Numerical methods for dynamical systems

Alexandre Chapoutot

ENSTA Paris
master CPS IP Paris

2020-2021

Part I

Multi-step methods for IVP-ODE

Initial Value Problem of Ordinary Differential Equations

Consider an IVP for ODE, over the time interval $[0, t_{\text{end}}]$

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \text{with} \quad \mathbf{y}(0) = \mathbf{y}_0$$

IVP has a unique solution $\mathbf{y}(t; \mathbf{y}_0)$ if $f : \mathbb{R}^n \to \mathbb{R}^n$ is Lipschitz in $\mathbf{y}$

$$\forall t, \forall \mathbf{y}_1, \mathbf{y}_2 \in \mathbb{R}^n, \exists L > 0, \quad \| f(t, \mathbf{y}_1) - f(t, \mathbf{y}_2) \| \leq L \| \mathbf{y}_1 - \mathbf{y}_2 \| \quad .$$

Goal of numerical integration

- Compute a sequence of time instants: $t_0 = 0 < t_1 < \cdots < t_n = t_{\text{end}}$
- Compute a sequence of values: $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n$ such that

$$\forall \ell \in [0, n], \quad \mathbf{y}_\ell \approx \mathbf{y}(t_\ell; \mathbf{y}_0) \quad .$$

- s.t. $\mathbf{y}_{\ell+1} \approx \mathbf{y}(t_\ell + h; \mathbf{y}_\ell)$ with an error $\mathcal{O}(h^{p+1})$ where
  - h is the integration **step-size**
  - p is the **order** of the method

Simulation algorithm

```
Data: $f$ the flow, $\mathbf{y}_0$ initial condition, $t_0$ starting time, $t_{\text{end}}$ end time, $h$
      integration step-size
$t \leftarrow t_0$;
$\mathbf{y} \leftarrow \mathbf{y}_0$;
while $t < t_{\text{end}}$ do
    Print($t$, $\mathbf{y}$);
    $y \leftarrow$ Euler($f$,$t$,$\mathbf{y}$,$h$);
    $t \leftarrow t + h$;
end
```

with, the Euler's method defined by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + hf(t_n, \mathbf{y}_n) \quad \text{and} \quad t_{n+1} = t_n + h \ .$$

Multi-step methods

Recall: single-step methods solve IVP using one value $\mathbf{y}_n$ and some values of f.

A multi-step method approximate solution $\mathbf{y}_{n+1}$ of IVP using k previous values of the solution $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, ..., $\mathbf{y}_{n-k-1}$.

Different methods implement this approach

- Adams-Bashforth method (explicit)
- Adams-Moulton method (implicit)
- Backward difference method (implicit)

The general form of such method is

$$\sum_{j=0}^{k} \alpha_j \mathbf{y}_{n+j} = h \sum_{j=0}^{k} \beta_j f(t_{n+j}, \mathbf{y}_{n+j}) \ .$$

with α_j and β_j some constants and $\alpha_k = 1$ and $|\alpha_0| + |\beta_0| \neq 0$

Polynomial interpolation

A quick remainder on polynomial interpolation

Starting point:

- a function $f(t)$
- a sequence of n time instants $t_1, t_2, \dots, t_n$.
- a sequence of points $f_1 = f(t_1)$, $f_2 = f(t_2)$, $\dots$, $f_n = f(t_n)$

Goal

- Find a polynomial p of order n approximating f and passes through the $(n+1)$ function values
$$p(t_i) = f_i$$

Theorem (Uniqueness of the Interpolating Polynomial)

Given n unequal points $x_1, x_2, \dots, x_n$ and arbitrary values $f_1, f_2, \dots, f_n$ there is at most one polynomial p of degree less or equal to $n-1$ such that $p(x_i) = f_i$, $i = 1, \dots, n$.

