LECTURE 4

Numerical calculus and differential equations

Discussion of assignment set A1 solutions, & A2 problems.

Random walks, and the diffusion of fragrance. Random walks, and the ubiquity of $1/\sqrt{N}$ scaling.

Integration of functions. Accuracy of n-th order method, vs. the Monte Carlo (MtC) method. Finding π by MtC.

PDE (partial diff. equations), example:
Diffusion equation-based unsharp-masking in image processing – stencil method. Detailed look at timing program execution.

(please see our code page https://planets,utsc.utoronto.ca/~pawel/progD57)

Numerical Calculus: Differentiation (Assign. 1.1-2)

Numerical Calculus: Integration

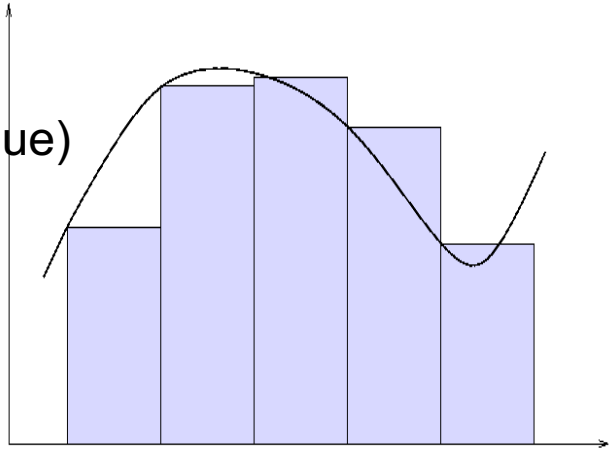
Developing coefficients c_0, c_1, ... of approximation polynomials in a *small* interval covered by just several computational points (samples of data or math function).

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.

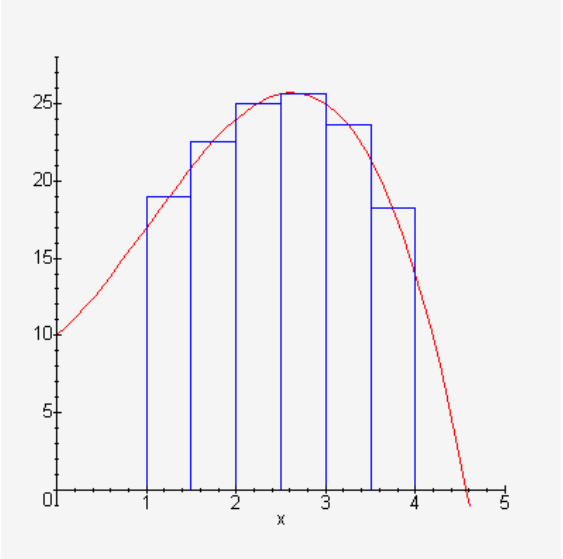
Require that method is of order m, and write m+1 equations (for n=0,1,2,...,m) binding m+1 coefficients of the method. Solve for c_n.

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)

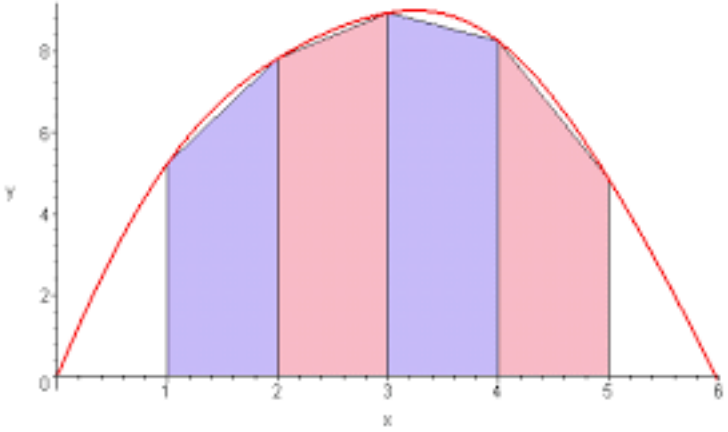
Euler(left value)
1st order

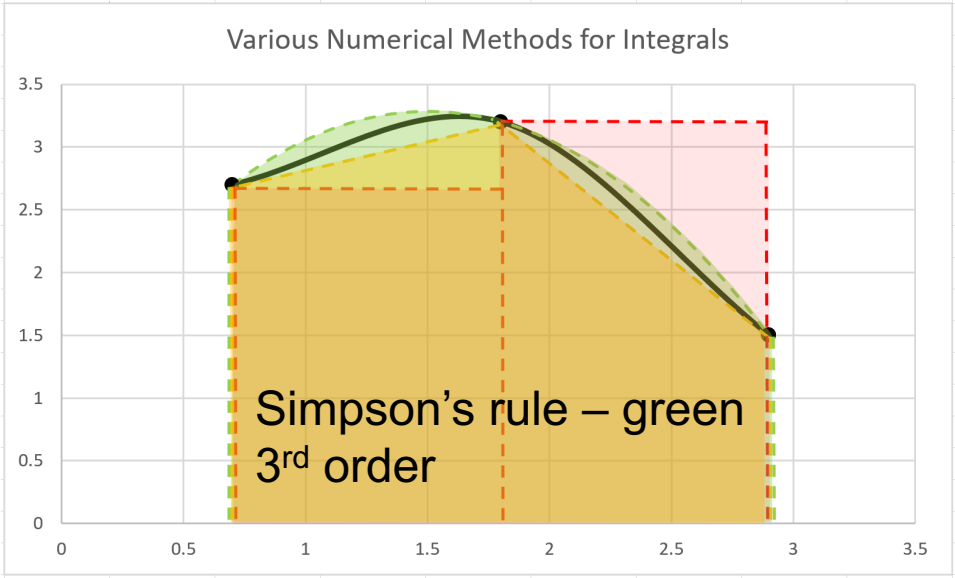


Midpoint method
2nd order



Trapezoid method
2nd order





Numerical integration of functions

see http://planets.utsc.utoronto.ca/~pawel/progD57 (also art-1: ~/progD57)

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

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,...

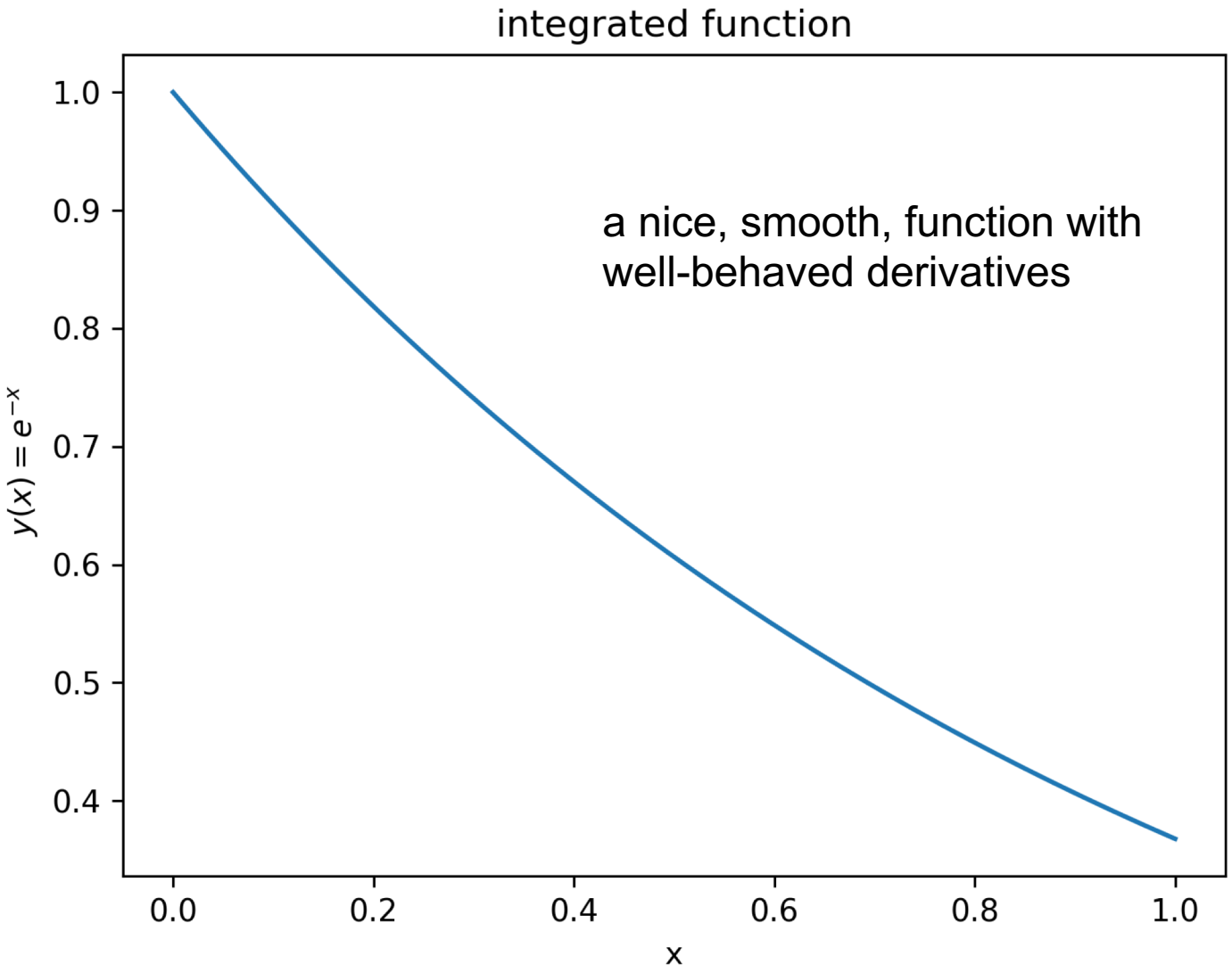
- Euler (left) $S = (f_0 + f_1 + f_2 + \ldots f_{n-1})\, h$
- Euler (right) $S = (f_1 + f_2 + f_3 + \ldots f_n)\, h$
- Midpoint $S = (f_{1/2} + f_{3/2} + \ldots + f_{n-3/2} + f_{n-1/2})\, h$, $f_{1/2} := f(\tfrac{1}{2}(x_0+x_1))$ etc.
- Trapezoid $S = (\tfrac{1}{2} f_0 + f_1 + f_2 + f_3 + \ldots + f_{n-1}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

