Interpolation (in Chemistry)

Guido Falk von Rudorff

Motivation

Potential energy surfaces

- Underlying function expensive
- Intermediate values required

Curve fitting

- Simpler / representative function
- Computationally more efficient evaluation of arbitrary functions
- Analyse coefficients
  - Curvature of potentials
  - Particle diffusion

Typical approach

1D case

- Polynomials (possibly piecewise)
  - Cubic functions, mostly
- Differentiable, and derivatives are used for boundary conditions
- Simple functional form: fast

Higher dimensional case

- Nearest neighbor value (fast, and increasingly efficient in higher dimensions)
- Basis functions
  - Radial basis function (RBF) in *scipy*
  - Hard to accelerate
- Volume “close by” decreases quickly with higher dimensions
  - Careful as to whether it makes sense at all

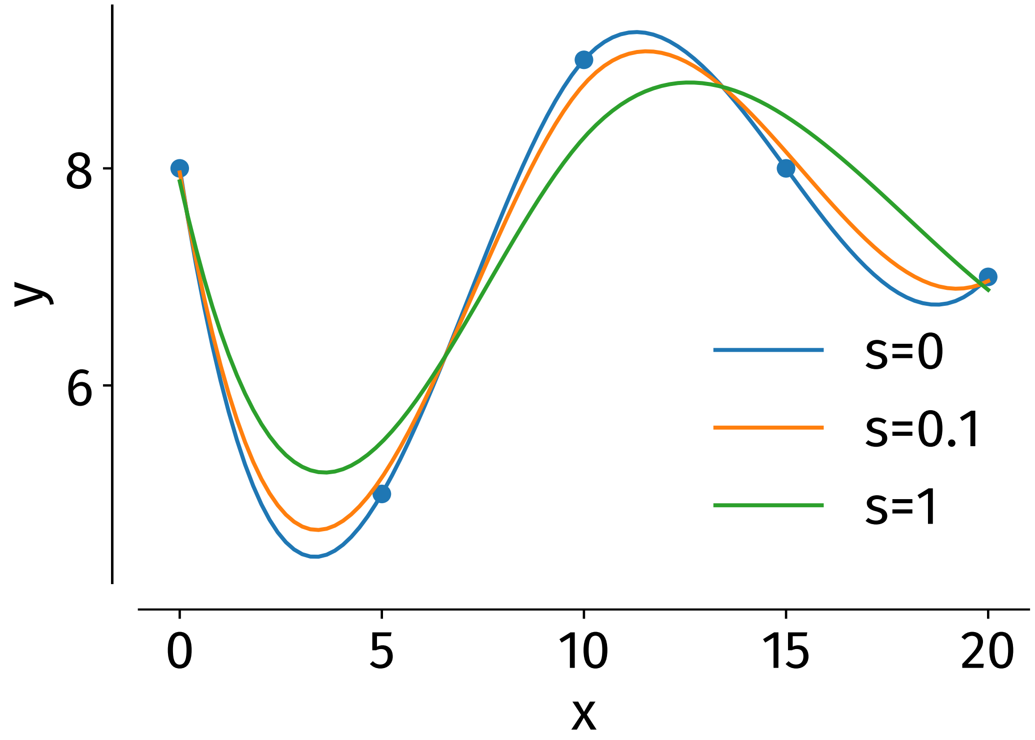
Main classes

Exact

- At the reference points, the interpolant has the reference values
- Prone to overfitting
- Noise problematic
- Can be numerically unstable