Note: different algorithms in function of the monomial basis

Polynomial interpolation: basis

Standard basis

We consider

$$p(x) = a_0 + a_1 x_+ a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

we have to find a_i such that $p(x_i) = f(x_i)$ so the **Vandermond matrix**

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \dots & \vdots & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}$$

Lagrange basis

We consider

$$p(x) = f(x_0)\ell_0(x) + f(x_1)\ell_1(x) + \cdots + f(x_n)\ell_n(x)$$

such that

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_i}{x_i - x_j}$$

Polynomial interpolation: error

Interpolation error

If f is $n+1$ continuously differentiable on $[a, b]$ then

$$E_n(x) = (x - x_0)(x - x_1)\dots(x - x_n)\frac{f^{(n+1)}(\xi)}{(n+1)!}$$

with $\xi \in]a, b[$

Comments:

- Vandermond matrix is not use as it is ill-conditioned
- Lagrange interpolation is useful when f change but not x_i

Remark

For our purpose to define multi-step methods, **equidistant time instants** will be considered!

Multi-step methods: Adams family

Adams-Bashforth method – 1

Integral form of IVP

$$\dot{\mathbf{y}} = f(t,\mathbf{y}) \quad \mathbf{y}(t_0) = \mathbf{y}_0 \Leftrightarrow$$
$$\mathbf{y}(t) = \mathbf{y}_0 + \int_{t_0}^{t} f(s,\mathbf{y}(s))ds \Rightarrow \mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} f(s,\mathbf{y}(s))ds$$

Ingredients:

- We denote by $t_i = t_n + ih$ the grid of points in time
- We assume given numerical approximations: $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, ..., $\mathbf{y}_{n-k+1}$ of the exact solution.

we can use $\mathbf{y}_i$, $i = n-k+1, \ldots, n$, to approximate $f(t,\mathbf{y}(t))$ using $f(t_i,\mathbf{y}_i) \equiv \mathbf{f}_i$.

We can use polynomial interpolation with points:

$$\{(t_i, f_i) : i = n-k+1, \ldots, n\}$$

to approximate integral.

Adams-Bashforth method – 2

We have $\mathbf{n+1}$ distinct (equidistant) points

$$(t_0, \mathbf{f}_0), (t_1, \mathbf{f}_1), \cdots, (t_n, \mathbf{f}_n)$$

with $\mathbf{f}_i = f(t_i, \mathbf{y}_i)$

Adams-Bashforth method is defined by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \sum_{i=0}^{\mathbf{n}} \mathbf{f}_i \ell_i(s)ds = \mathbf{y}_n + \sum_{i=0}^{\mathbf{n}} \mathbf{f}_i \int_{t_n}^{t_{n+1}} \ell_i(s)ds$$

Example of first Adams-Bashforth methods of order k:

- $k = 1$: $\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{hf}_n$ (explicit Euler method)
- $k = 2$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\frac{3}{2}\mathbf{f}_n - \frac{1}{2}\mathbf{f}_{n-1}\right)$
- $k = 3$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\frac{23}{12}\mathbf{f}_n - \frac{16}{12}\mathbf{f}_{n-1} + \frac{5}{12}\mathbf{f}_{n-2}\right)$
- $k = 4$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\frac{55}{24}\mathbf{f}_n - \frac{59}{24}\mathbf{f}_{n-1} + \frac{37}{24}\mathbf{f}_{n-2} - \frac{9}{24}\mathbf{f}_{n-3}\right)$

Adams-Bashforth's method – 3

```
from sympy import *

t = Symbol('t', real=True, positive=True)
h = Symbol('h', real=True, positive=True)
tn = Symbol('t_n', real=True, positive=True)
tnm3 = tn - 3*h
tnm2 = tn - 2*h
tnm1 = tn - h
tnp1 = tn + h

fnm3 = Symbol('f_{n-3}', real=True)
fnm2 = Symbol('f_{n-2}', real=True)
fnm1 = Symbol('f_{n-1}', real=True)
fn = Symbol('f_n', real=True)
yn = Symbol('y_n', real=True)
ynp1 = Symbol('y_{n+1}', real=True)

points_order_1 = [ (tn, fn) ]
points_order_2 = [ (tnm1, fnm1), (tn, fn) ]
points_order_3 = [ (tnm2, fnm2), (tnm1, fnm1), (tn, fn) ]
points_order_4 = [ (tnm3, fnm3), (tnm2, fnm2), (tnm1, fnm1), (tn, fn) ]
```