- 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 $\sim N\, h^p \sim h^{p-1}$ (method of order p-1)

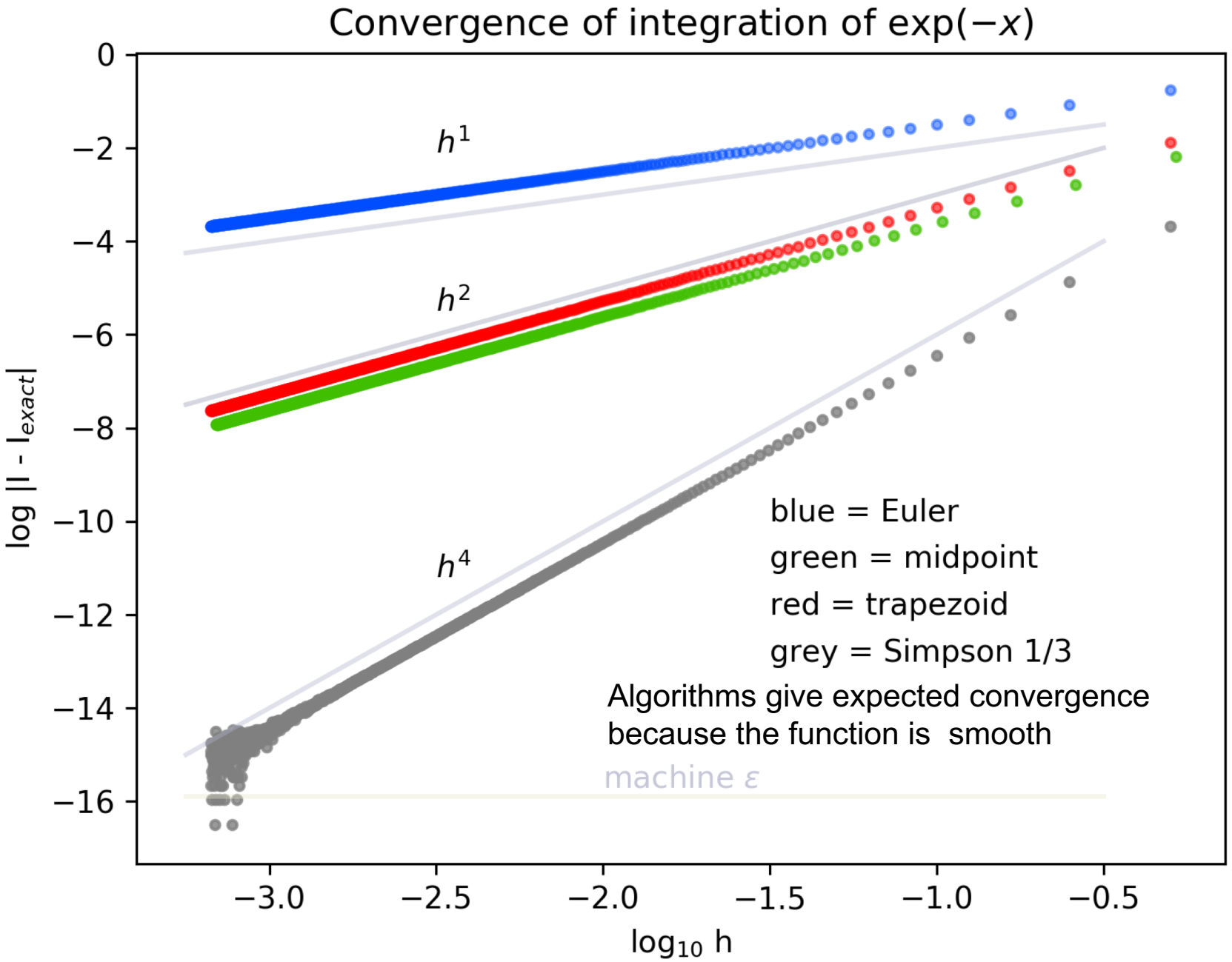
- Therefore, on the interval [a,b] we lose one power of h in accuracy scaling, or in other words one unit in the order of the method.

