

```

> restart;
  with(plots):
  with(linalg):
Warning, the name changecoords has been redefined
Warning, the protected names norm and trace have been redefined and unprotected

```

– Pagodenfunktion

Eine Lösung mit Hilfsfunktion:

```

> Phi := proc(t::numeric)
  if t <= -1 or t >= 1 then return 0
  elif t >= 0 then return 1-t
  else return 1+t
  end if;
end proc;

Φ := proc(t::numeric)
  if t ≤ -1 or 1 ≤ t then return 0 elif 0 ≤ t then return 1 - t else return 1 + t end if
end proc

> pagode := proc(x::numeric, y::numeric)
  return Phi(x)*Phi(y);
end proc;

  pagode := proc(x::numeric, y::numeric) return Φ(x)*Φ(y) end proc

```

Und eine Pagodenfunktion ohne Hilfsfunktion:

```

> pagode2 := proc(x::numeric, y::numeric)
  local erg;
  if x <= -1 or x >= 1 then erg := 0
  elif x >= 0 then erg := 1-x
  else erg := x+1
  end if;

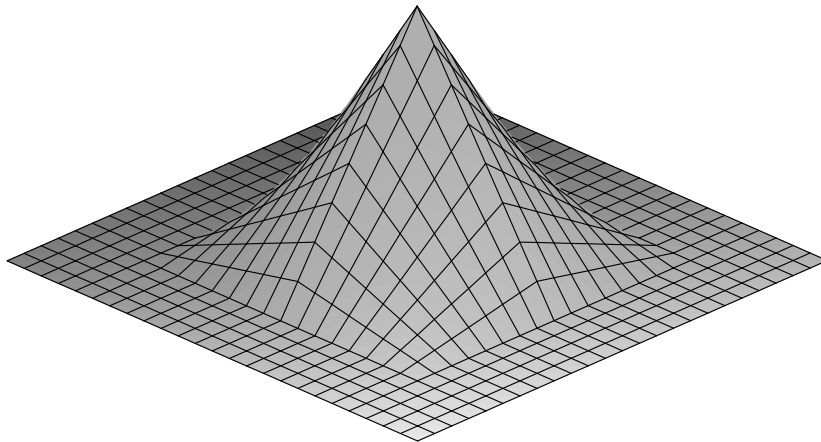
  if y <= -1 or y >= 1 then erg := 0
  elif y >= 0 then erg := erg*(1-y)
  else erg := erg*(y+1)
  end if;

  return erg;
end proc;

pagode2 := proc(x::numeric, y::numeric)
local erg;
  if x ≤ -1 or 1 ≤ x then erg := 0 elif 0 ≤ x then erg := 1 - x else erg := 1 + x end if;
  if y ≤ -1 or 1 ≤ y then erg := 0
  elif 0 ≤ y then erg := erg*(1 - y)
  else erg := erg*(y + 1)
  end if;
  return erg
end proc

> plot3d(pagode, -1.5 .. 1.5, -1.5 .. 1.5);

```



– Binomialkoeffizienten

– Iterative Variante

```

> binom1 := proc(n::numeric, k::nonnegint)
  local zaehler, nenner, i;
  zaehler := 1;
  nenner := 1;
  for i from 1 to k do
    zaehler := zaehler * (n+1-i);
    nenner := nenner * i;
  end do;
  return zaehler/nenner;
end proc;
binom1 := proc(n::numeric, k::nonnegint)
local zaehler, nenner, i;
  zaehler := 1;
  nenner := 1;
  for i to k do zaehler := zaehler*(n + 1 - i); nenner := nenner*i end do;
  return zaehler / nenner
end proc
[ > binom1(4,0);
1
[ >

```

– Rekursive Variante

```

> binom2 := proc(n::nonnegint, k::nonnegint)
    if k=0 or k=n then return 1
    else return binom2(n-1,k-1) + binom2(n-1,k)
    end if;
end proc;
binom2 := proc(n::nonnegint, k::nonnegint)
    if k=0 or k=n then return 1
    else return binom2(n-1, k-1)+binom2(n-1, k)
    end if
end proc
> binom2(4,2);
6
>

