Lecture 19: Ordinary Differential Equations: Introduction

3.016

Reading:
Kreyszig Sections: 1.1, 1.2, 1.3 (pages2–8, 9–11, 12–17)

Differential Equations: Introduction

Ordinary differential equations are relations between a function of a single variable, its derivatives, and the variable:

$$F\left(\frac{d^n y(x)}{dx^n}, \frac{d^{n-1} f(x)}{dx^{n-1}}, \ldots, \frac{d^2 y(x)}{dx^2}, \frac{dy(x)}{dx}, y(x), x\right) = 0 \tag{19-1}$$

A *first-order* Ordinary Differential Equation (ODE) has only first derivatives of a function.

$$F(\frac{dy(x)}{dx}, y(x), x) = 0 \tag{19-2}$$

A *second-order* ODE has second and possibly first derivatives.

$$F\left(\frac{d^2 y(x)}{dx^2}, \frac{dy(x)}{dx}, y(x), x\right) = 0 \tag{19-3}$$

For example, the one-dimensional time-independent Shrödinger equation,

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) = E\psi(x)$$

or

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) - E\psi(x) = 0$$

3.016 Home
Full Screen
Close
Quit

is a second-order ordinary differential equation that specifies a relation between the wave function, $\psi(x)$, its derivatives, and a spatially dependent function $U(x)$.

Differential equations result from physical models of anything that varies—whether in space, in time, in value, in cost, in color, etc. For example, differential equations exist for modeling quantities such as: volume, pressure, temperature, density, composition, charge density, magnetization, fracture strength, dislocation density, chemical potential, ionic concentration, refractive index, entropy, stress, etc. That is, almost all models for physical quantities are formulated with a differential equation.

The following example illustrates how some first-order equations arise:

Iteration: First-Order Sequences

notebook (non-evaluated) pdf (evaluated) html (evaluated)

Sequences are developed in which the next iteration only depends on the current value; in this most simple case simulate exponential growth and decay.

1: *ExampleFunction* taking two arguments is defined: the first argument represents the iteration and the second represents a single parameter expressing how the current iteration grows. The value at the $i+1^{\text{th}}$ iteration is the sum of the value of the i^{th} plus α times value of the i^{th}iteration. If this is a bank account and interest is compounded yearly, then the i^{th}iteration is the value of an account after i years at a compounded annual interest rate of α. This function has improved performance (but consumes more memory) by storing its intermediate values. Of course, the function would iterate for ever if an initial value is not specified...

4: Because the initial value and the 'growth factor' α deterimine all subsequent iterations, it is sensible to 'overload' *ExampleFunction* to take an extra argument for the intial value: here, if *ExampleFunction* is called with three arguments *and the first argument is zero*, then the initial value is set; otherwise it is a recursive definition with intermediate value storage.

5: *Trajectory* is an example of a function that builds a list by first-order iteration; its resulting list structure is plotted with `ListPlot`.

8: To visualize the behavior as a function of its initial value, several plots can be superposed with `MultipleListPlot` from the `MultipleListPlot` package. If $\alpha > 0$, the function goes to $\pm\infty$ depending on the sign of the initial value. For a fixed α every point in the plane belongs to one and only one trajectory associated with an initial value and that α.

9: If $\alpha < 0$, the function asymptotically goes to zero, independent of the initial value. In this case as well, the plane is completely covered by non-intersecting trajectories.

Suppose a function, F[i], changes proportional to its current size, i.e., F[i+1] = F[i] + αF[i]

```
1  ExampleFunction[i_, alpha_] := ExampleFunction[i, alpha] =
     ExampleFunction[i - 1, alpha] +
       alpha * ExampleFunction[i - 1, alpha]
```

The function needs some value at some time (an initial condition) from which it obtains all its other values:

```
2  ExampleFunction[0, 0.25] = π / 4
```

```
3  ExampleFunction[18, 0.25]
```

ExampleFunction[0,0.25] = π/4 above serves as an initial value for the function. The initial value and α determine the value at any later time. The initial value can be expressed as another parameter for the function:

```
4  ExampleFunction[0, alpha_, InitialValue_] := InitialValue
   ExampleFunction[Increment_Integer, alpha_,
     InitialValue_] := ExampleFunction[i, alpha, InitialValue] =
       ExampleFunction[i - 1, alpha, InitialValue] +
         alpha * ExampleFunction[i - 1, alpha, InitialValue]
```

```
5  Trajectory[alpha_, Steps_, InitialValue_] := Table[
     ExampleFunction[i, alpha, InitialValue], {i, 0, Steps - 1}]
```

```
6  ListPlot[Trajectory[0.01, 300, 0.0001], PlotJoined → True]
```

```
7  << Graphics`MultipleListPlot`
```

