Lecture 8

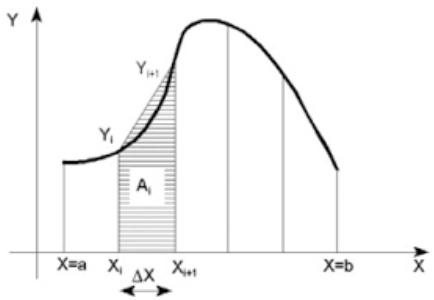
◆ Roots, integration, and linear algebra

As always, study & learn Python from scripts discussed in this lecture, cf. http://planets.utsc.utoronto.ca/~pawel/pyth.

Formal proof of convergence of secant and Newton's method

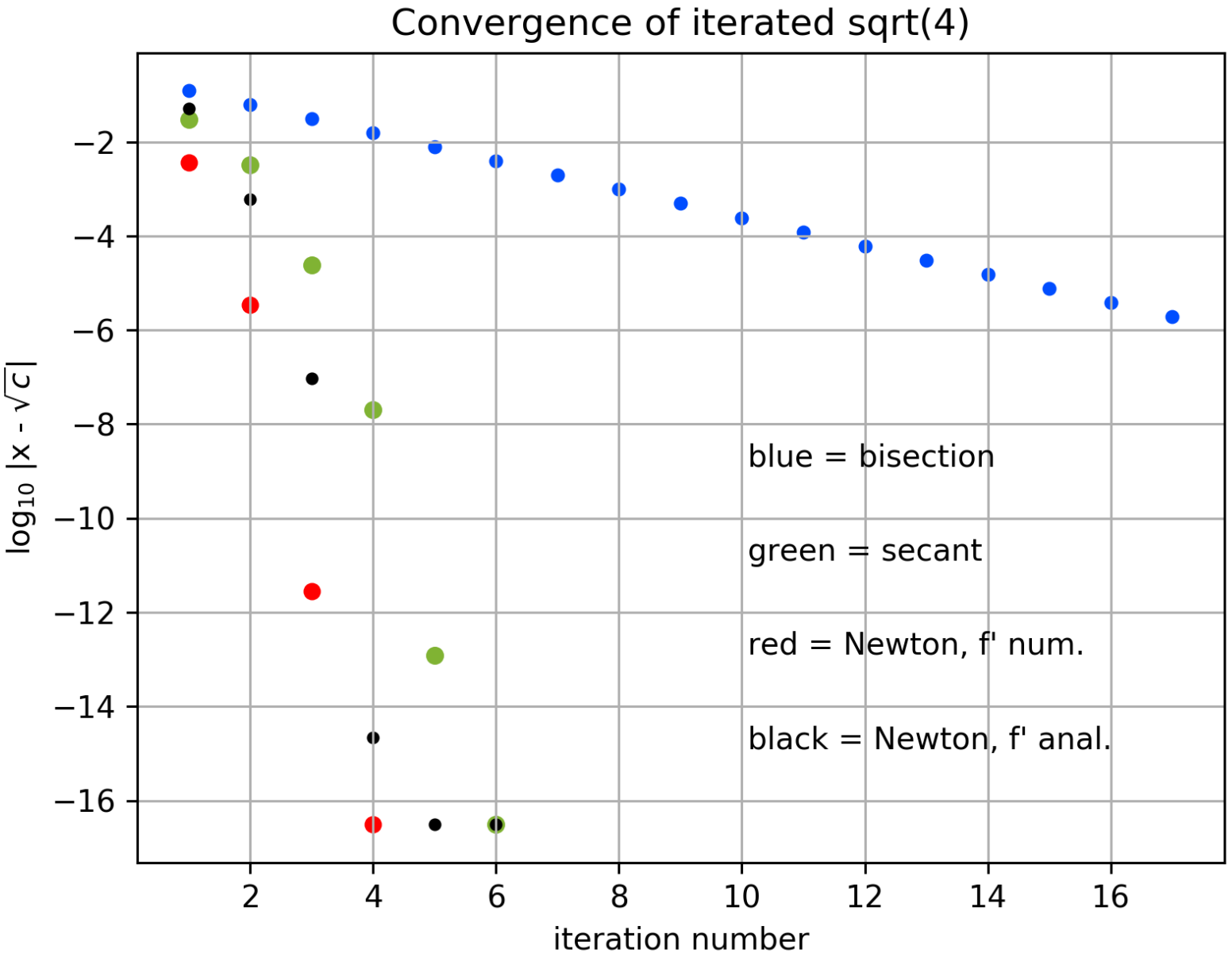
Which method is quicker: secant or Newton-Raphson-Simpson?

- Numerical Calculus: Rules of integration faster than MtCarlo
- Interpolating polynomials and integration: page 48 Turner(2018)
  - ❖ Euler's method
  - ❖ Trapezoid rule
  - ❖ Simpson's rule



- Numerical Linear Algebra
  - NumPy method to solve NxN linear equations
  - Applications: data fitting and denoising

$f = x^2 - c = 0, \quad c = 4,$ by **Newton's method ***

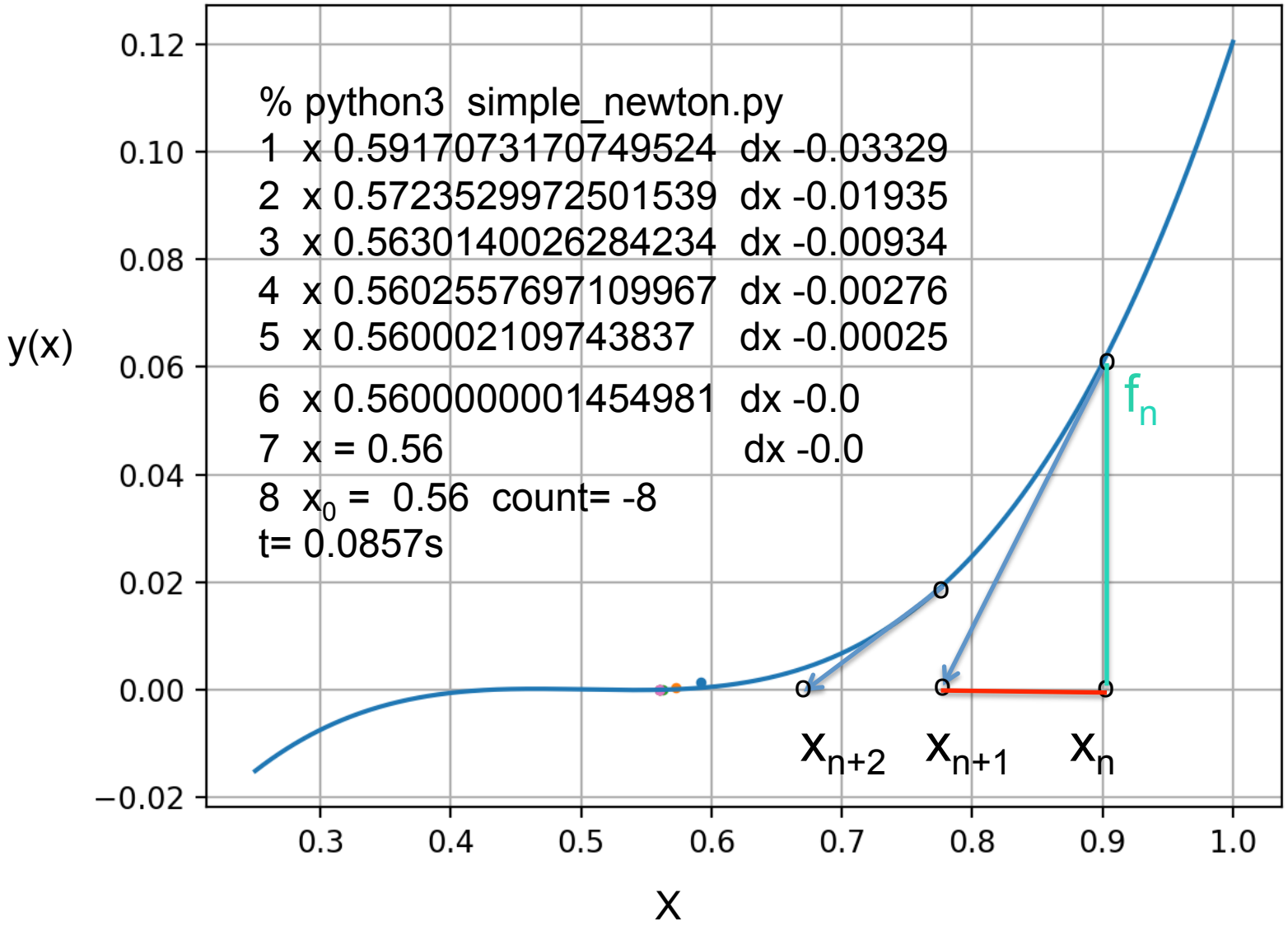


*) actually, Isaac Newton neither invented nor used this method.

- Reminder: zero finding by Newton's method

simple_newton.p

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$
$$f(x_n)=f'(x_n)(x_{n+1}-x_n)$$



- **Formal proof of quadratic convergence of Newton's method:**
- $x_{n+1} = x_n - f_n/f'_n$
- Let's define z as zero of function f: f(z)=0 and Taylor-expand function f around x_n:
- $0 = f(z) = f(x_n) + (z-x_n) f'(x_n) + (z-x_n)^2 f''(x_n)/2 + \ldots$
- Divide by $f'(x_n)$ and rearrange,
- $z - x_n + f(x_n)/f'(x_n) = - (z-x_n)^2 f''(x_n)/2 + \ldots$
- Using the blue equation on the left, we prove the quadratic relation between the consecutive error terms:

