Modellbildung und Simulation SS2011
Lineare Iterationsverfahren

```
> restart;
> with(plots):
> with(LinearAlgebra):
```

Hilfsfunktionen: Bilder malen

bild malt einen Vektor als stueckweise lineare Funktion ueber dem Einheitsintervall.
Der Parameter **p_options** ist eine Liste von Plot-Optionen, z.B. **[color=BLUE,thickness=2]**

```
> bild := proc(u::Vector,p_options::list(equation))
  local i,m;
    m := Dimension(u)+1;
    plot([[0,0],seq([i/m,u[i]], i=1..m-1),[1,0]],style=line,op
  (p_options))
  end;
```

$$bild := \mathbf{proc}(u::Vector, p\_options::(list(equation))) \quad \textbf{(1.1)}$$

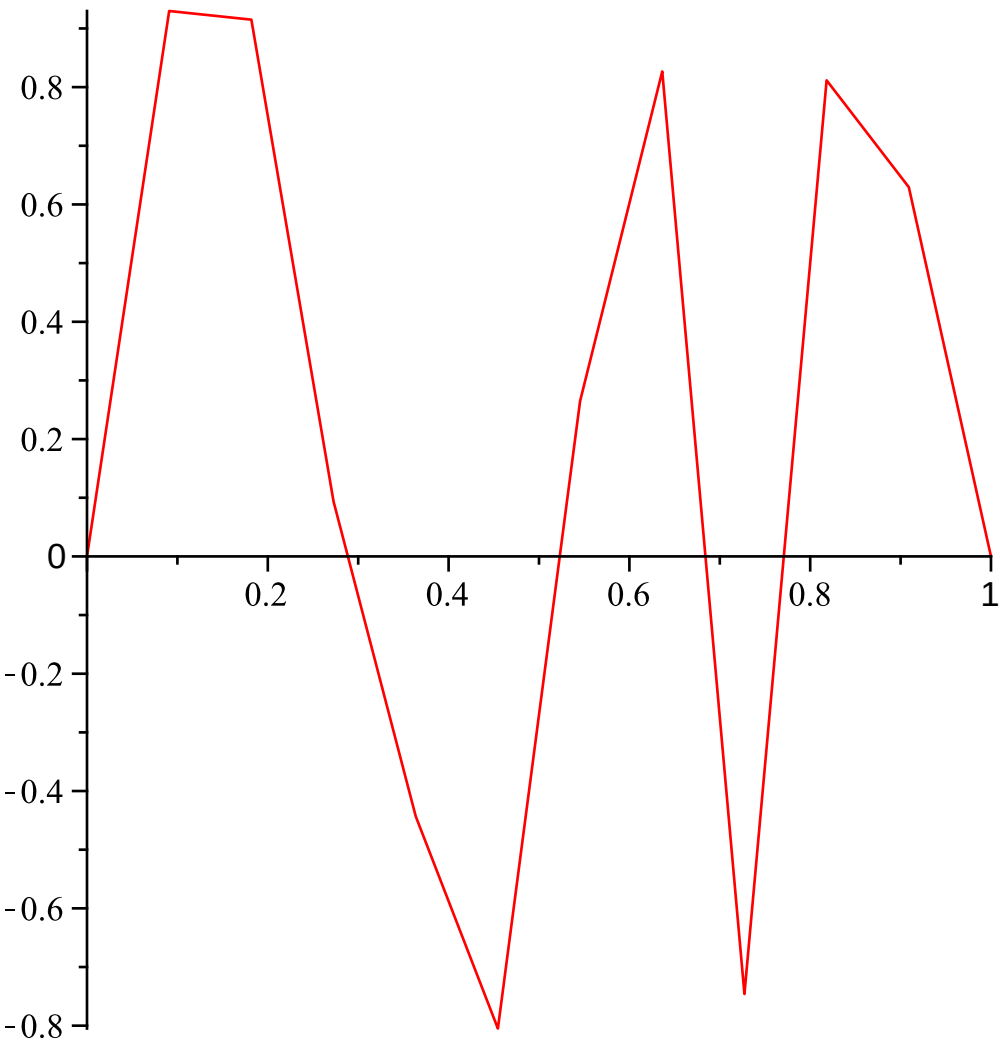
$$\mathbf{local}\ i, m;$$

$$m := LinearAlgebra:\text{-}Dimension(u) + 1;$$

$$plot([[0, 0], seq([i/m, u[i]], i = 1 ..m - 1), [1, 0]], style = line, op(p\_options))$$

$$\mathbf{end\ proc}$$

```
> bild(RandomVector(10,generator=-1.0 .. 1.0),[]);
```



bilder malt eine Liste von Bildern wie oben in ein Diagramm

```
> farben := proc(i::posint)
     return [RED,BLUE,GREEN,BLACK][(i-1) mod 4 +1]
  end;
```

farben := **proc**(*i*::*posint*) **(1.2)**

return [*RED*, *BLUE*, *GREEN*, *BLACK*][*mod*(*i* − 1, 4) + 1]

end proc

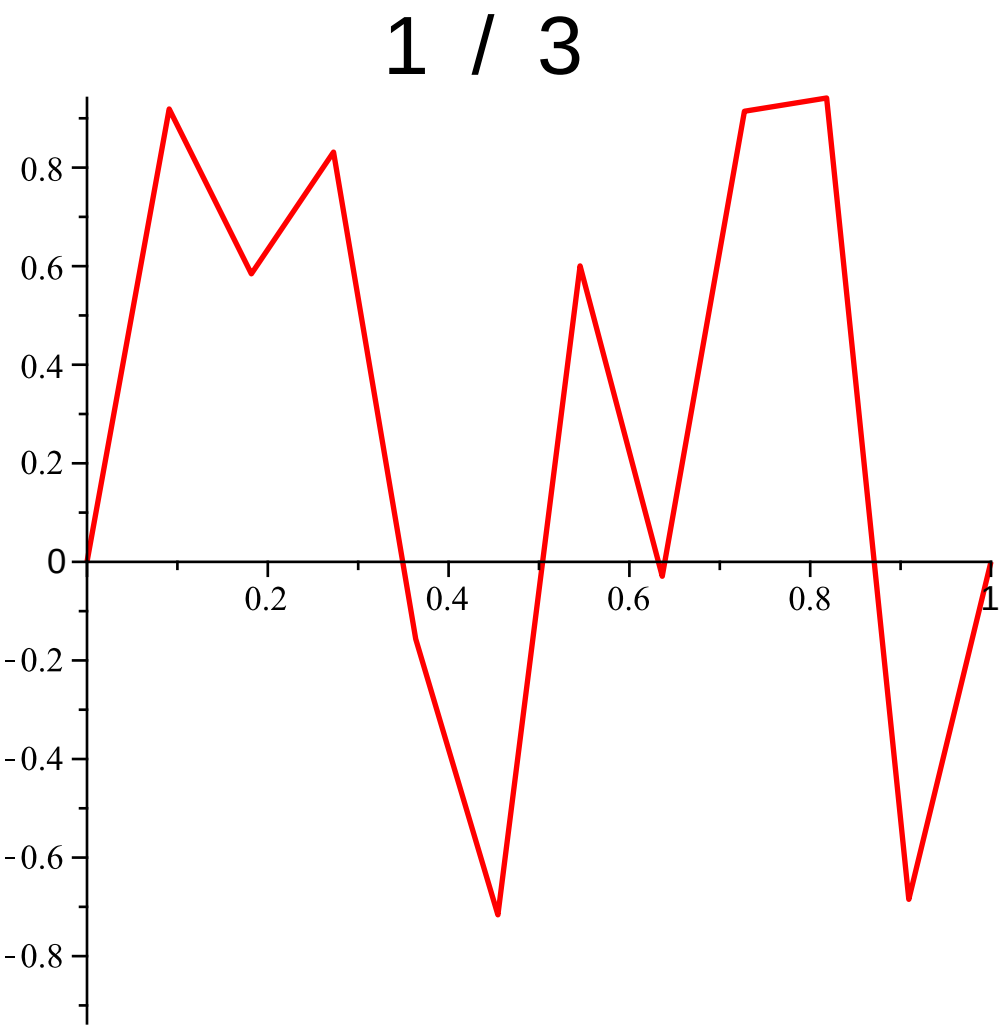
```
> bilder := proc(l::list(Vector))
    display([seq(bild(l[i],
                [color=farben(i),
                 title=sprintf("%d / %d", i, nops(l)),
                 titlefont=[HELVETICA,24],
                 thickness=2]),
               i=1..nops(l))],
       insequence=true);
  end;
```