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



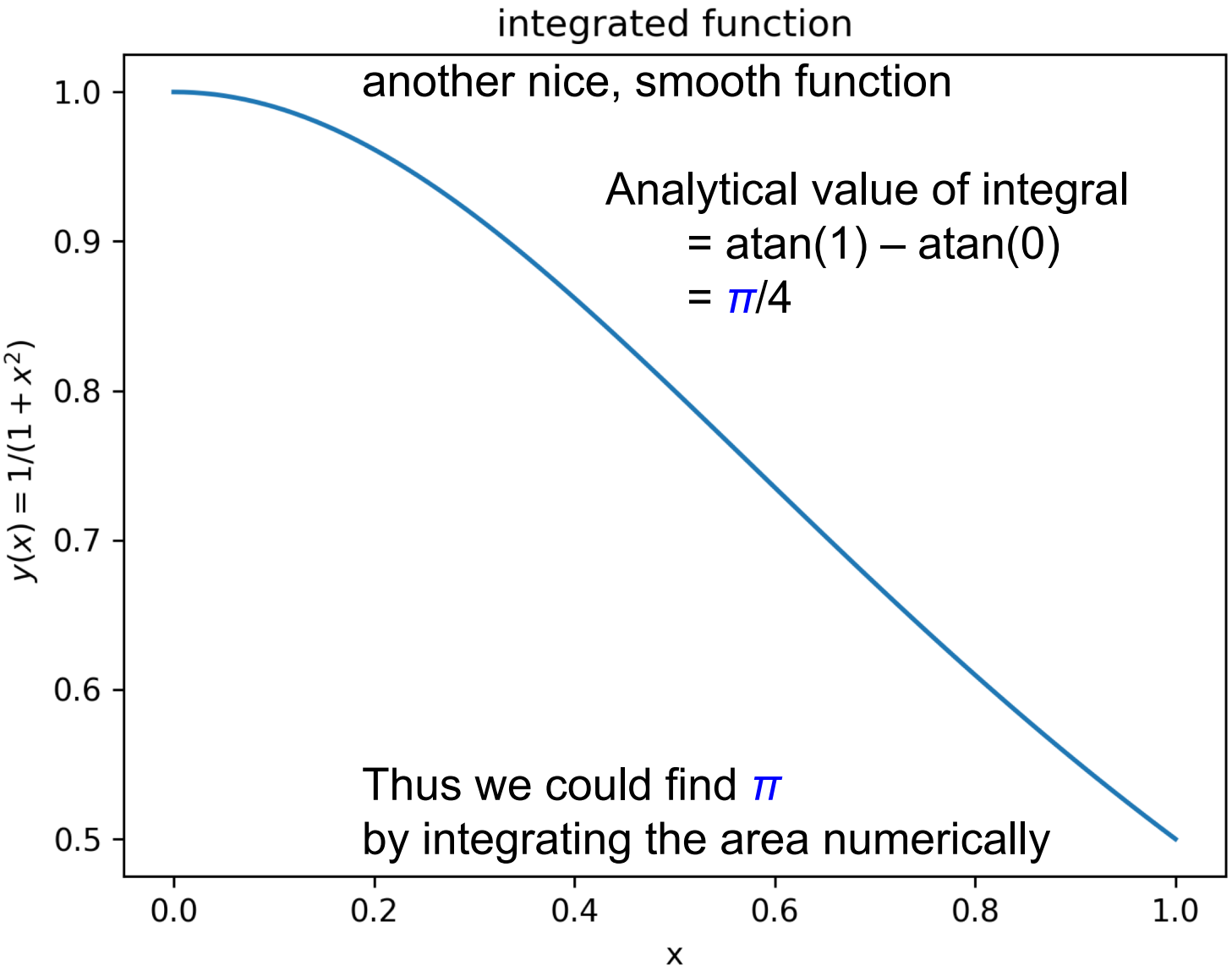
integ-p124-exp.py

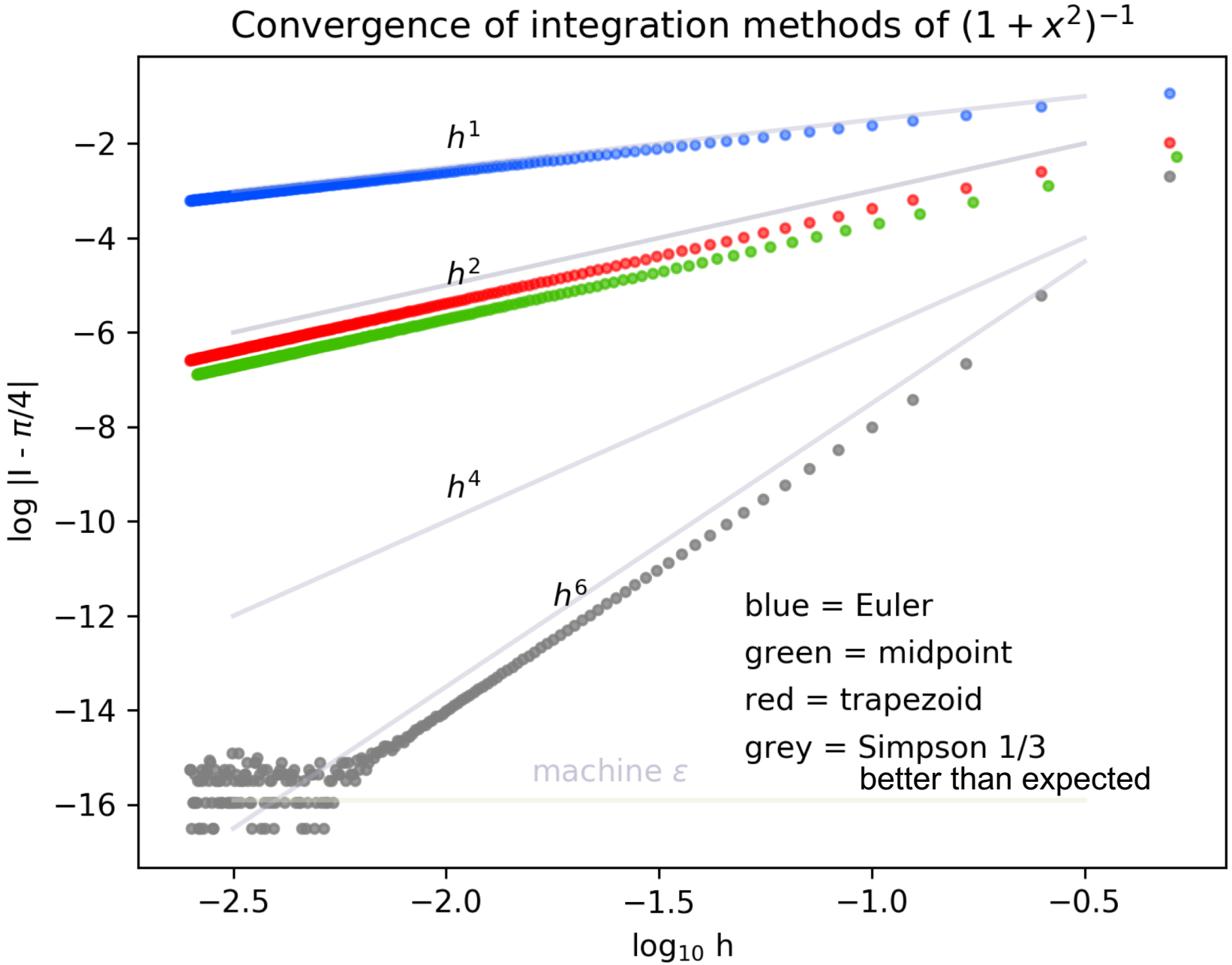
Everything works here as advertised!



Find area under function $1/(1+x^2)$ from 0 to 1:

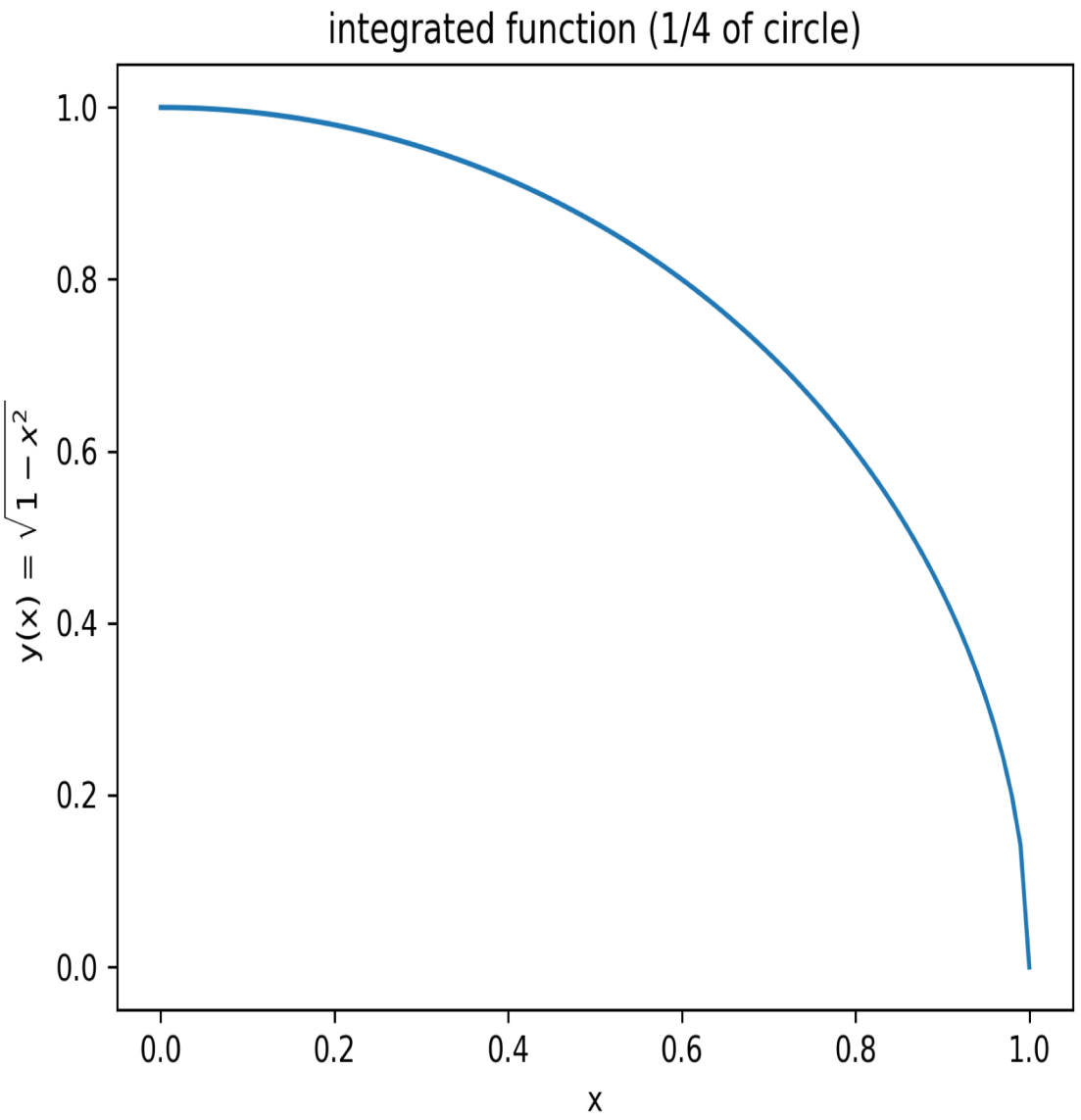
integ-p124-arc.py



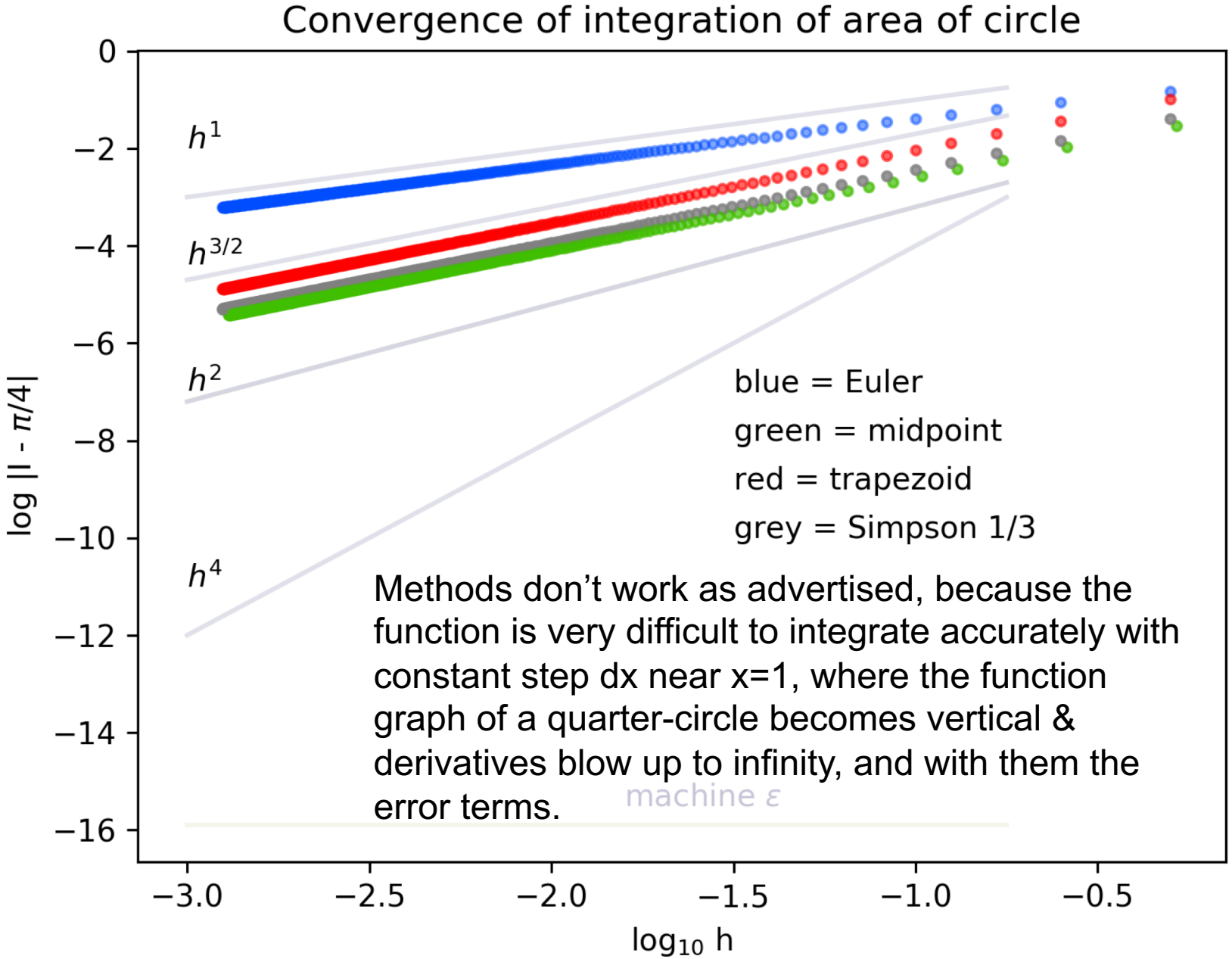


Again, everything works as advertised or better (Simpson's rule error~h^6 instead of h^4)!