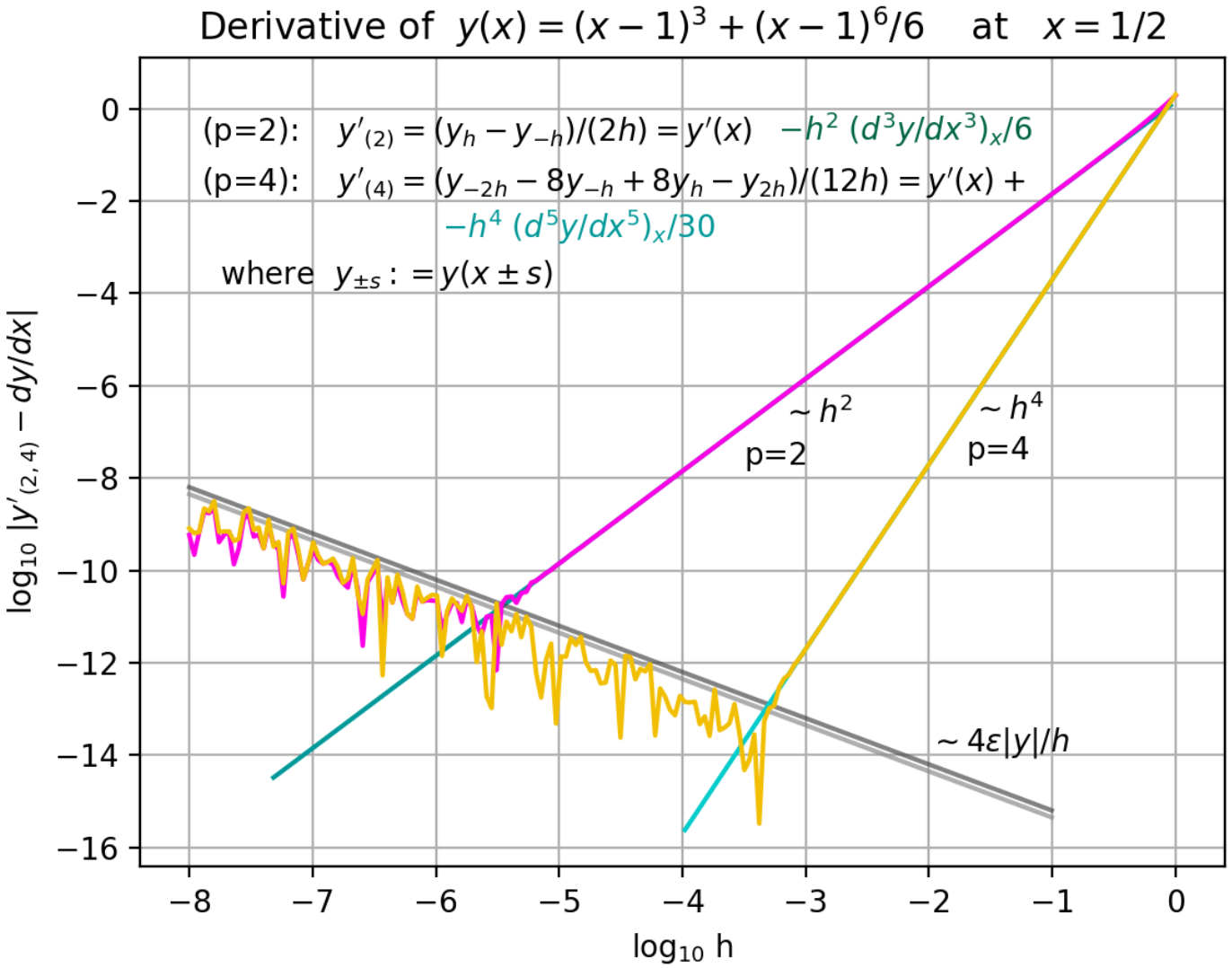
$$|\mathbf{z - x_{n+1}}| = | -(\mathbf{z-x_n})^2 f''(x_n)/2f'(x_n) + \ldots| = |f''(\xi)/2f'(x_n)| \; |\mathbf{z-x_n}|^2,$$

where ξ is somewhere between x_n and z, such that the whole colored term is asymptotically a constant.

In words: asymptotically the number of accurate digits doubles in every iteration.

- *Note: it is possible to start too far from the attraction range of the iteration, making f' too small or zero, thus preventing the success. But if the method starts converging, it converges quadratically, i.e. very fast.*

Reminder from Lecture 7. We will see below that numerical integration error behaves somewhat similarly (drops with h^p, then rises as h^{-1} due to roundoff, as h decreases.*)*
But sometimes there are problems. Integration is more difficult than differentiation! We can always differentiate functions, but not always integrate.



Numerical calculus: integration

see http://planets.utsc.utoronto.ca/~pawel/pyth integ-p124-*.py

- Theory discussed in texbook#2
- Turner et al., Springer 2018
- Section 3.2, p. 48

Peter R. Turner · Thomas Arildsen
Kathleen Kavanagh

Applied
Scientific Computing

With Python

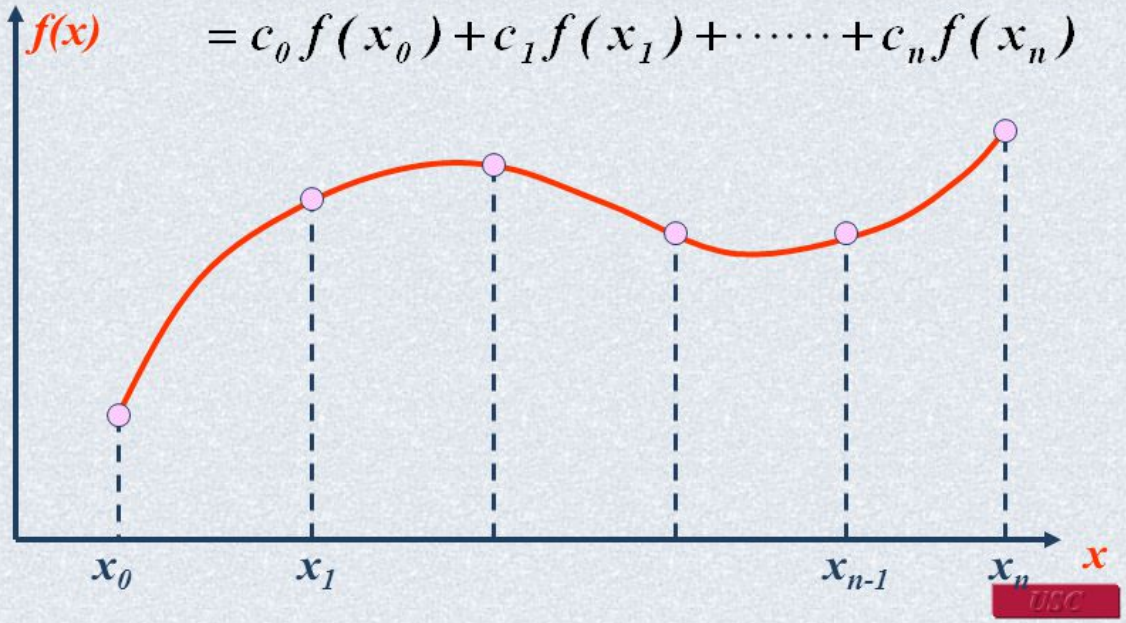
Basic idea of how we integrate in small intervals of x

Basic Numerical Integration

- **Weighted sum of function values**

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} c_i f(x_i)$$

$$= c_0 f(x_0) + c_1 f(x_1) + \cdots\cdots + c_n f(x_n)$$



Numerical Calculus II: Integration

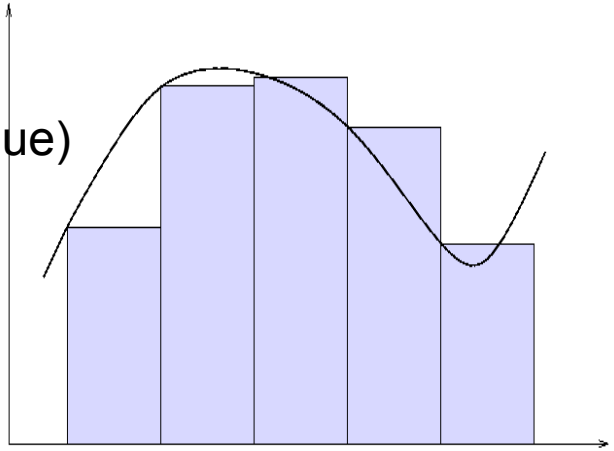
Developing coefficients c_0, c_1, ... of approximation polynomials in a *small* interval covered by just several computational points (samples of data or math function).

Def.: Order of the method is *m* if it integrates polynomials or piecewise polynomials up to order m exactly, and exhibits error proportional to h^{m+1} where *h* is the interval's width, for other functions.

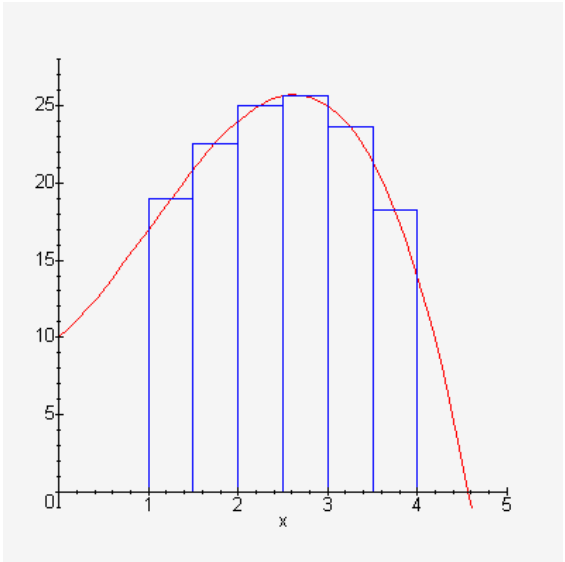
Require that method is of order m, and write m+1 equations (for n=0,1,2,...,m) binding m+1 coefficients of the method. Solve for c_n.

