Blatt 12

Aufgabe 46

```
> restart;
> with(plots):
> with(VectorCalculus):
> BasisFormat(false):
> f := (x, y) -> 4*x^2 - 3*x*y;
> param := t -> ( cos(t), sin(t) );
> g := (f@param)(t);  # f eingeschränkt auf den Einheitskreis
> dg := diff(g, t);   # Ableiten
> d2g := diff(dg, t);
> # Test
> d2g - diff(g, t$2);
> krit := solve({ dg = 0 }, t);
> # Die Lösungen werden im Interval [ -pi, pi ] gesucht. Test
> for kk from 1 to nops([ krit ]) do
    'dg'(rhs(krit[kk][1])) = simplify(subs(krit[kk], dg));
  od;
> # Werte der 2. Ableitung (hinreichendes Kriterium 2. Ordnung)
> d2g_krit := seq(simplify(subs(krit[kk], d2g)), kk = 1..nops([
  krit ]));
> # Auswerten
> minMax[-1] := "Maximum":
> minMax[1] := "Minimum":
> minMax[0] := "???": # Bei 2. Ableitung = 0 müssen weitere
  Kriterien hinzugenommen werden
> for kk from 1 to nops([ krit ]) do
    p   := rhs(krit[kk][1]);
    gp  := subs(krit[kk], g);
    typ := minMax[sign(d2g_krit[kk])];
    print(t = p, 'g'('t') = gp, d2g_krit[kk], typ);
  end do:
> # Schön darstellen
> p_f := plot3d(f(x, y), x = -1.1 .. 1.1, y = -1.1 .. 1.1);
> p_g := spacecurve([ param(t)[1], param(t)[2], g ], t = -Pi ..
  Pi, color = black, thickness = 3);
> # Kritische Punkte, eingesetzt in die Parametrisierung
> kp3d := [ seq(subs(krit[kk], [ cos(t), sin(t), g ]), kk = 1..
  nops([ krit ])) ]:
> p_p := pointplot3d(kp3d, symbol=circle,  symbolsize = 50, color
  = yellow) ;
> display([ p_f, p_g, p_p ]);
```

Aufgabe 47

```
> restart;
> with(LinearAlgebra):
> with(ArrayTools):
> with(VectorCalculus):
> BasisFormat(false):
> SetCoordinates('cartesian'[x, y]);
> f := (3*x^2 + x + y - 3*y^2) * exp(-(x^2 + y^2));
> x0 := [ 0, 0 ];
> dx := [ x - x0[1], y - x0[2] ];
```

Manuelle Methode

(von Hand -- auf Papier! -- alle Summanden aufstellen)

```
> Tf[3] := simplify(
    subs([ x = x0[1], y = x0[2] ], f) +
    subs([ x = x0[1], y = x0[2] ], diff(f, x)) * dx[1] + subs([
  x = x0[1], y = x0[2] ], diff(f, y)) * dx[2] +
    (1/2)*(
     subs([ x = x0[1], y = x0[2] ], diff(f, x, y)) * dx[1] * dx
  [2] +
     subs([ x = x0[1], y = x0[2] ], diff(f, y, x)) * dx[2] * dx
  [1] +
     subs([ x = x0[1], y = x0[2] ], diff(f, x$2)) * dx[1]^2 +
     subs([ x = x0[1], y = x0[2] ], diff(f, y$2)) * dx[2]^2
    ) +
    (1/6)*(
     3 * subs([ x = x0[1], y = x0[2] ], diff(f, x$2, y)) * dx
  [1]^2 * y +
     3 * subs([ x = x0[1], y = x0[2] ], diff(f, y$2, x)) * dx
  [2]^2 * x +
     subs([ x = x0[1], y = x0[2] ], diff(f, x$3)) * dx[1]^3+
     subs([ x = x0[1], y = x0[2] ], diff(f, y$3)) * dx[2]^3));
```