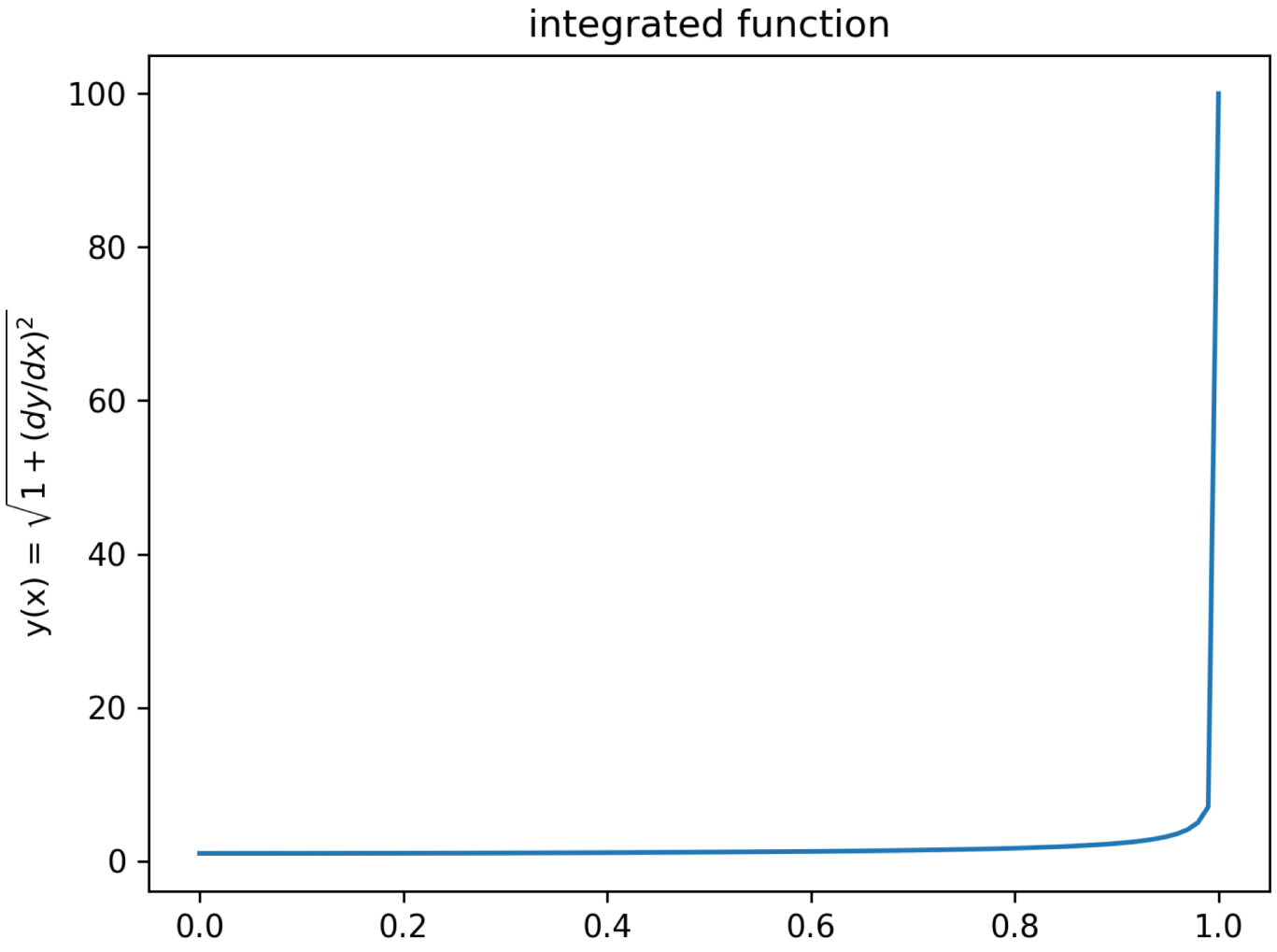
Integrate ¼ area of circle from 0 to 1: integ-p124-Acirc.py



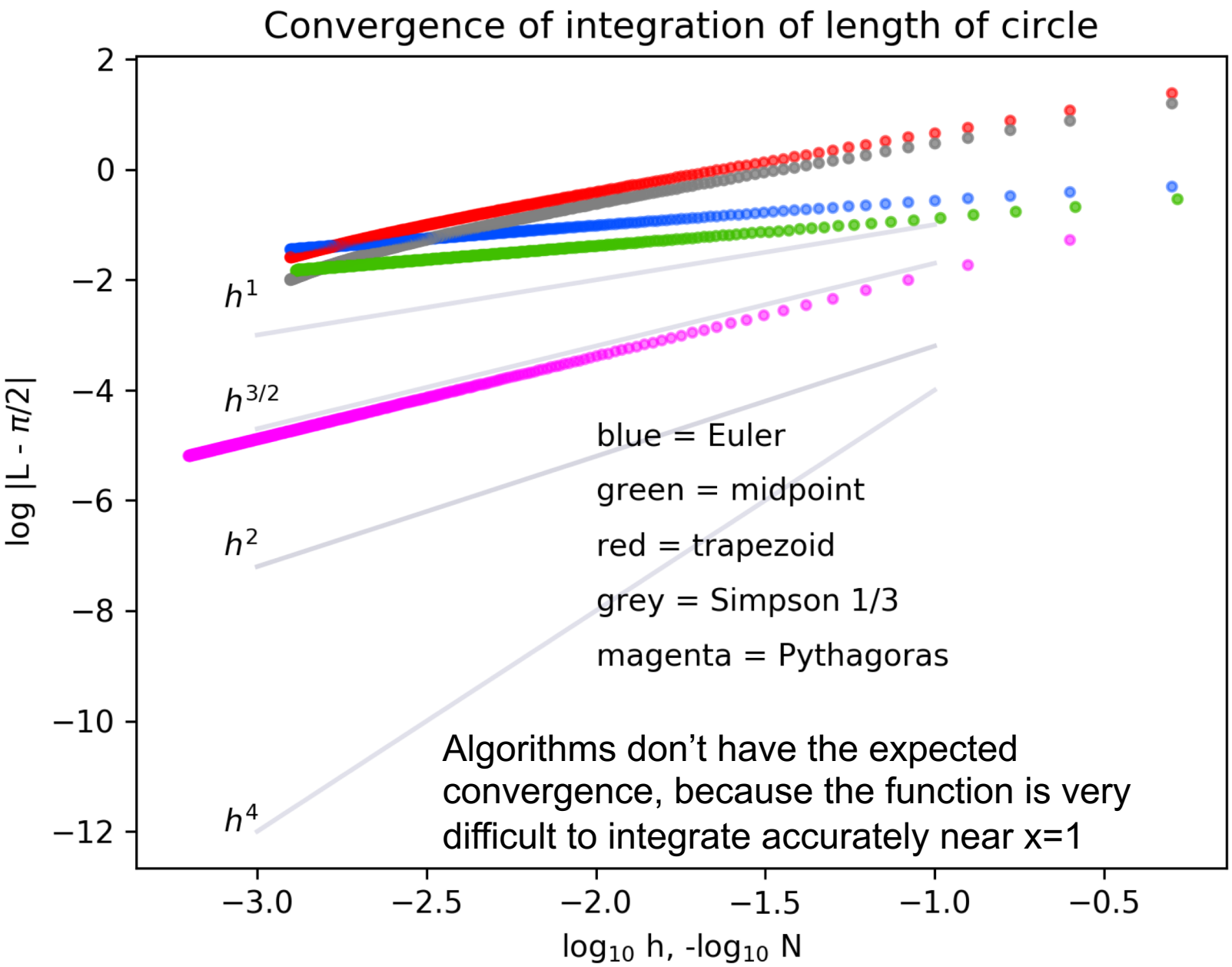
Notice the steep descent
near the end.
This will cause problems..


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
machine ε
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 with constant step dx near x=1, where the function graph of a quarter-circle becomes vertical & derivatives blow up to infinity, and with them the error terms.
−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



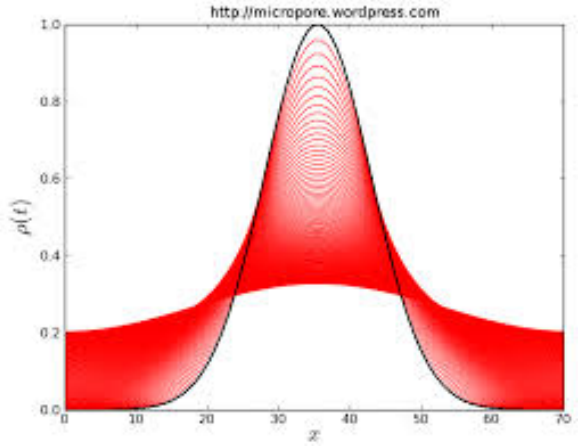
The sharp peak of dy/dx and other derivatives at x=1 promises trouble in this analytical approach to line integral giving the length of the circular curve (cf. the vertical axis label).