Approximate

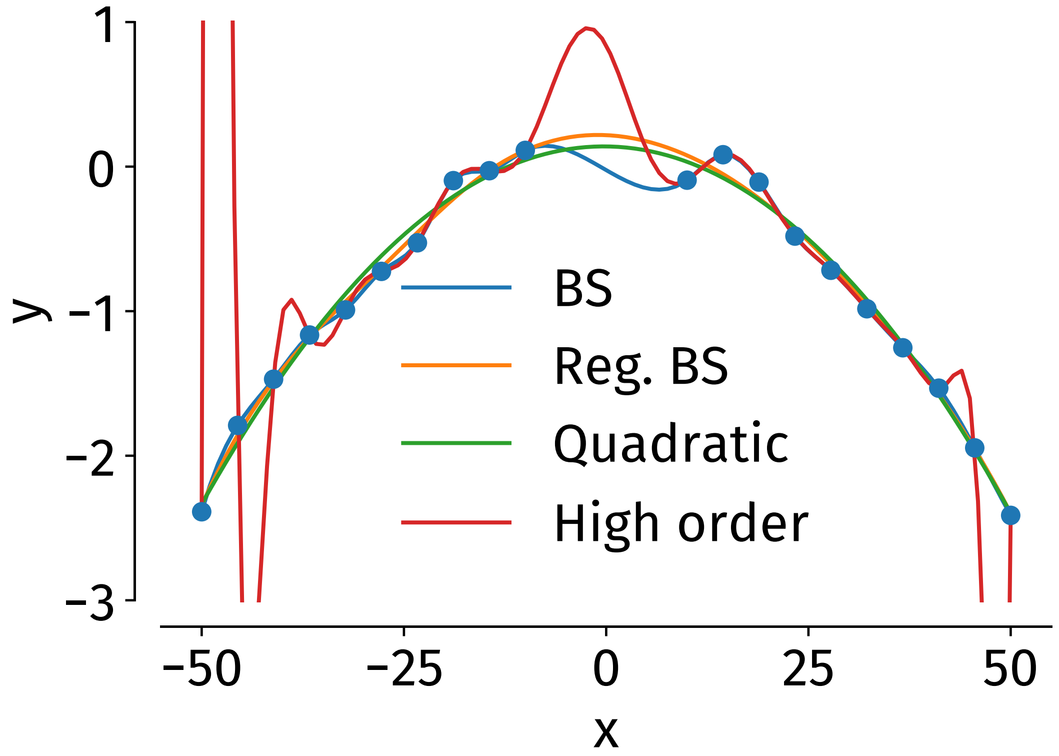
- (Almost) close to the reference values
- More regularized (Parameter *s*)
- Noise acceptable
- If oversimplified: wrong results



Overfitting

Danger of misleading results

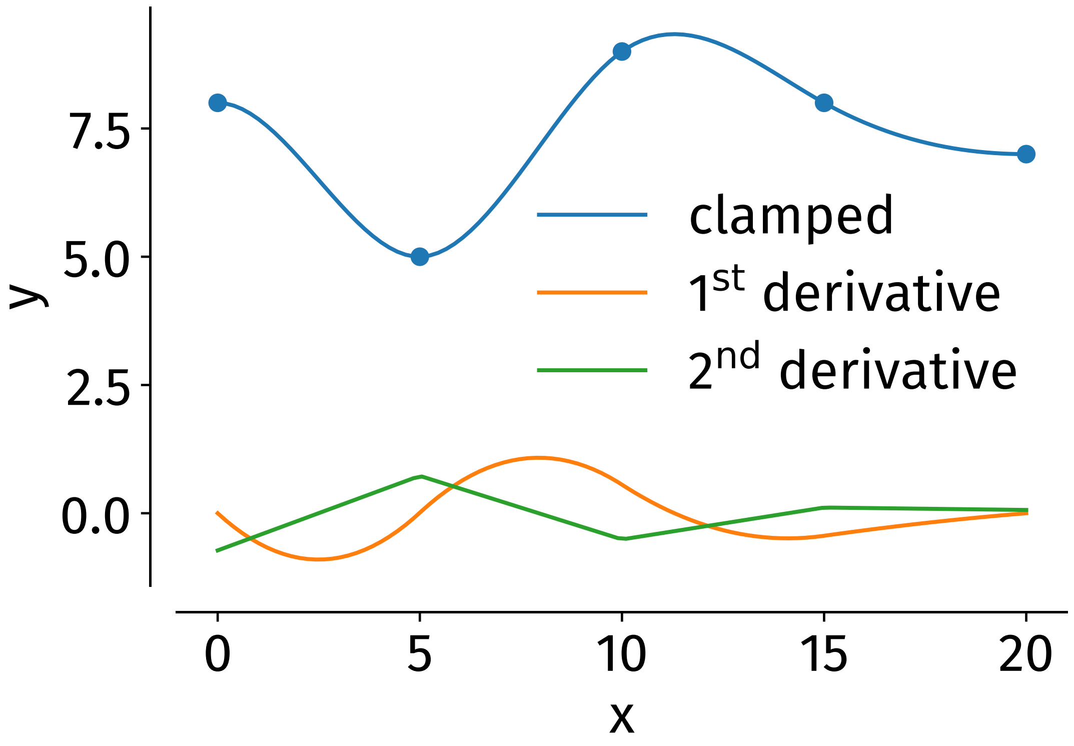
- High order polynomials are ill-conditioned
  - Runge's phenomenon
  - Do not pick them
- Best fit comes from the underlying system
  - Here: quadratic model
- Piecewise B-splines (BS)
  - Well-behaved but not error free
  - At least fewer numerical errors
- Regularized B-splines (Reg. BS)
  - Less subject to noise