Adams-Bashforth's method – 4

```
def lagrange_basis (time, points):
    acc = 1
    for point in points:
        if (time != point[0]):
            acc = acc * (t - point[0])/(time - point[0])
        else:
            acc = point[1]*acc
    return acc

def lagrange (points):
    acc = 0
    for point in points:
        acc = acc + lagrange_basis (point[0], points)
    return acc

def build_adams (points):
    pl = lagrange(points)
    return simplify(integrate(pl, (t, tn, tnp1)))

print ("##␣Order␣1")
formula1 = build_adams (points_order_1)
print (latex(Eq(ynp1, yn + formula1)))
```

Explicit Adams-Bashforth formulae

- This is an **explicit** ODE solver
- Each integration step involves **only one evaluation of** f
- Using past values of f for order n we use $n - 1$ past values
- Adams-Bashforth algorithm of order n can only be used after $n - 1$ previous steps (**not self starting method**)

Adams-Moulton method – 1

We have $\mathbf{n+2}$ distinct (equidistant) points

$$(t_0, \mathbf{f}_0), (t_1, \mathbf{f}_1), \cdots, (t_n, \mathbf{f}_n), (t_{n+1}, \mathbf{f}_{n+1})$$

with $\mathbf{f}_i = f(t_i, \mathbf{y}_i)$

Adams-Moulton method is defined by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \sum_{i=0}^{\mathbf{n+1}} \mathbf{f}_i \ell_i(s) ds = \mathbf{y}_n + \sum_{i=0}^{\mathbf{n+1}} \mathbf{f}_i \int_{t_n}^{t_{n+1}} \ell_i(s) ds$$

Example of first Adams-Moulton methods of order k:

- $k = 1$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}_{n+1}$ (implicit Euler method)
- $k = 2$: $\mathbf{y}_{n+1} = \frac{h}{2}(\mathbf{f}_n + \mathbf{f}_{n+1}) + \mathbf{y}_n$
- $k = 3$: $\mathbf{y}_{n+1} = \frac{h}{12}(8\mathbf{f}_n + 5\mathbf{f}_{n+1} - \mathbf{f}_{n-1}) + \mathbf{y}_n$
- $k = 4$: $\mathbf{y}_{n+1} = \frac{h}{24}(19\mathbf{f}_n + 9\mathbf{f}_{n+1} - 5\mathbf{f}_{n-1} + \mathbf{f}_{n-2}) + \mathbf{y}_n$

Adams-Moulton method – 3

```
from sympy import *

t = Symbol('t', real=True, positive=True)
h = Symbol('h', real=True, positive=True)
tn = Symbol('t_n', real=True, positive=True)
tnm2 = tn - 2*h
tnm1 = tn - h
tnp1 = tn + h

fnm2 = Symbol('f_{n-2}', real=True)
fnm1 = Symbol('f_{n-1}', real=True)
fn = Symbol('f_n', real=True)
fnp1 = Symbol('f_{n+1}', real=True)
yn = Symbol('y_n', real=True)
ynp1 = Symbol('y_{n+1}', real=True)

points_order_1 = [ (tnp1, fnp1) ]
points_order_2 = [ (tn, fn), (tnp1, fnp1) ]
points_order_3 = [ (tnm1, fnm1), (tn, fn), (tnp1, fnp1) ]
points_order_4 = [ (tnm2, fnm2), (tnm1, fnm1), (tn, fn), (tnp1, fnp1) ]
```

Adams-Moulton method – 4

```
def lagrange_basis (time, points):
    acc = 1
    for point in points:
        if (time != point[0]):
            acc = acc * (t - point[0])/(time - point[0])
        else:
            acc = point[1]*acc
    return acc

def lagrange (points):
    acc = 0
    for point in points:
        acc = acc + lagrange_basis (point[0], points)
    return acc

def build_adams (points):
    pl = lagrange(points)
    return simplify(integrate(pl, (t, tn, tnp1)))

print ("## Order 1")
formula1 = build_adams (points_order_1)
print (latex(Eq(ynp1,yn + formula1)))
```