bilder := **proc**(*l*::(*list*(*Vector*))) **(1.3)**

plots:-*display*([*seq*(*bild*(*l*[*i*], [*color* = *farben*(*i*), *title* = *sprintf*("%d / %d", *i*, *nops*(*l*)),

```
    titlefont = [HELVETICA, 24], thickness = 2]), i = 1 ..nops(l)) ], insequence = true)
end proc
```

```
> bilder([seq(RandomVector(10,generator=-1.0 .. 1.0),i=1..3)]);
```



Residuum zum Differenzenstern $\frac{[-1, 2, -1]}{h^2}$

Hier ist die Matrix A ausprogrammiert; wir brauchen sie nur in der Form (x,b) -> b - Ax, berechnen also das Residuum von Ax=b:

```
> residuum := proc(x::Vector,b::Vector)::Vector;
  local i,m,hh,Ax;
    m := Dimension(x)+1;
    if m=2 then Ax := Vector(1,8*x[1])
    else
      hh := 1/m^2;
      Ax := Vector(m-1);
      Ax[1] := (2*x[1] - x[2])/hh;
```

```
      for i from 2 to m-2 do
         Ax[i] := (2*x[i]-x[i-1]-x[i+1])/hh;
      end do;
      Ax[m-1] := (2*x[m-1] - x[m-2])/hh;
    end if;
    return b - Ax;
  end;
```

$$residuum := \mathbf{proc}(x::Vector, b::Vector)::Vector; \qquad (2.1)$$

```
  local i, m, hh, Ax;
  m := LinearAlgebra:-Dimension(x) + 1;
  if m = 2 then
    Ax := Vector(1, 8 * x[1])
  else
    hh := 1 / m^2;
    Ax := Vector(m - 1);
    Ax[1] := (2 * x[1] - x[2]) / hh;
    for i from 2 to m - 2 do Ax[i] := (2 * x[i] - x[i - 1] - x[i + 1]) / hh end do;
    Ax[m - 1] := (2 * x[m - 1] - x[m - 2]) / hh
  end if;
  return b - Ax
end proc
```

Jacobi-Verfahren

Ein Schritt des (mit Faktor α gedaempften) Jacobi-Verfahrens: $x^{i+1} = x^i + \alpha\, D_A^{(-1)}\,(b - A\,x^i)$.

Die Parameter b und α werden durchgereicht, das Ergebnis ist die Sequenz x^{i+1}, b, α, die wieder als Parameter von **jacobi** auftreten kann. Das erlaubt die Hintereinanderausfuehrung von j Iterationen mittels **jacobi@@j**.

Wenn man nur an der Iterierten x^{i+1} interessiert ist, schreibt man halt **jacobi(x,b,alpha)[1]**.

```
> jacobi := proc(x::Vector,b::Vector,alpha::numeric)::Vector,
  Vector,numeric;
  local r,hh;
    r := residuum(x,b);
    hh := 1.0/(Dimension(x)+1)^2;
    return x + alpha/(2/hh)*r, b, alpha
  end;
```

$$jacobi := \mathbf{proc}(x::Vector, b::Vector, \mathrm{alpha}::numeric)::Vector, Vector, numeric; \qquad (3.1)$$

```
  local r, hh;
```

```
    r := residuum(x, b);
    hh := 1.0 / ((LinearAlgebra:-Dimension(x) + 1)^2);
    return x + 1/2 * alpha * hh * r, b, alpha
end proc
```

Eigenvektoren von A und damit der Jacobi-Iteration:

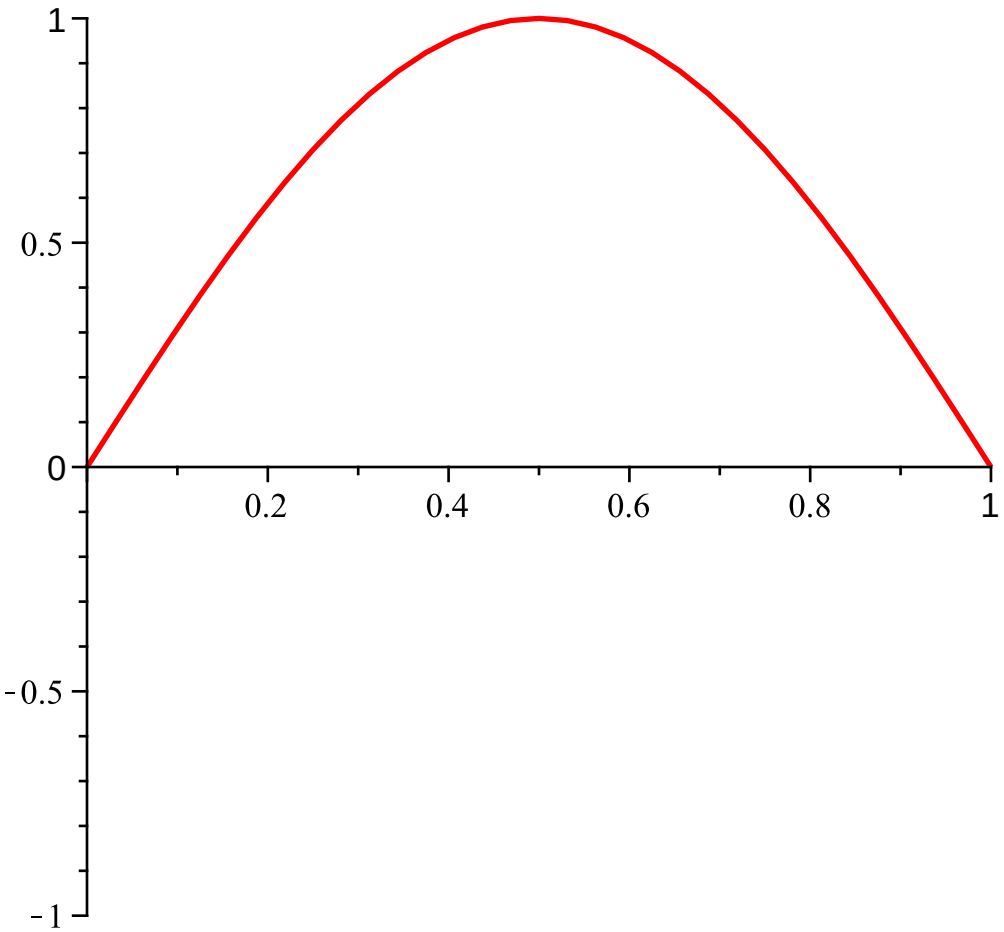
Wir wissen: der Vektor η^k mit Komponenten $\left(\eta^k\right)_i = \sin\left(\frac{i\,k\,\pi}{m}\right)$ ist Eigenvektor von A zum Eigenwert $\lambda_k = 4\,m^2 \sin\left(\frac{k\,\pi}{2\,m}\right)^2$, mithin Eigenvektor der Iterationsmatrix M des Jacobi-Verfahrens zum Eigenwert $1 - \frac{\alpha\,\lambda_k}{2\,m^2}$.

```
> eigv := proc(m::posint, k::posint)::Vector;
    return Vector(m-1, i->evalf(sin(i*k*Pi/m)))
  end;
```

```
eigv := proc(m::posint, k::posint)::Vector;
    return Vector(m - 1, i→evalf(sin(i * k * Pi / m)))
end proc
```

