PDEs and Fourier Transforms!

PHYS 250 (Autumn 2019) – Lecture 12

David Miller

Department of Physics and the Enrico Fermi Institute
University of Chicago

November 13, 2019

Outline

Reminders from last time

We took the simple examples that we had derived in Lecture 10 for extending Newton's methods to higher orders and then had a little hands on session for both ODEs and subsequent PDEs.

Extension of Newton's methods to high-order Runge-Kutta and then PDEs

- **Runge-Kutta:**
  - We tested the prediction that we would see $\mathcal{O}(h^2)$ and $\mathcal{O}(h^4)$ scaling for the higher order (*success!*)
  - We implemented the pendulum simulation and looked at the departure from the simple solution at long times
- **Partial differential equations:**
  - Started discussing the concepts and families of PDEs (elliptic, hyperbolic, parabolic)
  - Looked at a simple example of an elliptic equation: Laplace

Today we will follow-up with PDEs and then look at Fourier Transforms, but using the ODE example of a pendulum!

Outline

Runge-Kutta family of algorithms

The aim of Runge-Kutta methods is to eliminate the need for repeated differentiation of the differential equations. Because no such differentiation is involved in the **first-order Taylor series** expression:

$$\vec{y}(x+h) = \vec{y}(x) + \vec{y}'(x)h = \vec{y}(x) + \vec{F}(x,\vec{y})h \tag{1}$$

This first-order version is referred to as **Euler's method (aka Euler's Rule)**.

Effectively, what this is doing is to start with the known initial value of the dependent variable, $y_0 \equiv y(x=0)$, and then use the derivative function $f(x,y)$ to find an approximate value for y at a small step $x = h$ forward in time; that is, $y(x=h) \equiv y_1$.

We know from our discussion of differentiation that the error in the forward-difference algorithm is $\mathcal{O}(h)$, and so this too is the error in Euler's rule.

How can we test this?

Code up the algorithm!

ExplicitEuler

```
    for i, x_i in enumerate(x[:-1]):

        h = x[i+1] - x_i
        y[i+1] = y[i] + h*func(x_i, y[i], args)

    return y
```

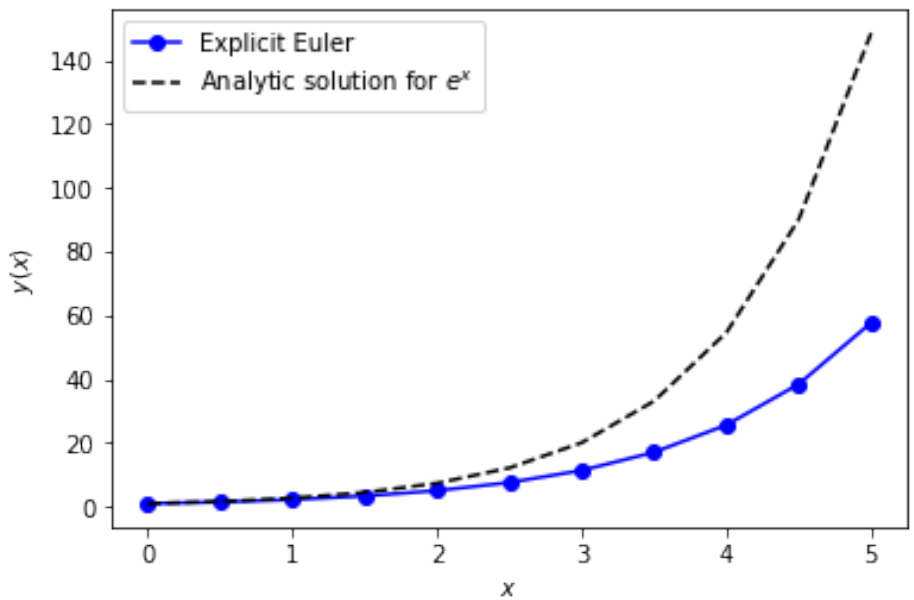
Code up the algorithm!

