Lecture 19: Ordinary Differential Equations: Introduction

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,

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

or

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

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:

Iterative Application of Function

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Iteration: First-Order Sequences from a Fixed Boundary Condition

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

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

```
ExplFun[i_, α_] :=
 ExplFun[i, α] =
  ExplFun[i - 1, α] +
   α * ExplFun[i - 1, α]
```
1

In the above, the symbol is assigned (ExplFun[i,alpha] = ...) as part of the function definition, so that intermediate values are ``remembered.''
The function needs some value at some time (an initial condition) from which it obtains all its other values:

```
ExplFun[0, 0.25] = π / 4
```
2

```
ExplFun[18, 0.25]
```
3

1: *ExpleFun* 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.

2: Of course, the function would iterate for ever if an initial value is not specified; and so it is specified here.

3: For, example this would produce the 18^{th} iteration of growth with a compounding rate of 25% with $\pi/4$ at the initial state.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Iteration: First-Order Sequences with a Generalized Boundary Condition

The previous example is generalized so that the iteration function is generalized for an arbitrary initial values.

```
ExplFun[0 , α_,
  InitVal_] := InitVal
ExplFun[Inc_Integer,
  α_, InitVal_] :=
 ExplFun[Inc, α, InitVal] =
  ExplFun[Inc - 1, α,
    InitVal] + α * ExplFun[
     Inc - 1, α, InitVal]
```

1

```
Traj[
  Steps_Integer?Positive,
  α_, InitVal_] :=
 Traj[Steps, α, InitVal] =
  AppendTo[Traj[Steps - 1,
    α, InitVal], ExplFun[
    Steps, α, InitVal]]
Traj[0, _, _] = {};
```

2

```
Traj[12, .01, .001]
```

3

We define a function, Evolve, producing an interactive tool with input : Initial values and α.producing an interactive tool with input : Initial values and α.

A

```
Evolve[300]
```

5

1: Because the initial value and the 'growth factor' α determine all subsequent iterations, it is sensible to 'overload' *ExplFun* (i.e., define the function to behave differently depending on the number and type of its arguments) to take an extra argument for the initial value. Here, if *ExplFun* 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.

2: *Traj* is an example of a function that builds a list by first-order iteration. It produces a result that is suitable for input to `ListPlot`. The second part of the definition defines the 0^{th} item of the list to always be an empty list, no matter what other values are passed to it. *Traj* does its work by calculating new pairs with the help of *ExplFun* and then recursively appends the current value to the growing list.

3: Here is an example which will produce a list of twelve entries, starting from the first iteration of 0.001 with growth factor of 1%.

8: To visualize the behavior as a function of its initial value, an interactive function, *Evolve* , is defined (definition suppressed in notes, but available via the links). It takes an argument for the maximum number of iterations, and the initial value and growth factor are controlled with `Manipulate`.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Space-Covering Sequences: Families of Trajectories

The previous example is generalized so that the iteration function is generalized for an arbitrary initial values. several plots. Once the growth rate is fixed, we visualize how each curve "belongs" to a particular initial value; the set of all initial values generates a family of curves that fill the plane—each point belongs to one and only one trajectory.

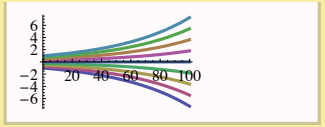
Plotting a bunch of curves for the same positive α value, but each corresponding to a different initial value.

```
PlotTrajs[α_] := Block[
  {$RecursionLimit = 10^4},
  ListPlot[Evaluate[
    Table[Traj[300, α, iv],
     {iv, -1, 1, 0.25}]],
   PlotRange → All,
   Joined → True,
   PlotStyle → Thick]]
```

1

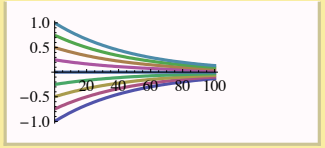
```
PlotTrajs[0.02]
```

2




```
PlotTrajs[-0.02]
```

3



1: *PlotTrajs* is a function that provides a visualization of trajectories for an input growth. It works by generating a set of initial values to pass to *Traj* and then plots them with `ListPlot`.