(4.1)

```
> bilder([seq(eigv(32,k),k=1..31)]);
```



Beispiele fuer Konvergenz des Jacobi-Verfahrens

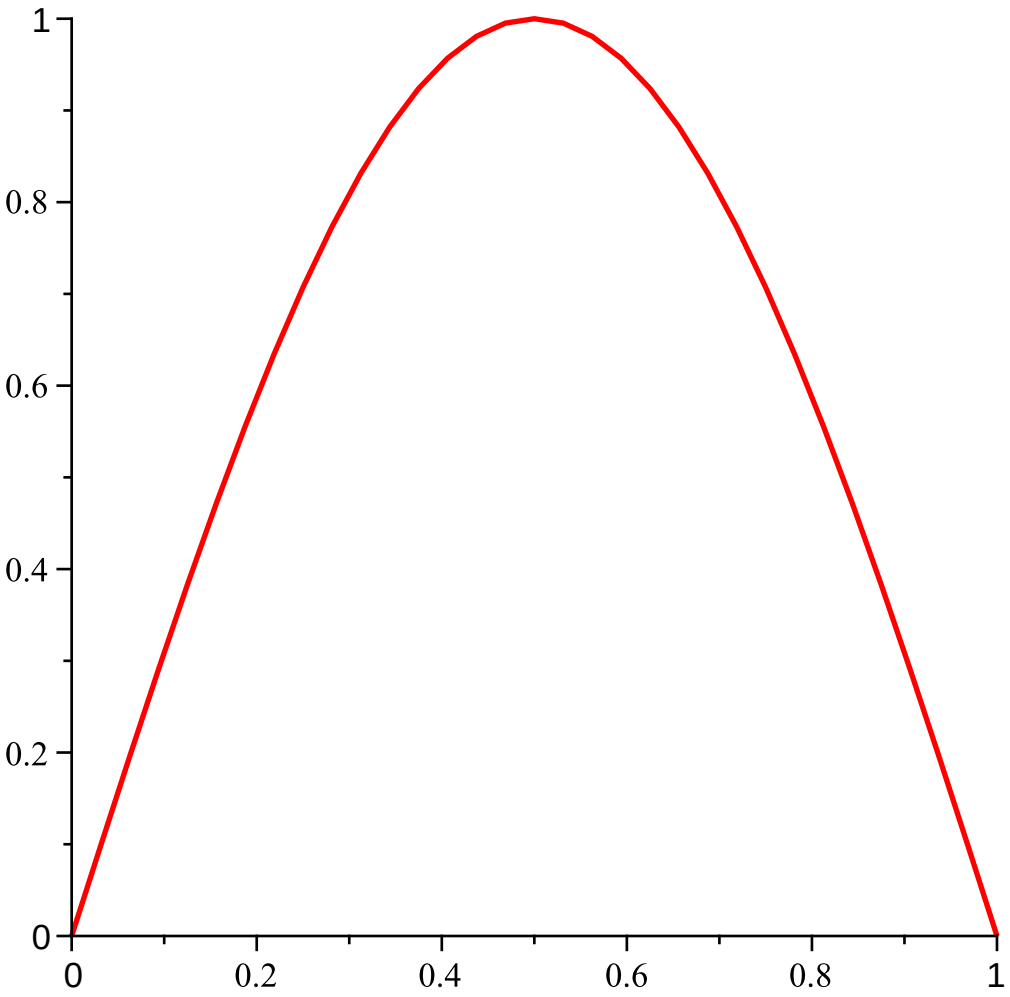
Wir legen die Zahl der Teilintervalle fest (hier beliebig; spaeter beim Mehrgitter muss es eine Zweierpotenz sein):

```
> m_ex := 32;
```

$$m\_ex := 32 \quad \textbf{(5.1)}$$

Wir waehlen einen der Eigenvektoren als Startloesung x^0, was auch der Startfehler e^0 ist...

```
> e0 := eigv(m_ex,1):
> bild(e0,[thickness=2]);
```



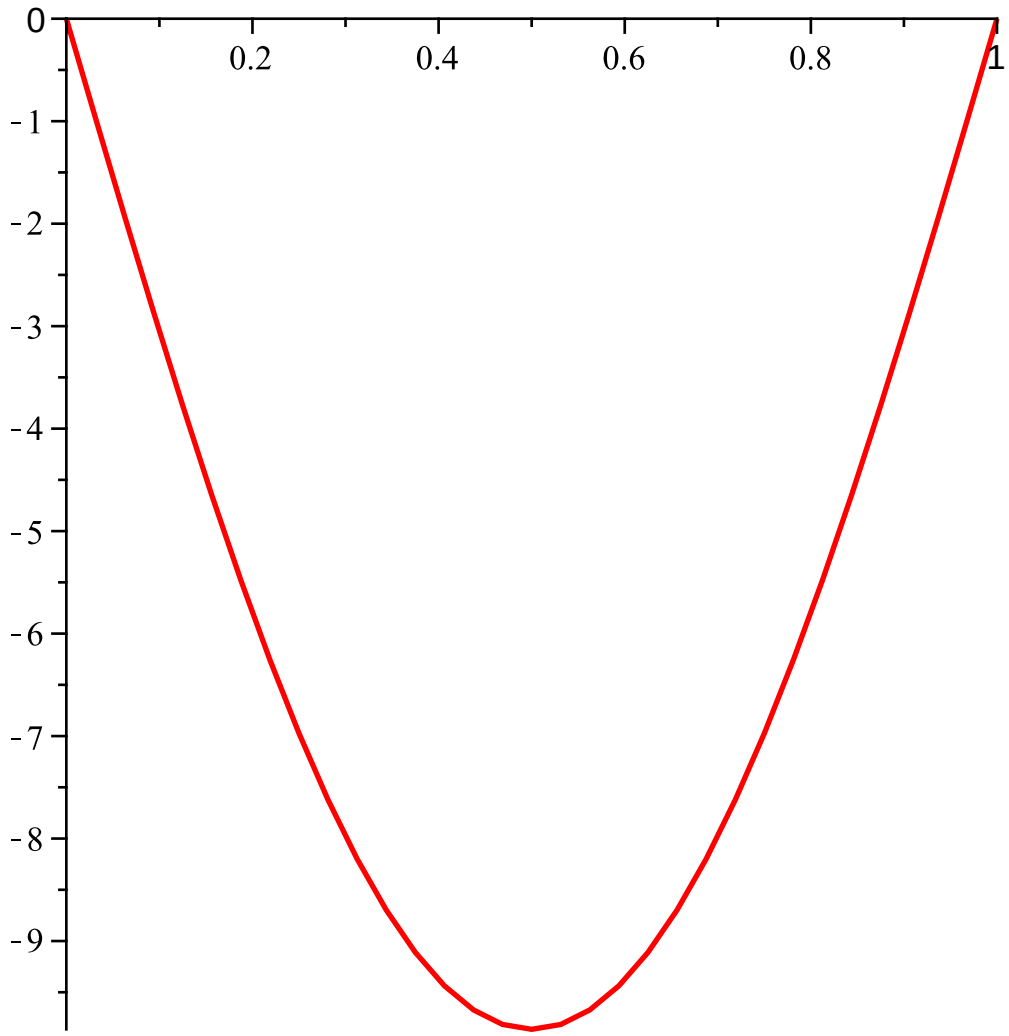
... weil wir als rechte Seite 0 waehlen:

```
> b0 := Vector(Dimension(e0),0);
```