```

– Listen

Zunächst eine iterative Variante:

```

> teilliste := proc(l1::list, l2::list)
    local i, j, gleich;
    for i from 0 to nops(l2)-nops(l1) do
        # teste ob l1 der Anfang von l2 ist,
        # wenn man die ersten i Zeichen in l2 weglässt.
        # Dazu zählt man z.B. die gleichen Zeichen
        gleich := 0;
        for j from 1 to nops(l1) do
            if l2[i+j] = l1[j]
                then gleich := gleich + 1
            end if;
        end do;
        if gleich = nops(l1) then return true end if;
    end do;
    return false;
end proc;
teilliste := proc(l1::list, l2::list)
local i, j, gleich;
    for i from 0 to nops(l2) - nops(l1) do
        gleich := 0;
        for j to nops(l1) do if l2[i+j]=l1[j] then gleich := gleich + 1 end if end do;
        if gleich = nops(l1) then return true end if
    end do;
    return false
end proc
> teilliste([7,8],[4,2,3,4,5,6,7,8]);
true
Es gibt auch eine recht nette, rekursive Lösung:
> teilliste_rek := proc(l1::list, l2::list)
    local i;
    # l1 kann keine Teilliste sein,

```

```

# wenn sie mehr Elemente als l2 hat:
if nops(l1) > nops(l2) then return false end if;

# fängt l2 mit l1 an?
# falls nicht: prüfe ob rest(l2) mit l1 anfängt!
for i from 1 to nops(l1) do
  if l1[i] <> l2[i]
    then return teilliste_rek(l1,l2[2..-1]) end if;
end do;
return true;
end proc;
teilliste_rek := proc(l1::list, l2::list)
local i;
  if nops(l2) < nops(l1) then return false end if;
  for i to nops(l1) do if l1[i] ≠ l2[i] then return teilliste_rek(l1, l2[2 .. -1]) end if end do
  ;
  return true
end proc
> teilliste_rek([2,3,5],[4,2,3,4,5,6,7,8]);
teilliste_rek([2,3,4],[4,2,3,4,5,6,7,8]);

                                false
                                true
[ >

```

– Bäume

Suche nach dem größten Knoten:

Zunächst zwei Bäume der Tiefe 1 ("Blätter"):

```

> b1 := [10, []];
> b2 := [7, []];

```

```

                                b1 := [10, []]

```

```

                                b2 := [7, []]

```

Ein Baum der Tiefe 2:

```

> b1[2] := [eval(b1), b2]: b1;

```

```

                                [10, [[10, []], [7, []]]]

```

Und ein Baum der Tiefe 3:

```

> b3 := [3, [b1, b2, b2]];

```

```

                                b3 := [3, [[10, [[10, []], [7, []]]], [7, []], [7, []]]

```

```

> maxknoten := proc(baum)

```

```

  local max, sohn, maxsohn;

```

```

  max := baum[1];

```

```

  for sohn in baum[2] do

```

```

    maxsohn := maxknoten(sohn);

```

```

    if maxsohn > max then max := maxsohn end if;

```

```

  end do;

```

```

  return max;

```

```

end proc;

```

```

maxknoten := proc(baum)

```

```

local max, sohn, maxsohn;
    max := baum[1];
    for sohn in baum[2] do
        maxsohn := maxknoten(sohn); if max < maxsohn then max := maxsohn end if
    end do;
    return max
end proc
> maxknoten(b3);

```

10

– Matrizen

Ist eine Matrix eine M-Matrix?

```

> Mmatrix := proc(a::matrix, n::posint)
    local zeile, spalte, sum;
    for zeile from 1 to n do
        sum := 0;
        for spalte from 1 to n do
            sum := sum + a[zeile, spalte];
        end do;
        if sum <= 0 then return false end if;

        for spalte from 1 to n do
            if zeile <> spalte and a[zeile, spalte] >= 0
            then return false
            end if;
        end do;
    end do;
    return true;
end proc;

Mmatrix := proc(a::matrix, n::posint)
    local zeile, spalte, sum;
        for zeile to n do
            sum := 0;
            for spalte to n do sum := sum + a[zeile, spalte] end do;
            if sum ≤ 0 then return false end if;
            for spalte to n do if zeile ≠ spalte and 0 ≤ a[zeile, spalte] then return false end if
            end do
        end do;
        return true
    end proc
>

```