wdv-notes 140
Stand: 30.NOV.1992 (2.)
Wiss.Datenverarbeitung
FREIE UNIVERSITÄT BERLIN

Mathematica: Sammlung von WDV-Prozeduren – Teil 1.

© 1990–1992 *Karl-Heinz Dittberner*

Pro

Das wissenschaftliche Programmier-System ***Mathematica*** der Firma Wolfram Research, USA (wdv-notes Nr. 71) hat sich inzwischen auf allen Computer-Plattformen – auch auf dem PC – als ein anspruchsvolles Werkzeug zur mathematischen Behandlung theoretischer und numerischer Probleme durchgesetzt.

In diesem Merkblatt werden vollständige Prozeduren und Algorithmen in ***Mathematica*** V 2.0 dargestellt, die von der Abt. Wiss. Datenverarbeitung seit 1990 im Rahmen anderer Fragestellungen entwickelt worden sind. Sämtliche Prozeduren sind sorgfältig getestet und mit Beispielen versehen.

1. Ausgabe eines „genormten“ Plots:

Mit der Prozedur **NPlot** läßt sich eine Funktion, deren Amplitude im Bereich [–1, +1] liegt, im Bereich [*xmin*, *xmax*] grafisch darstellen (plotten). Die Beschriftung der x-Achse erfolgt im Abstand von *dx*. Verschiedene Plotparameter sind voreingestellt, was sich erweitern läßt.

In[1]:=

```
NPlot [  function_, xmin_, xmax_, dx_] :=
  Plot [  function, {x, xmin, xmax},
          Ticks –> {Range [  Floor [xmin],
                             Floor [xmax], dx],
                    Table [ 0.2 i, { i, –5, 5 } ] },
          PlotStyle   –> { Thickness [0.006] },
          DefaultFont –> {"Helvetica", 10 (*pt*) }  ]
```

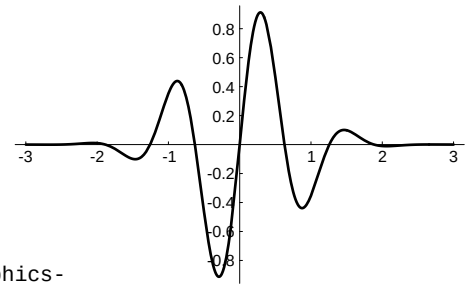
Beispiel:

Mit dem folgenden Aufruf von **NPlot** wird das Wavelet „Morlet-Sinus“ (wdv-notes Nr. 91) dargestellt.

In[2]:=

```
NPlot [ Exp[–x^2] Sin[5 x], –3, 3, 1]
```

Out[2]=




```
-Graphics-
```

2. Eingabe einer Zahl mit Prüfung:

Mit der folgenden Prozedur **GetNumber** wird ein Benutzer aufgefordert, eine Zahl einzugeben, die einer bestimmten Bedingung genügt. Ist diese nicht erfüllt, wird nach einem Hinweis der Benutzer erneut aufgefordert, die Zahl einzugeben. Wird nur die Return-Taste betätigt, dann wird der im Aufruf vorgegebene Wert (*default*) verwendet.

In[3]:=

```
GetNumber  [ question_String,
             condition_:(True&), default_ ] :=
    Module [ {answer},
      While [ True,
        answer = Input ["•" question];
        If [ answer === Null, answer = default];
        If [ NumberQ [answer] &&
             condition [answer], Break [ ] ];
        If [ answer === EndOfFile, Break [ ] ];
        Print ["• Enter a number that satisfies: ",
               condition]
        ];  answer  ]
```

Beispiel:

In[4]:=

```
zahl = GetNumber ["Enter an odd number:",
                  OddQ, 25 ]
```

```
• Enter a number that satisfies: OddQ
• Enter a number that satisfies: OddQ
```

Out[4]=

```
25
```

3. Gradient eines Skalarfeldes :

Mit der Prozedur **grad** kann der Gradient einer skalaren Ortsfunktion (Skalarfeld) $\varphi(x, y, z)$ berechnet werden (wdv-notes Nr. 46). Das Ergebnis ist ein Vektorfeld.

In[5]:=

```
grad [ scalarField_, variable_List ] :=
          D [ scalarField, # ] &  /@  variable
```

Beispiel:

In[6]:=

```
vector = grad [ phi [x, y, z],  {x, y, z} ]
```

Out[6]=

```
   (1,0,0)          (0,1,0)          (0,0,1)
{phi     [x,y,z], phi      [x,y,z], phi     [x,y,z]}
```

4. Divergenz eines Vektorfeldes:

Die Divergenz eines Vektorfeldes ist ein Skalarfeld (wdv-notes Nr. 46) und kann mit der Prozedur **div** ermittelt werden.