Implicit Adams-Moulton formulae

- This is an **implicit** ODE solver
- Each integration step involves **only one evaluation of** f but requires solution of algebraic equations
- Using past values of f for order n we use $n-1$ past values
- Adams-Moulton algorithm of order n can only be used after $n-1$ previous steps (**not self starting method**)

Adams-Bashforth-Moulton formulae

Predictor-Corrector methods, example of third order:

$$\begin{aligned}
\text{predictor:}\quad & \mathbf{f}_k = f(t_k, \mathbf{y}_k) \\
& \mathbf{y}^P_{k+1} = \mathbf{y}_k + \frac{h}{12}\left(23f_k - 16\mathbf{f}_{k-1} + 5\mathbf{f}_{k-2}\right) \\
\text{corrector:}\quad & \mathbf{f}^P_{k+1} = f(t_{k+1}, \mathbf{y}^P_{k+1}) \\
& \mathbf{y}^P_{k+1} = \mathbf{y}_k + \frac{h}{12}\left(5\mathbf{f}^P_{k+1} + 8\mathbf{f}_k - \mathbf{f}_{k-1}\right)
\end{aligned}$$

Note: this algorithm is explicit.

Note: we need two evaluations of f per step.

Adams-Bashforth-Moulton formulae - 2

Predictor-Corrector methods, two general forms

- $P(EC)^m$
- $P(EC)^m E$

Note that:

- the corrector methods (usually implicit) can be iterated a few number of times to increase accuracy
- in $P(EC)^m E$, the last evaluation

In that case, instead using Newton method for the implicit method we use a functional iteration approach.

Estimation of the Local Truncation Error

Adams-Bashforth or Adams-Moulton methods are defined from a polynomial interpolation.
Recall, for Adams-Bashforth we have

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_{t_n}^{t_{n+1}} \sum_{i=0}^{n} \mathbf{f}_i \ell_i(s) ds = \mathbf{y}_n + \sum_{i=0}^{n} \mathbf{f}_i \int_{t_n}^{t_{n+1}} \ell_i(s) ds$$

In consequence, it is possible to compute the remainder of the integral, for example
Example of first Adams-Bashforth methods of order k:

- $k = 1$: $\mathbf{y}_{n+1} = \mathbf{y}_n + hf_n$, LTE is $\frac{h^2}{2}\ddot{\mathbf{y}}(\xi)$
- $k = 2$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\frac{3}{2}f_n - \frac{1}{2}f_{n-1}\right)$, LTE is $\frac{5h^3}{12}\mathbf{y}^{(3)}(\xi)$
- $k = 3$: $\mathbf{y}_{n+1} = \mathbf{y}_n + h\left(\frac{23}{12}f_n - \frac{16}{12}f_{n-1} + \frac{5}{12}f_{n-2}\right)$, LTE is $\frac{3h^4}{8}\mathbf{y}^{(4)}(\xi)$

We can do the same for Adams-Moulton methods

Estimation of the Local truncation error

In case of Predictor-Corrector, we can estimate the local truncation error *i.e.*, the distance between the true solution and the numerical one.
For example, PC with AB3 and AM3 we get:

$$\mathbf{y}(t_{n+2}) - \tilde{\mathbf{y}}_{n+2} = \frac{5}{12} h^3 \mathbf{y}^{(3)}(\xi_{\mathrm{AB3}})$$
$$\mathbf{y}(t_{n+2}) - \mathbf{y}_{n+2} = -\frac{1}{12} h^3 \mathbf{y}^{(3)}(\xi_{\mathrm{AM3}})$$