The integration points may be uniformly spaced in the x-interval or their positions in the interval may also be unknown (there must be exactly m+1 unknown x's and c's – read all about Gauss integration formulae on p. 56 of textbook)

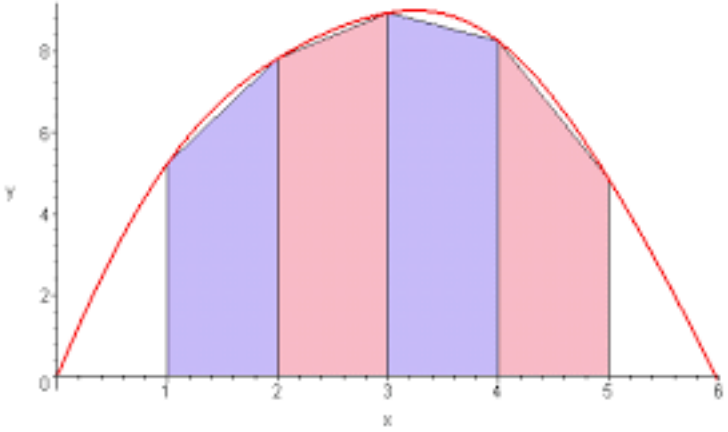
Euler(left value)
1st order

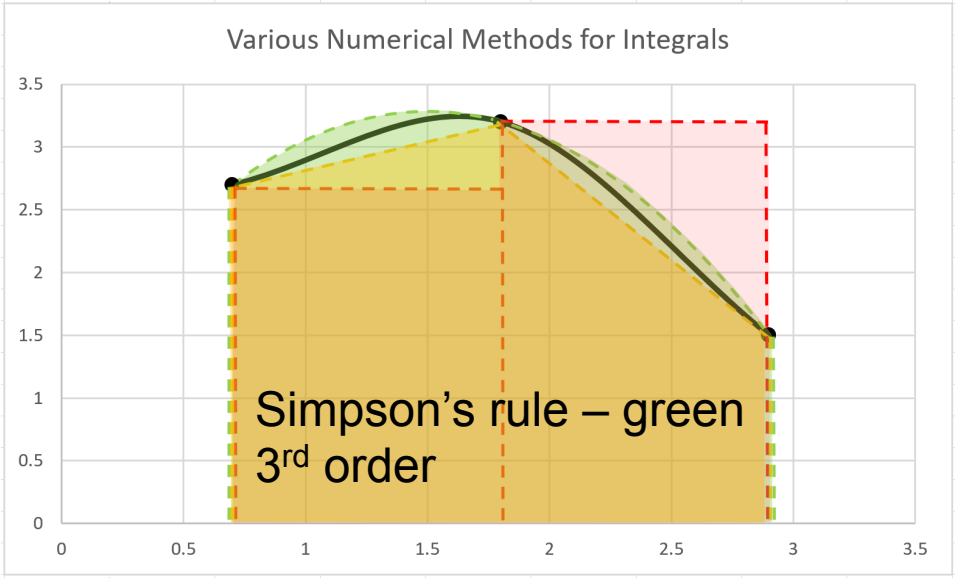


Midpoint method
2nd order



Trapezoid method
2nd order





Numerical integration of functions

see http://planets.utsc.utoronto.ca/~pawel/pyth

- Integrate exp(-x) from 0 to 1: integ-p124-exp.py
- Integrate $(1+x^2)^{-1}$ from 0 to 1: integ-p124-arc.py
- Integrate the length of a curtain: integ-p124-cur.py
- Integrate ¼ circle (area): integ-p124-Acirc.py
- Integrate ¼ circle (length): integ-p124-Lcirc.py

We implement and compare these methods :

- ❑ Euler's method
- ❑ Midpoint method
- ❑ Trapezoid rule
- ❑ Simpson's rule (so-called 1/3 rule, a 3-point rule)
- ❑ Pythagoras rule for line integrals (which you know from midterm)

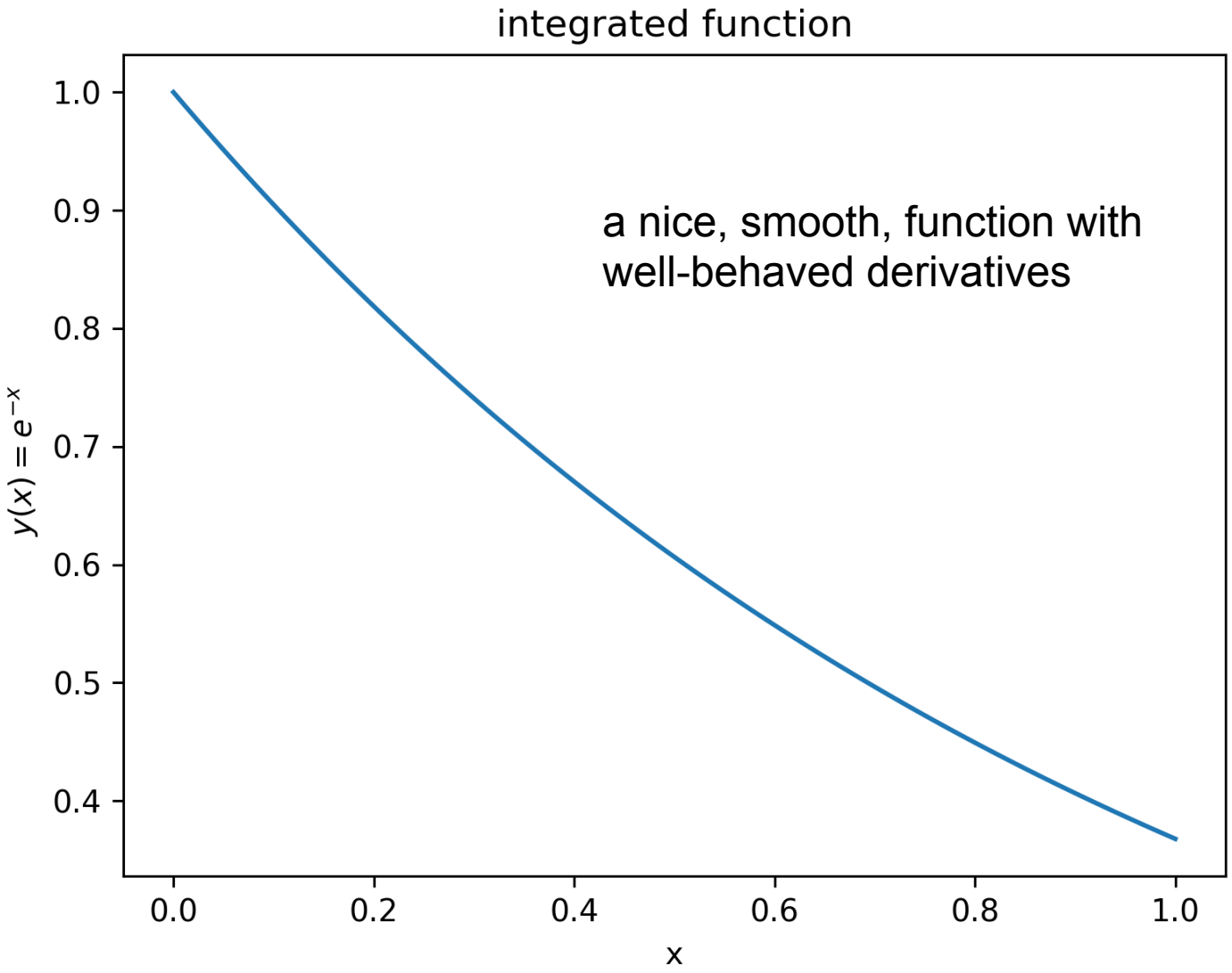
Composite integration formulae – combining N small sub-intervals of width h.

N+1 points forming N uniform intervals covering x-interval [a,b].
We have equal integration steps $h = (b-a)/N = x_{n+1} - x_n$, for all intervals n=0,1,2,...

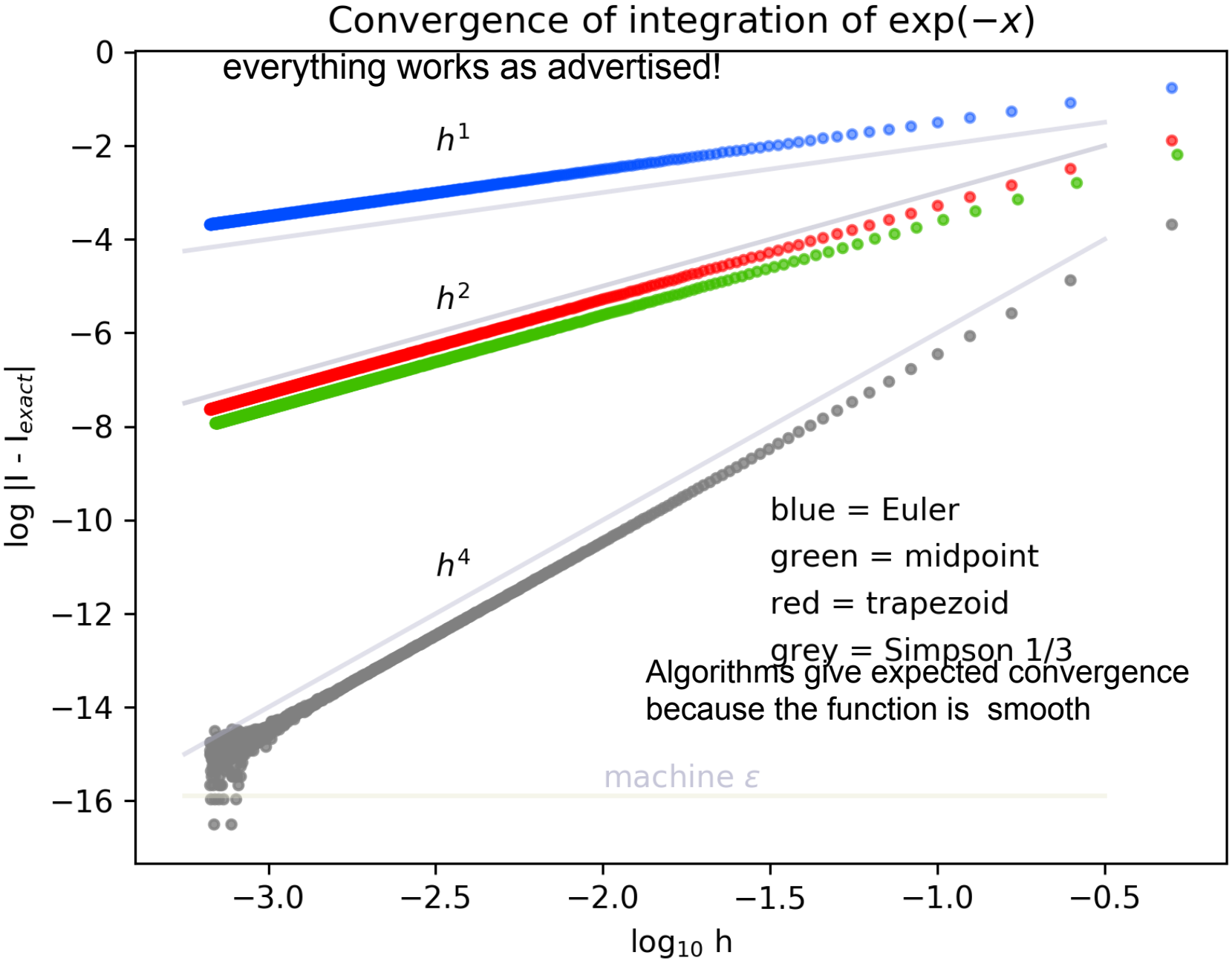
- Euler (left) $S = (f_0 + f_1 + f_2 + \ldots f_{n-1})\, h$
- Euler (right) $S = (f_1 + f_2 + f_3 + \ldots f_n)\, h$
- Midpoint $S = (f_{1/2} + f_{3/2} + \ldots + f_{n-3/2} + f_{n-1/2})\, h$, $f_{1/2} := f((x_0+x_1)/2)$ etc.
- Trapezoid $S = ((1/2)f_0 + f_1 + f_2 + f_3 + \ldots + f_{n-1} + (1/2)f_n)\, h$
- Simpson's 1/3: $S = (f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 4f_{n-1} + f_n)\, h$
- Pythagoras length summation: Sum of all $[(x_{n+1}-x_n)^2 + (f_{n+1}-f_n)^2]^{1/2}$

