation
- Based on linear functions
- `scipy.integrate.trapz`

When to use
- Periodic functions



```
import scipy.integrate as sci

def function(xs):
    return xs**2

xs = np.linspace(0, 1, 40)
ys = function(xs)
sci.trapz(ys, xs)
```

```
0.3334429103659873
```



Monte Carlo

Approximation

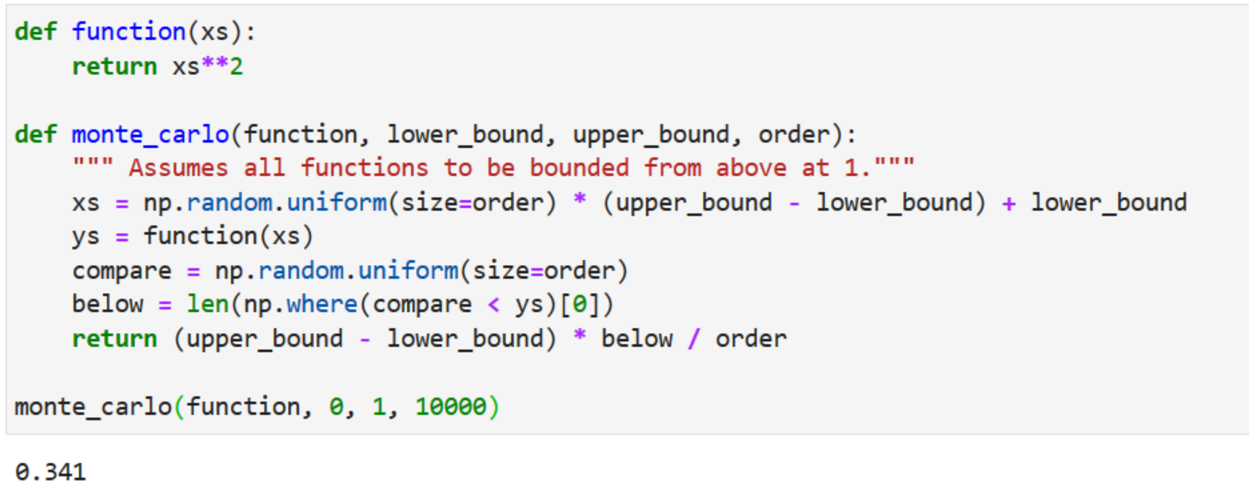
- Random points

When to use

- High-dimensional
- Highly irregular functions
- Evaluation cheap

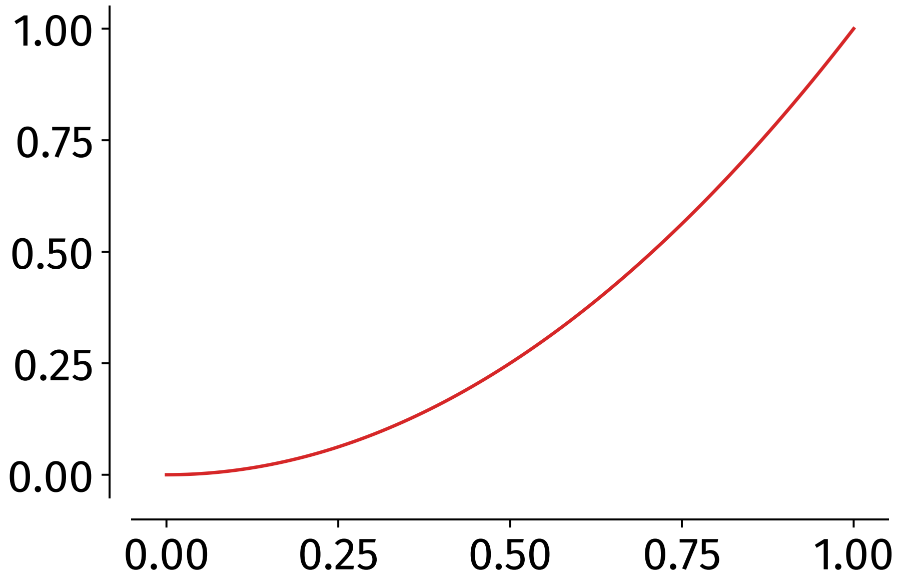
Caveat

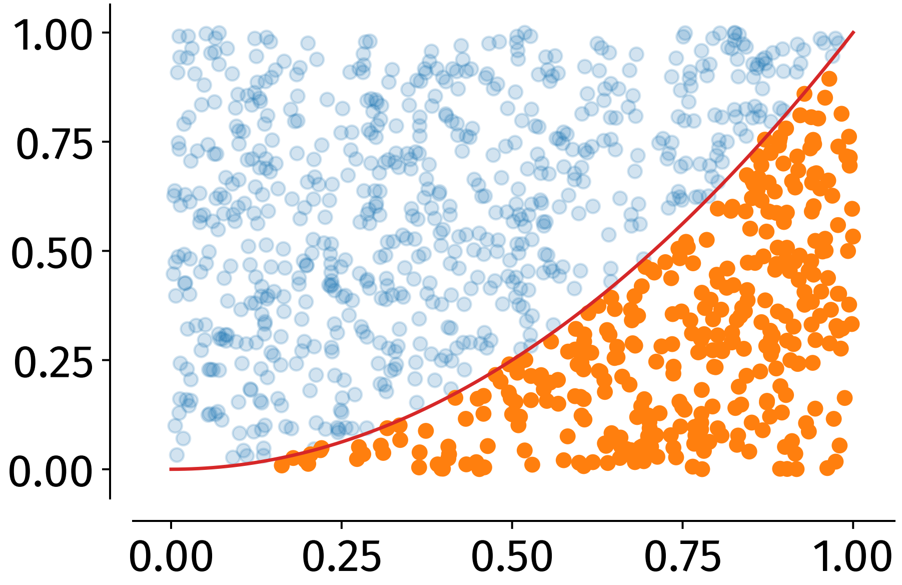
- Slow convergence



```
def function(xs):
    return xs**2

def monte_carlo(function, lower_bound, upper_bound, order):
    """ Assumes all functions to be bounded from above at 1."""
    xs = np.random.uniform(size=order) * (upper_bound - lower_bound) + lower_bound
    ys = function(xs)
    compare = np.random.uniform(size=order)
    below = len(np.where(compare < ys)[0])
    return (upper_bound - lower_bound) * below / order

monte_carlo(function, 0, 1, 10000)
```

```
0.341
```