Assuming $\mathbf{y}^{(3)}(\xi_{\mathrm{AM3}}) \approx \mathbf{y}^{(3)}(\xi_{\mathrm{AB3}})$ on the time interval, we get

$$\mathbf{y}_{n+2} - \tilde{\mathbf{y}}_{n+2} \approx \frac{1}{2} h^3 \mathbf{y}^{(3)}(\xi) \Longrightarrow$$
$$\mid \mathbf{y}(t_{n+2}) - \mathbf{y}_{n+2} \mid \approx \frac{1}{12} h^3 \mathbf{y}^{(3)}(\xi_{\mathrm{AM3}}) \approx \frac{1}{6} \mid \mathbf{y}_{n+2} - \tilde{\mathbf{y}}_{n+2} \mid$$

Once this value is obtained, we can control the step-size as for embedded Runge-Kutta methods.

Summary on Adams Family

- These methods are of almost arbitrary order
- Very efficient for non-stiff problem once the starting problem is solved.
- These formula cannot be used to solve stiff problem !
  Except for AM1 and AM2

Adams-Bashforth's method – Implementation

```
def heun_one_step (f, t, y, h):
    y1 = y + h * f(t, y)
    return y + h * 0.5 * ( f(t, y) + f(t+h, y1))

def solve (f, t0, y0, tend, nsteps):
    h = (tend - t0) / nsteps; y = []
    ynm2 = y0
    ynm1 = heun_one_step (f, t0+h, ynm2, h)
    yn = heun_one_step (f, t0+2*h, ynm1, h)
    fnm2 = f(t0, ynm2)
    fnm1 = f(t0+h, ynm1)
    y.append(ynm2); y.append(ynm1)
    time = np.linspace(t0+2*h, tend, nsteps-2)
    for t in time:
        y.append(yn)
        fn = f(t, yn)
        yn = yn + h / 12.0 * (23.0 * fn - 16.0 * fnm1 + 5.0 * fnm2)
        fnm2 = fnm1
        fnm1 = fn
    return [ np.linspace(t0, tend, nsteps), y]

def dynamic (t, y):
    return np.array([-y[1], y[0]])

[t, y] = solve (dynamic, 0.0, np.array([1., 0.]), 2*np.pi*10, 500)
```

Multi-step methods: BDF

Backward Differentiation Formula – 1

We have $n+2$ distinct (equidistant) points

$$(t_0, \mathbf{y}_0), (t_1, \mathbf{y}_1), \cdots, (t_n, \mathbf{y}_n), (t_{n+1}, \mathbf{y}_{n+1})$$

We can interpolate the solution $y(t)$ of IVP ODE from these points:

$$\mathbf{p}(t) = \sum_{i=0}^{n+1} \mathbf{y}_i \ell_i(t)$$

We can differentiate this polynomial in order to be equal to f

$$\dot{\mathbf{p}}(t) = f(t, \mathbf{y})$$

We evaluate this a time $t_{n+1} = t_n + h$ that is

$$\dot{\mathbf{p}}(t_{n+1}) = f(t_{n+1}, \mathbf{y}_{n+1})$$

Backward Differentiation Formula – 1

All the methods

- $f(t_{n+1}, \mathbf{y}_{n+1}) = \frac{1}{h}(-\mathbf{y}_n + \mathbf{y}_{n+1})$ (implicit Euler method)
- $f(t_{n+1}, \mathbf{y}_{n+1}) = \frac{1}{2h}(-4\mathbf{y}_n + 3\mathbf{y}_{n+1} + \mathbf{y}_{n-1})$
- $f(t_{n+1}, \mathbf{y}_{n+1}) = \frac{1}{6h}(-18\mathbf{y}_n + 11\mathbf{y}_{n+1} + 9\mathbf{y}_{n-1} - 2\mathbf{y}_{n-2})$
- $f(t_{n+1}, \mathbf{y}_{n+1}) = \frac{1}{12h}(-48\mathbf{y}_n + 25\mathbf{y}_{n+1} + 36\mathbf{y}_{n-1} - 16\mathbf{y}_{n-2} + 3\mathbf{y}_{n-3})$
- $f(t_{n+1}, \mathbf{y}_{n+1}) =$
  $\frac{1}{60h}(-300\mathbf{y}_n + 137\mathbf{y}_{n+1} + 300\mathbf{y}_{n-1} - 200\mathbf{y}_{n-2} + 75\mathbf{y}_{n-3} - 12\mathbf{y}_{n-4})$
- $f(t_{n+1}, \mathbf{y}_{n+1}) =$
  $\frac{1}{60h}(-360\mathbf{y}_n + 147\mathbf{y}_{n+1} + 450\mathbf{y}_{n-1} - 400\mathbf{y}_{n-2} + 225\mathbf{y}_{n-3} - 72\mathbf{y}_{n-4} + 10\mathbf{y}_{n-5})$