ExplicitEuler

```
    for i, x_i in enumerate(x[:-1]):

        h = x[i+1] - x_i
        y[i+1] = y[i] + h*func(x_i, y[i], args)

    return y
```



Improve the accuracy

We know that the accuracy depends on h so make that smaller?

ExplicitEuler

```
for N in [5, 10, 20, 40]:
    x = np.linspace(0., x_max, N+1) # Time steps
    y = ExplicitEuler(exp, y_0, x, solve_args)
    plt.plot(x, y, '-o', label='%d steps'%N)

plt.plot(x,np.exp(solve_args['a']*x),'k--',label='Known')
plt.xlabel(r'$x$'), plt.ylabel(r'$y$')
plt.legend(loc=2)
```

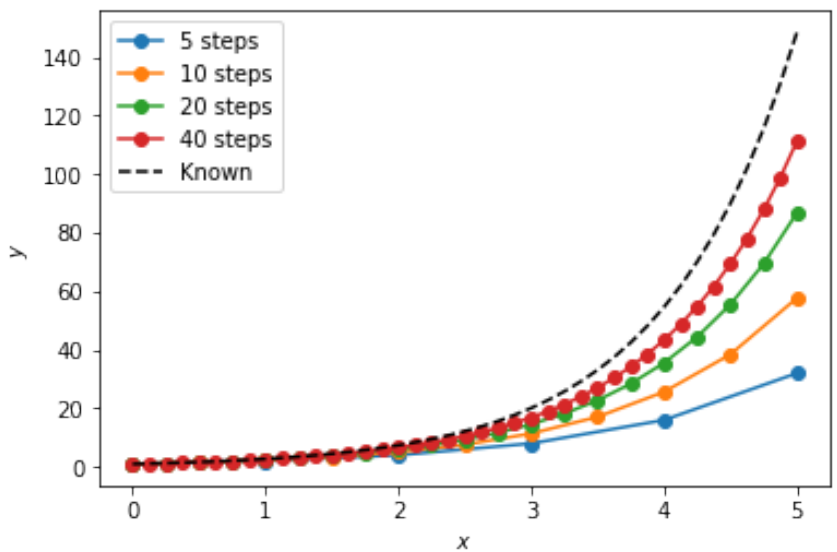
Improve the accuracy

We know that the accuracy depends on h so make that smaller?