Plotting curves for a range of intial values, but fixed α >0

```
8  MultipleListPlot[Trajectory[.01, 300, -.5],
     Trajectory[.01, 300, .5], Trajectory[.01, 300, 1],
     Trajectory[.01, 300, 1.5], PlotJoined → True]
```

A similar plot for negative α value. In either case each point in space correstponds a particular initial value for a fixed α

```
9  MultipleListPlot[Trajectory[-.01, 300, -.5],
     Trajectory[-.01, 300, .5], Trajectory[-.01, 300, 1],
     Trajectory[-.01, 300, 1.5], PlotJoined → True]
```

The previous example is generalized to a discrete change Δt of a continous (i.e., time-like) parameter t. The following example demonstrates *first-order Euler finite differencing* or *Euler integration*. It is an integration approximation because the method uses a finite time step Δt to approximate $f(t) = \int_{t_0} tA(t)dt$ for a known first-order differential equation $df/dt = A(t)$ where $f(t_0)$ is an *initial condition*. In this example, the iteration sequence approximates

$$(f(t=0), f(\Delta t), f(2\Delta t), \ldots) \approx$$

$$\left(f(t=0), f_{\text{aprx}}(\Delta t) = f(t=0) + \left.\frac{df}{dt}\right|_{t=0} \Delta t, f_{\text{aprx}}(2\Delta t) = f_{\text{aprx}}(2\Delta t) + \left.\frac{df}{dt}\right|_{t=\Delta t} \Delta t, \ldots \right)$$

$$= (f(t=0), A(t=0)\Delta t, A(t=\Delta t)\Delta t, \ldots) \tag{19-4}$$

3.016 Home

Full Screen

Close

Quit

First-Order Finite Differences

notebook (non-evaluated) pdf (evaluated) html (evaluated)

Two functions that 'grow' lists by using simple forward Euler finite differencing are constructed.

1: The function *ForwardDifferenceV1* is defined with four arguments: argument 1 is a placeholder for *another function* that determines how each increment changes (i.e., the function df/ft); argument 2 is the initial value; argument 3 is the distrete forward difference (i.e., Δt); argument 4 indicates the size of the list that will be returned. The function uses `Module` to hide an internal variable representing the current value of the list, and `AppendTo` to incrementally grow the list. In a following example, `NestList` and `NestWhileList` will be used to generate more efficient functions than the ones generate by `AppendTo` here.

2: *exampleFunction* is defined to to pass to sequence-generating functions—it plays the role of df/dt in Eq. 19-4.

5: `ListPlot` will produce a plot of the exemplary result which is a list of length 500.

7: *ForwardDifferenceV1* is unsatisfactory because the x-axis of the plot is the iteration and not the time-like variable that is more physical; so it is generalized with *ForwardDifferenceV2* which also takes arguments for the initial value and the final value of the continuous parameter. This function returns a list containing lists $(x_i, f(x_i))$ also suitable an argument to `ListPLot`.

Create a function to return a list of values by forward differencing with a function (these examples were modified from those found in ``Computer Science with *Mathematica*'' by Roman E. Maeder, Cambridge University Press, (2000).)

```
1  ForwardDifferenceListV1[AFunction_,
     InitialValue_, delta_, ListLength_Integer] :=
   Module[{TheResultList = {InitialValue},
     TheValue = InitialValue},
    Do[TheValue = TheValue + delta AFunction[TheValue];
     AppendTo[TheResultList, TheValue],
     {ListLength}]; TheResultList]
```

```
2  exampleFunction[x_] := 0.1 x
```

```
3  result =
     ForwardDifferenceListV1[exampleFunction, 1, 0.01, 500];
```

```
4  result // Short
```

```
5  ListPlot[result, PlotJoined → True]
```

Write another version of this forward difference function that returns a list of values
and the "x" value for subsequent use in ListPlot, this one will take x_0 and $y(x_0)$ as an argument in a list

```
6  Clear[ForwardDifferenceListV2]
```

```
7  ForwardDifferenceListV2[
     AFunction_, x0_, fx0_, delta_, Xlast_] :=
   Module[{TheResultList = {{x0, fx0}},
     TheValue = fx0, CurrentX = x0},
    While[CurrentX < Xlast,
     CurrentX = CurrentX + delta;
     TheValue = TheValue + delta AFunction[TheValue];
     AppendTo[TheResultList, {CurrentX, TheValue}]];
    TheResultList]
```

```
8  result =
     ForwardDifferenceListV2[exampleFunction, 0, 1, 0.01, 4];
```

```
9  result // Short
```

```
10 ListPlot[result]
```

3.016 Home
Full Screen
Close
Quit

Nested Operations

notebook (non-evaluated) pdf (evaluated) html (evaluated)

The concept of nesting operations to produce finite differences is demonstrated. The MATHEMATICA® notion of a *pure function* is utilized with an example.