In[7]:=

```
div [ vector_List, variable_List ] :=
          Inner [ D, vector, variable, Plus ]
```

Beispiel:

In[8]:=

```
scalar = div [ vector, { x, y, z } ]
```

Out[8]=

```
  (0,0,2)          (0,2,0)          (2,0,0)
phi     [x,y,z] + phi     [x,y,z] + phi     [x,y,z]
```

5. Lösung eines DGL-Systems:

Mit ***Mathematica*** lassen sich auch nichtlineare Differentialgleichungen (DGL), wie sie in der Natur häufig anzutreffen sind, lösen. Im folgenden wird dieses am Beispiel von DGL's vom Typ „van der Pol“ gezeigt. Dazu werden zunächst die DGL's mit den Anfangsbedingungen definiert.

In[9]:=

```
vanderPol = { y'[ t ] == z[ t ],
              z'[ t ] == (1–y[ t ]^2) z[ t ] – y[ t ],
              y[ 0 ] == 20, z[ 0 ] == 0 };
```

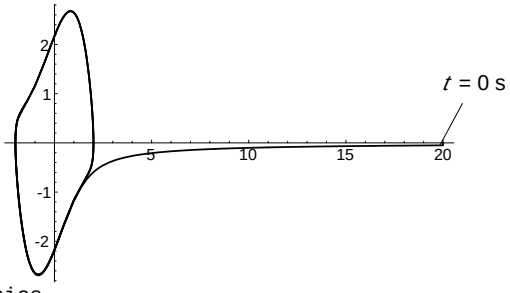
In[10]:=

```
vdPol = NDSolve [ vanderPol, { y[ t ], z[ t ] },
                  { t, 0, 220}, MaxSteps –> 4000 ];
```

In[11]:=

```
ParametricPlot [  Evaluate [ { y[ t ], z[ t ] } /.
                    vdPol [[1]]  ],  { t, 0, 220 },
          DefaultFont –> {"Helvetica", 10},
          PlotPoints  –> 150,
          PlotRange   –> All,
          PlotStyle   –> {Thickness [0.004] }  ]
```

Out[11]=




```
-Graphics-
```

Das Ergebnis ist im Phasenraum $z = f(y)$ für die ersten 220 Sekunden dargestellt, wobei $z = x' = dx/dt$ ist.

6. Ermittlung der Funktionalmatrix:

Gegeben sei eine Liste von Ausdrücken (f_1, f_2, …, f_n), die Funktionen der Variablen (x_1, x_2, …, x_m) beschreiben. Die (n × m)-Funktionalmatrix der partiellen Ableitungen läßt sich dann mit der folgenden Prozedur **JacobiMatrix** ermitteln (wdv-notes Nr. 87).

In[12]:=
```
JacobiMatrix [ funct_List, variable_List ] :=
    Outer [ D, funct, variable ] // MatrixForm
```

Beispiel:

In[13]:=
```
fm = JacobiMatrix [ { f1[x, y], f2[x, y], f3[x,y] }, {x, y} ]
```

Out[13]//MatrixForm=
```
   (1,0)      (0,1)
f1     [x, y]  f1     [x, y]

   (1,0)      (0,1)
f2     [x, y]  f2     [x, y]

   (1,0)      (0,1)
f3     [x, y]  f3     [x, y]
```

7. Quadratische Abweichung von Daten:

Mit der Prozedur **SquaredDev** kann die Summe der Abstandsquadrate zwischen einer Datenliste und einer vorgegebenen analytischen Funktion berechnet werden.

In[14]:=
```
SquaredDev [ data_List, function_, var_ ] :=
    Apply [ Plus, Map [ (# [[ 2 ]] – (function /.
                var –> # [[ 1 ]] ) )^2&, data ] ]
```

Beispiel:

In[15]:=
```
dev = SquaredDev [ { {–1, 2}, {1, 1}, {3, –1} },
                   1/ (2 + x Exp [x] ), x ]
```

Out[15]=
```
          1   2           1   2             1     2
(2 - -------)  + (1 - -------)  + (-1 - ---------)
           1           2 + E                   3
     2 - -                             2 + 3 E
         E
```

In[16]:=
```
N [ dev ]
```

Out[16]=
```
3.57802
```

8. Nichtlineare Kurvenanpassung:

Mit der folgenden Prozedur **NFit** lassen sich die Parameter einer vorgegebenen nichtlinearen analytischen Funktion aufgrund von Daten bestimmen (Curve-Fitting).