Breakdown of normal convergence rules of all the discussed methods *except* Pythagoras summation, which is less sensitive to derivatives. Error dominated by a few h intervals near x~1.

Application of numerical calculus to PDEs

Differentiation formulae (cf. assignment 1.1 & 1.2) provide ways to compute n-dimensional stencils for 1st and 2nd derivatives.
They include Laplacian operators such as
$(d^2/dx^2 + d^2/dy^2)\ f(x,y)$ in 2D, or
$(d^2/dx^2 + d^2/dy^2 + d^2/dz^2)\ f(x,y,z)$ in 3D.



$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} = \nabla \cdot \nabla = \nabla^2$$

This enables us to find curvature of functions that change in time.
In fact, time evolution of thing that diffuse, such as thermal energy (=temperature T), or concentration of fragrance in a room, or molecules in a container, is governed by diffusion equation, which says that:

$$\frac{\partial C(x,t)}{\partial t} = D\frac{\partial^2 C(x,t)}{\partial x^2}$$

or

The rate of change is equal to the divergence of the product of the diffusivity and the gradient

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u)$$

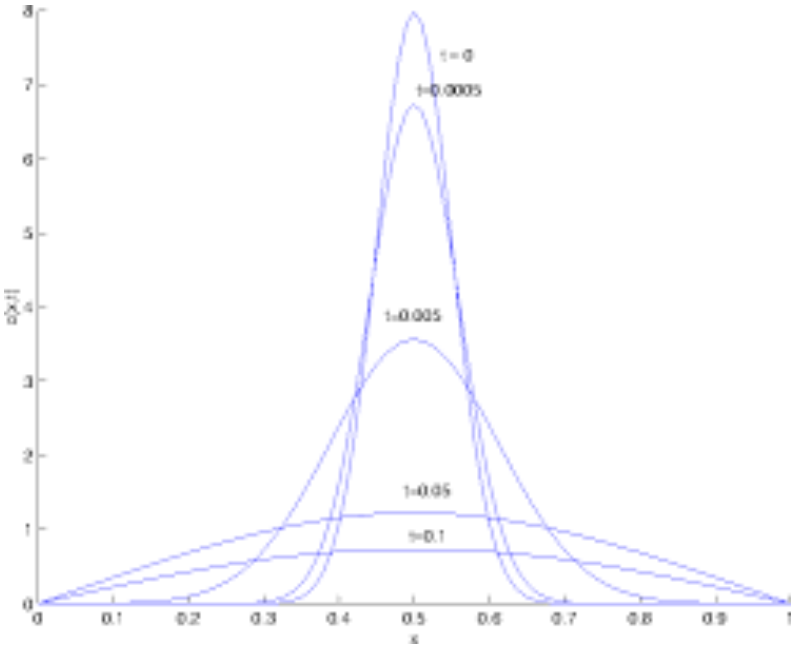
Application of numerical calculus: Diffusion equations

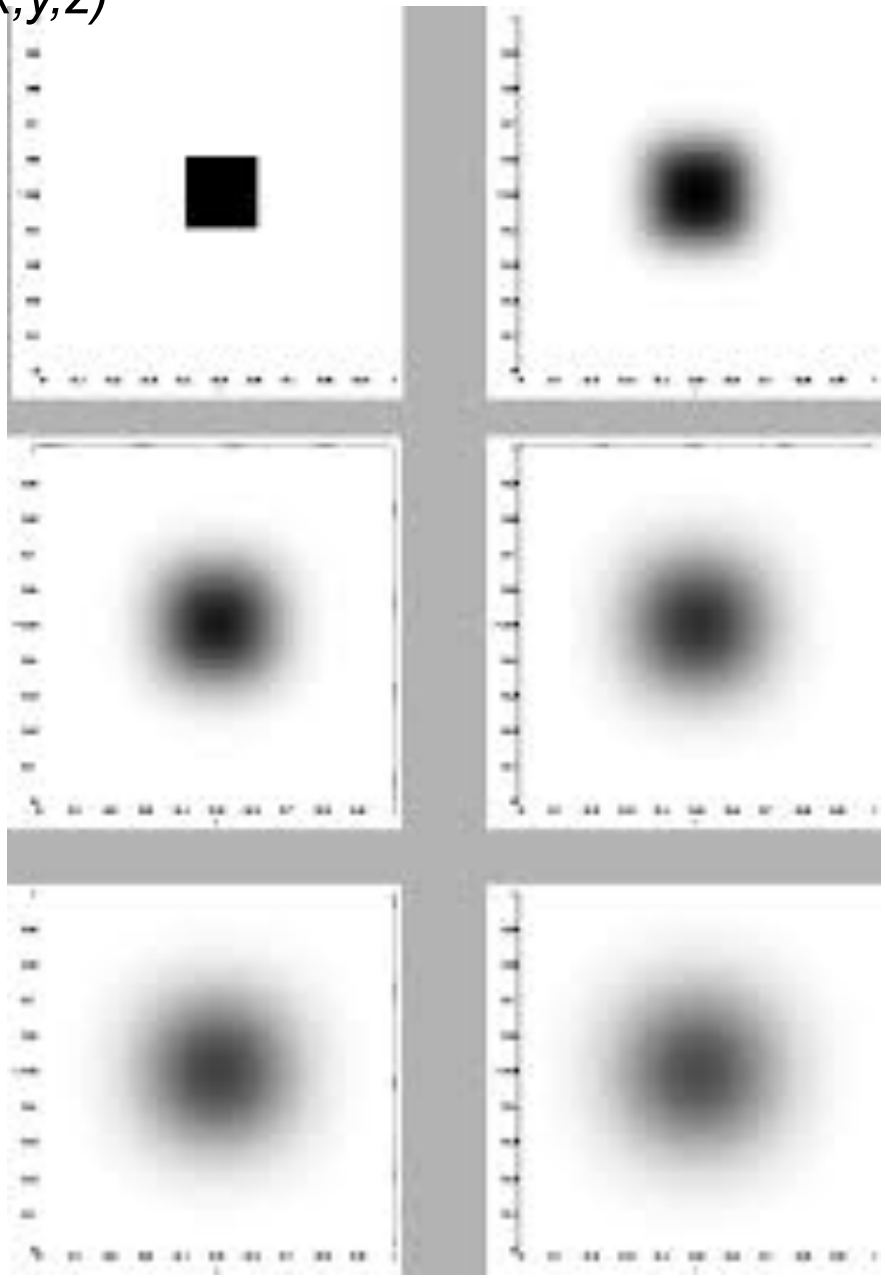
3D: $df/dt = D\,(d^2/dx^2 + d^2/dy^2 + d^2/dz^2)\,f(x,y,z)$

2D: $df/dt = D\,(d^2/dx^2 + d^2/dy^2)\,f(x,y)$

We will derive the criterion of stability for the numerical solution (2^{nd} order in both t and x)

$$\frac{\partial C}{\partial t} = D\nabla^2 C,$$

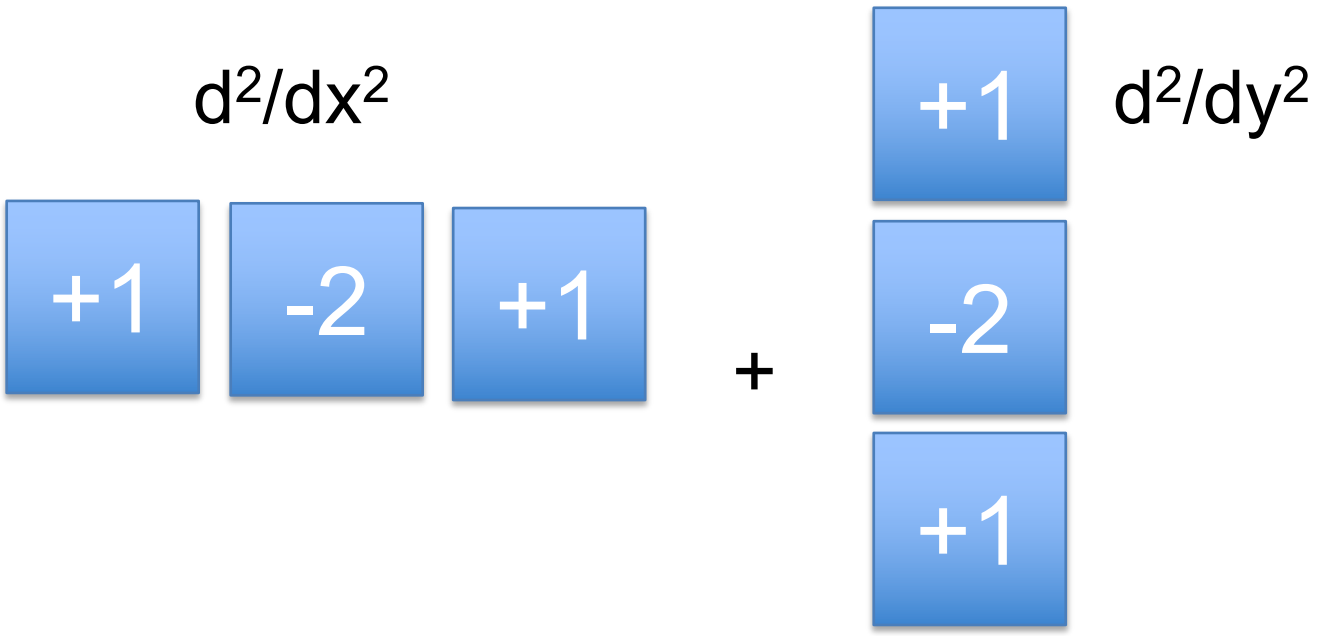




Application of numerical calculus: Image processing, blurring images

laplacian-4.py in our code repository

Used the basic second derivative stencil to do
$(d^2/dx^2 + d^2/dy^2)\ F(x,y)$ on an image $F(x,y)$.