1: Construct a function, *StepOnce* , that operates on its first argument (the pair $\{x_i, y_i(x_i)\}$) with an input function dy/dx and a discrete increment δ. The function returns a list (the pair $\{x_{i+1}, y(x_{i+1})\}$) representing the finite difference approximation at "time" $x_i + \delta$.

2: Here, a specific case is developed by defining a function that explicitely defines both the input function (here *exampleFunction*) and the fixed increment (here $\Delta t = 0.01$). The result is a function that takes a single x-y pair.

3: Instead of the forward difference techniques that used `AppendTo` to grow a list of x-y pairs—here, `NestList` is used to build a list (`result`) by repeatedly (400 times) applying a function to the result at the previous iteration.

5: Here, `NestList` is called with a *pure function* which is indicated by the & that appears the `StepOnce[#,exampleFunction,0.01]&` definition; the # is a placeholder for the functions argument. `NestList` calls this pure function repeatedly starting with the first argument (here $\{0, 1\}$) and stores intermediate values in a list.

6: In the next few steps, the goal is to generate such trajectories for a variety of initial conditions. This is achieved by creating and saving plots as *Graphics Objects*. In this step, rules (*DisplayLater* and *DisplayNow*) are defined that can be passed to plotting functions that control whether the created Graphics Object is displayed or not.

7: A function is defined that creates a Graphics object for a trajectory as a function of its sole argument representing the initial value. dd

8: A list of trajectories for initial values $(-4, -3.5, \ldots, 3.5, 4.0)$ are plotted together.

Because each iteration is the same, the iteration can be considered a functional operation, for the case considered above, $\{x_{i+1}, y_{i+1}\} = \{x_i + \delta, y_i + \delta\, f(y_i)\}$. Therefore a "Incrementing Operator" can be obtained that updates the values:

```
1 StepOnce[{x_, y_}, AFunction_, delta_] :=
    {x + delta, y + delta AFunction[y]}
```

Then, the trajectory should be obtained from:
{{xo,yo},StepOnce[{xo,yo}], StepOnce[StepOnce[{xo,yo}]], ...}
This is what the built-in *Mathematica* function
NestList[function, initialvalue,depth] does:

```
2 OurStepOnce[{x_, y_}] :=
    StepOnce[{x, y}, exampleFunction, 0.01]
3 result = NestList[OurStepOnce, {0, 1}, 400];
4 ListPlot[result]
```

Using a *Mathematica* trick of a ``pure function'' one can eliminate the intervening function (OurStepOnce) definition:

```
5 ListPlot[
    NestList[StepOnce[#, exampleFunction, 0.01] &, {0, 1}, 400]]
6 DisplayLater = DisplayFunction → Identity;
  DisplayNow = DisplayFunction → $DisplayFunction;
7 lp[i_] :=
    ListPlot[NestList[StepOnce[#, exampleFunction, 0.01] &,
      {0, i}, 400], DisplayLater];
```

This will plot a family of related curves, each for a different starting value of the iterated function:

```
8 Show[Table[lp[i], {i, −4, 4, .5}], DisplayNow]
```

To summarize what was done up to now, we've seen how a given function can be changed incrementally by stepping forward the independent variables and calculating a corresponding change in the function's value. By doing so, we trace out trajectories in space, the paths of which depend on the starting values of the independent variables.

Geometrical Interpretation of Solutions

The relationship between a function and its derivatives for a first-order ODE,

$$F(\frac{dy(x)}{dx}, y(x), x) = 0 \tag{19-5}$$

can be interpreted as a level set formulation for a two-dimensional surface embedded in a three-dimensional space with coordinates (y', y, x). The surface specifies a relationship that must be satisfied between the three coordinates.

If $y'(x)$ can be solved for exactly,

$$\frac{dy(x)}{dx} = f(x, y) \tag{19-6}$$

then $y'(x)$ can be thought of as a height above the x-y plane.