- Notice that integration errors accumulate and will be N times larger than single-interval error for basic formulae. $N = (b-a)/h \sim h^{-1}$.
- If the error on one sub-interval was $\sim h^p$, then the error on bigger interval [a,b], made of N sub-intervals, will be $\sim N\, h^p \sim h^{p-1}$.
- Therefore, on the large interval [a,b] we lose

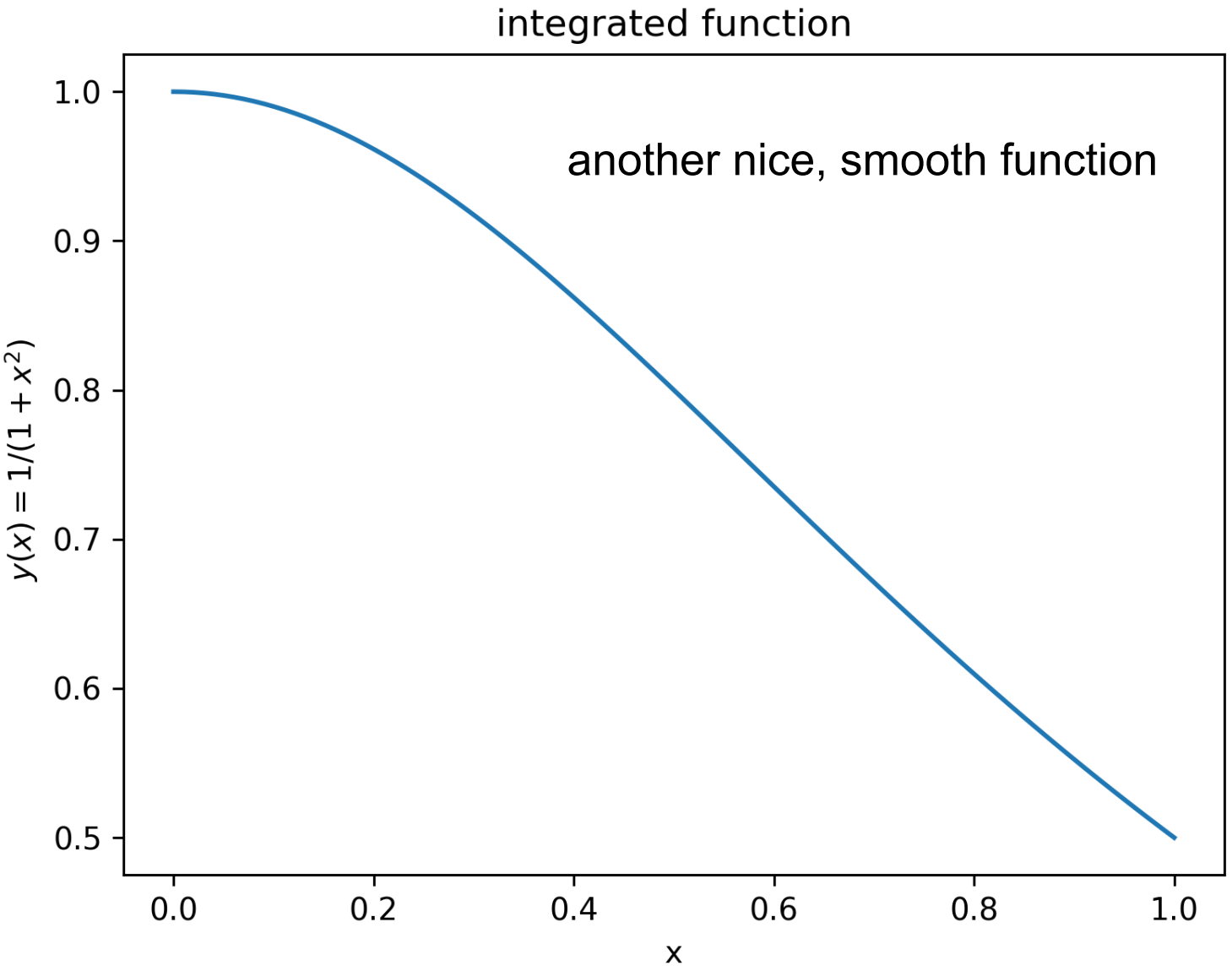
Integrate exp(-x) from 0 to 1: integ-p124-exp.py

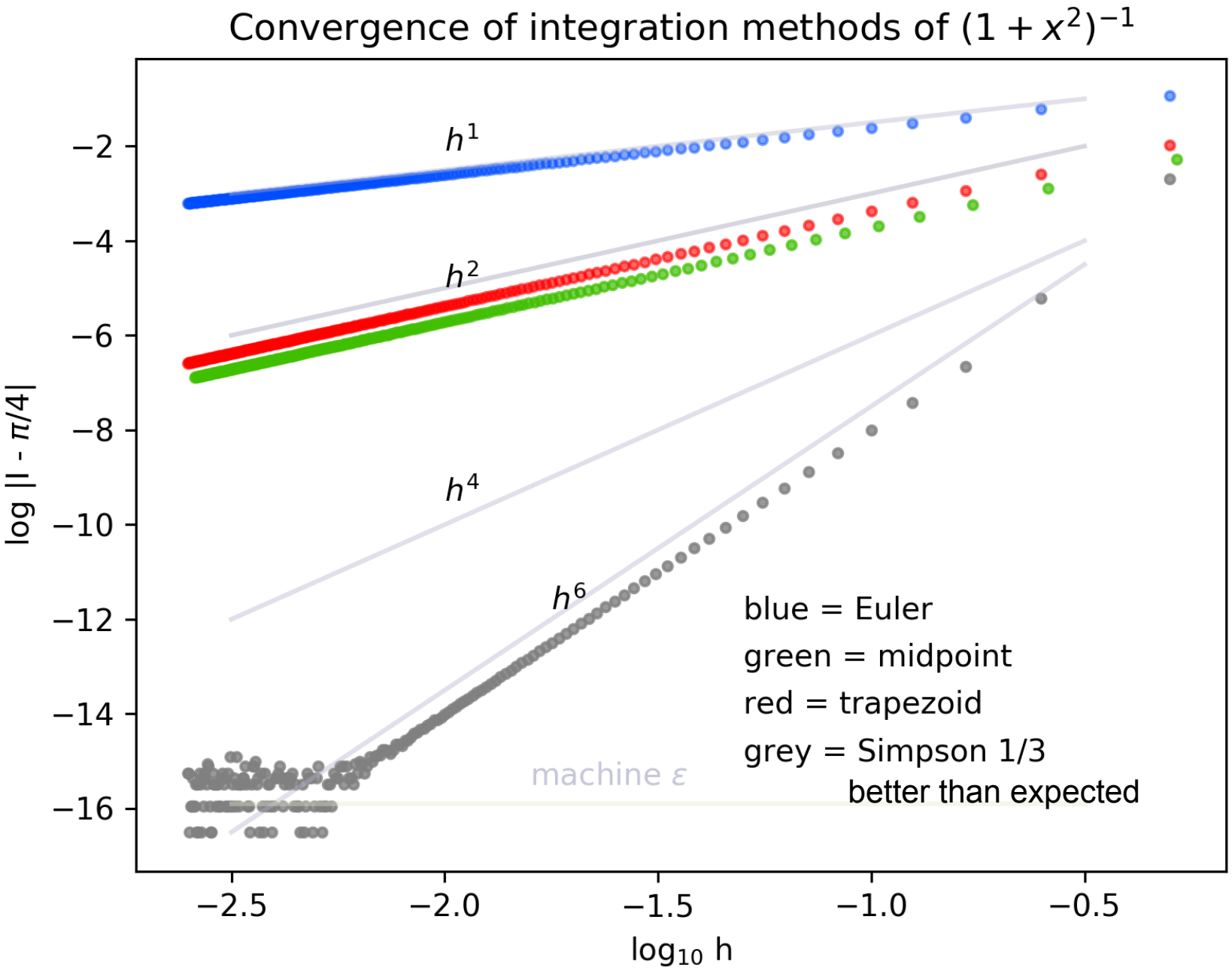


integ-p124-exp.py



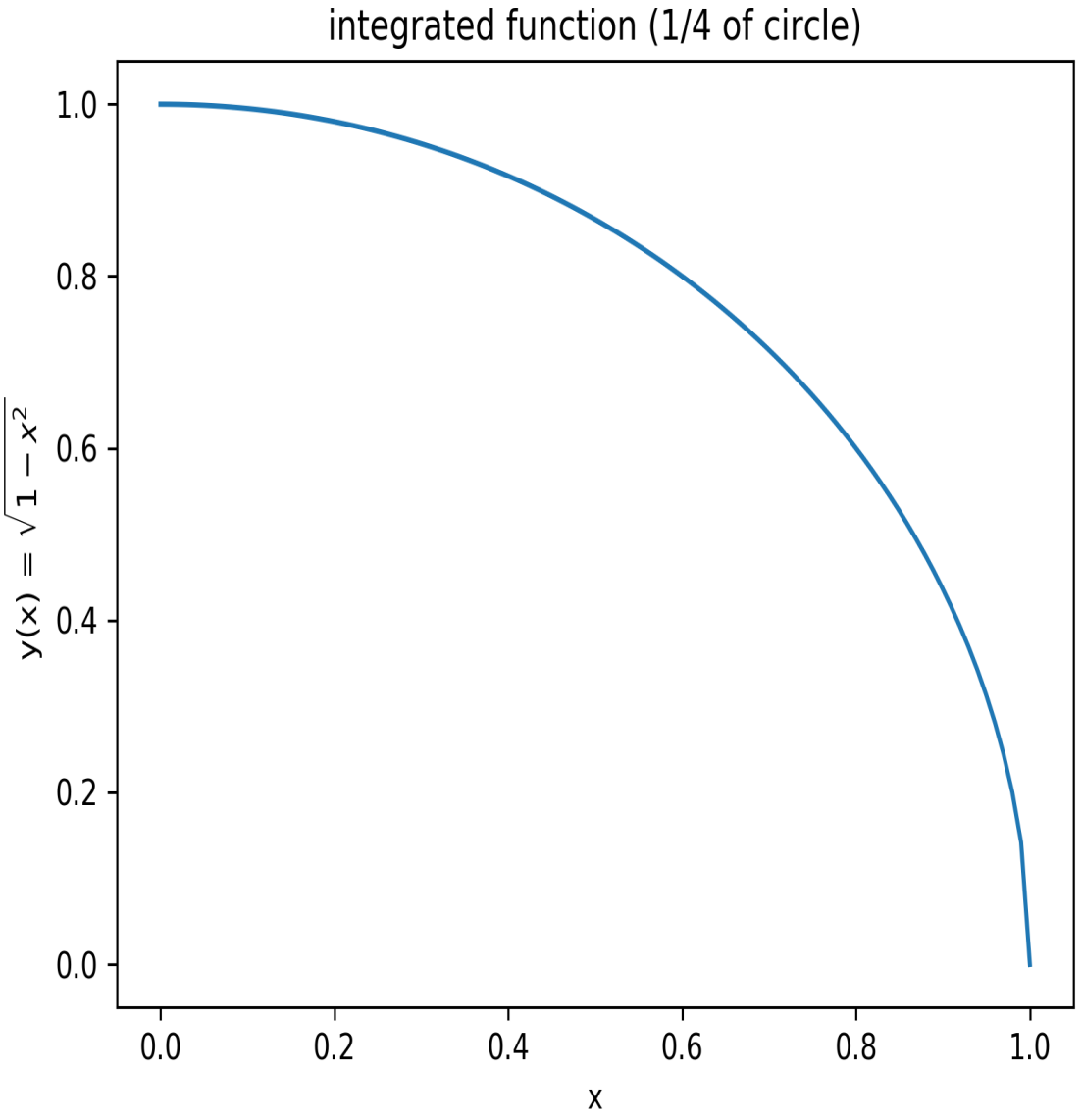
Integrate $1/(1+x^2)$ from 0 to 1: integ-p124-arc.py