Families

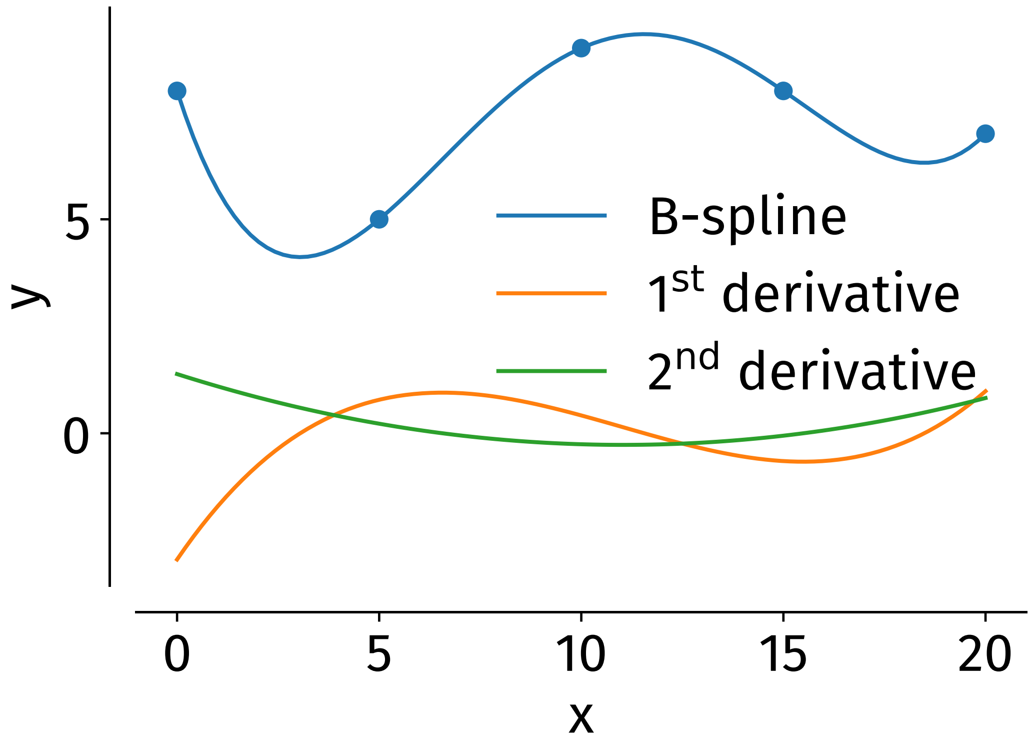
Cubic splines

- Piecewise polynomials
- Match first and second derivative
- Only first derivative smooth
- No requirement for particular spacing



B-spline

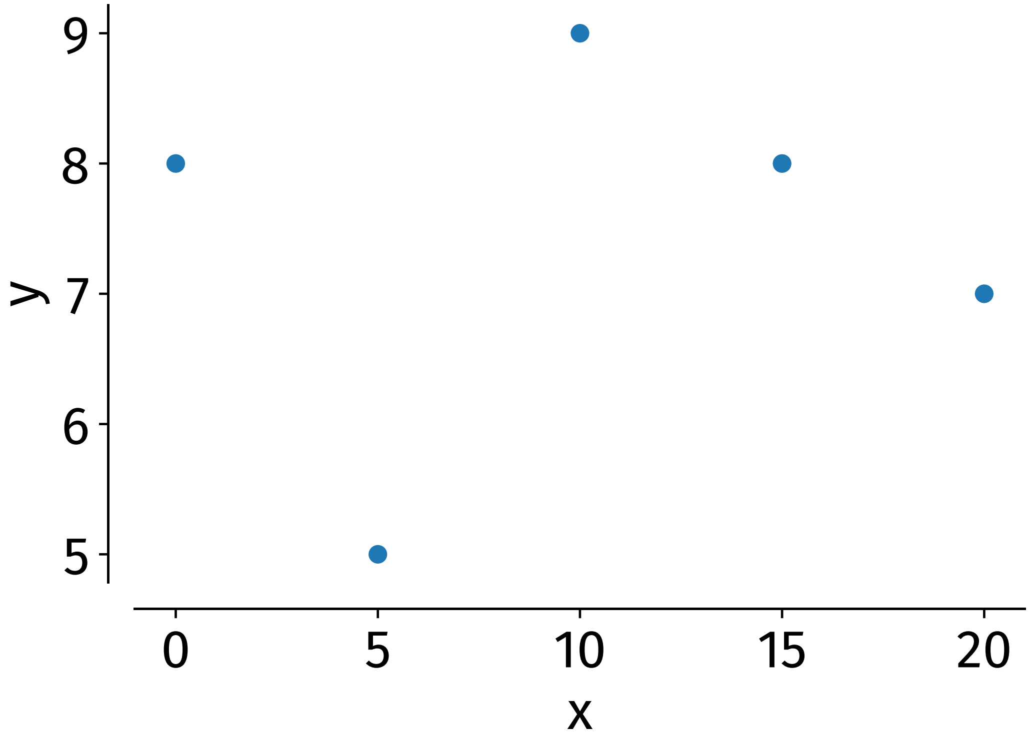
- Piecewise polynomial order n
- Match n-2 first derivatives
- Control points rather than data points
- No requirement for particular spacing