For a very simple example, consider Newton's law of cooling which relates the change in temperature, dT/dt, of a body to the temperature of its environment and a *kinetic coefficient* k:

$$\frac{dT(t)}{dt} = -k(T - T_o) \tag{19-7}$$

It is very useful to "non-dimensionalize" variables by scaling via the physical parameters. In this way, a single ODE represents *all* physical situations and provides a way to describe universal behavior in terms of the single ODE. For Newton's law of cooling, this can be done by defining non-dimensional temperatures and time with $\Theta = T/T_o$ and $\tau = kt$, then if T_o and k are constants:

$$\frac{d\Theta(\tau)}{d\tau} = (1 - \Theta)$$

The Geometry of First-Order ODES

notebook (non-evaluated) pdf (evaluated) html (evaluated)

The surface representation provides a useful way to think about differential equations—much can be inferred about a solution's behavior without computing the solution exactly. This is shown for a simple case of Newton's law of cooling Equation 19 and an artificial case.

1: For first-order ODEs, behavior is dominated by whether the derivative term is positive or negative. Here, a 3D graphics object is created for a gray-colored horizontal plane at $z = 0$. This is achieved by combining (in a list) the `SurfaceColor` directive in a `Graphics3D` object, and then using `Plot3D` to create the plane with delayed display.

2: This will create the surface associated with Newton's law of cooling with the zero plane. This case is very simple. The sign of the change of Θ depends only the sign of $1 - \Theta$ and therefore $d\Theta/dt = 0$ is the parametric curve (a line in this case) $(d\Theta/dt = 0, \Theta = 1, \tau)$. That is, if $\Theta = 1$ at any time τ it will stay there at all subsequent times (also, at all previous times as well). Because $\Theta(\tau)$ will always increase when $\Theta < 1$ and will always decrease when $\Theta > 1$, the solutions will asymptotically approach $\Theta = 1$.

4: The asymptotic behavior can be further visualized by plotting a first-order difference representation of how the solution is changing in time, i.e, $(d\tau, d\Theta) = d\tau\left(1, \frac{d\Theta}{d\tau}\right)$ This can be obtained with `PlotVectorField` from the `PlotField` package. Here the magnitude of the arrows is scaled by setting $d\tau = 1$.

5: A more complex case $\frac{dy}{dt} = y \sin\left(\frac{yt}{1+y+t}\right)$ can be visualized as well and the behavior can be inferred whether the derivative lies above or below the zero-plane (i.e., the sign of the derivative).

6: `PlotVectorField` provides another method to follow a solution trajectories.

Newton's law of cooling $\frac{dT}{dt} = -k(T - T_o)$ can be written in the non-dimensional form $\frac{d\Theta}{d\tau} = 1\text{-}\Theta$

In the general case, $\frac{d\Theta}{d\tau}$ will depend on both Θ and t, i.e., $\frac{d\Theta}{d\tau} = \frac{d\Theta}{d\tau}(\Theta,t)$. This the equation of a surface in three dimensions, as shown in the following plot (in this specific case there is no t dependence):

```
1 ZeroPlane[xmin_, xmax_, ymin_, ymax_] :=
    {Graphics3D[SurfaceColor[GrayLevel[0.6]]],
     Plot3D[0, {tau, xmin, xmax}, {Θ, ymin, ymax},
      PlotPoints -> 4, DisplayFunction -> Identity]};
```

```
2 Show[Plot3D[1 - Θ, {tau, -1, 1}, {Θ, -2, 3},
    AxesLabel → {"τ", "Θ", "dΘ/dτ"}, DisplayFunction -> Identity],
   ZeroPlane[-1, 1, -2, 3], DisplayFunction ->
    $DisplayFunction, ViewPoint -> {17.830, 10.191, 4.064}]
```

```
3 << Graphics`PlotField`
```

```
4 PlotVectorField[{1, 1 - Θ}, {tau, 0, 4},
   {Θ, -2, 4}, Axes → True, AxesLabel → {"τ", "Θ"}]
```

Slightly more complicated example: $\frac{dy}{dt} = y \sin(\frac{yt}{1 + t + y})$, (dt,dy) = dt(1,ysin $\frac{yt}{1 + t + y}$))

```
5 Show[Plot3D[y Sin[ y t / (t + y + 1)], {t, 0, 10}, {y, 0, 10},
    AxesLabel → {"t", "y", "dy/dt"}, PlotPoints → 40,
    DisplayFunction -> Identity], ZeroPlane[0, 10, 0, 10],
   DisplayFunction -> $DisplayFunction,
   ViewPoint -> {14.795, 5.556, 13.731}]
```

```
6 PlotVectorField[{1, y Sin[ y t / (t + y + 1)]}, {t, 0, 10},
   {y, 0, 10}, Axes → True, AxesLabel → {"t", "y"}]
```

Separable Equations

If a first-order ordinary differential equation $F(y', y, x) = 0$ can be rearranged so that only one variable, for instance y, appears on the left-hand-side multiplying its derivative and the other, x, appears only on the right-hand-side, then the equation is said to be 'separated."