In[17]:=
```
NFit [ data_List, function_, var_, startValue__ ] :=
    FindMinimum [
        Evaluate [ SquaredDev [ data, function,
                   var ] // Expand ], startValue ]
```

Beispiel:

In[18]:=
```
par = NFit [ { {–1, 2}, {1, 1}, {3, –1} },
             1/ (a + b x Exp [ x ] ), x, {a, 0.5}, {b, 0} ]
```

Out[18]=
```
{1.17214, {a -> 0.575268, b -> 0.221474}}
```

9. Biologischer Encoder:

Es besteht die Vermutung, daß die Encoder-Kennlinien von biologischen Rezeptoren einem fraktalen Gesetz etwa in der Form $y = a/x^D$ gehorchen (D = fraktale Dimension). Mit der Prozedur **BioEncoder** können die Parameter einer solchen Kennlinie aus gemessenen Daten bestimmt werden.

In[19]:=
```
BioEncoder [ data_List, startValue__ ] :=
  FindMinimum [
      Evaluate [ SquaredDev [ data, a / (x)^d, x] //
                 Expand ], startValue ]
```

Beispiel:

In[20]:=
```
datas = { { 50, 52.3}, {100, 27.3}, {150, 19.2},
          {200, 15.0}, {250, 12.5}, {300, 10.7},
          {350,  9.4}, {400,  8.5}, {450,  7.7} };
```

In[21]:=
```
param = BioEncoder [ datas, {a, 1}, {d, 1} ]
```

Out[21]=
```
{1.09663, {a -> 1704.2, d -> 0.892067}}
```

Für diesen Encoder (Muskelspindel des Frosches) ergibt sich somit als fraktale Dimension $D \approx 0{,}89$.

10. Berechnung der diskreten Faltung:

Mit der Prozedur **Convolve** kann die (exakte) diskrete Faltung (Convolution) eines abgetasteten Signals *s* mit einem Kernel *k* berechnet werden.

In[22]:=
```
Convolve [ s_List, k_List ] :=
    Module [ { addend = Join [ s,Table [ 0,
                { Length [ k ] – 1} ] ], i },
        Sum[ RotateRight [ addend, i–1] * k [[ i ]],
                { i, 1, Length [ k ] } ] ]
```

Beispiel:

In[23]:=
```
sig = Convolve [ {2, 2, 3, 3, 4}, {1, 1, 2} ]
```

Out[23]=
```
{2, 4, 9, 10, 13, 10, 8}
```

11. DGL-Integration nach Runge-Kutta:

Mit den Prozeduren **RungeKutta** sowie **RKStep** lassen sich Differentialgleichungen (DGL) numerisch integrieren. Der folgende Code basiert auf den Angaben in R. Maeder: „Programming in Mathematica“ (Addison-Wesley 1990).

In[24]:=
```
RKStep [ f_, v_, v0_, dt_ ] :=
    Module [ { k1, k2, k3, k4 },
        k1 = dt N [ f /. Thread [ v –> v0] ];
        k2 = dt N [ f /. Thread [ v –> v0 + k1/2] ];
        k3 = dt N [ f /. Thread [ v –> v0 + k2/2] ];
        k4 = dt N [ f /. Thread [ v –> v0 + k3] ];
        v0 + (k1 + 2 k2 + 2 k3 + k4) / 6 ]

RungeKutta [ f_List, v_List, v0_List, { t1_, dt_ } ] :=
    NestList [ RKStep [ f, v, #, N [ dt ] ]&, N [ v0 ],
               Round [ N [ t1/dt ] ] ] /;
    Length [ f ] == Length [ v ] == Length [ v0 ]
```

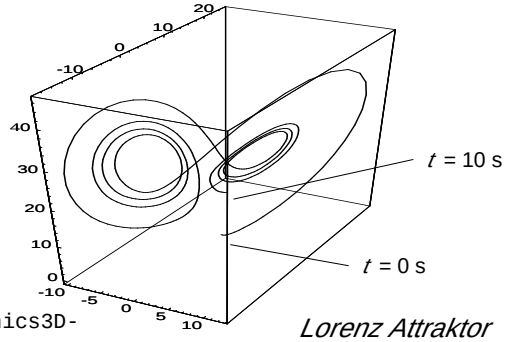
Beispiel:

Der Lorenz Attraktor – ein „chaotisches“ System – wird durch ein nichtlineares DGL-System beschrieben:

$x' = -3\,(x - y)$; $y' = -x\,z + 26{,}5\,x - y$; $z' = x\,y - z$. Die Lösung erfolgt mit den Anfangswerten $x = 0$, $y = 5$, $z = 0$ für die ersten 10 Sekunden und wird als Raumkurve dargestellt.

In[25]:=
```
LorenzAttractor = {–3 (x-y), –x z + 26.5 x – y, x y – z };
attractor = RungeKutta [ LorenzAttractor, {x, y, z},
                         {0, 5, 0}, {10, 0.02} ];
Show [ Graphics3D [ { Line [attractor] } ],
       Axes –> Automatic,
       DefaultFont –> {"Helvetica", 10} ]
```

Out[25]=



-Graphics3D-

Lorenz Attraktor