Python

Interpolation
- `scipy.interpolate`
- Many interfaces available

Groups
- Cubic splines (Exact)
- B-splines (Approximate)

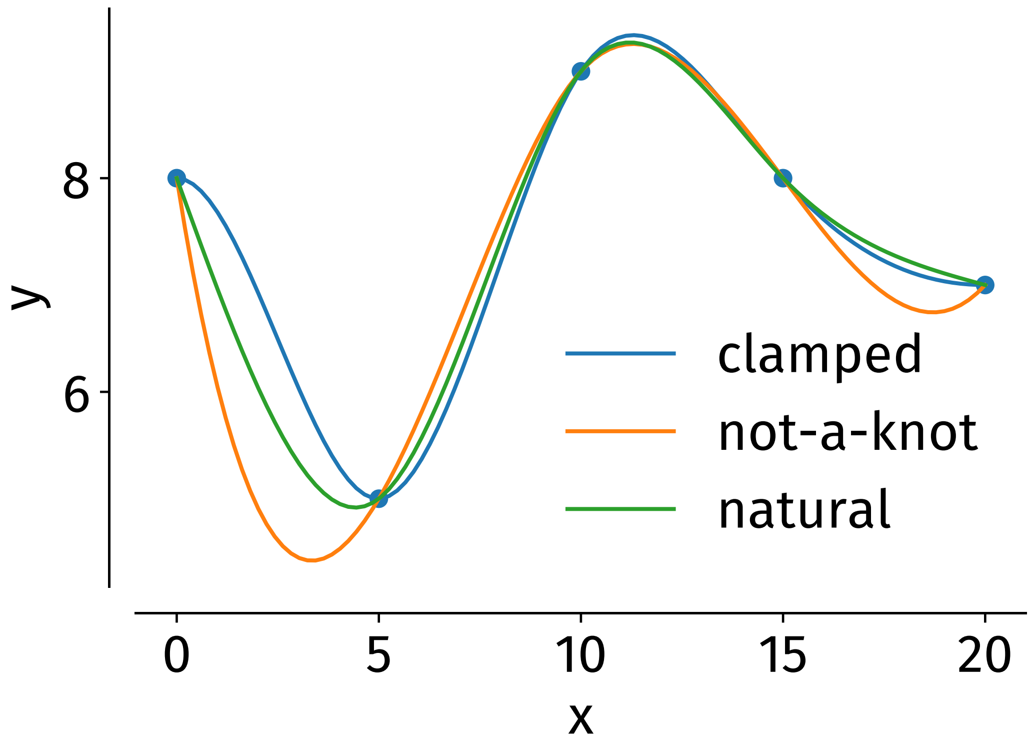


Python: cubic splines

Code

```python
import scipy.interpolate as sci
import numpy as np

xs = (0, 5, 10, 15, 20)
ys = (8, 5, 9, 8, 7)

xss = np.linspace(0, 20, 100)
plt.scatter(xs, ys)
cspline = sci.CubicSpline(xs, ys, bc_type="not-a-knot")
plt.plot(xss, cspline(xss))
```

- Boundary conditions important
  - *clamped*: First derivative 0
  - *not-a-knot*: first and second polynomial are the same
  - *natural*: Second derivative 0
  - *periodic*: If data is periodic
- Object can be called like a function