$$b0 := \begin{bmatrix} 1 .. 31\ Vector_{column} \\ Data\ Type: anything \\ Storage: rectangular \\ Order: Fortran\_order \end{bmatrix} \quad \textbf{(5.2)}$$

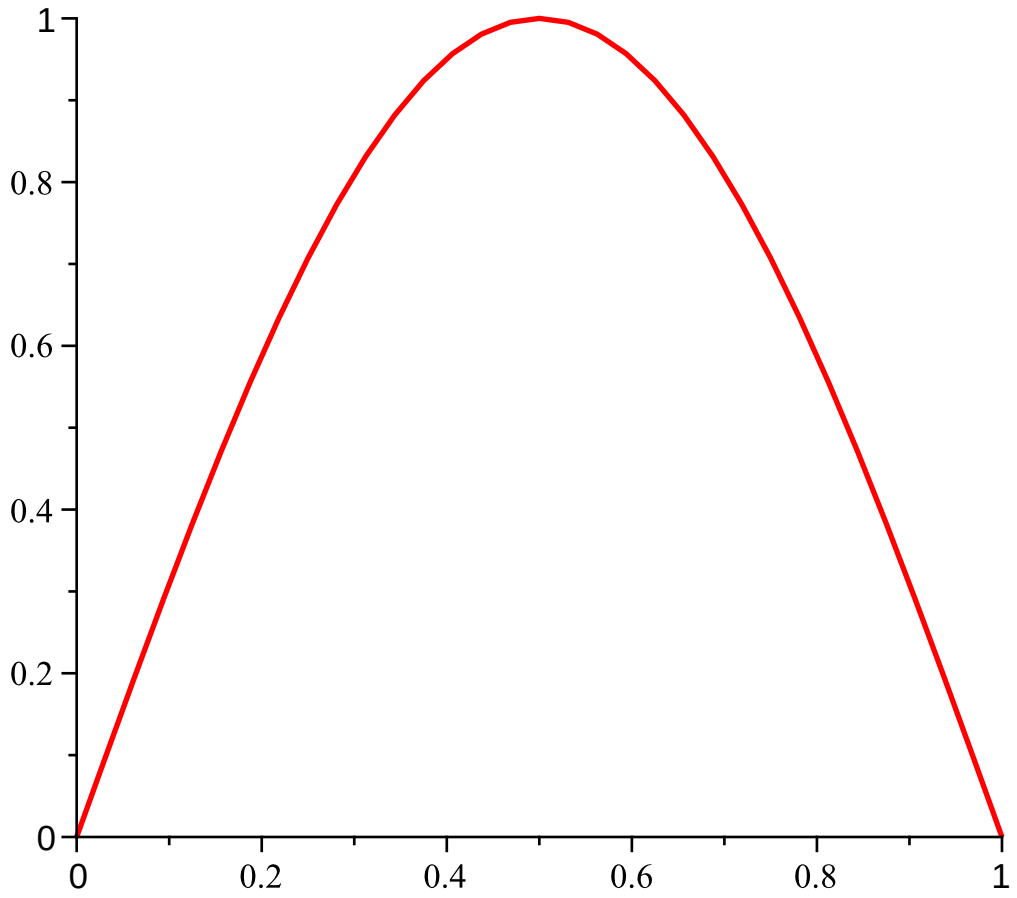
So sieht das Residuum am Anfang aus:

```
> bild(residuum(e0, b0),[thickness=2]);
```



Und das ist das Ergebnis der ersten 10 Jacobi-Iterationen (also x^0 bis x^{10}):

```
> bilder([seq((jacobi@@j)(e0, b0, 1)[1],j=0..10)]);
```



Diesen Prozess automatisieren wir: diese Prozedur erzeigt ein Bild mit der Startloesung e^0 und den ersten **nj** Iterierten x^j.

```
> konvbild := proc(e0::Vector, alpha::numeric, nj::posint)
  local x, s, j, b_null;
    x := e0;
    s := x;
    b_null := Vector(Dimension(e0),0);
    for j from 1 to nj do
      x := jacobi(x, b_null, alpha)[1];
      s := s, x;
    end do;
    bilder([s])
  end;
```

$$konvbild := \mathbf{proc}(e0::Vector, \text{alpha}::numeric, nj::posint) \quad \textbf{(5.3)}$$