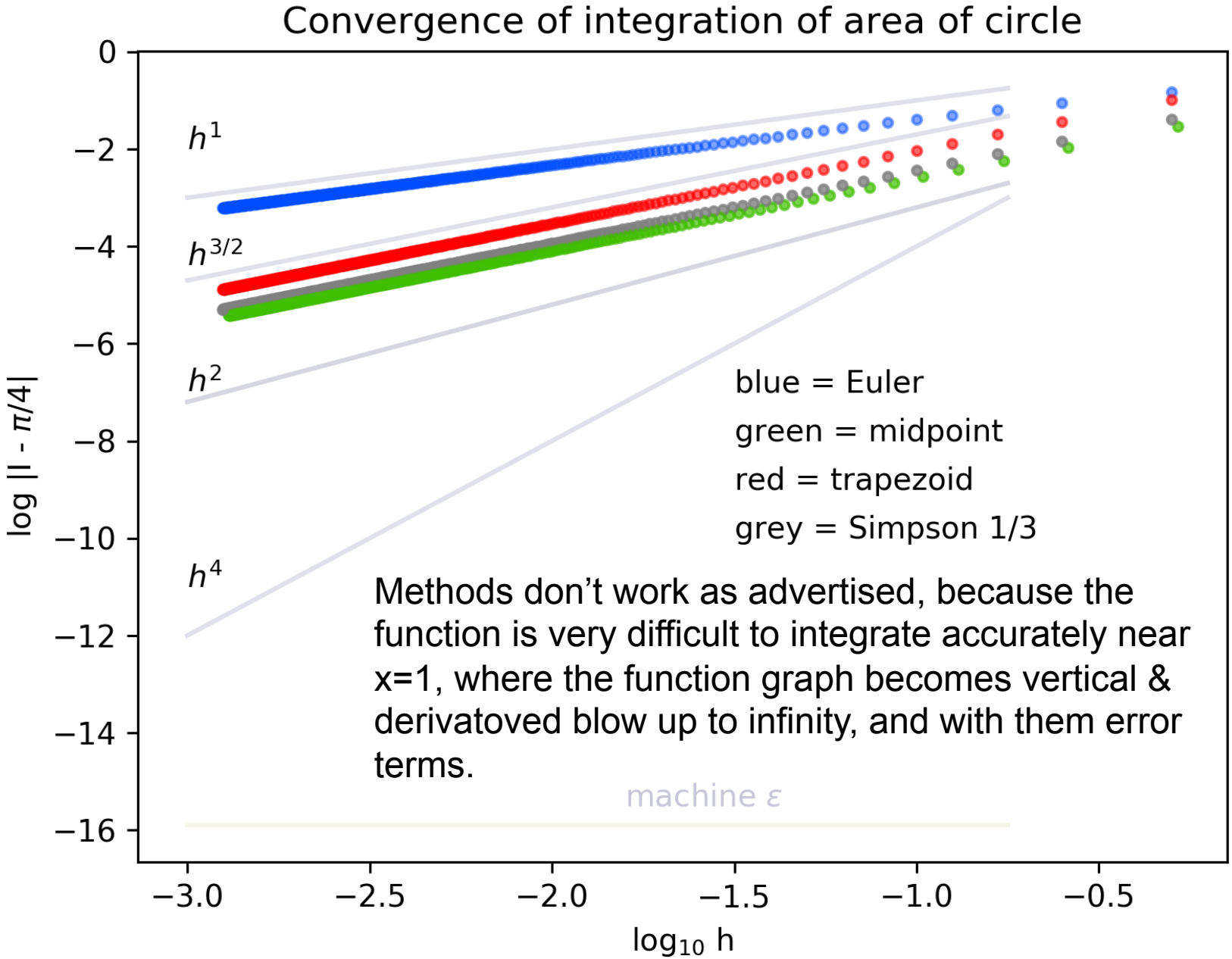


Again, everything works as advertised or better (Simpson's rule error~h^6 instead of h^4)!

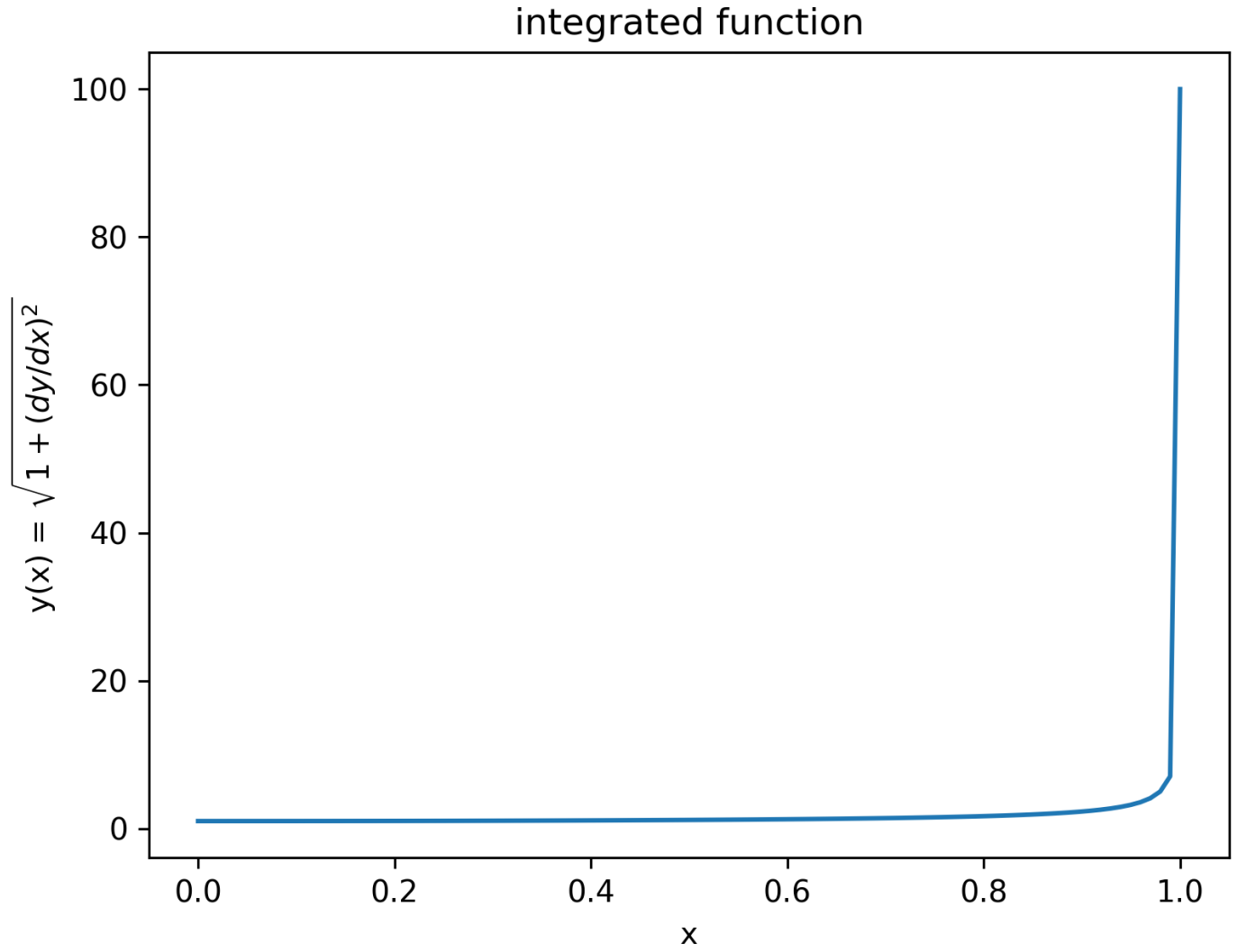
Integrate ¼ circle from 0 to 1: integ-p124-Acirc.py