BDF – 2

```
from sympy import *

t = Symbol('t', real=True, positive=True)
h = Symbol('h', real=True, positive=True)
tn = Symbol('t_n', real=True, positive=True)
tnm5 = tn - 5*h
tnm4 = tn - 4*h
tnm3 = tn - 3*h
tnm2 = tn - 2*h
tnm1 = tn - h
tnp1 = tn + h

ynm5 = Symbol('y_{n-5}', real=True)
ynm4 = Symbol('y_{n-4}', real=True)
ynm3 = Symbol('y_{n-3}', real=True)
ynm2 = Symbol('y_{n-2}', real=True)
ynm1 = Symbol('y_{n-1}', real=True)
yn = Symbol('y_n', real=True)
ynp1 = Symbol('y_{n+1}', real=True)
fnp1 = Symbol('f_{n+1}', real=True)

points_order_1 = [ (tn, yn), (tnp1, ynp1) ]
points_order_2 = [ (tnm1, ynm1), (tn, yn), (tnp1, ynp1) ]
points_order_3 = [ (tnm2, ynm2), (tnm1, ynm1), (tn, yn), (tnp1, ynp1) ]
```

BDF – 3

```
def lagrange_basis (time, points):
    acc = 1
    for point in points:
        if (time != point[0]):
           acc = acc * (t - point[0])/(time - point[0])
        else:
            acc = point[1]*acc
    return acc

def lagrange (points):
    acc = 0
    for point in points:
        acc = acc + lagrange_basis (point[0], points)
    return acc

def build_bdf (points):
    pl = lagrange(points)
    return simplify(pl.diff(t).subs(t, tnp1))

print ("## Order 1")
formula1 = build_bdf (points_order_1)
print (latex(Eq(fnp1, formula1)))
```

Solving implicit equation - 1

BDF can be write:

$$\mathbf{y}_{k+1} = \alpha_i h f_{k+1} + \sum_{j=1} i\beta_{ij}\mathbf{y}_{k-j+1}$$

Functional iteration

$$\mathbf{y}_{k+1}^{\ell+1} = \mathbf{y}_k + \alpha_i h f(t_{k+1}, y_{k+1}^{\ell}) + \text{cst}$$

Note:

- initial estimate of $\mathbf{y}_{k+1}^{0}$ can be given by a predictor method.
- Functional iteration converges is

$$\begin{aligned}\mathbf{y}_{k+1}^{\ell+1} - \mathbf{y}_{k+1}^{\ell} &= \alpha_i h \left(f(t_{k+1}, \mathbf{y}_{k+1}^{\ell}) - f(t_{k+1}, \mathbf{y}_{k+1}^{\ell-1}) \right) \\ &= \alpha_i h \mathcal{J}_f()(\mathbf{y}_{k+1}^{\ell} - \mathbf{y}_{k+1}^{\ell-1})\end{aligned}$$

that is if $\mid \alpha_i h \mathcal{J}_f \mid < 1$

In some problems (e.g., stiff) we have $\mid \mathcal{J}_f \mid \gg 1$ so $h <\mid (\alpha_i \mathcal{J}_f)^{-1} \mid \ll 1$.