$$\mathbf{local}\ x, s, j, b\_null;$$

$$x := e0;$$

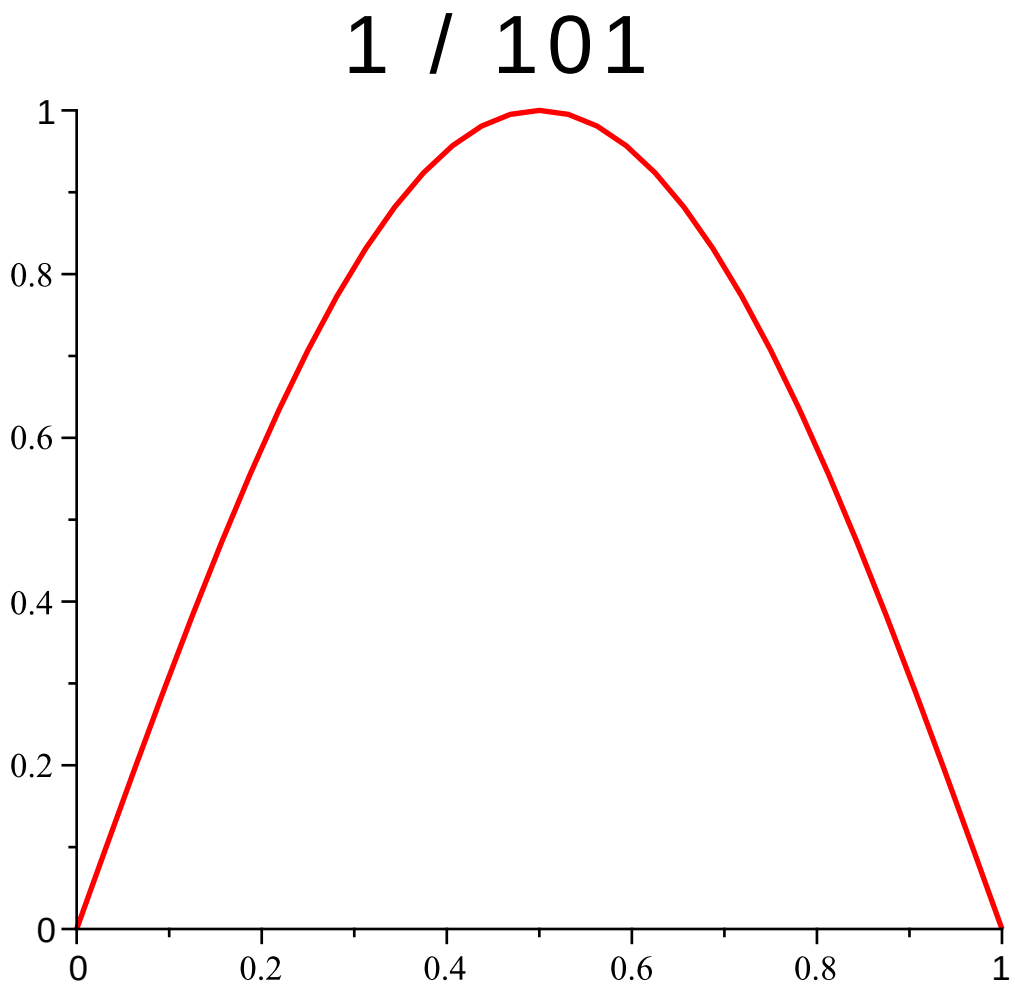
$$s := x;$$

$$b\_null := Vector(LinearAlgebra:\text{-}Dimension(e0), 0);$$

```
    for j to nj do x := jacobi(x, b_null, alpha)[1]; s := s, x end do;
    bilder([s])
end proc
```

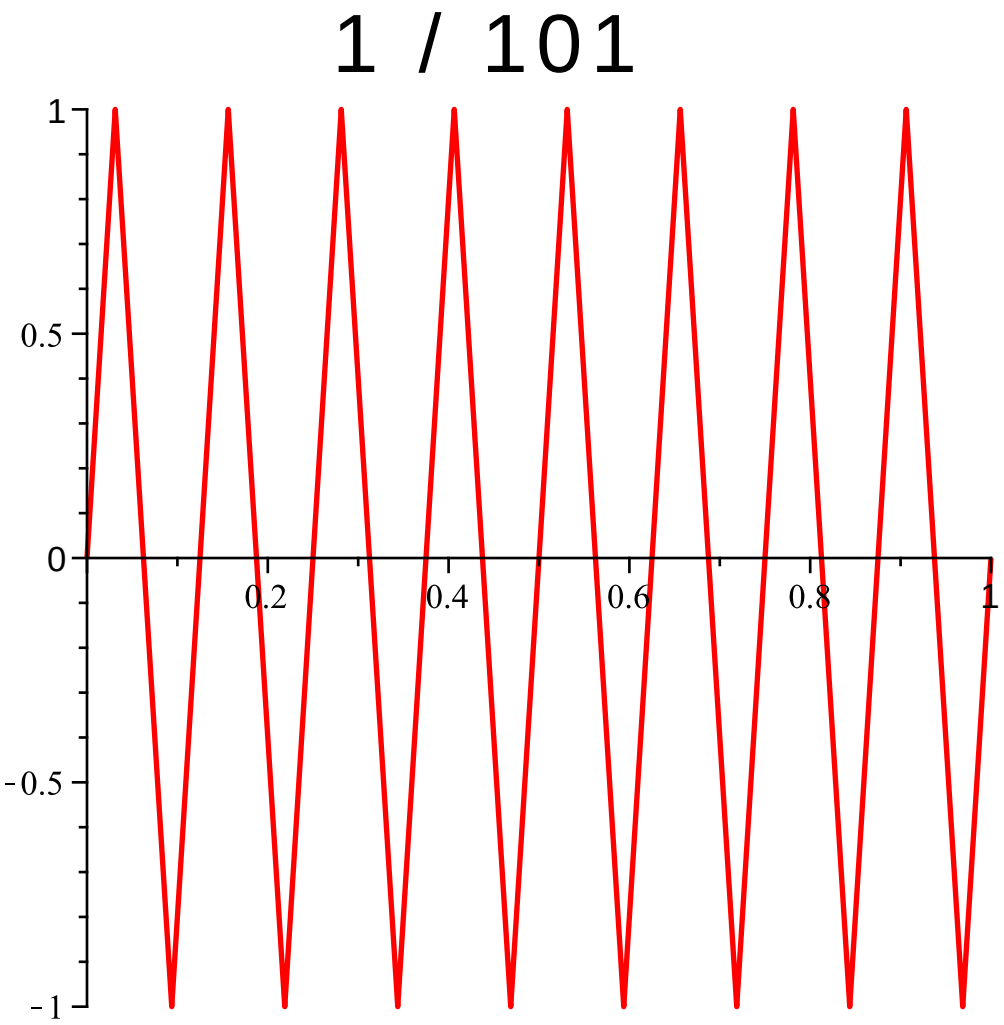
Nochmal mit dem Startvektor von vorhin:

```
> konvbild(e0,0.5,100);
```



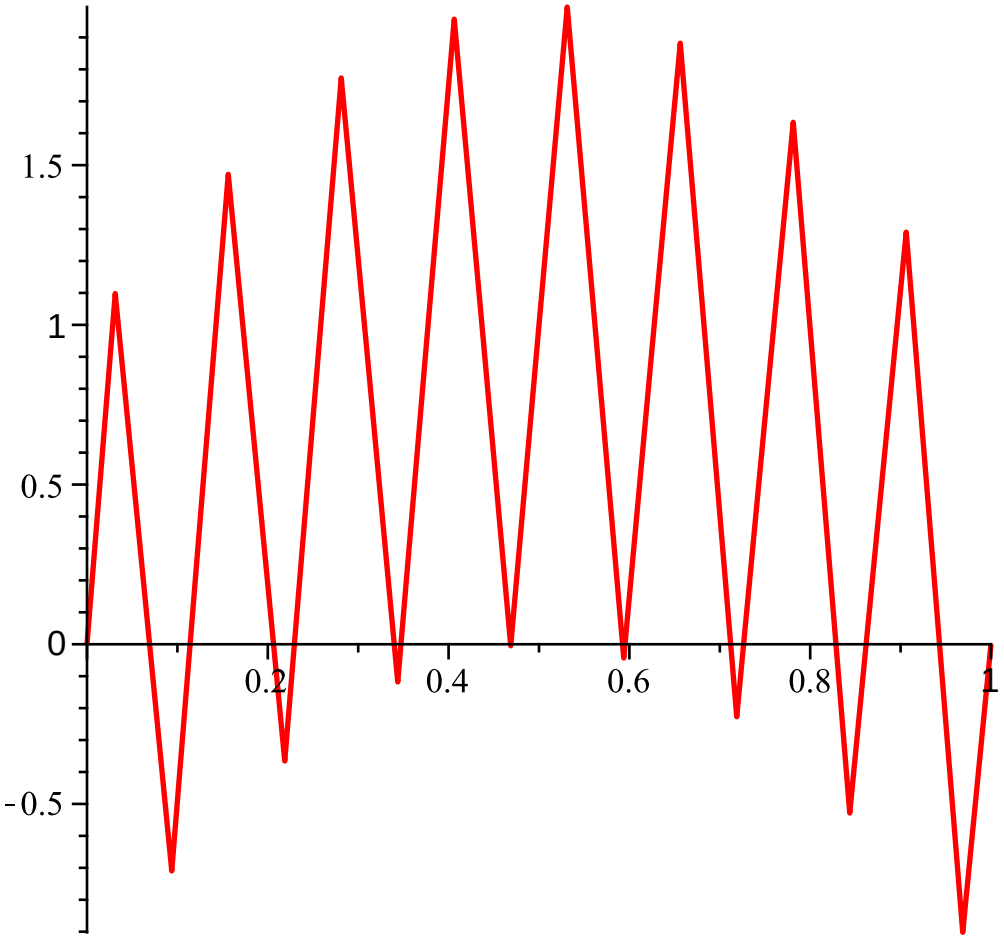
Ein zweites Beispiel aus der Mitte des Spektrums:

```
> e0neu := eigv(m_ex,round(m_ex/2)):
> konvbild(e0neu,0.5,100);
```