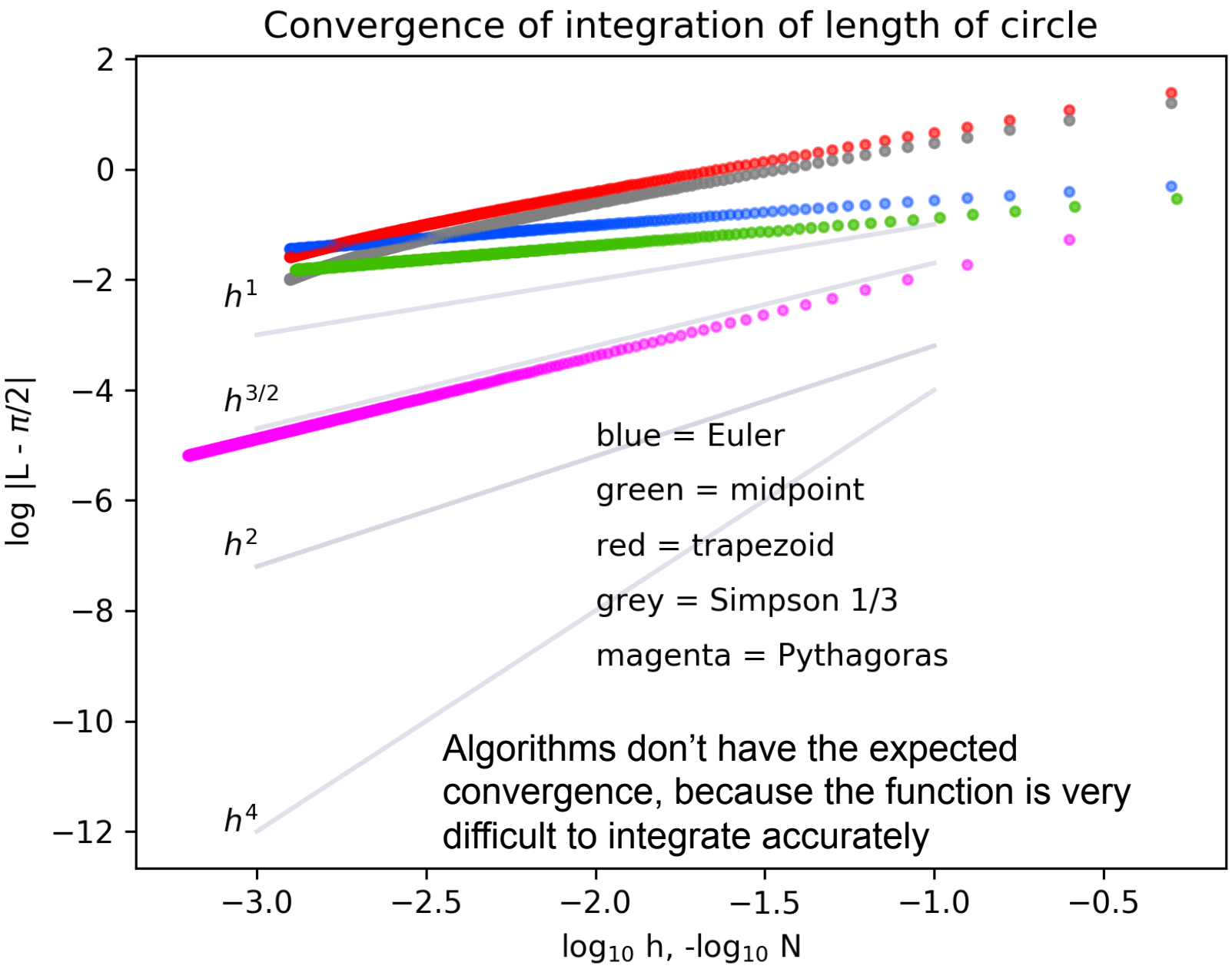
Notice steep descent near the end; this will cause problems..


Convergence of integration of area of circle
0
−2
−4
−6
−8
−10
−12
−14
−16
log |I - π/4|
h^1
h^3/2
h^2
h^4
blue = Euler
green = midpoint
red = trapezoid
grey = Simpson 1/3
Methods don't work as advertised, because the function is very difficult to integrate accurately near x=1, where the function graph becomes vertical & derivatoved blow up to infinity, and with them error terms.
machine ε
−3.0
−2.5
−2.0
−1.5
−1.0
−0.5
log10 h

Integrate length of ¼ circle x=0...1 integ-p124-arc.py



The sharp peak of dy/dx and other derivatives at x=1 promises trouble in this analytical approach to line integral giving the length of the circular curve (cf. the vertical axis label).



Breakdown of normal convergence rules of all the discussed methods *except* Pythagoras summation, which is not sensitive to derivatives. Error dominated by a few h intervals near x~1.

Computer Linear Algebra

- Read Turner et al., chapter 4, p. 81
- If you do not remember basic linear algebra (vectors, matrix notation, what is a linear system of equations, matrix inversion by Gauss elimination, linear systems and their solutions, what is an eigenvalue and eigenvector), please ask TA during tutorial, after first reading the link below.

Carl Friedrich Gauss (1777-1855)

- https://saylordotorg.github.io/text_intermediate-algebra/s06-05-matrices-and-gaussian-eliminat.html

Computer Linear Algebra

Examples of some scientific problems leading to linear systems of equations:

- Fitting theory to data
- De-noising measurements if underlying trends understood and can be modeled mathematically
- Solving discretized differential equations of science and engineering (many examples, as most of science since about 200 years ago uses the language of differential equations)
- Finding eigenvalues such as energy E in Schroedinger eq. of quantum mechanics, or rates of growth of spiral density waves in galaxies
- Finding modes in mechanical systems, e.g. shapes of the fastest growing modes in disk galaxies, distribution of stress and dilatation in vibrating objects.
- Flight simulators (the so-called panel methods)
- In addition, matrix operations are ubiquitous in: signal and image processing, multidimensional visualization and simulation,
- as well as machine learning (so-called AI = artificial intelligence)

Computer Linear Algebra

- To begin working with Python, start with this matrix equation

A x = b [matrix A times vector x equals vector b]

A linear system of N eqs. for N unknowns, A is a square NxN matrix, b the vector of constants of length N, x is the vector of N unknowns.

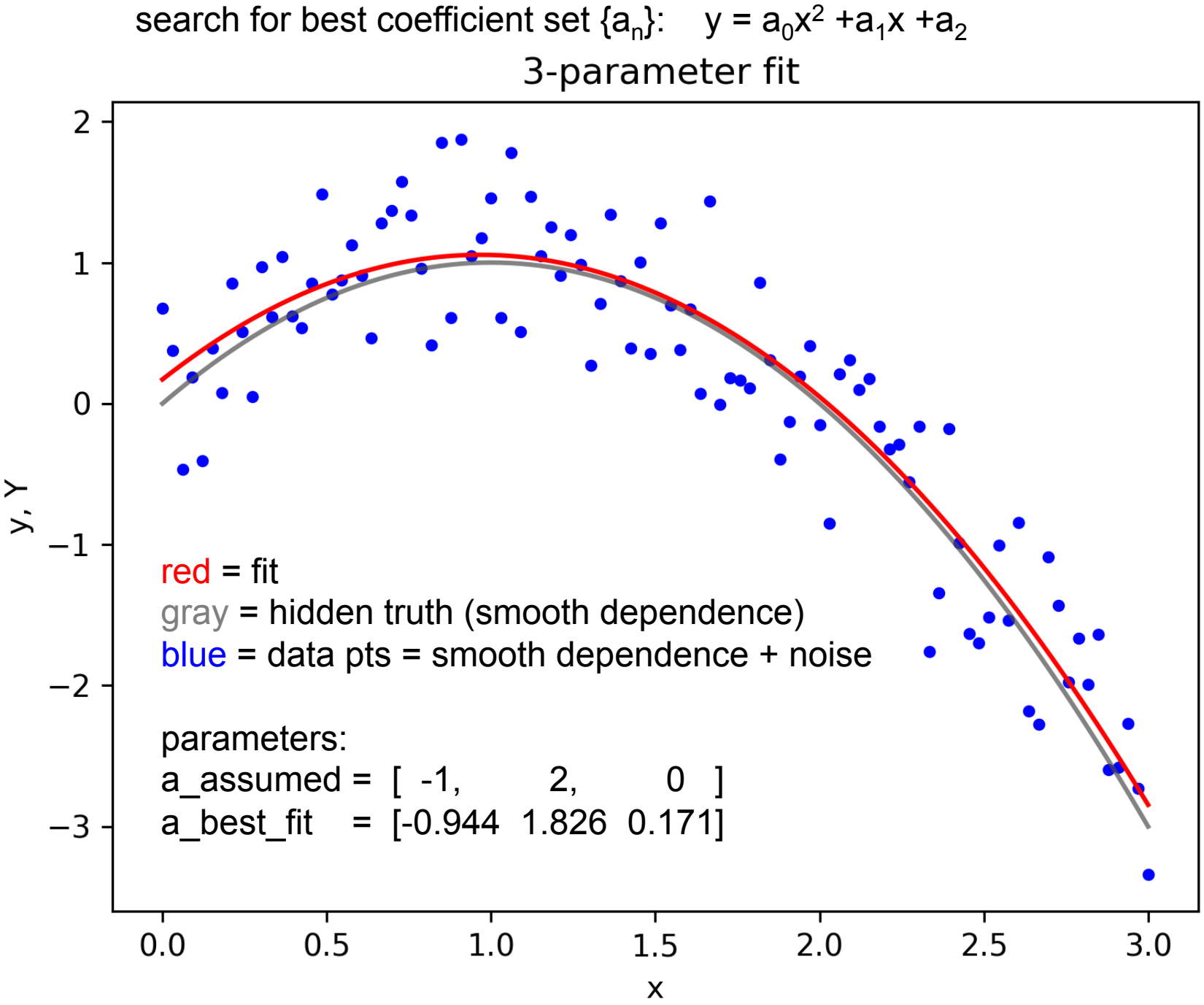
x = np.linalg.solve(A,b) # NumPy solves the system

- Consider a least-squares, 3-parameter fit. What is it?
- A method to find parameters of curves that pass closest to experimental data points (minimize sum of squares of differences)
- Least Squares method will be presented in detail in Lecture 9. Meanwhile see how well it works while finding coefficients of a linear combination of complicated functions, providing the best fit to a synthetic data set.