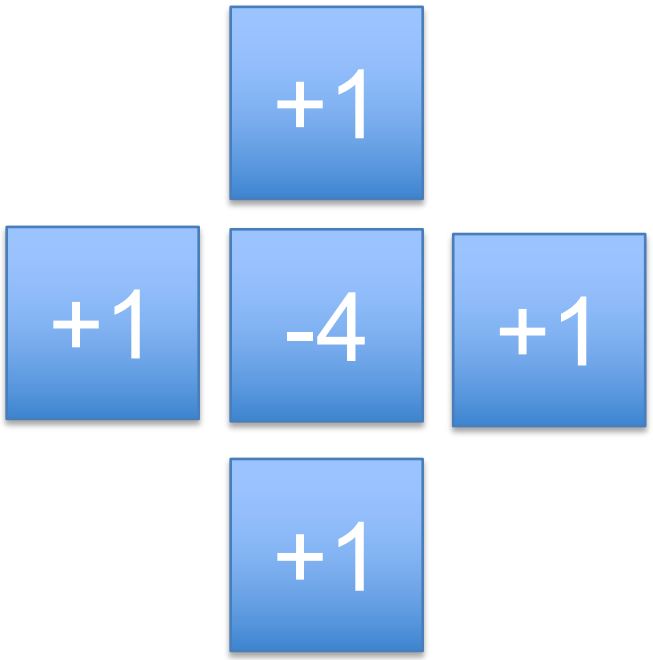
Application of numerical calculus: Image processing, blurring images

laplacian-4.py numerically solves a 2-D heat (diffusion) eq.

The basic second derivative stencil to do
$(d^2/dx^2 + d^2/dy^2)\, F(x,y)$ on an image $F(x,y)$.

ś

=



Stencil for Laplacian operator in 2D. Discretization scheme:

$$F(x,y,z,t+dt) = F(x,y,z,t) + dt\, D\, h^{-2}\, [F(x+dx) + F(x-dx) + F(y+dy) + F(y-dy) - 4F(x,y,z,t)]$$

Application of numerical calculus: Image processing, blurring

Solving $dF/dt = (d^2/dx^2 + d^2/dy^2)\, F(x,y)$,
beginning with an image F(x,y). Diffusion equation is discretized as

$$F(x,y,t+dt) = F(x,y,t) + dt\, D\, h^{-2} * \\ * [F(x+dx,y,t) + F(x-dx,y,t) + F(x,y+dy,t) + F(x,y-dy,t) - 4\, F(x,y,t)]$$

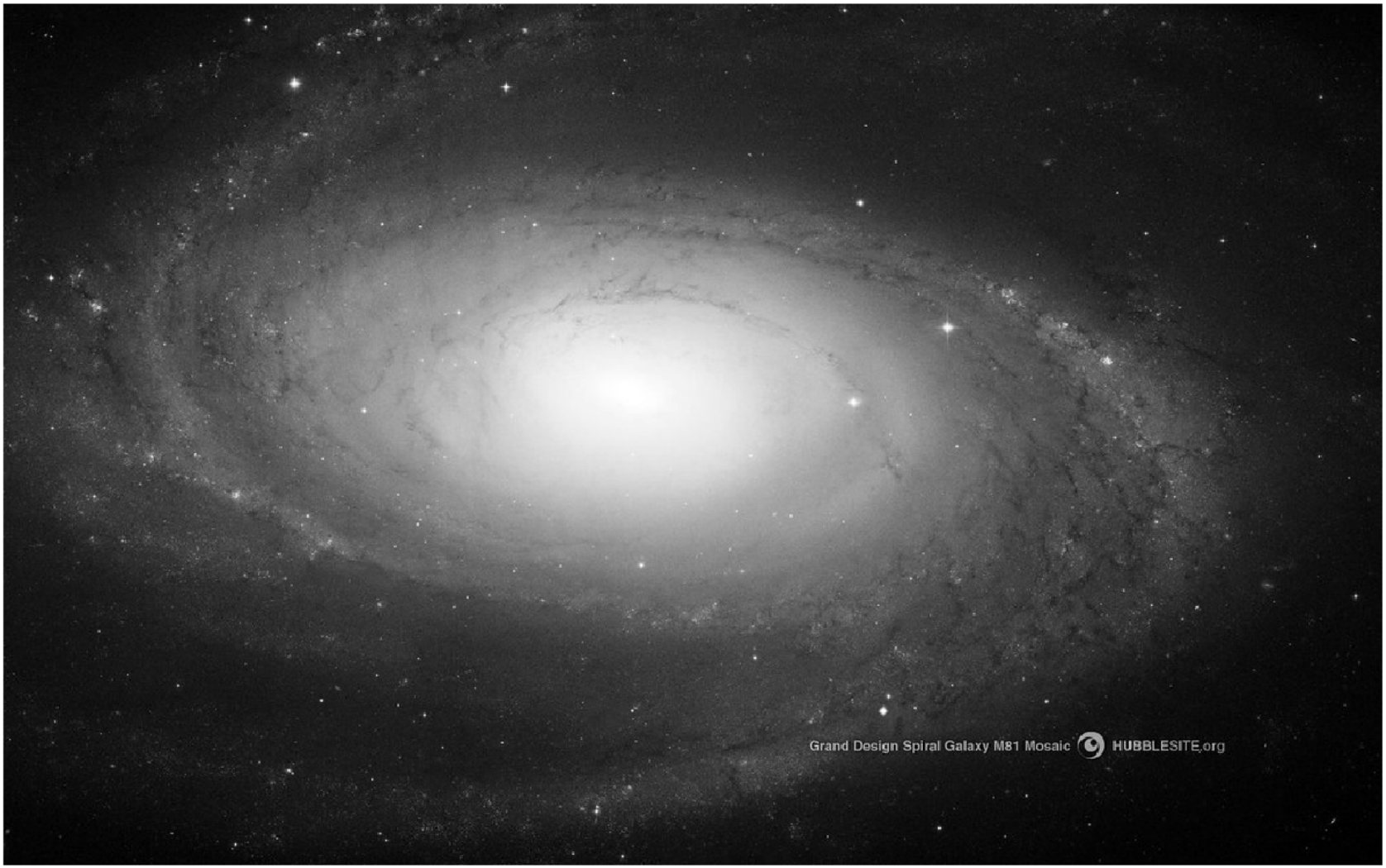
Let $q := dt\, D\, h^{-2}$, a nondimensional quantity.

$$F(x,y,t+dt) = (1-4q)\, F(x,y,t) + q\, [F(x+dx,y,t) + F(x-dx,y,t) + F(x,y+dy,t) + F(x,y-dy,t)]$$

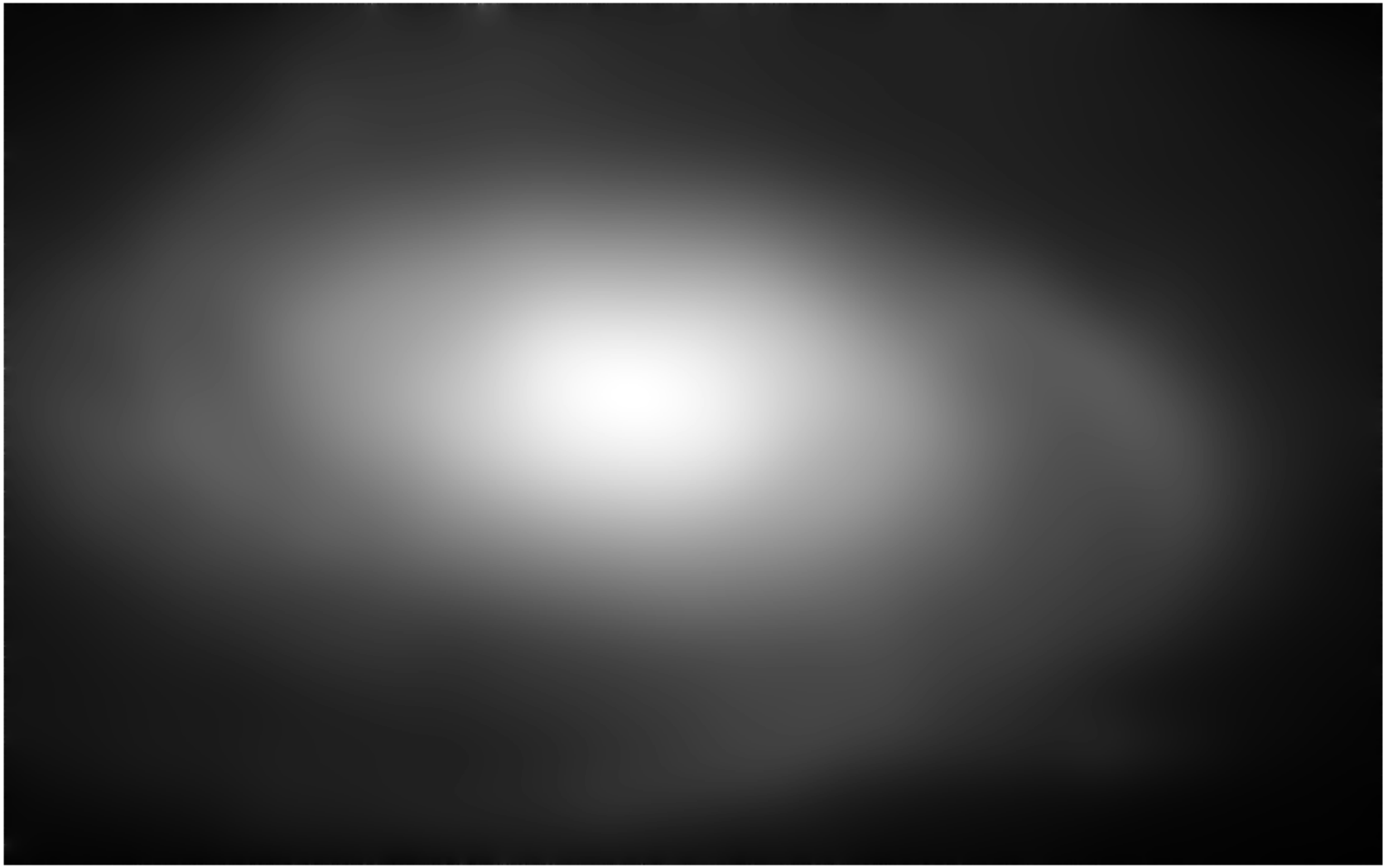
For stability of calculation, q < 1/4, otherwise negative F(x,y,t+dt) can appear, which for temperature, or concentration of particles, or images, is non-physical. We can prove the necessary criterion of stability: q < 1/(2n), where n = dimensionality of space, like this: Above we had 1-4q standing, in fact, for 1-2nq, and we need 1-2nq>0, so q < 1/(2n). Maximum *reasonable* q (you can call it: practical, sufficient criterion) is however 1/5 not ¼, and that q results in spreading of one single peak (1 pixel) to neighboring 5 pixels, each getting value 1/5: $1 - 4q = 1 - 4/5 = 1/5 = q$
If we went with the allowable q=1/4, we'd spread a single pixel into 4 neighbor cells but leave zero value in the center, in next time step. That would still work but it's weird.