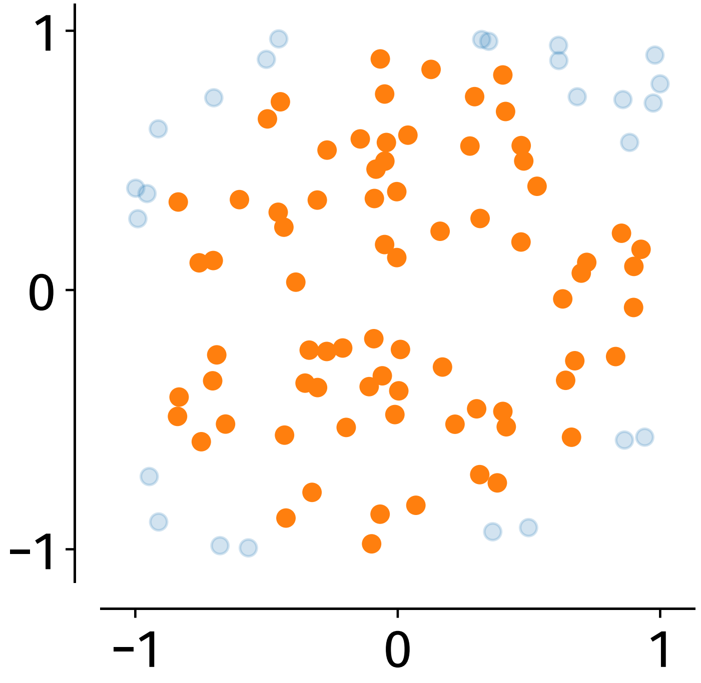


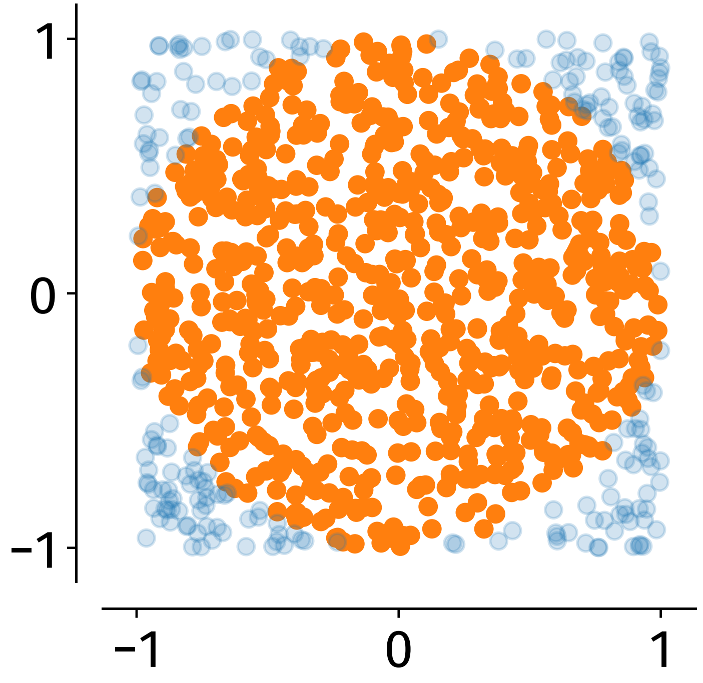


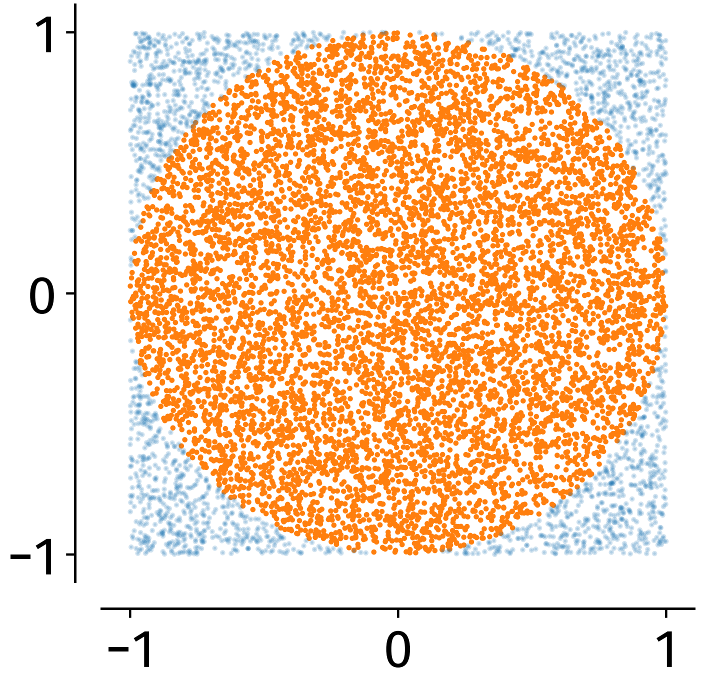
Monte Carlo

Slow convergence

- Compare base area (blue rectangle) to integral area (orange)
- Random numbers can be expensive
- Method of last resort