Solving implicit equation - 2

BDF can be write:

$$\mathbf{y}_{k+1} = \alpha_i h f_{k+1} + \sum_{j=1} i \beta_{ij} \mathbf{y}_{k-j+1}$$

at each step we try to solve:

$$\mathcal{F}(\mathbf{y}_{k+1}) = \alpha_i h f_{k+1} - \mathbf{y}_{k+1} + \sum_{j=1}^{i} \beta_{ij} \mathbf{y}_{k-j+1} = 0$$

Newton operator

$$\mathbf{y}_{k+1}^{\ell+1} = \mathbf{y}_{k+1}^{\ell} - \mathcal{H}^{-1} \mathcal{F}(\mathbf{y}_{k+1}^{\ell})$$

with $\mathcal{H}$ is a matrix defined by:

$$\mathcal{H} = \mathcal{I} - \alpha_i \mathcal{J}_j \cdot h$$

with $\mathcal{J}$ the Jacobian of f evaluated at point $\mathbf{y}_{k+1}^{\ell}$.

Solving implicit equation - 3

- Industrial code does not reevaluate the Jacobian at each step (use the error evaluation as indicator)
- Industrial code has options to deal with Jacobian:
  - providing analytically expression
  - numerical approximations

  speed and range of convergence are influenced by the quality of the Jacobian

The full Jacobian can be approximate by (for each state variable)

$$\frac{\partial f(t, \mathbf{y})}{\partial y_i} \approx \frac{f_{\text{pert}} - f}{\delta y_i}$$

Note: Usually a **quasi-Newton** method is used *i.e.*, the Jacobian is only computed at the begin of the Newton iteration.

Note: strategies to update this computation are usually present in industrial code solver.

Order condition

Theoretical definition of the method order

A general linear multi-step method can be written as

$$\sum_{j=0}^{k} \alpha_j \mathbf{y}_{n+j} = h \sum_{j=0}^{k} \beta_j f_{n+j}$$

with

- $f_{n+j} = f(t_{n+j}, \mathbf{y}_{n+j})$
- $\alpha_k = 1$ (normalization)
- $|\alpha_0| + |\beta_0| \neq 0$

The first and second characteristic polynomial of a linear multi-step method are defined by

$$\rho(\zeta) = \sum_{j=0}^{k} \alpha_j \zeta^j, \qquad \sigma(\zeta) = \sum_{j=0}^{k} \beta_j \zeta^j$$

with $\zeta \in \mathbb{C}$

Theoretical definition of the method order

Linear difference operator

$$\mathcal{L}[z(x);h] = \sum_{j=0}^{k}[\alpha_j z(x+jh) - \beta_j z'(x+jh)] \quad \text{with} \quad z(x) \in C^1$$

After expansion around x we can write

$$\mathcal{L}[z(x);h] = C_0 z(x) + C_1 h z^{(1)}(x) + \cdots + C_q h^q z^{(q)}(x) + \cdots$$

where C_i are constants

Theorem

A linear multi-step and its associated linear difference operator are of order p if $C_0 = C_1 = \cdots = C_p = 0$ and $C_{p+1} \neq 0$.

We know the values of C_i *e.g.*, $C_0 = \sum_{j=0}^{k} \alpha_j$, $C_1 = \sum_{j=0}^{k}(j\alpha_j - \beta_j)$, and $C_q = \sum_{j=0}^{k}\left[\frac{1}{q!}j^q\alpha_j - \frac{1}{(q-1)!}j^{q-1}\beta_j\right]$

Variable order and variable step-size multi-step methods

Step-size control

Problem: Interpolation polynomial for multi-step methods uses equidistant values
Changing the step-size break the equidistant assumption

But from interpolation polynomial we can compute approximation of successive derivative of **y** at time t_n