Let's apply such a blurring procedure to astronomical image of grand-design spiral galaxy M81. laplacian-4.py

M81 galaxy

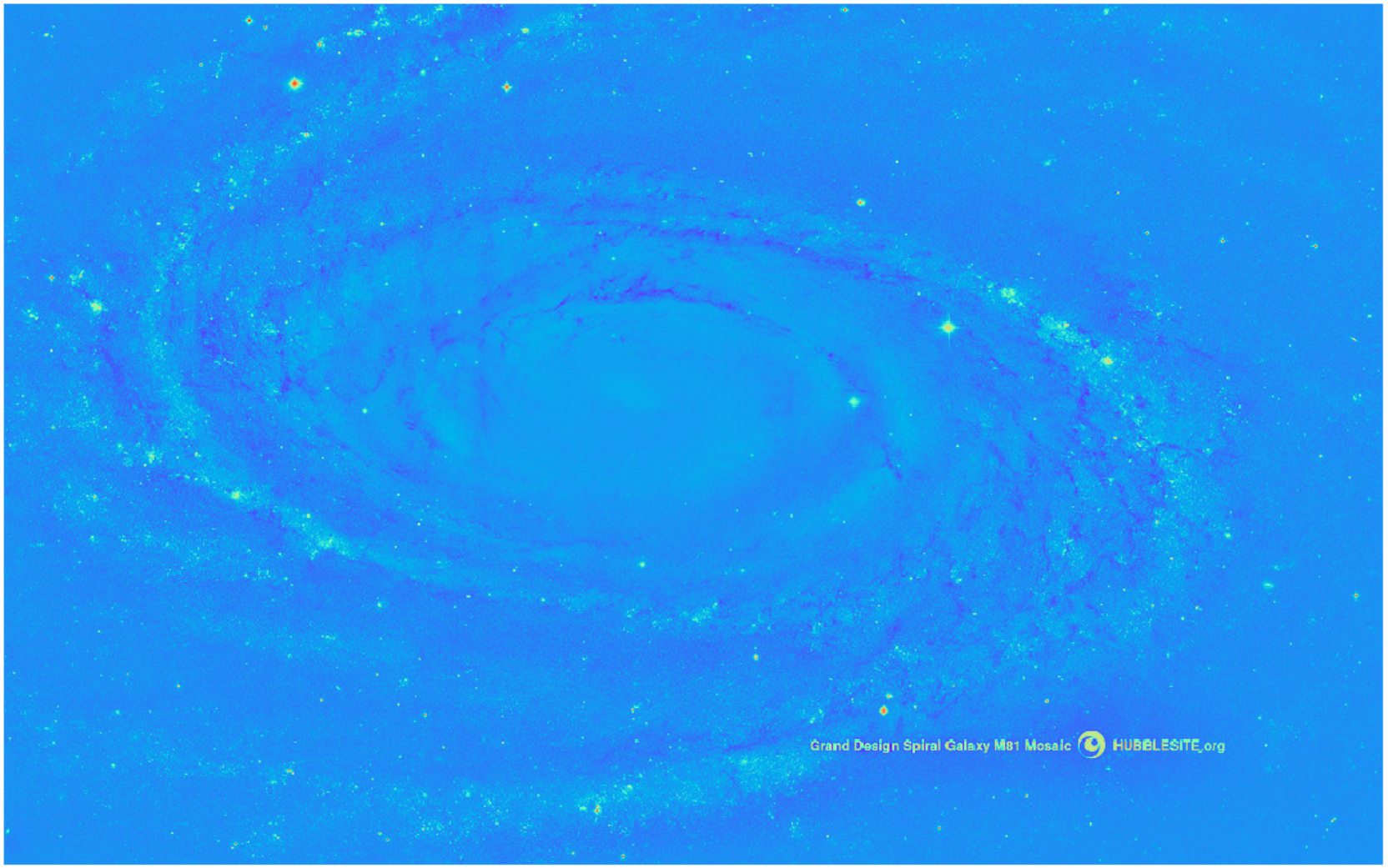


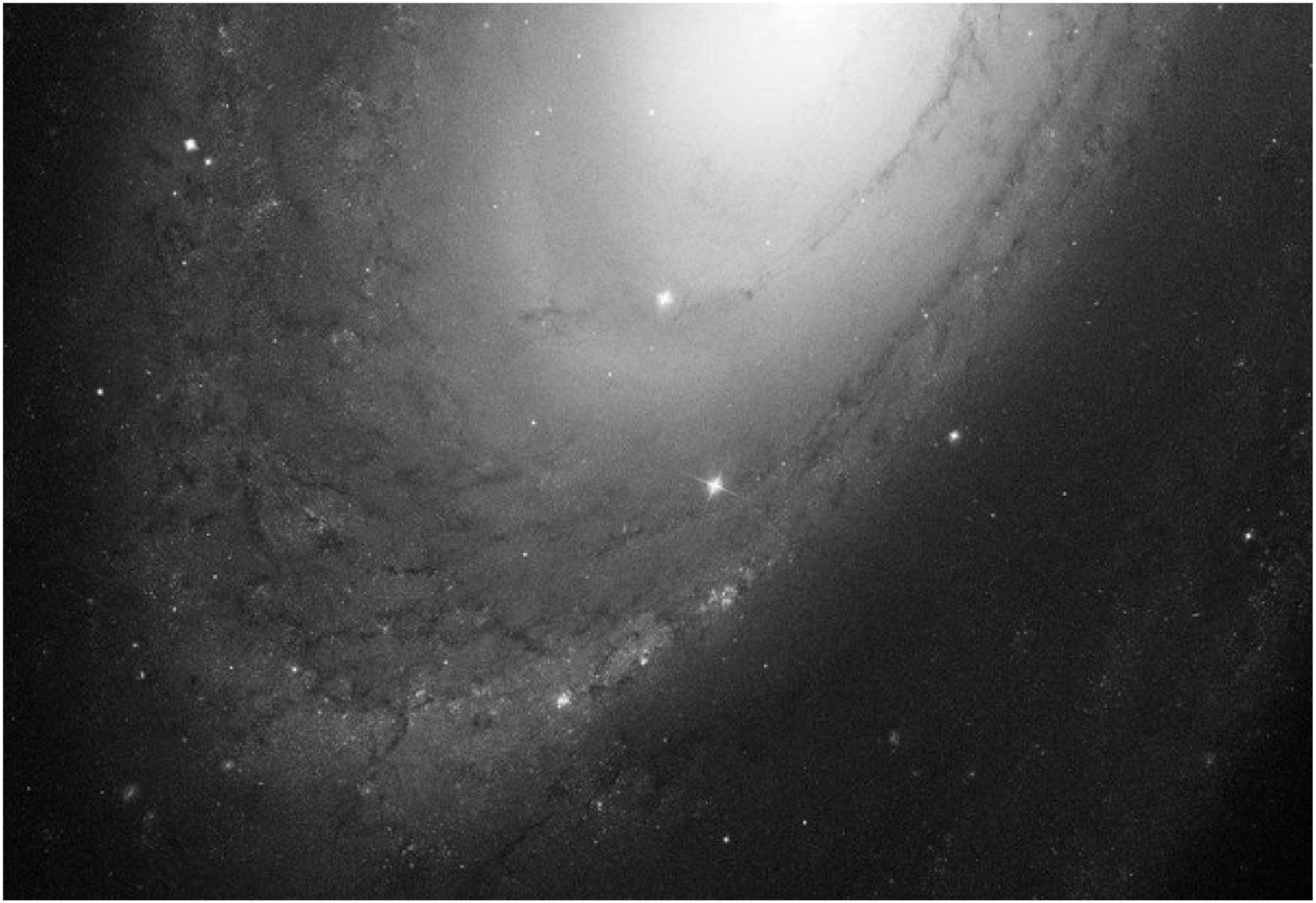
M81 after N=1200 Laplace iter. q=0.2

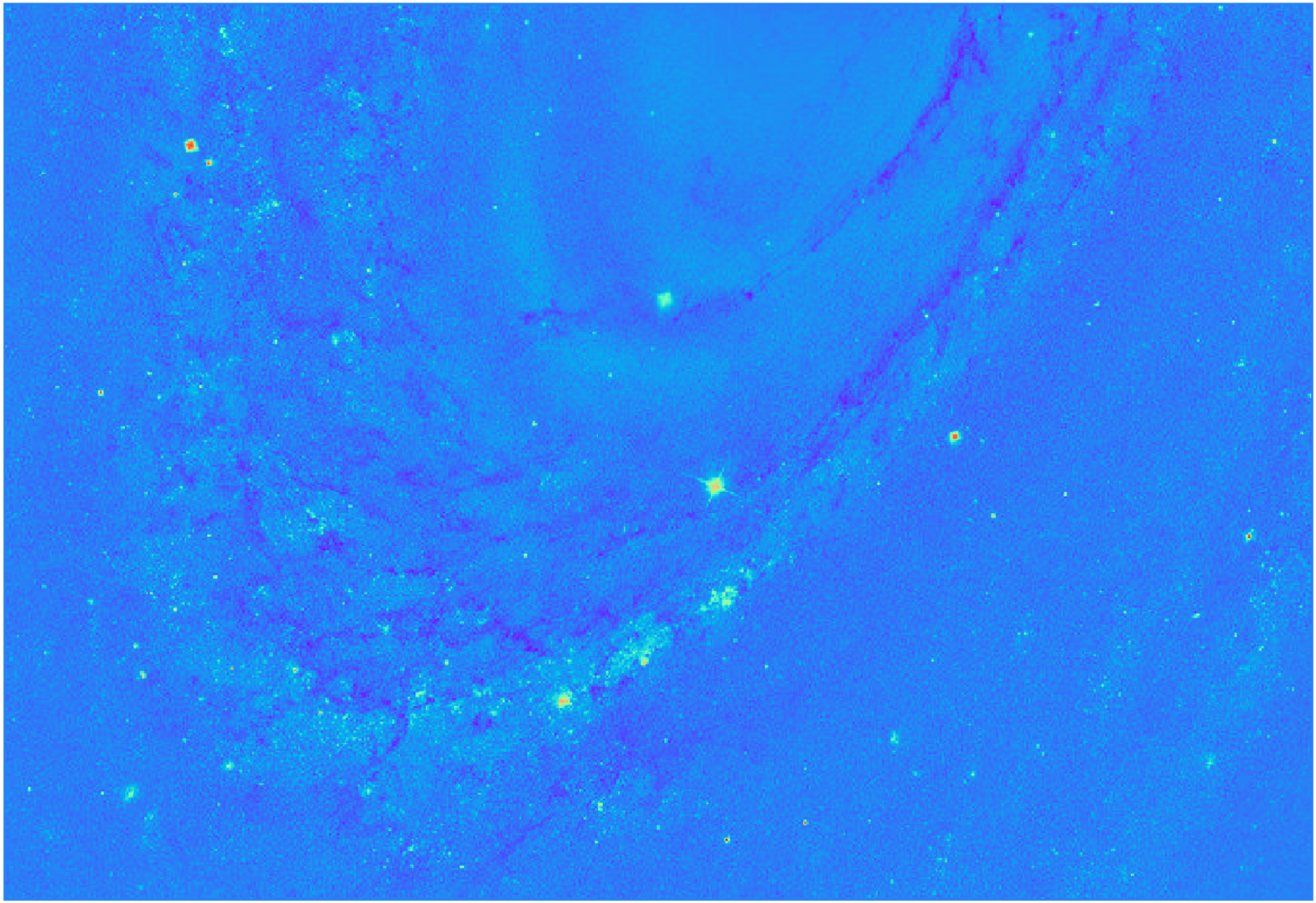


Unsharp masking technique:

Subtract a blurred image from the original image to remove the smooth background, enhance visibility of overexposed features.







Finally, consider programs in other languages below,
placed in art-1 subdirectory ~/progD57, i.e. /home/phyd57/progD57/
(art-1 has CentOS 6, and tcsh by default)

They perform the same task: to blur the monochromatic NxM pixel image,
according to a 3x3 cross stencil of a Laplacian operator Δf.

```
      q
q  (1-4q)  q      stencil of Δf operator, sum of coefficients must be & is = 1.
      q
```

Read, copy, compile with all possible compilers, and execute these programs

laplacian-5.py	laplacian-5t.py
ifor-laplace3-sp.f90	ifor-laplace3-dp.f90
cudafor-laplace3-sp.f95	cudafor-laplace3-dp.f95

(python3 programs, run on your own machine, as all modules aren't installed on art-1)

Example compilation with Intel ifort compiler. Code is in double precision (dp)

```
art-2[174]:~/progD57$ ifor-laplace3-dp.x
t=  2.797E-04 s       3654 fps,  val=  0.2769022      host OMP
t=  2.691E-04 s       3716 fps,  val=  0.2769022      host OMP
t=  3.757E-04 s       2661 fps,  val=  0.2769022      host OMP i
t=  3.264E-04 s       3063 fps,  val=  0.2769022      host slices
t=  2.532E-04 s       3949 fps,  val=  0.2769022      host slices 1
t=  9.322E-04 s       1072 fps,  val=  0.2769022      host squares
t=  8.695E-04 s       1150 fps,  val=  0.2769022      host
t=  3.720E-03 s        268 fps,  Numpy (written in C or F)
t=  0.1562400 s            6.4 fps,  plain Python
```

Time t is the execution of 1 Laplacian blurring pass on array of 1 M pixels in seconds. The corresponding number of sweeps per second is given as fps = frames per sec.

Results show that this problem is solved extremely slowly by 2 nested Python loops (6.4 Mpix/s), 42 times faster by NumPy methods of vectorized array operations (268 Mpix/s), but even much much faster in Fortran (different methods mentioned in the rightmost column).

This however is beaten by single precision versions of the program and by a CUDA (GPU version)!

Example compilation with pgf95 PGI compiler. Code in double precision (dp) run on CPU = i7 6 cores, 4GHz overclock, and Nvidia GTX 1080ti graphics card.

```
art-2[177]:~/progD57$ nvidia-smi –p 1
(it's good to make sure that the so-called persistence mode of the
Nvidia driver is on, or switch it on as above)
art-2[177]:~/progD57$ cudafor-laplace3-dp.x