Pitfalls

Functions of local support

- Line shapes
- Hard to find relevant region

High local curvature

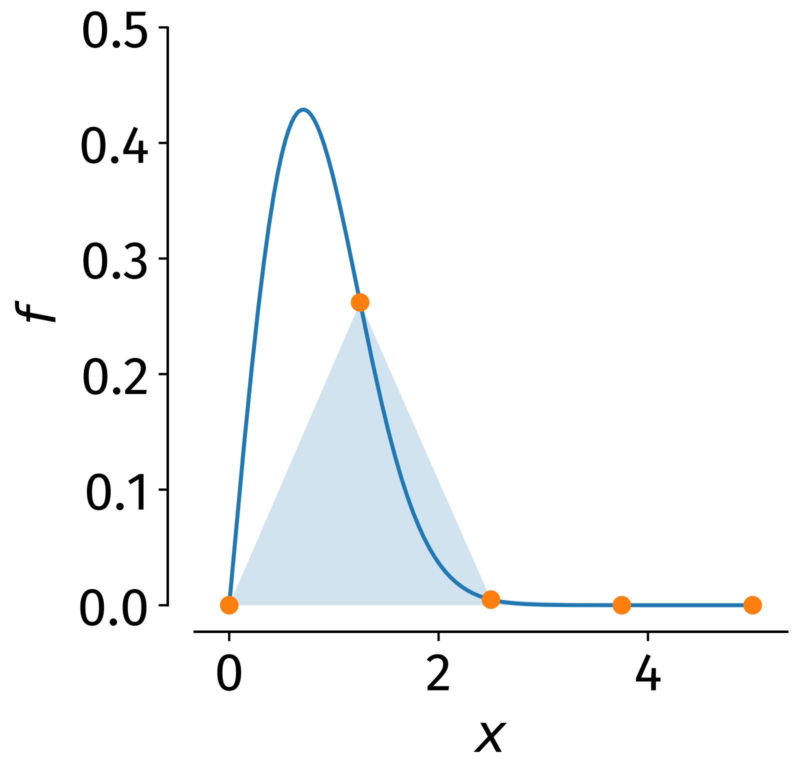
- Hard to capture with polynomials

Higher dimensions

- Grid scaling in N dimensions: g^N

High orders

- Runge's phenomenon



Pitfalls

Functions of local support

- Line shapes
- Hard to find relevant region

High local curvature

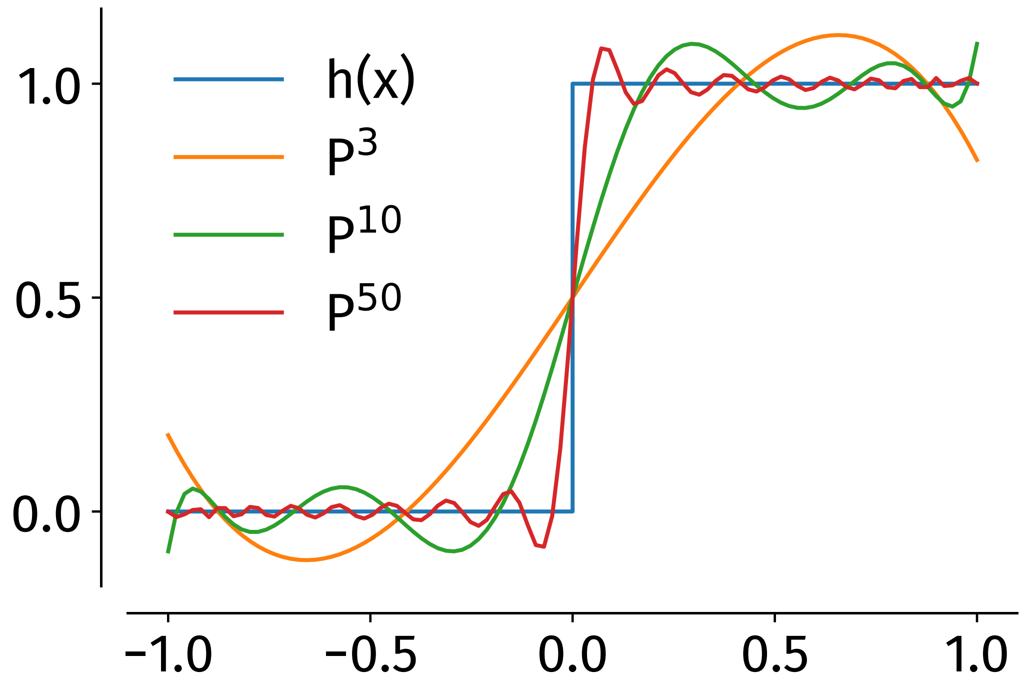
- Hard to capture with polynomials

Higher dimensions

- Grid scaling in N dimensions: g^N

High orders

- Runge's phenomenon



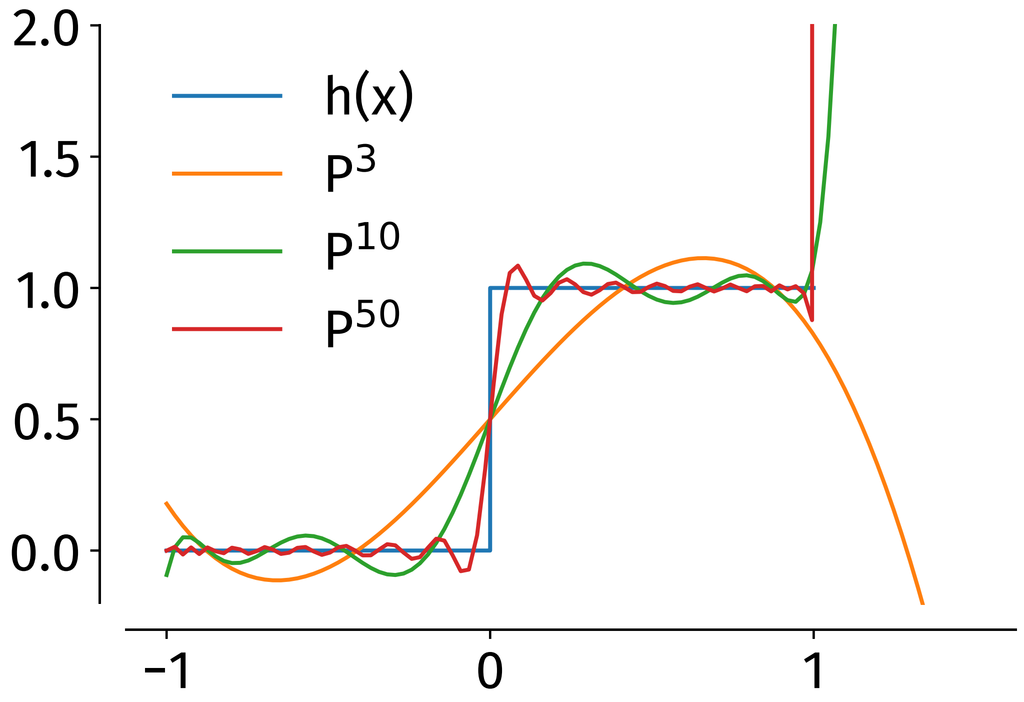
Pitfalls