For example starting from the set of points

$$(t_0, \mathbf{y}_0), (t_1, \mathbf{y}_1), \cdots, (t_n, \mathbf{y}_n),$$

We have

$$\begin{aligned}
\dot{\mathbf{y}}_n &= \frac{1}{6h}\left(11y_n - 18y_{n-1} + 9y_{n-2} - 2y_{n-3}\right)\\
\ddot{\mathbf{y}}_n &= \frac{1}{h^2}\left(2y_n - 5y_{n-1} + 4y_{n-2} - y_{n-3}\right)\\
\mathbf{y}_n^{(3)} &= \frac{1}{h^3}\left(y_n - 3y_{n-1} + 3y_{n-2} - y_{n-3}\right)
\end{aligned}$$

Step-size control

Truncating after the cubic term and evaluation at $t = t_k$ (i.e., $s = 0.0$)

$$\begin{pmatrix} \mathbf{y}_n \\ h\dot{\mathbf{y}}_n \\ \frac{h^2}{2}\ddot{\mathbf{y}}_n \\ \frac{h^3}{6}\mathbf{y}_n^{(3)} \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 6 & 0 & 0 & 0 \\ 11 & -18 & 9 & -2 \\ 6 & -15 & 12 & -3 \\ 1 & -3 & 3 & -1 \end{pmatrix}\begin{pmatrix} \mathbf{y}_n \\ \mathbf{y}_{n-1} \\ \mathbf{y}_{n-2} \\ \mathbf{y}_{n-3} \end{pmatrix}$$

We call **Nordsieck vector** of 3 order the one of the left.

Expressing the derivatives in function of h_{new} we get:

$$\begin{pmatrix} \mathbf{y}_n \\ h_{\text{new}}\dot{\mathbf{y}}_n \\ \frac{h_{\text{new}}^2}{2}\ddot{\mathbf{y}}_n \\ \frac{h_{\text{new}}^3}{6}\mathbf{y}_n^{(3)} \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{h_{\text{new}}}{h_{\text{old}}} & 0 & 0 \\ 0 & 0 & \left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^2 & 0 \\ 0 & 0 & 0 & \left(\frac{h_{\text{new}}}{h_{\text{old}}}\right)^3 \end{pmatrix}\begin{pmatrix} \mathbf{y}_k \\ h_{\text{old}}\dot{\mathbf{y}}_n \\ \frac{h_{\text{old}}^2}{2}\ddot{\mathbf{y}}_n \\ \frac{h_{\text{old}}^3}{6}\mathbf{y}_n^{(3)} \end{pmatrix}$$

Step-size control

In consequence,

$$\begin{pmatrix} \mathbf{y}_n \\ \mathbf{y}_{n-1} \\ \mathbf{y}_{n-2} \\ \mathbf{y}_{n-3} \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & -2 & 4 & -8 \\ 1 & -3 & 9 & -27 \end{pmatrix} \begin{pmatrix} \mathbf{y}_n \\ h_{\text{new}} \dot{\mathbf{y}}_n \\ \frac{h_{\text{new}}^2}{2} \ddot{\mathbf{y}}_n \\ \frac{h_{\text{new}}^3}{6} \mathbf{y}_n^{(3)} \end{pmatrix}$$

Hence we can compute a new equidistant sequence of state values using the new step-size h_{new}.

- Three matrix multiplications are used to change the step-size
- In consequence, multi-step methods use a more conservative step size than RK methods

Order control of multi-step methods

- order control is cheap in linear multi-step methods
  - decrease the order by one, forget one element of the history
  - increasing the order by one, add one element
- In consequence, we can make multi-step method **self starting**.

  but the numerical precision of the previous steps is very important for the stability of the method

  we can also use Runge-Kutta methods to accurately compute the *m* first steps.

Start-up difficulties

It is easy to change order: increase or decrease the state history vector. A basic algorithm would be:

- start with order 1 method
- use order 2 method during the second step
- in the next step use order 3 method
- etc. until the appropriate order is reached

Drawback:

- low orders produce low accurate value

Other idea: use ERK methods as starting point but what about stiff problems and the initial step-size?

Conclusion

Multi-step methods are interesting because

- they are computationally cheaper than Runge-Kutta methods
- they can vary in order

but

- variation of the step-size is possibly more computationally involve
- the properties of stability are weaker than Runge-Kutta methods (may be sufficient for most of the problems)