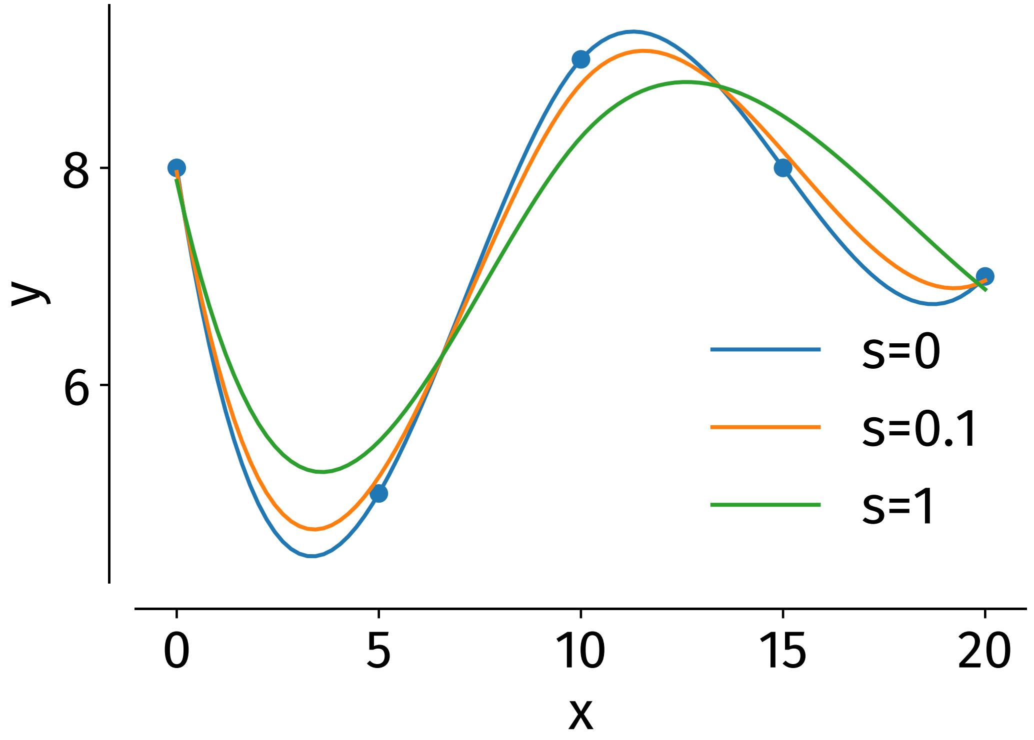


Python: B-splines

Code

```
import scipy.interpolate as sci
import numpy as np

xs = (0, 5, 10, 15, 20)
ys = (8, 5, 9, 8, 7)

xss = np.linspace(0, 20, 100)
plt.scatter(xs, ys)
bspline = sci.UnivariateSpline(xs, ys, s=0.1)
plt.plot(xss, bspline(xss))
```

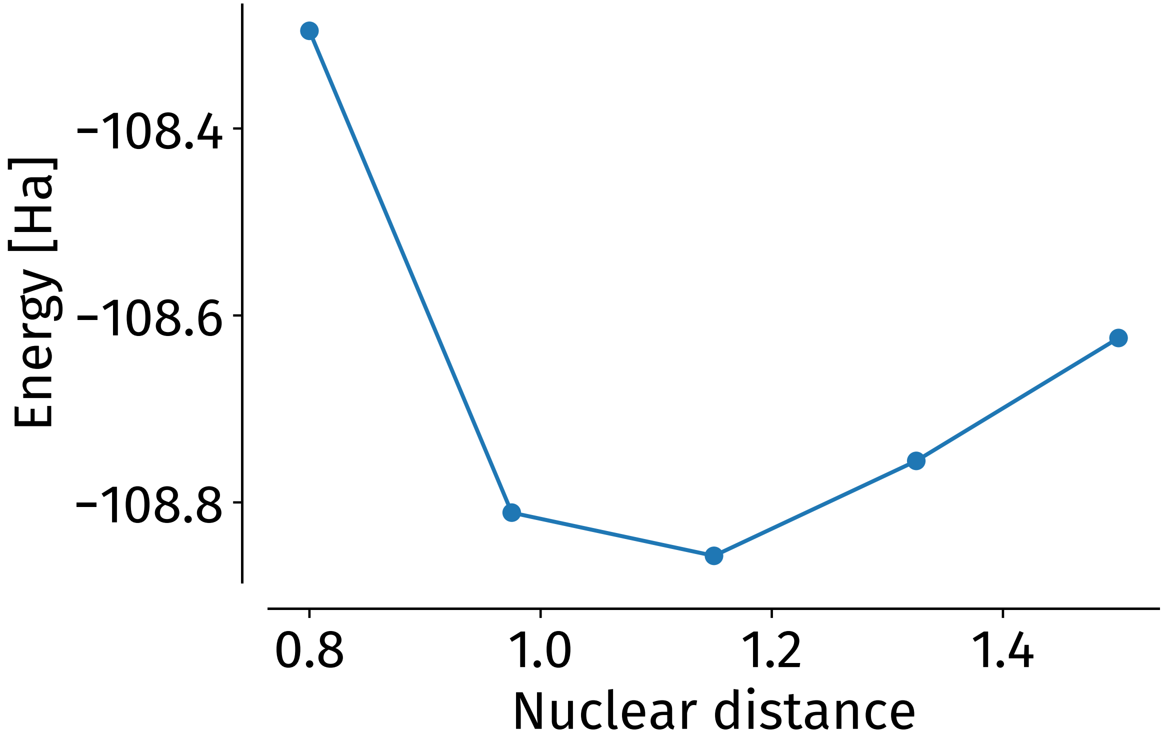
- s as regularizer
- By default: cubic B-splines
- Object can be called like a function