Direkte Methode

(nach der Difinition)

```
> Gf := Gradient(f);
> Hf := Hessian(f);
> # Aber was ist mit der 3. Ableitung?
> # Alternativ (direkt mit der Definition höherer Ableitungen
  # mehrerer Veränderlicher):
> xy[1] := x; xy[2] := y;
> df := Array(1 .. 2, i -> diff(f, xy[i]));
> # Test
> 'df' - 'Gf' = convert(df, 'Vector') - Transpose(Gf);
> d2f := Array(1 .. 2, 1 .. 2, (i, j) -> diff(f, [ xy[i], xy[j] ]
  ));
> # Test
> 'd2f' - 'Hf' = convert(d2f, Matrix) - Hf;
```

```
> # Die dritte Ableitung ist ein 3-Tensor!
> d3f := Array(1 .. 2, 1 .. 2, 1 .. 2, (i, j, k) -> diff(f, [ xy
  [i], xy[j], xy[k] ]));
> # So sieht die erste Zeile der ersten Ebene der 3. Ableitung
  aus:
> d3f(1, 1..2, 1);
> # Taylorpolynom:
  # T_k[f,x_0](x) = sum_i=0^k D^i f(x_0) [(x - x_0), ..., (x -
  x_0)]/i!
> # Auswerten der Ableitungen
> auswerten := (f, v) -> subs([ x = v[1], y = v[2] ], f);
> # Prozedur um einen k-Tensor auf k Vektoren anzuwenden,
  # die als Array übergeben werden
> anwenden := proc (f, vs)
    description "Wende einen k-Tensor auf k Vektoren an.";
    local k, ind, ii, kk, curSumd, curSum;

    # Anzahl Vektoren/Stufe des Tensors
    k := nops(vs);

    # alle möglichen Index-k-Tupel des Tensors
    ind := indices(f);

    curSum := 0;
    for ii in ind do
      # f[ii[1]][ii[2]]...[ii[k]] auswerten
      curSumd := f(seq(ii[ll], ll = 1..nops(ii)));

      for kk from 1 to nops(vs) do
        # ii-ter Eintrag des kk-ten Vektors
        curSumd := curSumd * vs[kk][ii[kk]];
      end do;
      curSum := curSum + curSumd;
    end do;
    return curSum;
  end proc:
> #f(seq(ii[ll], ll = 1..nops(ii))) * product(dx[ii[i]], i = 1..
  nops(ii));
> # Summanden der Taylorentwicklung (nur zum Ansehen):
> 'f'('x0') = auswerten(f, x0);
> 'df'('x0') * ('x' - 'x0') = anwenden(auswerten(df,  x0), [ dx ]
  ) ;
> ('x' - 'x0')^'T' *'d2f'('x0') * ('x' - 'x0') = anwenden
  (auswerten(d2f, x0), [ dx, dx ]);
> 'd3f'('x0')['x' - 'x0','x' - 'x0','x' - 'x0'] = anwenden
  (auswerten(d3f, x0), [ dx, dx, dx ]);
> # Jetzt zusammensetzen
```

```
> Tf_alt[3] := auswerten(f, x0)/factorial(0) + anwenden(auswerten
  (df, x0), [ dx ])/factorial(1)
       + anwenden(auswerten(d2f, x0) , [ dx, dx ])/factorial(2)
      + anwenden(auswerten(d3f, x0) , [ dx, dx, dx ])/factorial
  (3);
> Tf[3] := simplify(Tf[3]);
```

Auf die 1D-Version zurückführen

(schreibe g[x_0,x](t) := f(x_0 + t*(x - x_0) und bestimme die Taylor-Entwicklung von g. Dann ergbit sich T_k(f, x_0)(x) = T_k(g[x_0,x], 0)(1).)

```
> # Darf jemand anderes machen ;-).
> # Probe
> Tf[3] - Tf_alt[3];
```

Plotten

```
> plot3d([ f, Tf[3] ], x=-2..2, y=-2..2, view=-5..5,
        color = [ 'red', 'blue' ]);
> #plot3d([ f, Tf[3] ], x=-1..1, y=-1..1, color = [ 'red', 'blue'
  ]);
```