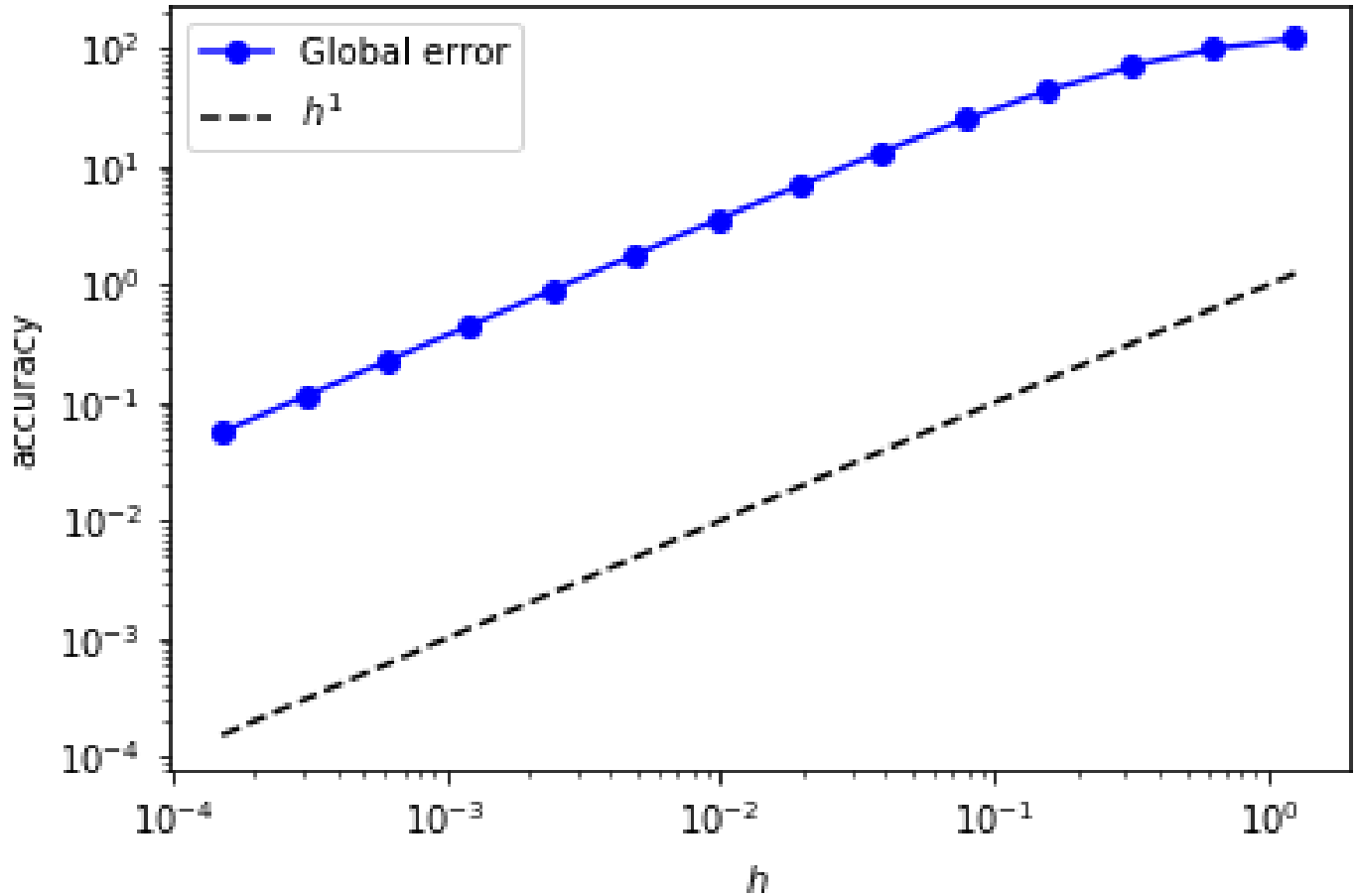
ExplicitEuler

```
for N in [5, 10, 20, 40]:
    x = np.linspace(0., x_max, N+1) # Time steps
    y = ExplicitEuler(exp, y_0, x, solve_args)
    plt.plot(x, y, '-o', label='%d steps'%N)

plt.plot(x,np.exp(solve_args['a']*x),'k--',label='Known')
plt.xlabel(r'$x$'), plt.ylabel(r'$y$')
plt.legend(loc=2)
```



Error of the method



Second order Runge-Kutta family of algorithms

Recall that the key insight was to expand $f(x,y)$ in a Taylor series about the **midpoint of the integration interval** and retain two terms:

$$f(x,y) \simeq f(x_{n+1/2}, y_{n+1/2}) + (x - x_{n+1/2}) \frac{df}{dx}(x_{n+1/2}) + \mathcal{O}(h^2) \qquad (2)$$

As you recall from the **finite central difference** algorithm, only **odd powers** of h remain, and thus when used inside the integral above, the terms with $(x - x_{n+1/2})^{n \in \text{odd}}$ vanish. We are left with

$$y_{n+1} \simeq y_n + hf(x_{n+1/2}, y_{n+1/2}) + \mathcal{O}(h^2) \qquad (3)$$

How can we test this?

Outline

1. *Reminders*
   - Reminders from Lecture 11
2. *Hands on with the Runge-Kutta family of methods*
   - Reminders
   - Code up the algorithm
   - Test the accuracy of the methods
3. *Partial Differential Equations*
   - Statement of the problem
   - Classes and boundary conditions for PDEs
   - Example: electrostatic potentials

What's so different about PDEs?

Physical quantities such as temperature, pressure, gravitational and electromagnetic forces, and more, may vary continuously in both space and time.

As such, the functions or **fields,** $U(x, y, z, t)$ that describe these quantities must contain **independent space and time variations**.

As time evolves, the changes in $U(x, y, z, t)$ at any one position affect the field at neighboring points.