2: 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.

3: 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.

3.016 Home
Full Screen
Close
Quit

Forward Differencing Methods: Explicit Methods

The previous example is generalized to a discrete change Δt of a continuous (i.e., time-like) parameter t. The following example demonstrates the simplest method of numerically solving a simple first-order ODE. *first-order explicit finite differencing* or *Euler integration.*

We begin by approximating the derivative dy/dt at time t with a finite difference approximation:

$$\Delta y/\Delta t = [y(t + \Delta t) - y(t)]/[(t + \Delta t) - t] \tag{19-4}$$

We can write down a formula for $y(t + \Delta t)$ in terms of current values at t, and thus 'project y into the future. Suppose we use fixed small time steps Δt and the short-hand $y_n = y(n\Delta t)$, $y_{n+1} = y(n\Delta t + \Delta t)$. Now, we must determine which value to use for $f(y(t))$ in $dy/dt = f(y)$: the current value $f(y_n)$, the future value $f(y_{n+1})$, an average value $([f(y_n) + f(y_{n+1})]/2$, or something else. The simplest thing to do is use the current value and then every term (but y_{n+1} is in terms of n:

$$y_{n+1} = y_n + \Delta t f(y_n) \tag{19-5}$$

This is called explicit forward-differencing or Euler's method,

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

First-Order Finite Differences: Method 1 Explicit Finite Differences

We implement this simple method described in Eq. 19-5 be creating a function which 'projects' the current value of y into the future.

Approximate f(y) with f(y$_{i-1}$);

```
PushMethod1[f_,
  {ti_, yi_}, Δt_] :=
 {ti + Δt, yi + Δt f[yi]}
```
1

```
FuncEx[y_] := -Sin[y]
```
2

```
PushMethod1[
 FuncEx, {0, 1}, .01]
```
3

```
PushMethod1[FuncEx,
 PushMethod1[FuncEx,
  {0, 1}, .01], .01]
```
4

```
Nest[PushMethod1[FuncEx,
   #, .01] &, {0, 1}, 2]
```
5

```
NestList[
 PushMethod1[FuncEx,
   #, .01] &, {0, 1}, 2]
```
6

```
NestWhileList[
 PushMethod1[FuncEx,
   #, .01] &, {0, 1},
 (First[#] < 0.03) &]
```
7

1: The function *PushMethod1* takes three arguments: argument 1 is a place-holder for *another function* that determines how each increment changes (i.e., the function $f = dy/ft$); argument 2 is the current value; argument 3 is the discrete forward difference (i.e., Δt).

2: *FuncEx* is defined to to pass to sequence-generating functions—it plays the role of $f(y_n)$ in Eq. 19-5.

3: For example, this pushes a value `{0,1}` by $\Delta t = 0.01$ into the future with `FuncEx[1]`.

4: Calling the function, *PushMethod1* , recursively on itself (once) pushes the value iteravely into the future (twice).

5: We can generalize this recursion method by using `Nest` (`Nest[f,x,3]` *rightarrow* `f[f[f[x]]]`). However, we must turn *PushMethod1* into a function of a single argument, so there is no ambiguity about which value is being iteratively pushed forward. This is done by creating a *Pure Function* version of *PushMethod1* . The pure function is indicated by the trailing ampersand, `&`, and the `#` becomes a place holder for the single argument. Thus, `Nest[(PushMethod[FuncEx,#,0.01])&, {0,1}, 2]` nests *PushMethod1* with *fixed* first and third arguments (`FuncEx` and `0.01`) on the initial value `{0,1}` twice.

6: `NestList` is another version of `Nest`, but it stores each increment in a growing list and returns a list structure.