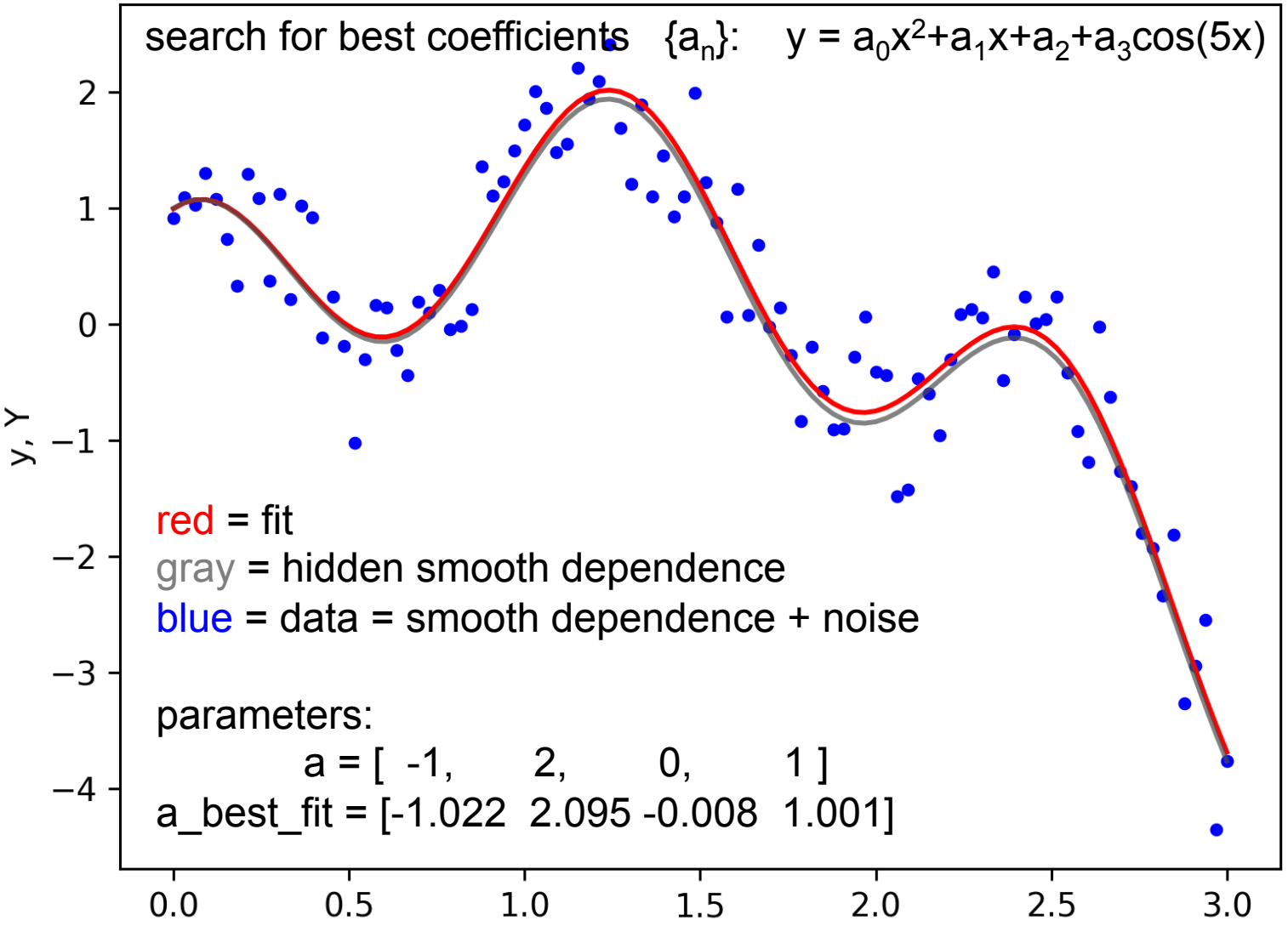
- fit-3par.py - see our Python code repository.

Least squares fit (parabola fit to noisy data): fit-3par.py

- "Hidden truth" or a theoretical dependence is in the form
- y_{th} = a[0] f(0,x) + a[1] f(1,x) + a[2] f(2,x)
- = a[0] x*x + a[1] x + a[2]
- a = a_{th} = [-1, 2, 0] # assumed values. Q: can they be # recovered by the program?
- x-axis is 100 uniformly spread values from 0 to 3.
- Noise added to simulate observational errors. Y holds 100 y-values:
- Y = -x*x+2*x + (np.random.rand(M)+np.random.rand(M)-1)
- Find set of parameters a = a_best_fit giving the best fit by requiring
- E = np.sum((y_{th}-Y)**2) → minimum
- Taking N derivatives after all a[n] yields N equations with N unknown a's:
- A a = b [N x N system of linear equations]
- A[m,n] = np.sum(f(n,x)*f(m,x)) # see L9 for more details
- b = np.sum(f(n,x)*Y)
- a = A^{-1} b # formal solution in terms # of the inverse matrix A^{-1}

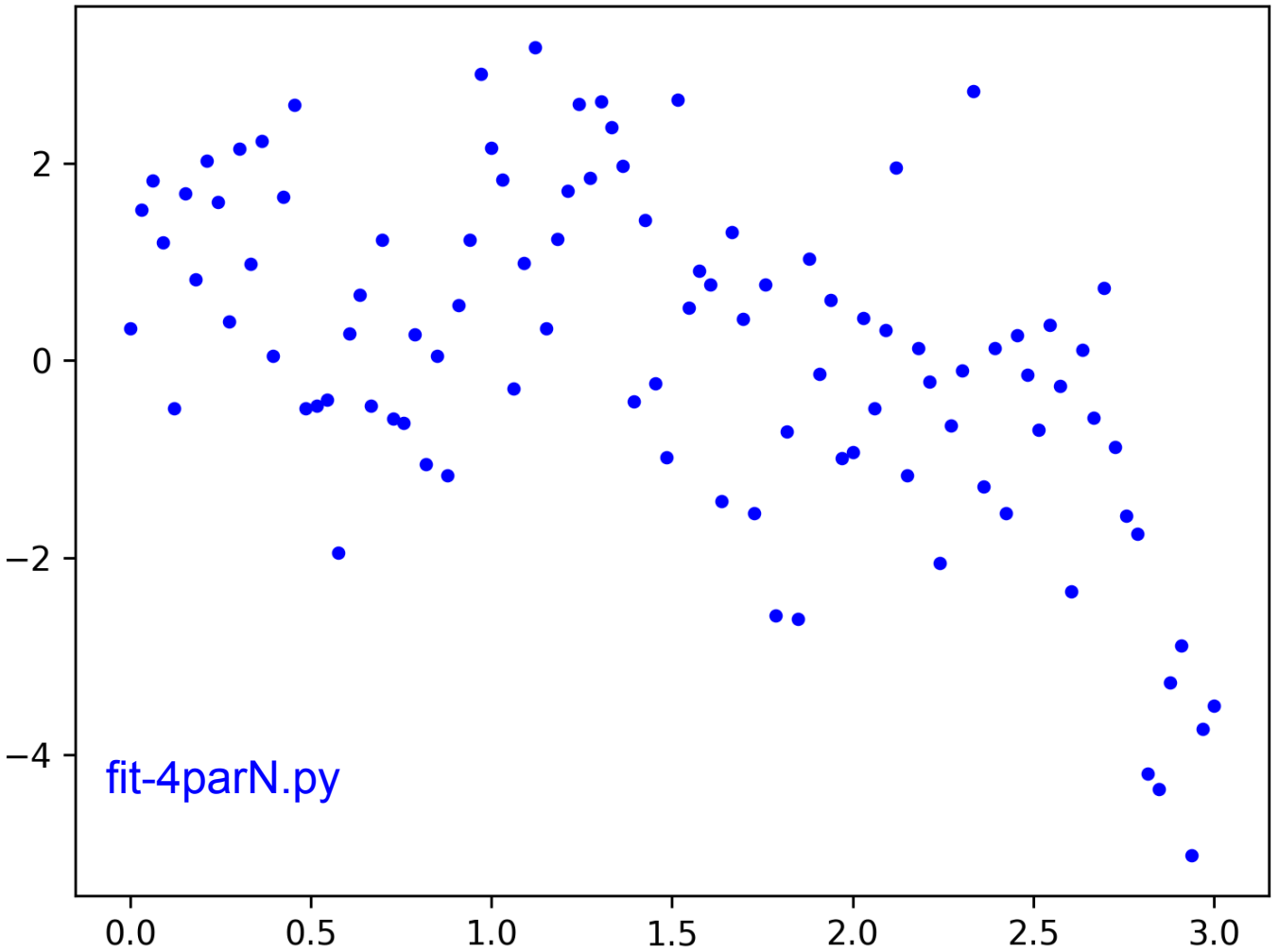

search for best coefficient set {a_n}: y = a_0x^2 +a_1x +a_2
3-parameter fit
red = fit
gray = hidden truth (smooth dependence)
blue = data pts = smooth dependence + noise
parameters:
a_assumed = [-1, 2, 0]
a_best_fit = [-0.944 1.826 0.171]
y, Y
x
2
1
0
−1
−2
−3
0.0
0.5
1.0
1.5
2.0
2.5
3.0

fit-4par.py. 4-parameter fit to data: y = x*(2-x)+cos(5x)+noise



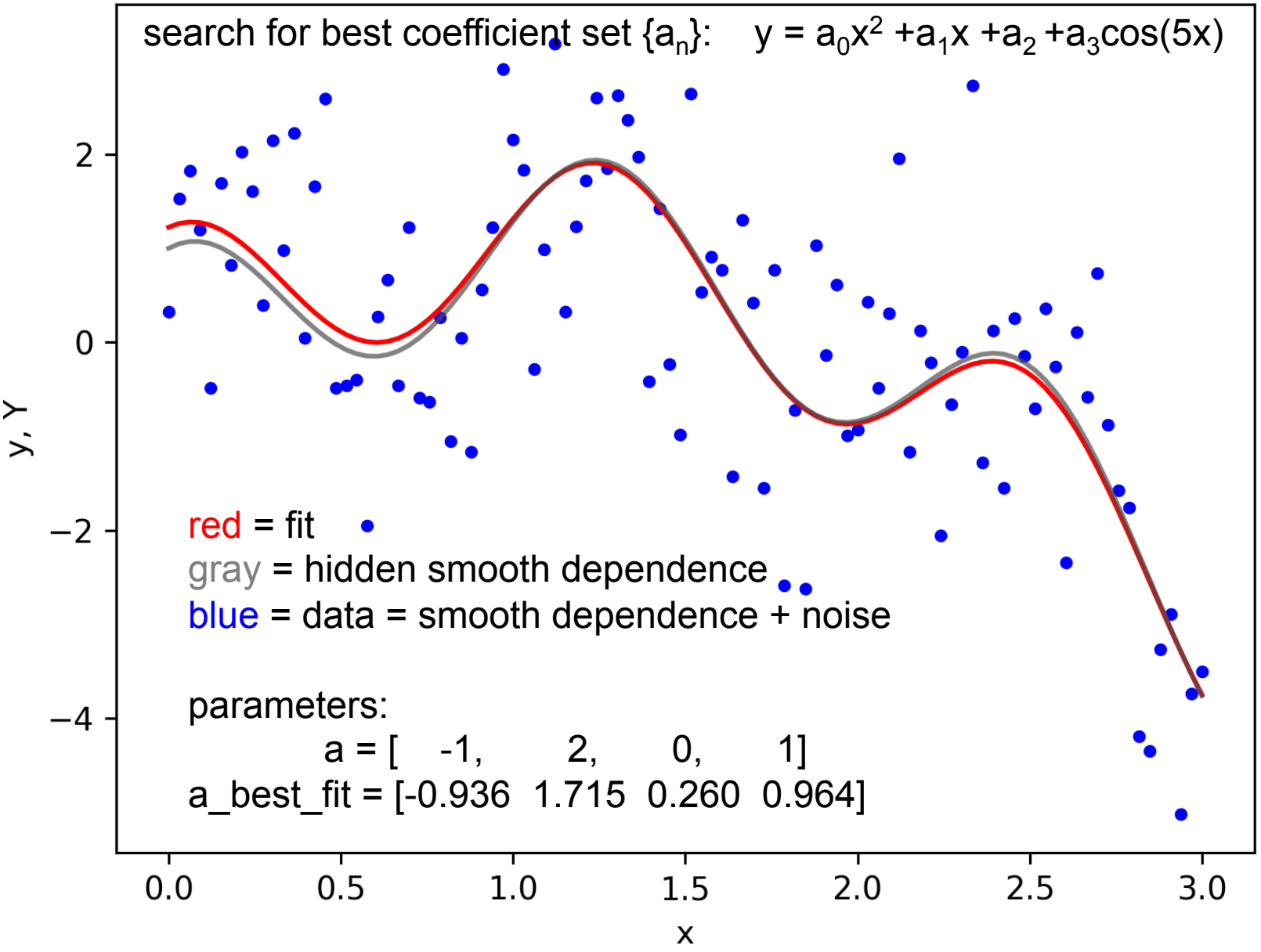
Finds the best combination of a parabola and sin(5x) that fits the data. Uncovers correctly all the parameters gray curve that was used to generate the blue points. This effectively de-noises the data set. Notice that we used the knowledge of what kind of functions to use, but that is often clear from the scientific context of a problem.

