Week 6: Initial Value Problems

Amath 301

TA Session

Today

0. IVP solvers - a classification
1. IVP solvers availible to you
2. Exploiting solvers for clean code

IVP solvers

Many criteria exist that you can classify solvers/problems.

- Stiff or non-stiff? How important is the step size to accuracy?
  - Stiff: very very very important
  - non-Stiff: Eh?
  - A more qualitative measure. Hard to define for nonlinear systems
- Explicit or implicit solver?
  - Can the update step be expressed entirely in terms of function evaluations?
  - yes: Explicit
  - No: implicit (usually have to solve systems of equations to do an update step)

Lots of named methods - but really only a few ideas

Runge-Kutta methods

(For your learning, not for HW/Exams)

Given a system of ODEs $\frac{dy}{dt} = f(t, y)$

$$y_{n+1} = y_n + h \sum_{i=1}^{n} b_i k_i$$

where

$$k_i = f\left(t_n + c_i h, y_n + \sum_{j=1}^{n} a_{i,j} k_i\right)$$

Depends on 3 groups of parameters: $\{b_i\}$, $\{c_i\}$, $\{a_{i,j}\}$

Can describe hundreds of methods with this framework!

Butcher Tableaus

(Named after John Butcher - used this notation in a book)

Uniquely describes the scheme.

$$
\begin{array}{c|ccccc}
c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
c_3 & a_{3,1} & a_{3,2} & \cdots & a_{3,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_n & a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\
\hline
 & b_1 & b_2 & \cdots & b_n
\end{array}
$$

Some simplification: Autonomous (not time dependent) problems - don't care about c_i

For explicit methods: $C_1 = 0$, A strictly lower-triangular.

Semi-implicit: include diagonal. Fully implicit: full matrix (hard!)

It looks like a matrix!

One final detail:

Higher order schemes offer two benefits:

- Larger step sizes/better behavior on stiff problems
- Lower order approximations give continuous (very accurate) interpolations!

This last part is useful for HW's

Some names you'll see in documentation:

- RK45 - Do a 4th order and a 5th order step. Estimate error and performs a more efficient step
- RK23 - Same as RK45, but 2nd/3rd order steps. Faster to compute, less accurate
- DOP853 - Explicit RK 8th order. DOmard & Prince
- Dorpi5 - Implicit variant of RK45. Less popular
- Radau - Stiff solver. 5th order Runge Kutta method
- BDF - backwards differentiation - finite difference scheme derived, not an RK method. Implicit!
- LSODA - BDF variant - not an RK method, not popular. Implicit!

Matlab

Matlab has even more RK schemes! Very helpful naming rules

Nonstiff methods:

- `ode45` - RK45 scheme
- `ode23` - Crude RK23
- `ode113` - Matlab Magic variable order method
- `ode78` - RK8 variant - 7th order interpolant
- `ode89` - RK9 variant - 8th order interpolant

In practice: try `ode45` first, then `ode113`

Matlab

Fully implicit:

- `ode15i` - Variable order based on backwards differentiation scheme, up to 5th order

Stiff solvers: (not RK methods usually)

- `ode15s` - Variable order multistep
- `ode23s` - Single step solver, estimates jacobian matrix at each step
- `ode23t` - Fancy trapezoid rule based solver
- `ode23tb` - Trapezoid plus backwards differentiation

`ode15i` and `ode23tb` work great on stiff problems - solvers of last resort!
If all 10 methods fail - you've probably got a research problem

Scipy/Python

One high level function to rule them all!

```
from scipy.integrate import solve_ivp
```

Choose the solver by specifying `method=`

- `RK45` (default)
- `RK23`
- `DOP853`
- `Radau` Radau IIA method (stiff problems)
- `BDF` (stiff problems)
- `LSODA` (stiff problems)

Matlab implementation

```
f1 = @(t,x) [x(1); 100*x(2)];  % A column vector comes out
f2 = @(t,x) [x(1) + x(2); 100*x(2) - x(1)];  % system of ODEs
[t,y1] = ode45(f1, [0,2], [1,1])  % fn handle, time values, f1(0)
[t,y2] = ode45(f2, [0:0.1:2], [1,1])  % fn handle, time values, f1(0)
```

Always this order! Time, then state.

Python code

```
from scipy.integrate import solve_ivp
f1 = lambda t, x: [x[0], 100 * x[1]]
f2 = lambda t, x: [x[0] + x[1], 100 * x[1] - x[0]]
soln1 = solve_ivp(f1, [0, 2], [1,1])  # fn, [t0, t1], IC
soln1a = solve_ivp(f1, [0, 2], [1,1], t_eval=[0.1, 0.5, 1.5])  # fn, [t0, t1], IC
soln2a = solve_ivp(f2, [0, 2], [1,1], t_eval=[0.1, 0.5, 1.5])  # fn, [t0, t1], IC
```

Always this order! Time, then state.

Output is much more robust!