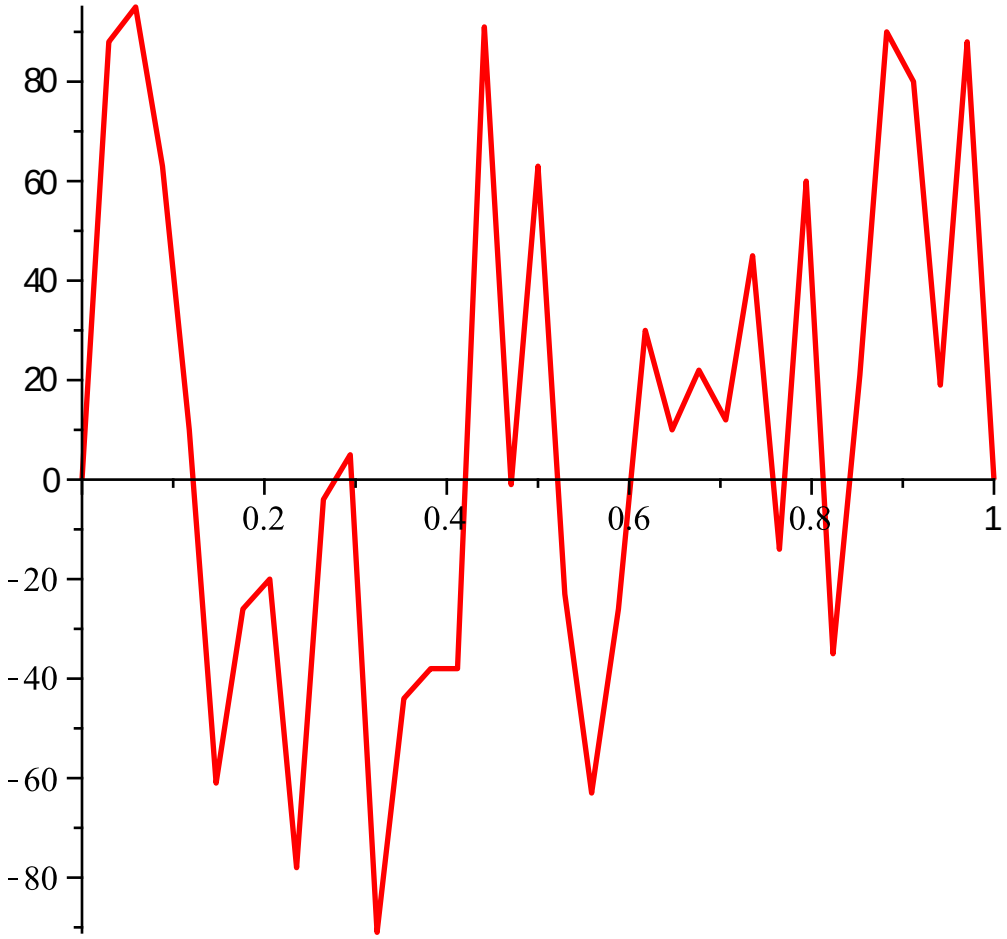
Und beide zusammen:

```
> konvbild(e0+e0neu,0.5,100);
```



Auch ein zufaelliger Startwert laeuft schnell in Richtung "renitenteste Fehleranteile":

```
> konvbild(RandomVector(m_ex+1),0.5,100);
```



Mehrgitterverfahren

Nun bauen wir uns ein einfaches Mehrgitterverfahren (das waere fuer dieses Modellproblem mit seiner Tridiagonalmatrix nicht noetig, aber man sieht daran das Prinzip)

Restriktion der rechten Seite auf das $2h$-Gitter

Hier triviale Restriktion: die Komponenten mit ungeradem Index wegwerfen. Der Parameter b_fein muss die Laenge $2^l - 1$ mit einem $1 < l$ haben, das Ergebnis hat die Laenge $2^{l-1} - 1$.

```
> restriktion := proc(b_fein::Vector)::Vector;
  local m_grob;
    m_grob := (Dimension(b_fein)+1)/2;
    Vector(m_grob-1, i -> b_fein[2*i])
  end;
```

$$restriktion := \mathbf{proc}(b\_fein::Vector)::Vector; \quad \textbf{(6.1.1)}$$

$$\mathbf{local}\ m\_grob;$$

$$m\_grob := 1/2 * LinearAlgebra:\text{-}Dimension(b\_fein) + 1/2;$$

```
    Vector(m_grob - 1, i→b_fein[2*i])
end proc
```

Prolongation der Korrektur auf das $\frac{h}{2}$-Gitter

Hier lineare Interploation: Komponenten mit geradem (Feingitter-)Index bekommen den entsprechenden Grobgitterwert, die ungeraden werden linear aus ihren Nachbarwerten interpoliert:

```
> prolongation := proc(x_grob::Vector)::Vector;
  local x_fein,m_grob,i;
    m_grob := Dimension(x_grob)+1;
    x_fein := Vector(2*m_grob-1, 0);
    for i from 1 to m_grob-1 do
      x_fein[2*i] := x_grob[i];
      x_fein[2*i-1] := x_fein[2*i-1]+x_grob[i]/2;
      x_fein[2*i+1] := x_fein[2*i+1]+x_grob[i]/2;
    end do;
    x_fein;
  end;
```

```
prolongation := proc(x_grob::Vector)::Vector;
    local x_fein, m_grob, i;
    m_grob := LinearAlgebra:-Dimension(x_grob) + 1;
    x_fein := Vector(2 * m_grob - 1, 0);
    for i to m_grob - 1 do
        x_fein[2 * i] := x_grob[i];
        x_fein[2 * i - 1] := x_fein[2 * i - 1] + 1 / 2 * x_grob[i];
        x_fein[2 * i + 1] := x_fein[2 * i + 1] + 1 / 2 * x_grob[i]
    end do;
    x_fein
end proc
```

(6.2.1)

Test fuer Prolongation und Restriktion

Ein Beispielvektor:

```
> x_ex := eigv(4,1);
```

$$x\_ex := \begin{bmatrix} 0.7071067810 \\ 1. \\ 0.7071067810 \end{bmatrix}$$

(6.3.1)