$$g(y)\frac{dy}{dx} = f(x) \qquad (19\text{-}8)$$
$$g(y)dy = f(x)dx$$

Each side of such an equation can be integrated with respect to the variable that appears on that side:

$$\int_{y(x_o)}^{y} g(\eta)d\eta = \int_{x_o}^{x} f(\xi)d\xi \qquad (19\text{-}9)$$

if the initial value, $y(x_o)$ is known. If not, the equation can be solved with an integration constant C_0,

$$\int g(y)dy = \int f(x)dx + C_0 \qquad (19\text{-}10)$$

where C_0 is determined from initial conditions. or

$$\int_{y_{\text{init}}}^{y} g(\eta)d\eta = \int_{x_{\text{init}}}^{x} f(\zeta)d\zeta \qquad (19\text{-}11)$$

where the initial conditions appear explicitely.

Using MATHEMATICA® 's Built-in Ordinary Differential Equation Solver

notebook (non-evaluated) pdf (evaluated) html (evaluated)

MATHEMATICA® has built-in exact and numerical differential equations solvers. DSolve takes a representation of a differential equation with initial and boundary conditions and returns a solution if it can find one. If insufficient initial or boundary conditions are specified, then "integration constants" are added to the solution.

1: `DSolve` operates like `Solve` . It takes a list of equations containing symbolic derivatives, the function to be solved for, and the dependent variable. In this case, the general solution of $\frac{dy(x)}{dx} = -xy(x)$ is returned as a list of rules. The solutions are be obtained by applying the rules (i.e., `y[x]/.dsol`).

2: The solution will depend on an integration constant(s) in general. If additional If more constraints (i.e., equations) are provided, then (provided a solution exists) the integration constant is determined as well.

3: The solution is plotted by turning the "solution rule" into a single list with `Flatten`. The plot is stored as a graphics object *exactplot* .

4: To see how finite differencing compares to the exact solution, the method from an earlier example is used. Here, a the forward differencing function is defined for the solution that was just obtained.

9: The previously defined forward differencing method is compared to the exact solution.

10: Here, the method is generalized to take an argument for the size Δt. `NestWhileList` is used with a pure function (where one arguments is fixed by passing through *res*). A pure function is also defined for the test of when to stop building the list—in this case, it stops when the first element in the list (accumulated time) exceeds 10.

11: An animation which will visualize the effect of time-step on accuracy of the Euler method is created. `Show` repeatedly called on *exactplot* (the exact solution) and a graphics object created from calling `ListPlot` on *res* with different time-steps.

```
1  dsol = DSolve[y'[x] + x * y[x] == 0, y[x], x]
```

Note that the solution is given as a rule, just like for the function **Solve**. Because no initial condition was specified, the solution involves an unknown constant, C[1].

```
2  dsol = DSolve[{y'[x] + Sin[x] * y[x] == 0, y[0] == 1}, y[x], x]
```

In this case an initial condition was specified for the differential equation, so there is no undetermined constant in the solution. The next statement extracts y(x) for plotting...

```
3  exactplot = Plot[y[x] /. Flatten[dsol], {x, 0, 10}]
4  ExampleFun[x_, y_] := -Sin[x] y
5  StepOnce[{x_, y_}, AFunctionXY_, delta_] :=
     {x + delta, y + delta AFunctionXY[x, y]}
6  OurStepOnce[{x_, y_}] :=
     StepOnce[{x, y}, ExampleFun, 0.01]
7  result = NestList[OurStepOnce, {0, 1}, 1000];
8  forwarddifferenceplot = ListPlot[result, PlotStyle → {Hue[1]}]
```

Now we superpose the exact solution with that obtained by the forward-differencing approximation.

```
9  Show[forwarddifferenceplot, exactplot]
```

Generalize to see how the step-size on the forward differencing scheme affects result

```
10 res[delta_] :=
     NestWhileList[(StepOnce[#, ExampleFun, delta]) &,
       {0, 1}, (#[[1]] < 10) & ]
11 Table[
     Show[exactplot, ListPlot[res[del],
       PlotStyle -> {Hue[0.75 del], Thickness[0.01]},
       PlotJoined -> True, DisplayFunction -> Identity],
     PlotRange -> {{0, 10}, {0, 1}}], {del, 0.01, 1, 0.02}]
```

While the accuracy of the first-order differencing scheme can be determined by comparison to an exact solution, the question remains of how to establish accuracy and convergence with the step-size δ for an arbitrary ODE. This is a question of primary importance and studied by Numerical Analysis.

3.016 Home

Full Screen

Close

Quit

Index