7: `NestListWhile` is another version of `NestList`, but with a switch to tell it when to stop 'Nesting.' We use this method to indicate "at what time" the nesting should stop, and not "after how many nests." For `NestListWhile`' test-argument, we use another pure function: it takes the current value of `{t,y}` and tests to see if `t` is less than `0.03`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

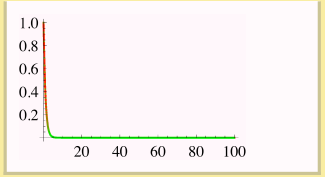
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing Trajectories from Explicit Forward Differences

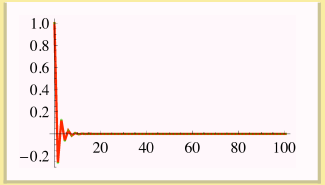
Examples of the explicit forward differencing function *PushMethod1* called recursively with `NestListWhile` are illustrated. An example of *Numerical Instability* appears.

A Function PlotM1, taking arguments for α and initial condition is used with NestListWhile and ListPlot to produce graphics with a red line and green points. **A**

`PlotM1[0.1, 1]` **2**



`PlotM1[1.5, 1]` **3**



A: *PlotM1* is defined which takes a first argument for a time-step, and a second argument is y_0. It uses `ListPlot` to create a trajectory, and show line segments between the computed points. (The definition is suppressed in class-notes, it is available via the links given above)

2: Here is an example of a stable numerical integration of a first-order ODE. We have not evaluated how *accurate* the numerical algorithm is, but only that it is well-behaved.

3: Using a larger time-step, we can see that the algorithm is becoming less well-behaved. This introduces the concept *maximum stable time-step*.

3.016 Home
Full Screen
Close
Quit

Forward Differencing Methods: Implicit Methods

As in the implicit method, we begin by approximating the derivative dy/dt at time t with a finite difference approximation:

$$\Delta y/\Delta t = [y(t+\Delta t) - y(t)]/[(t+\Delta t) - t] \tag{19-6}$$

However, in this case we will use the expected future value, y_{n+1} as the argument to $f(y)$.

$$\begin{aligned} y_{n+1} &= y_n + \Delta t f(y_{n+1}) \\ &= y_n + \Delta t \left[f(y_n) + \left.\frac{df}{dy}\right|_{y_n} (y_{n+1} - y_n) \right] \end{aligned} \tag{19-7}$$

Because y_{n+1} appears on both sides, we have to solve for it (this is the implicit step),

$$y_{n+1} = \frac{y_n + \Delta t (f(y_n) - \left.\frac{df}{dy}\right|_{y_n} y_n)}{1 - \Delta t \left.\frac{df}{dy}\right|_{y_n}} \tag{19-8}$$

This is called implicit forward-differencing.

First-Order Finite Differences: Method 1 Explicit Finite Differences

We implement this implicit method described in Eq. 19-8

Approximate f(y) with f(y_i); then solving the finite difference equation above,
$y_i = y_{i-1} + \Delta t\ [f(y_i)\ ]$.
So, $y_i = y_{i-1} + \Delta t\ (f(y_i) + f'(y_{i-1})dy)$
$y_i = y_{i-1} + \Delta t\ (f(y_i) + f'(y_{i-1})(y_i - y_{i-1})\ )$
$y_i = (y_{i-1} - \Delta t\ [f(y_{i-1}) - f'(y_{i-1})y_{i-1}])/$
$(1 - \Delta t\ f'(y_{i-1})\ )$

```
PushMethod2[f_,
  df_, {ti_, yi_},
  Δt_] := {ti + Δt,
  (yi + (Δt (f[yi] -
         df[yi] yi))) /
   (1 - Δt df[yi])}
```

1

```
dFuncEx[y_] :=
 Evaluate[D[FuncEx[y], y]]
```

2

```
NestList[
 PushMethod2[FuncEx,
   dFuncEx, #, 0.1] &,
 {0, 1}, 3]
```

3

We define a function, PlotM2, which takes arguments Δt and InitialCondition and then uses ListPlot with Blue lines and Gray points.