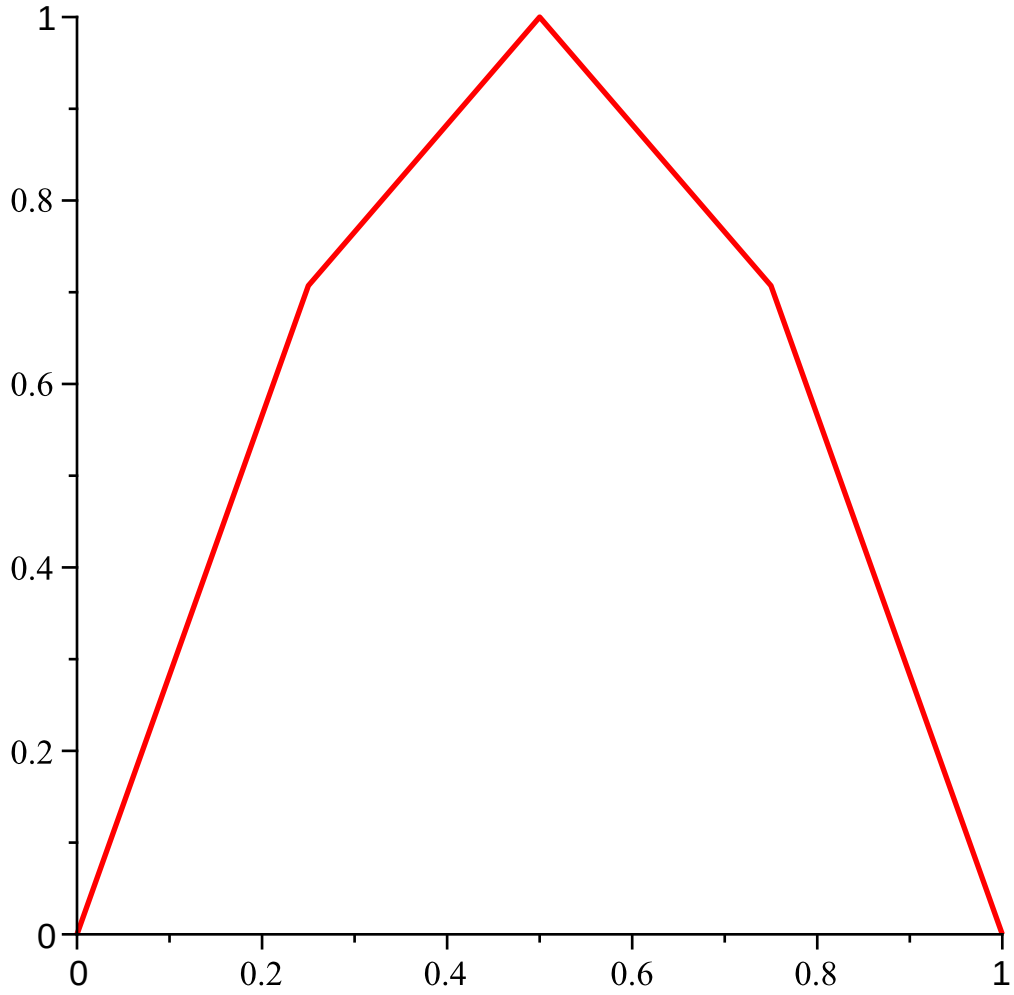
Linear interpoliert...

```
> prolongation(x_ex);
```

$$\begin{bmatrix} 0.3535533905 \\ 0.7071067810 \\ 0.8535533905 \\ 1. \\ 0.8535533905 \\ 0.7071067810 \\ 0.3535533905 \end{bmatrix} \quad \textbf{(6.3.2)}$$

... sieht man (spaetestens auf den zwieten Blick), dass man nichts sieht:

```
> bild(prolongation(x_ex),[thickness=2]);
```



Es gilt hier: Restriktion nach Prolobgation gibt die Identitaet:

```
> restriktion(prolongation(x_ex));
```

(6.3.3)

$$\begin{bmatrix} 0.7071067810 \\ 1. \\ 0.7071067810 \end{bmatrix} \quad \textbf{(6.3.3)}$$

Das Mehrgitterverfahren

Nun koennen wir alles zusammenbauen.
Die Parameter:

- **x**: Startvektor, also x^i
- **b**: rechte Seite
- **l**: Anzahl der Vergroeberungsschritte (die Laenge von **x** und **b** soll $2^k 2^l - 1$ mit einem $0 < k$ sein).
- **nu1**, **nu2**: zahl der Vor- bzw. Nachglaettungen

Das eigentliche Ergebnis ist der Vektor x^{i+1} wie bei der Prozedur **jacobi** werden die uebrigen Parameter (ausser **x**) einfach durchgereicht.

Als Glaetter wird das Jacobi-Verfahren mit $\alpha = \frac{1}{2}$ verwendet.

```
> mg := proc(x::Vector, b::Vector,
             l::nonnegint, nu1::nonnegint, nu2::nonnegint)::

       Vector,Vector,nonnegint,nonnegint,nonnegint;
  local i, xneu, xg, tmp;
    xneu:= (jacobi@@nu1)(x,b,0.5)[1];
    if l>0 then
      xneu := xneu
              + prolongation(
                  mg(Vector((Dimension(x)-1)/2,0),
                     restriktion(residuum(xneu,b)),
                     l-1, nu1, nu2)[1]
                );
    end if;
    (jacobi@@nu2)(xneu, b, 0.5)[1], b, l, nu1, nu2
  end;
```

$$mg := \textbf{proc}(x::Vector, b::Vector, l::nonnegint, nu1::nonnegint, nu2::nonnegint)::Vector, \quad \textbf{(6.4.1)}$$
$$Vector, nonnegint, nonnegint, nonnegint;$$
$$\textbf{local}\ i, xneu, xg, tmp;$$
$$xneu := `@@`(jacobi, nu1)(x, b, 0.5)[1];$$
$$\textbf{if}\ 0 < l\ \textbf{then}$$
$$xneu := xneu + prolongation(mg(Vector(1/2 * LinearAlgebra:-$$

```
        Dimension(x) - 1/2, 0), restriktion(residuum(xneu, b)), l - 1, nu1, nu2)[1])
    end if;
    `@@`(jacobi, nu2)(xneu, b, 0.5)[1], b, l, nu1, nu2
end proc
```

Beispielrechnungen

Fuer `l_max` Vergroeberungen...

```
> l_max := 5;
```

...brauchen wir mindestens soviele Teilintervalle:

```
> m_ex := 2^(l_max+1);
```

$$l\_max := 5$$

$$m\_ex := 64 \qquad \textbf{(6.5.1)}$$

Nun nehmen wir wieder (wie bei der Jacobi-Iteration) einen beliebigen Startwert und 0 als rechte Seite:

```
> e0 := eigv(m_ex,1):
> b0 := Vector(Dimension(e0),0):
```

Damit sind verschiedene Experimente moeglich:

- Mit l=0 bekommt man mit einem Aufruf einfach $v1 + v2$ Jacobi-Schritte ausgefuehrt.
- Mit zunehmendem l werden die niederfrequenten Anteile des Fehlers immer besser gedaempft...
- .. bis zum groesstmoeglichen l=l_max

Dazu lassen wir uns analog zum Jacobi-Verfahren Bilder malen:

```
> konvbild_mg := proc(e0::Vector,
                      l::nonnegint, nu1::nonnegint,
   nu2::nonnegint,
                      nj::posint)
   local x, s, j, b_null;
     x := e0;
     s := x;
     b_null := Vector(Dimension(e0),0);
     for j from 1 to nj do
       x := mg(x, b_null, l, nu1, nu2)[1];
       s := s, x;
     end do;
     bilder([s])
   end;
```