Python: Example

Task
- Find minimum
- Optimizer of unknown cost
- Find fixed-cost alternative

```
from pyscf import gto
import numpy as np
from pyscf.data import nist

def energy_N2(nuclear_distance_angstrom):
    mol = gto.M(atom=[['N', 0., 0., 0.0],
                      ['N', 0., 0., nuclear_distance_angstrom]],
                basis='6-31G', verbose=0)
    mf = mol.RHF().run()
    return mf.e_tot

xs = np.linspace(0.8, 1.5, 5)
ys = []
for x in xs:
    ys.append(energy_N2(x))
```



Python: Example

Task
- Find minimum
- Optimizer of unknown cost
- Find fixed-cost alternative

```python
import scipy.interpolate as sci
cspline = sci.CubicSpline(xs, ys)
print (cspline.derivative().roots())
```

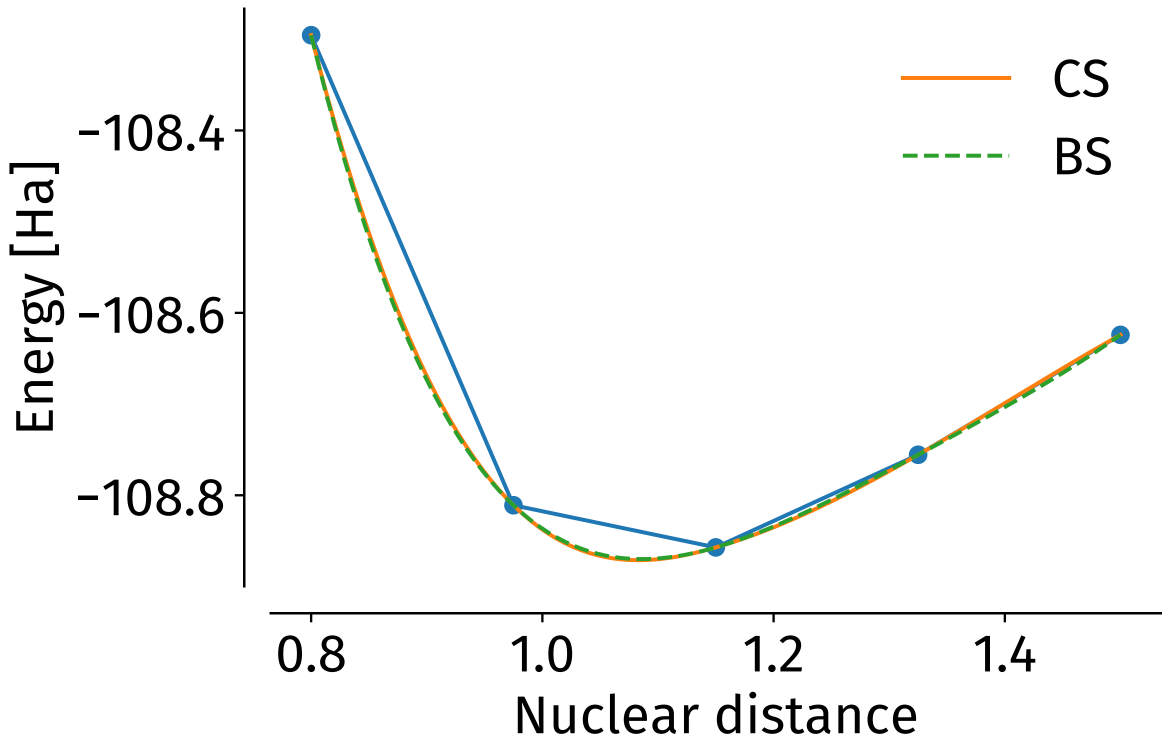
```
[1.08307817 1.75348628]
```

```python
import scipy.interpolate as sci
bspline = sci.UnivariateSpline(xs, ys, s=1, k=4)
print (bspline.derivative().roots())
```

```
[1.08593233]
```

```python
sco.minimize(energy_N2, x0=1.)
```

```
      fun: -108.86800502696096
 hess_inv: array([[0.14851732]])
      jac: array([5.7220459e-06])
  message: 'Optimization terminated successfully.'
     nfev: 24
      nit: 6
     njev: 8
   status: 0
  success: True
        x: array([1.08912625])
```