A

1: *PushMethod2* implements the implicit differencing strategy. However, we must also provide this method with a function representing the derivative of f.

2: We define the derivative function, but use `Evaluate` on the right-hand side of the delayed assignment (`:=`) so that the derivative operator `D` is not called each time the function is used.

3: We can use the `Nest`-family of functions as before.

A: A function to plot the implicit function results is defined for comparison to the explicit method.

3.016 Home
Full Screen
Close
Quit

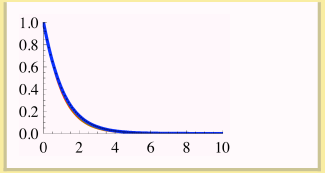
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Comparison of Implicit and Explicit Methods

We plot the results from the two different time-stepping methods and show that the implicit method is more stable. We still have not evaluated the accuracy of either method.

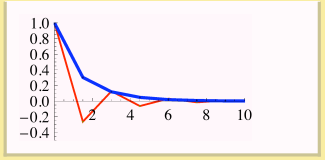
```
Show[PlotM1[0.1, 1],
 PlotM2[0.1, 1], PlotRange →
  {{0, 10}, {0, 1}}]
```

1




```
Show[PlotM1[1.5, 1.0],
 PlotM2[1.5, 1.0],
 PlotRange →
  {{0, 10}, {-0.5, 1}}]
```

2



Method 2 will fail if the step size is increased to 2

1: With a time step of $\Delta t = 0.1$, the two methods give results that are barely discernible. This gives us confidence in the hypothesis that the solutions are also accurate at this time step.

2: At larger time steps, the implicit method is more well-behaved. However, if the step size is made a little larger, both methods will become unstable.

3.016 Home

Full Screen

Close

Quit

Geometrical Interpretation of Solutions

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

$$F(\frac{dy(x)}{dx}, y(x), x) = 0 \tag{19-9}$$

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,

$$\frac{dy(x)}{dx} = f(x, y) \tag{19-10}$$

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:

$$\frac{dT(t)}{dt} = -k(T - T_o) \tag{19-11}$$

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:

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

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visual Understanding of the Behavior of First-Order ODES

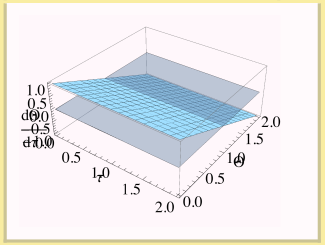
The surface representation provides a useful way to think about differential equations—much can be inferred about a solution's behavior without computing the solution exactly. This is shown for a simple case of Newton's law of cooling Equation 19 and an artificial case.

```
ZeroPlane[xmin_,
  xmax_, ymin_, ymax_] :=
Graphics3D[{Gray,
   Opacity[0.25], Cuboid[
    {xmin, ymin, -.001},
    {xmax, ymax, .001}]}]
```

1

```
Show[
 Plot3D[1 - Θ, {tau, 0, 2},
  {Θ, 0, 2}, AxesLabel →
   {"τ", "Θ", "dΘ/dτ"},
  DisplayFunction ->
   Identity],
 ZeroPlane[0, 2, 0, 2]]
```

2



1: For first-order ODEs, behavior is dominated by whether the derivative term is positive or negative. Here, a `Graphics3D` object is created for a gray-colored opaque horizontal plane (in reality we use a very thin slab) at $z = 0$. We will use this function to evaluate when the derivative is positive and the value is increasing or negative and the value is decreasing.