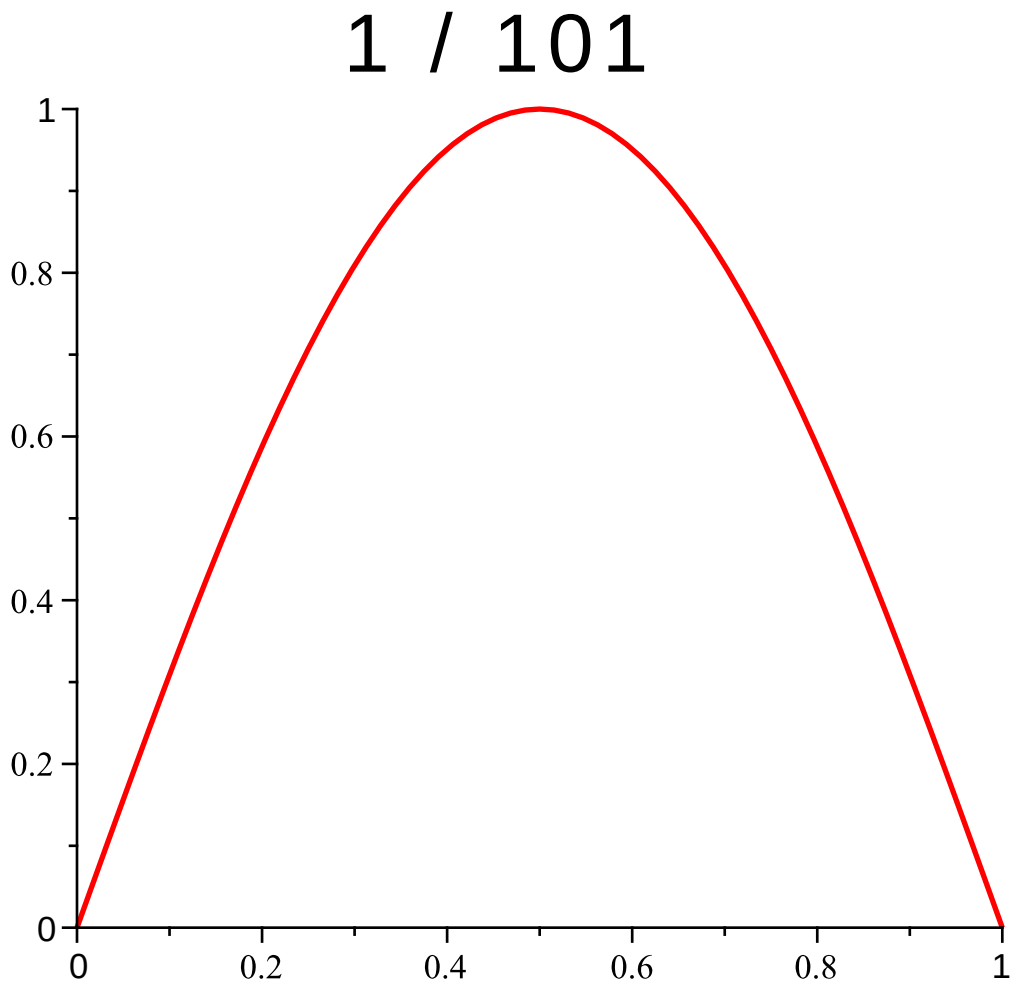
$konvbild\_mg := \mathbf{proc}(e0::Vector, l::nonnegint, nu1::nonnegint, nu2::nonnegint, nj::posint)$ **(6.5.2)**

```
    local x, s, j, b_null;
    x := e0;
    s := x;
```

```
    b_null := Vector(LinearAlgebra:-Dimension(e0), 0);
    for j to nj do x := mg(x, b_null, l, nu1, nu2)[1]; s := s, x end do;
    bilder([s])
end proc
```

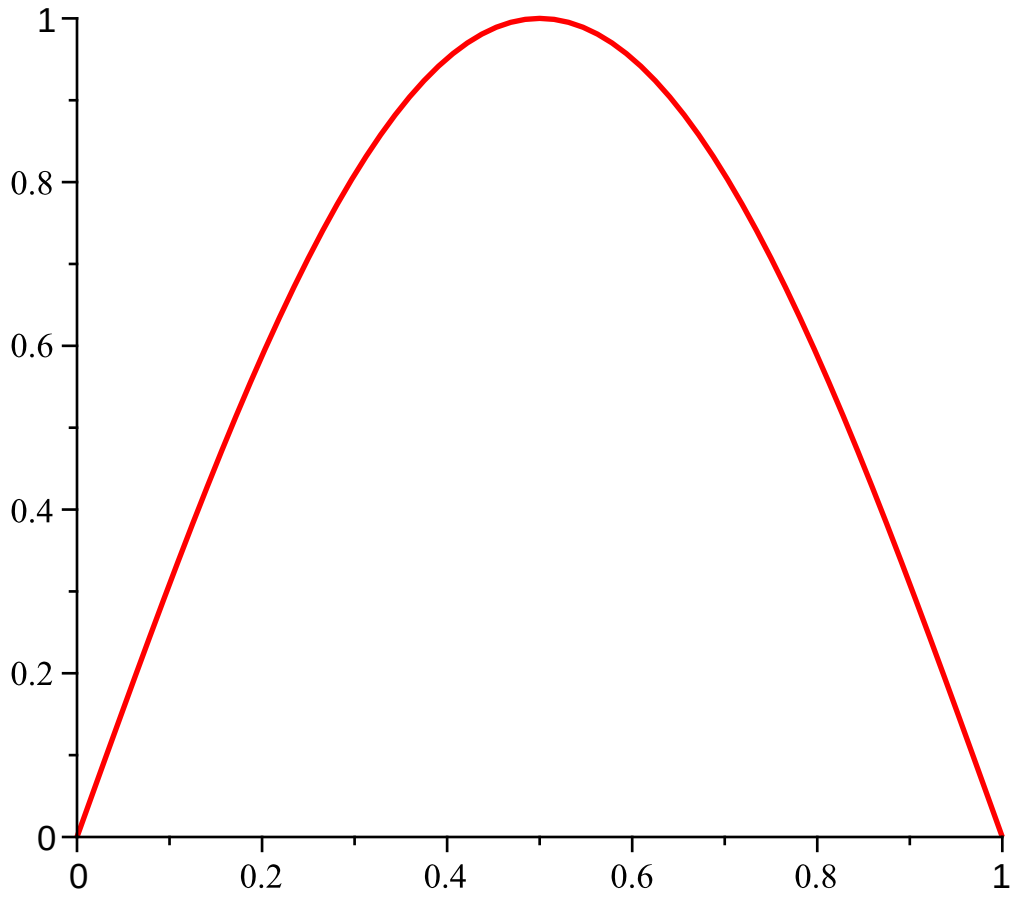
Der ganz normale Jacobi-Verfahren:

```
> konvbild_mg(e0, 0, 1, 0, 100);
```



Und die Mehrgitteridee im Einsatz (je 2 Vor- und Nachglaettungsschritte - unter Vernachlaessigung des Grobgitteraufwandes entspricht ein **mg** also 4 **jacobi**)

```
> konvbild_mg(e0, l_max, 2, 2, 10);
```


1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1