Aufgabe 48

```
> # Satz über die Implizite Funktion
> restart:
> with(LinearAlgebra):
> f := (x, y) -> exp(y) + y^3 + x^3 + x^2 - 1;
> implFun := proc(f, x)
    local Dyf, r, dim, g;
    # Bestimme den Rang des y-Anteils der Jacobi-Matrix
    Dyf := diff(f(x, y), y);
    dim := max(Dimension(convert([f(x,y)], Matrix)));
    r  := Rank(convert([Dyf], Matrix));
    printf("Dimension: %d, Rang: %d\n", dim, r);
    # Die implizite Funktion g, die als Lösung von f(x, g(x)) = 0
  gegeben
    # ist, gibt es, wenn der Rang voll ist.
    if r = dim then
      print("y-Anteil der Jacobi-Matrix ist invertierbar,
  implizite Funktion existiert.");
      g := rhs(solve({ f(x, g) = 0 }, { g })[1]);
    else
      print("y-Anteil der Jacobi-Matrix ist nicht invertierbar,
  implizite Funktion existiert NICHT.");
    end if;
    return g;
  end proc;
> g := implFun(f, x);
> plot(g, x = -3..2, y = -2..2, numpoints = 100);
> # kein sehr schöner Plot ...
```

```
> # Alternative
> with(plots):
> implicitplot({ f(x, y) = 0 }, x = -3..2, y=-5..5) ;
```

Aufgabe 49

```
> restart:
> with(LinearAlgebra):
> with(VectorCalculus):
> with(plots):
> BasisFormat(false):
> f := x^2 * y;
> g := x^2 + y^2 - 3;
> L := f + lambda * g;
> dL := [ diff(L, x), diff(L, y), diff(L, lambda) ];
> krit := seq(allvalues(s), s = solve({ dL[1] = 0, dL[2] = 0, dL
  [3] = 0 }, { x, y, lambda })):
> HL := Hessian(L, [ x, y ]);
> GradG := Gradient(g, [ x, y ]);
> # Wenn die letzte Spalte der Ausgabe positiv für alle Werte
  # von s[1], s[2] ist, so haben wir ein lokales Minimum; wenn
  # negativ, dann ein Maxiumum, wenn dem nicht so ist, haben
  # wir kein sicheres Kriterium.
> for kk from 1 to nops([ krit ]) do
    p   := subs(krit[kk], [ x, y ]);
    fp  := subs(krit[kk], f);
    typ := minMax[sign(d2g_krit[kk])];
    HLp := subs(krit[kk], HL);

    # Tangentialvektor bestimmen. Der Gradient steht
    # orthogonal auf dem Rand des Gebietes, d.h.
    # Tangentialen stehen orthogonal auf dem Gradienten.
     GradGp := subs(krit[kk], GradG);
    print("orthogonale:", GradGp);
     tans := [ solve({ GradGp[1] * s[1] + GradGp[2] * s[2] = 0 },
            { s[1], s[2] }) ];
    print("Tangentialraum:", tans);
     v := subs(tans[1], <s[1], s[2]>);
    kriterium := seq(Transpose(v).HLp.v, s in tans);
     print([x,y] = p, 'f'(['x','y']) = fp, kriterium);
  end do:
> # Jetzt noch plotten (optional)
> p_f := plot3d(f, x = -sqrt(3)*1.1..sqrt(3)*1.1, y = -sqrt(3)*
  1.1..sqrt(3)*1.1, view = -3..3);
> param := [ cos(t)*sqrt(3), sin(t)*sqrt(3) ];
> p_t := spacecurve([ param[1], param[2], subs([ x = param[1], y
  = param[2] ], f) ], t = -Pi..Pi, color = black, thickness = 3);
```

```
> # Kritische Punkte
> kp3d := [ seq(subs(krit[kk], [ x, y, f ]), kk = 1..nops([ krit
  ])) ];
> p_p := pointplot3d(kp3d, symbol=circle,  symbolsize = 50, color
  = yellow);
> display([ p_f, p_t, p_p ]);
```