Python: 2D

Interpolation on grid

- Bivariate spline *RectBivariateSpline*

Interpolation on irregular points

- *interp2d*
    - Linear mode: not smooth
    - Cubic mode: slow

Other tools in brief

Genetic algorithms

- Large-scale optimisation problem — *scipy, DEAP*
- Automatable optimisation

Automatic differentiation

- Get derivatives of python code — *autograd*
- No need to derive explicit expressions

Symbolic algebra

- Build mathematical expressions in code — *sympy*
- Reduces errors for long equations

Genetic Algorithms

Global Optimization

- Find minimum with little knowledge of search space
- Inspired by evolution
  - Genome — Vector describing the solution
  - Population — Set of trial solutions
  - Mutation — Random change of an existing solution
  - Crossover — Combine features of two solutions
  - Selection — Survival of the fittest
- Requirements
  - Fitness function (Target objective)
  - Representation (Solution vector or larger)
  - Random solutions
- When to use
  - Medium dimensionality
  - Human time available
- When not to use
  - Classification

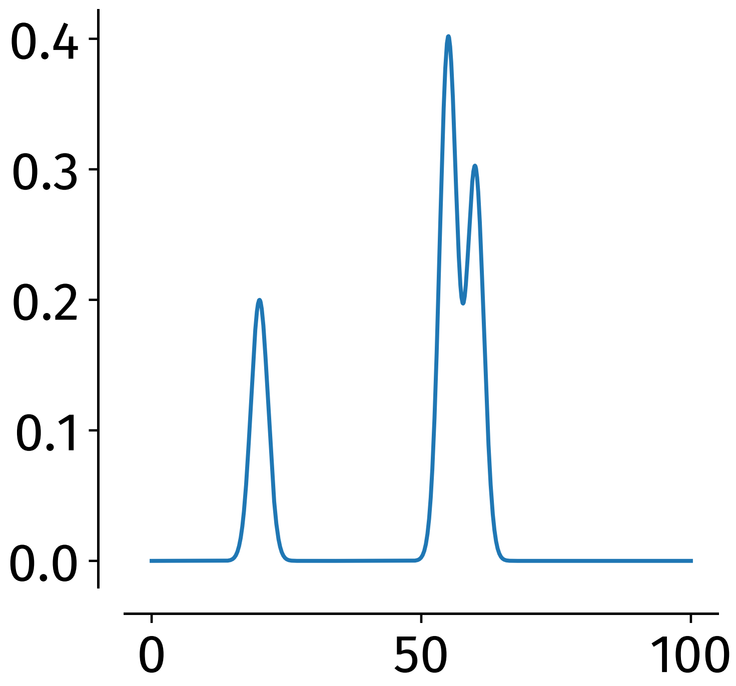
Genetic Algorithm: Example

Global Optimization

- Find three peaks

```
import numpy as np
import scipy.optimize as sco

xs = np.linspace(0, 100, 1000)

def func(xs, a, b, c, d, e, f):
    result = 0
    for scale, position in ((a, b), (c, d), (e, f)):
        result += scale * np.exp(-0.2 * (xs - position)**2)
    return result

def residuals(x0, xs, ys):
    return np.abs(func(xs, *x0) - ys).sum()

bfgs = sco.minimize(residuals, x0=(0.1, 10, 0.1, 20, 0.1, 30), args=(xs, ys), method="BFGS")
print (f"BFGS used {bfgs.nfev} evaluations and has accuracy {residuals(bfgs.x, xs, ys)}")

genetic = sco.differential_evolution(residuals,
                                     args=(xs, ys),
                                     bounds=((0,1), (0, 100), (0,1), (0, 100), (0,1), (0, 100)))
print (f"GA used {genetic.nfev} evaluations and has accuracy {residuals(genetic.x, xs, ys)}")
```

```
BFGS used 1132 evaluations and has accuracy 27.715548328799127
GA used 24807 evaluations and has accuracy 5.6151293195716914e-14
```