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 $\Theta = 1$.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

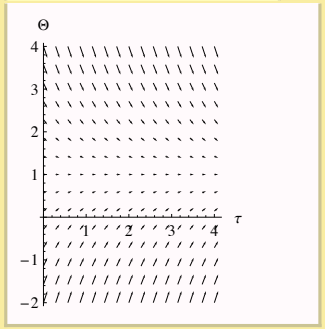
Visualizing the Geometry of Flows for First-Order ODES

By creating vector field which 'points' toward subsequent points as inferred from the ODE, we produce a very useful way to understand solution behavior for a variety of initial conditions, without computing a solution to the ODE. This is shown again for a simple case of Newton's law of cooling

Plot the vectorf-ield (dτ,dΘ) = dτ(1, $\frac{d\Theta}{d\tau}$) We can do so by plotting vectors of the form {dτ, dΘ} = dτ{1, $\frac{d\Theta}{d\tau}$} which will be proportional to the vector {1, 1-Θ}. This is done as follows:

```
Needs["VectorFieldPlots`"];
VectorFieldPlots`VectorFie\
  ldPlot[{1, 1 - Θ},
{tau, 0, 4},
{Θ, -2, 4}, Axes → True,
AxesLabel → {"τ", "Θ"},
ImageSize → Full]
```

1



1: 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 `VectorFieldPlot` from the `VectorFieldPlots` package. Here the magnitude of the arrows is scaled by setting $d\tau = 1$.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Geometry of Flows for First-Order ODES

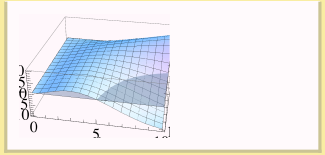
We utilize our visualization methods for intuitive understanding of the behavior of ODES for the case:

$$\frac{dy}{dt} = y \sin\left(\frac{yt}{1+y+t}\right)$$

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

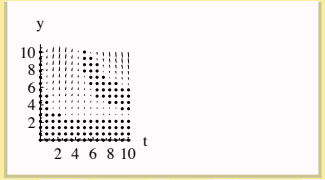
```
Show[Plot3D[
  y Sin[y t / (t + y + 1)],
  {t, 0, 10}, {y, 0, 10}, ,
  ZeroPlane[0, 10, 0, 10]]]
```

1



```
VectorFieldPlot[
 {1, y Sin[y t / (t + y + 1)]},
 {t, 0, 10}, {y, 0, 10}]
```

2



1: This case 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). Where $dy/dt < 0$, y decreases as time marches forward; thus it moves toward the intersection of the zero plane and the dy/dt-surface. We see that the slope of the surface evaluated along the curve of intersection determines whether there is an "attractor-manifold" in the ODE.

2: `VectorFieldPlot` provides another method to follow a solution trajectories: we plot vectors proportional to $dt(1, y\sin[yt/(1+y+t)])$.

3.016 Home
Full Screen
Close
Quit

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."

$$\begin{aligned} g(y)\frac{dy}{dx} &= f(x) \\ g(y)dy &= f(x)dx \end{aligned} \tag{19-12}$$

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

$$\int_{y(x_o)}^{y} g(\eta)d\eta = \int_{x_o}^{x} f(\xi)d\xi \tag{19-13}$$

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

$$\int g(y)dy = \int f(x)dx + C_0 \tag{19-14}$$

where C_0 is determined from initial conditions. or

$$\int_{y_{\text{init}}}^{y} g(\eta)d\eta = \int_{x_{\text{init}}}^{x} f(\zeta)d\zeta \tag{19-15}$$

where the initial conditions appear explicitly.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

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

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.

```
dsol = DSolve[
  {y'[t] == FuncEx[y[t]]},
  y[t], t]
```
1

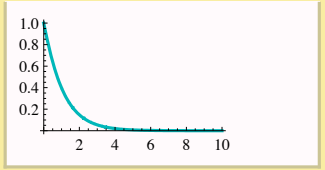
$$\{\{y[t] \to 2\,\text{ArcTan}\left[e^{-t+C[1]}\right]\}\}$$

```
dsol = DSolve[
  {y'[t] == FuncEx[y[t]],
   y[0] == 1}, y[t], t]
```
2

$$\{\{y[t] \to 2\,\text{ArcTan}\left[e^{-t}\,\text{Tan}\left[\frac{1}{2}\right]\right]\}\}$$