This means that the dynamic equations describing the dependence of U on four independent variables must be written in terms of partial derivatives, and therefore the equations must be **partial differential equations (PDEs)**, in contrast to ordinary differential equations (ODEs).

General form of PDEs

The most general form for a two-independent variable PDE is

$$A\,\frac{\partial^2 U}{\partial x^2} + 2B\,\frac{\partial^2 U}{\partial x \partial y} + C\,\frac{\partial^2 U}{\partial y^2} + D\,\frac{\partial U}{\partial x} + E\,\frac{\partial U}{\partial y} = F,$$

where A, B, C, and F are arbitrary functions of the variables x and y.

General form of PDEs

The most general form for a two-independent variable PDE is

$$A\,\frac{\partial^2 U}{\partial x^2} + 2B\,\frac{\partial^2 U}{\partial x \partial y} + C\,\frac{\partial^2 U}{\partial y^2} + D\,\frac{\partial U}{\partial x} + E\,\frac{\partial U}{\partial y} = F,$$

where A, B, C, and F are arbitrary functions of the variables x and y. These typically fall into three categories based on the structure of the derivatives in the expression:

Elliptic	*Parabolic*	*Hyperbolic*
$d = AC - B^2 > 0$	$d = AC - B^2 = 0$	$d = AC - B^2 < 0$
$\nabla^2 U(x) = -4\pi\rho(x)$	$\nabla^2 U(\mathbf{x}, t) = a\,\partial U/\partial t$	$\nabla^2 U(\mathbf{x}, t) = c^{-2}\partial^2 U/\partial t^2$
Poisson's	Heat	Wave

Boundary conditions for PDE classes

Separately, the boundary conditions will place specific constraints on the relationships between each of the terms, and often dictate whether the problem is even solvable.

Boundary Condition	*Elliptic (Poisson Equation)*	*Hyperbolic (Wave Equation)*	*Parabolic (Heat Equation)*
Dirichlet open surface	Underspecified	Underspecified	*Unique & stable (1-D)*
Dirichlet closed surface	*Unique & stable*	Overspecified	Overspecified
Neumann open surface	Underspecified	Underspecified	*Unique & Stable (1-D)*
Neumann closed surface	*Unique & stable*	Overspecified	Overspecified
Cauchy open surface	Nonphysical	*Unique & stable*	Overspecified
Cauchy closed surface	Overspecified	Overspecified	Overspecified

- **Dirichlet:** value of U on a surface
- **Neumann:** value of the normal derivative of U on a surface
- **Cauchy:** value of both U and U' on a surface

Example from your courses: Laplace in E&M

The Laplace equation is of fundamental importance in physics, and is most often first encountered in electrodynamics.

$$\nabla^2 V = 0. \tag{4}$$

In E&M, we often taught the solution to Laplace's equation in two dimensions, $V(x,y)$, which can be found by evaluating:

$$V(x,y) = \frac{1}{2\pi R} \oint_{\text{circle}} V dl \tag{5}$$

due to the lack of sources. In fact, even in common textbooks (e.g. Griffiths) you are taught that you can solve this iteratively, by evaluating this integral until a change in $V(x,y)$ on successive evaluations of the expression is smaller than some tolerance.

Method of relaxation

To solve our 2D PDE numerically, we divide space up into a lattice (recall: `meshgrid`!) and solve for V at each site on the lattice. Since we will express derivatives in terms of the finite differences in the values of V at the lattice sites, this is called....

Method of relaxation

To solve our 2D PDE numerically, we divide space up into a lattice (recall: `meshgrid`!) and solve for V at each site on the lattice. Since we will express derivatives in terms of the finite differences in the values of V at the lattice sites, this is called....**a finite-difference method**!

Method of relaxation

To solve our 2D PDE numerically, we divide space up into a lattice (recall: `meshgrid`!) and solve for V at each site on the lattice. Since we will express derivatives in terms of the finite differences in the values of V at the lattice sites, this is called....**a finite-difference method**!

To derive the finite-difference algorithm for the numeric solution of this PDE, we would follow the same path taken before, and add the two Taylor expansions of the potential to the right and left of (x, y) and above and below (x, y) and so on.

At this point, I think it's just worth "writing" some code...