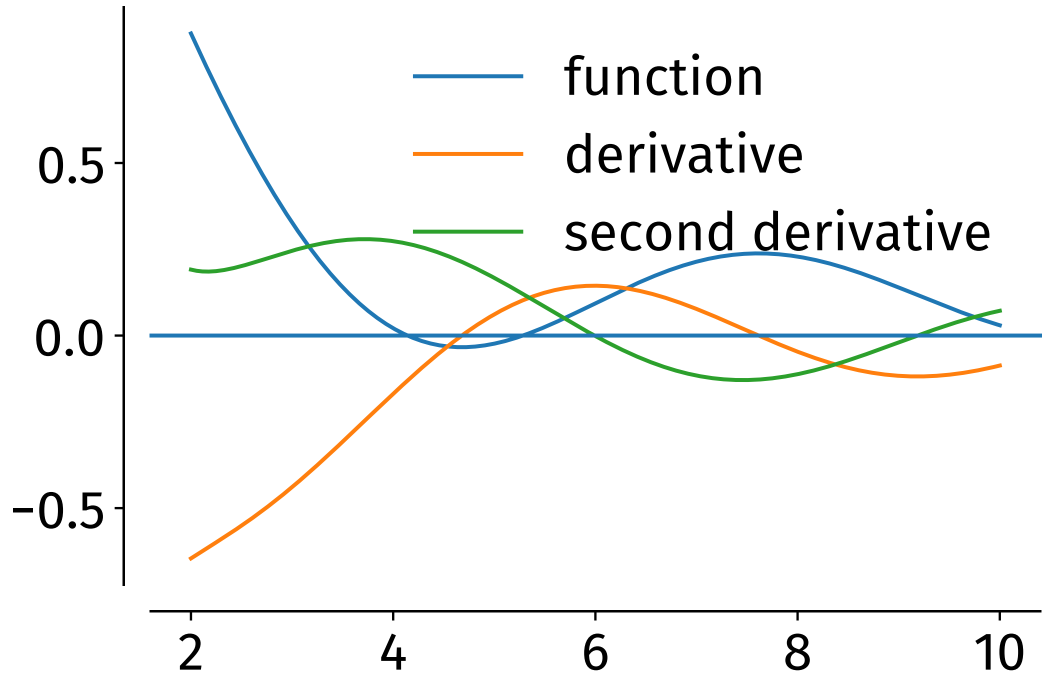


Automatic differentiation

Core idea
- Any code is a function f(x) -> y
- Follow program code, apply chain rule and get derivatives

When to use
- Machine learning (how to improve model?)
- Optimization without explicit gradients
- Derivatives to complex code

```python
import autograd.numpy as np
from autograd import elementwise_grad as egrad

def function(xs):
    result = xs * 0
    for i in range(2):
        result += np.sin(xs**i) / xs
    return result

xs = np.linspace(2, 10, 1000)
plt.plot(xs, function(xs), label="function")
plt.plot(xs, egrad(function)(xs), label="derivative")
plt.plot(xs, egrad(egrad(function))(xs), label="second derivative")
```



Library
- *autograd*

Symbolic algebra

Core idea

- Mathematical expressions as code
- Analytical differentiation or integration algorithms
- In python: can be mixed with other code components
- Switch between *numpy* functions and mathematical expression

```python
import sympy
x = sympy.Symbol('x')
```

```python
function = 0
for i in range(2):
    function += sympy.sin(x**i) / x
function
```

$$\frac{\sin(x)}{x} + \frac{\sin(1)}{x}$$

```python
derivative = sympy.diff(function)
derivative
```

$$\frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} - \frac{\sin(1)}{x^2}$$

```python
l = sympy.utilities.lambdify(x, derivative)
l(3)
```

```
-0.4391742758521222
```

Summary

Interpolation

- B-splines
- Cublic splines

Other tools

- Genetic algorithms
- Automatic differentiation
- Symbolic algebra