Functions of local support
- Line shapes
- Hard to find relevant region

High local curvature
- Hard to capture with polynomials

Higher dimensions
- Grid scaling in N dimensions: g^N

High orders
- Runge’s phenomenon



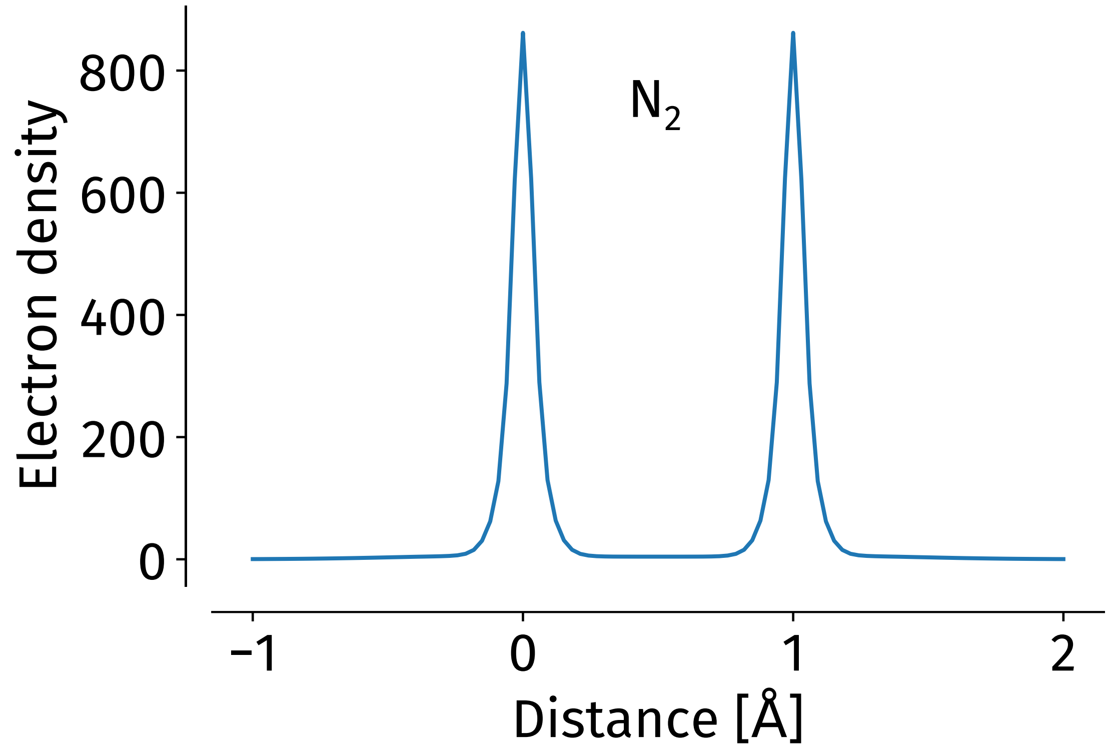
Molecules

Electron densities are peaked

- Local support
- Exponential decay
- Regular grids unsuitable
- Inherently 3D

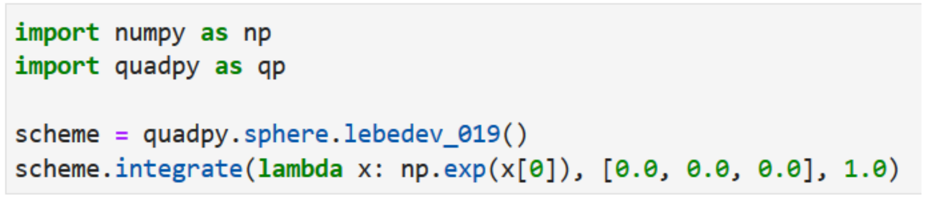
Custom grids (Becke-Lebedev)

- Spheres around atoms (Lebedev)
- Radii exponentially spaced (Becke)



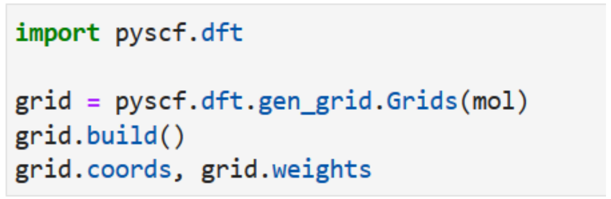
Molecules

Spherical and other grids
- quadpy package
- *pip install quadpy*



```python
import numpy as np
import quadpy as qp

scheme = quadpy.sphere.lebedev_019()
scheme.integrate(lambda x: np.exp(x[0]), [0.0, 0.0, 0.0], 1.0)
```

Molecular grids
- PySCF package
- *pip install pyscf*



```python
import pyscf.dft

grid = pyscf.dft.gen_grid.Grids(mol)
grid.build()
grid.coords, grid.weights
```

Summary

Methods

- Newton-Cotes
- Gauss
- Trapezoidal rule
- Monte Carlo
- Becke-Lebedev

Caveats

- High dimensions
- Local support
- High curvature

Python

- Numerical integration of functions
- … and for molecular geometries

ferchault @ferchault guido.vonrudorff.de