t=   5.7930E-05 s       17261 fps,  val=   0.3449662 CUDA kernel 0
t=   5.7928E-05 s       17262 fps,  val=   0.3449662 CUDA kernel 1
t=   3.1855E-04 s        3139 fps,  val=   0.3449662 host OMP slices
t=   3.7507E-04 s        2666 fps,  val=   0.3449662 host OMP
t=   8.8273E-04 s        1132 fps,  val=   0.3449662 host (CPU)
t=   3.7199E-03 s         268 fps,  Numpy dp
t=   0.1562400  s         6.4 fps,  Python out of the box
```

Time t refers to execution of one sweep of Laplacian blurring operator on array of 1 M pixels, and the corresp. fps or frames per sec. are shown (i.e. Mpix/s)

Results show that this problem is solved extremely fast by PGI CUDA Fortran (~1000x faster than Python, and 23x faster than Numpy)

This can only be beaten by a single precision CUDA run....

Example compilation with pgf95 PGI compiler. Code in single precision (sp) run on host CPU = i7, 6 cores, 4GHz overclock, and Nvidia GTX 1080ti GPU.

```
art-1[183]:~/progD57$  cudafor-laplace3-sp.x
 t=   3.4939E-05        28621 fps,  val=   0.3449661 CUDA kernel 0
 t=   3.4926E-05        28631 fps,  val=   0.3449661 CUDA kernel 1
 t=   3.4950E-05        28612 fps,  val=   0.3449661 host OMP slices
 t=   1.0031E-04         9963 fps,  val=   0.3449661 host OMP
 t=   3.6699E-04         3756 fps,  val=   0.3449661 host (CPU)
 t=   3.7199E-03          268 fps,  Numpy dp
 t=   1.5624E-01          6.4 fps,  Python as is
```

Maximum speed of Laplacian blurring seems to be 29k fps = 29 Gpix/s. (This value, however, is suspiciously constant in 3 different runs. A possible problem with the accuracy of internal timer! The speed values may be in error, but not their order of magnitude.)

If each pixel requires about 10 arithmetic operations to be updated using values from its neiborhood, then CUDA Fortran processes the image at about 290 GFLOPs.

This is an impressive speed, even though even not close to ~7 TFLOPs = 7000 GFLOP of which the 1080ti card is actually capable in computationally dominated tasks, applications that do not rely (as this code does) on the GPU-RAM bandwidth.

On the same server, the sp calculation can be done ~4500x faster on GPU than in (dp) Python on CPU, and ~100x faster than on the same CPU in Numpy.

This record can further be beaten by parallel computation on a multi-node cluster, especially with a faster hardware. (But the best hardware, CPUs and GPUs, in 2024 was no more than about 3 times faster than we have on art-1.)

We have proven that PYTHON IS NOT AN HPC LANGUAGE. That‘s ok, since it was never meant for large computations. This is also one of the main reasons why you take this course: to go beyond Python in programming, as well as encounter new algorithms, methods, and Linux shells.