The next statement extracts y (x) for plotting ..

```
ExactPlot =
 Plot[ y[t] /. dsol,
  {t, 0, 10}, PlotStyle →
   {Thick, Darker[Cyan]},
  PlotRange → All]
```
3



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 the example we used for finite differencing examples: $\frac{dy(x)}{dx} = FuncEx\,[y]$ `DSolve` returns a list of rules. The solutions are be obtained by applying the rules (i.e., `y[x]/.dsol`). The solution will depend on an integration constant(s) in general. MATHEMATICA® uses the symbols `C[1],C[2],etc` as place-holders for the integration constants.

2: If additional If more constraints (i.e., equations) are provided, then (provided a solution exists) the integration constant is determined as well. This is the exact solution to what we were numerically approximating above.

3: The solution is plotted by turning the "solution rule" into a plot-table `y[t] Flatten`. The plot is stored as a graphics object `ExactPlot`.

3.016 Home
Full Screen
Close
Quit

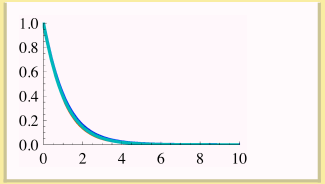
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Comparision of Exact Solutions to Finite Difference Methods

We compare the plots of implicit, explicit finite differencing to the exact solution obtained by `DSolve`.

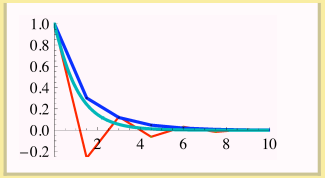
```
Show[PlotM1[0.1, 1],
 PlotM2[0.1, 1],
 ExactPlot, PlotRange →
  {{0, 10}, {0, 1}}]
```

1




```
Show[PlotM1[1.5, 1],
 PlotM2[1.5, 1],
 ExactPlot, PlotRange →
  {{0, 10}, {-0.25, 1}}]
```

2



1: To see how finite differencing compares to the exact solution, we plot all three trajectories together. The less-stable explicit method is more accurate for intermediate values of t.

2: This shows the comparison at larger time steps.

3.016 Home

Full Screen

Close

Quit

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

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Using MATHEMATICA® 's Differential Equation Solver on a First-Order ODE: Less Trivial Example

We solve $y'(x) + xy(x) = 0$ for a 'strange' condition $y'(5) = 1$ and plot the solution.

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

1

Boundary conditions other than y[0]:

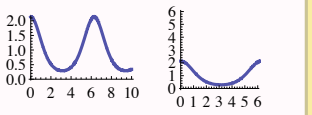
```
dsol = DSolve[
  {y'[x] + Sin[x] * y[x] ==
    0, y'[5] == 1}, y[x], x]
```

2

```
{{y[x] → -e^(-Cos[5]+Cos[x]) Csc[5]}}
```

```
GraphicsRow[
 {p = Plot[y[x] /. dsol,
    {x, 0, 10},
    PlotStyle → Thick],
  Show[p, PlotRange →
    {{0, 6}, {0, 6}},
   AspectRatio → 1]}]
```

3



1: This demonstrates the use of `DSolve`, because we have not supplied enough conditions to determine the solution exactly, MATHEMATICA® introduces all the undetermined constants of integration. In this case, there is only one undetermined constant.

2: Here, the solution is required to have a slope of unity at $x = 5$. If such a value is possible, then MATHEMATICA® will compute the corresponding value of `C[1]`.

3: This demonstrates how to extract the solution and plot it. It is plotted a second time with the same y and x scales so we can see that the slope is indeed one at $x = 5$.

3.016 Home

Full Screen

Close

Quit

Index