$y = a_0x^2 + a_1x + a_2 + a_3\cos(5x)$?



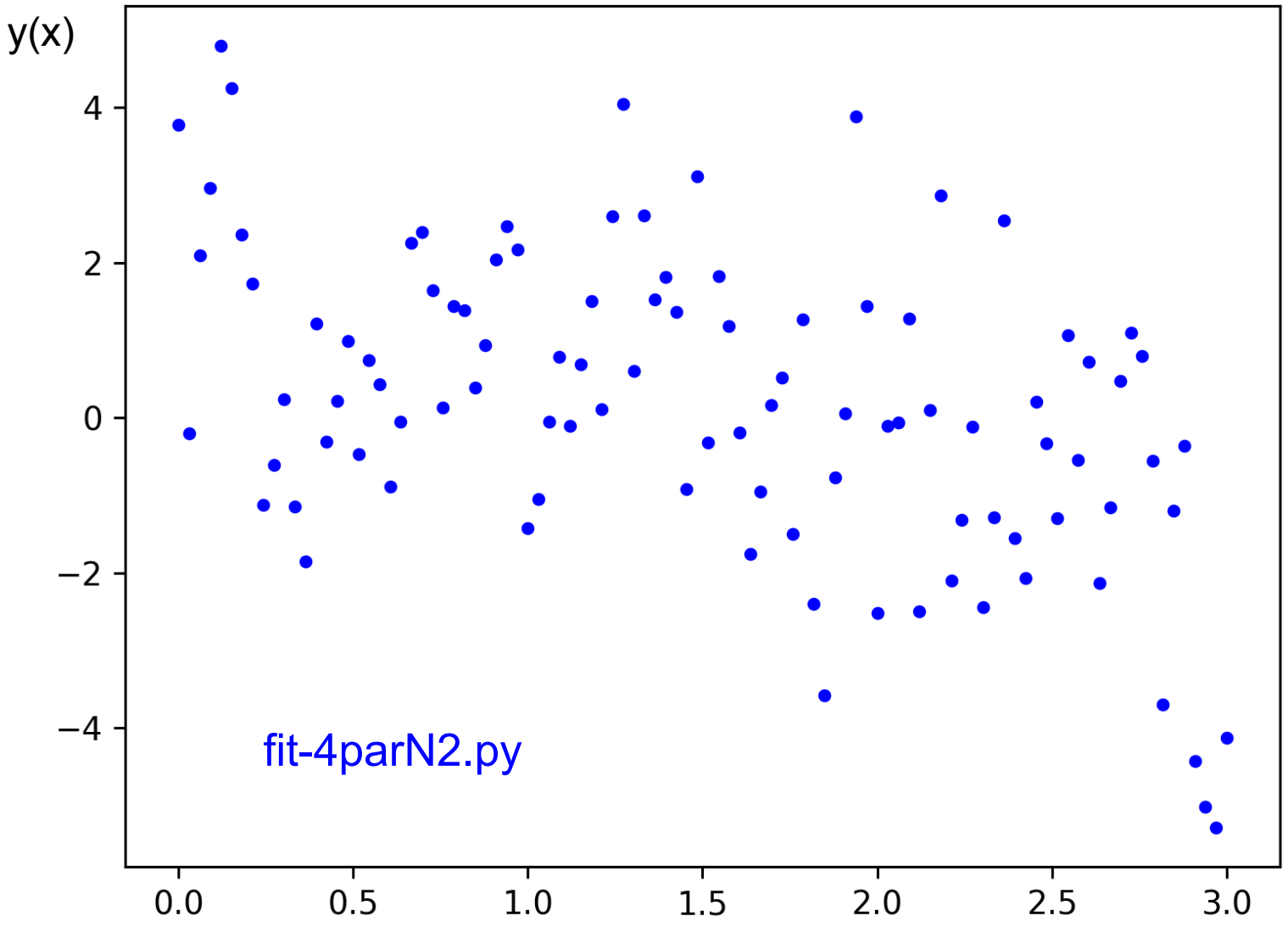
So much noise has been added here (varying in every execution of the code) that it becomes very hard to see what kind of periodic signal is hidden. Only something resembling a parabolic trend can be noticed. Let's now see how the linear algebra of the Least Squares method is going to recover for us the underlying regular variations.

fit-4parN.py 4-parameter fit to data: y = x*(2-x)+cos(5x)+noise

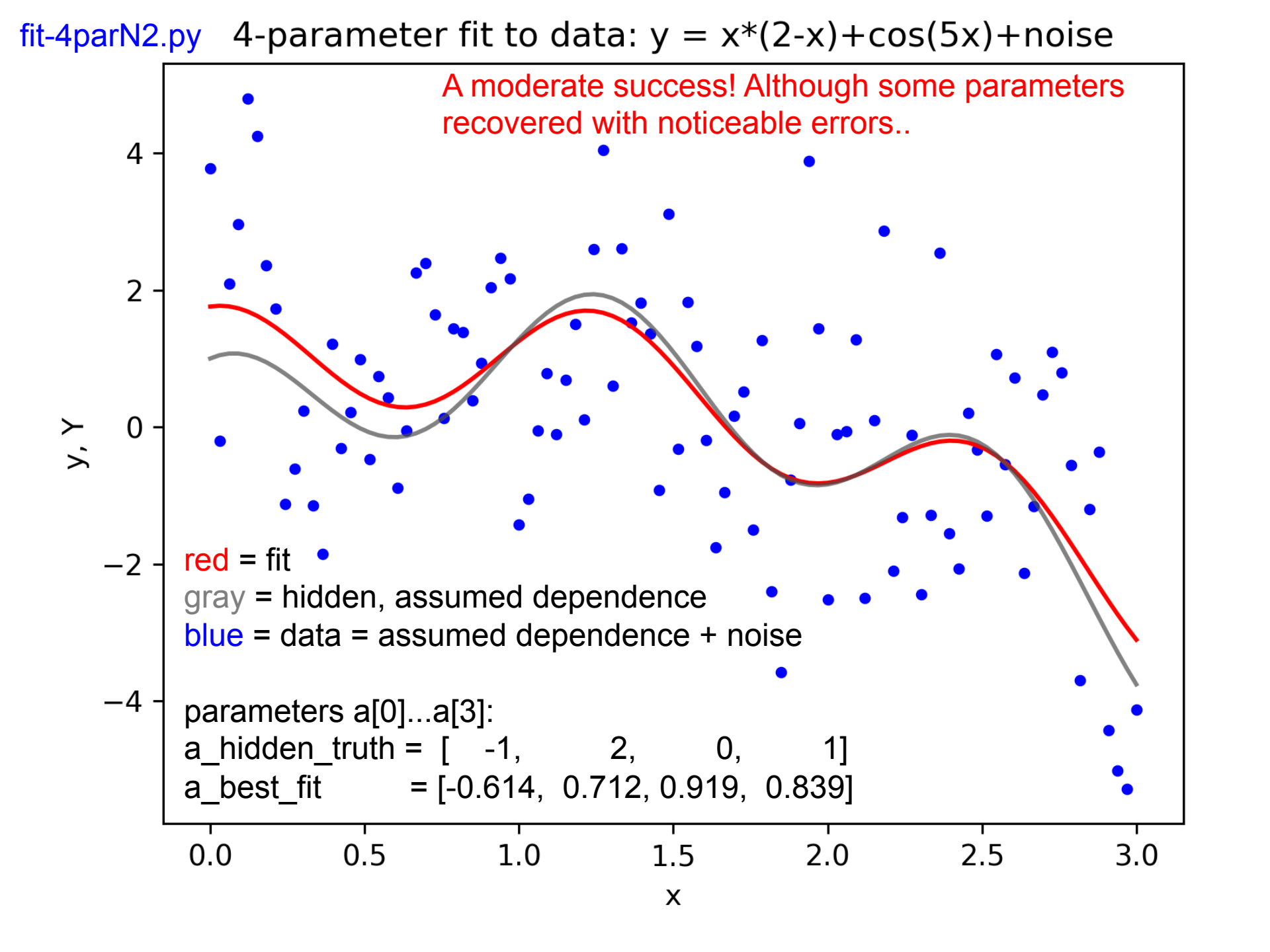


Very nice recovery (denoising) of the “true” variations obscured by strong noise

$y = a_0x^2 + a_1x + a_2 + a_3\cos(5x)$??



Let's add even more noise and test the strength of the method!
Can this data possibly be de-noised?


fit-4parN2.py
4-parameter fit to data: y = x*(2-x)+cos(5x)+noise
A moderate success! Although some parameters recovered with noticeable errors..
red = fit
gray = hidden, assumed dependence
blue = data = assumed dependence + noise
parameters a[0]...a[3]:
a_hidden_truth = [-1, 2, 0, 1]
a_best_fit = [-0.614, 0.712, 0.919, 0.839]
y, Y
x
4
2
0
−2
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0