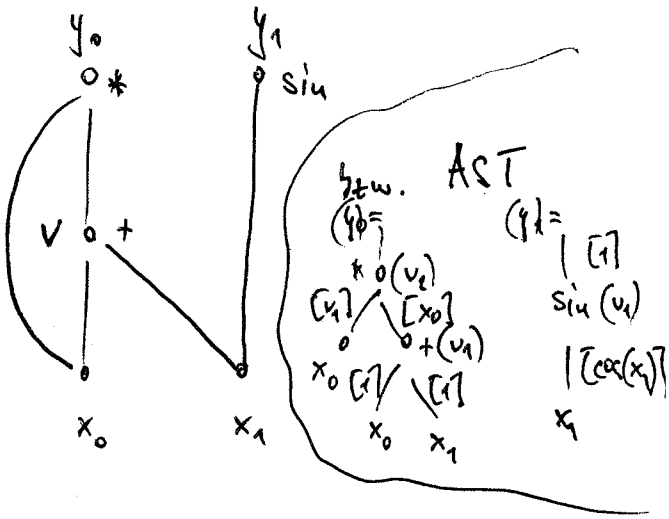
1. Vorlesung Compilerbau 27.6.2012

Numerische Simulationsprogramme $F: \mathbb{R}^n \to \mathbb{R}^m$, $y = F(x)$

$$y_0 = x_0 \cdot (x_0 + x_1)$$
$$y_1 = \sin(x_1)$$



$\Downarrow$

SAC

DAG

$$v = x_0 + x_1$$
$$y_0 = x_0 \cdot v$$
$$y_1 = \sin(x_1)$$

SAC: Single-Assignment Code

DAG: Directed Acyclic Graph

Was man gern wissen möchte: Wie sensitiv sind die Ausgaben bezüglich (sehr) kleiner Änderungen in den Werten der Eingaben?

$\Downarrow$

Ableitungen für Elementarfunktionen, z.B.

$$y = x_0 + x_1 \quad \Rightarrow \quad \frac{\partial y}{\partial x_0} = \frac{\partial y}{\partial x_1} = 1 \quad \Rightarrow \text{(lokaler Gradient)} \; \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

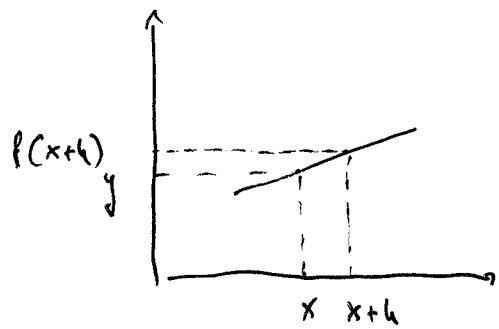
$$y = x_0 \cdot x_1 \quad \Rightarrow \quad \frac{\partial y}{\partial x_0} = x_1, \; \frac{\partial y}{\partial x_1} = x_0 \quad \Rightarrow \quad \begin{pmatrix} x_1 \\ x_0 \end{pmatrix}$$

$$y = \sin(x) \quad \Rightarrow \quad \frac{\partial y}{\partial x} = \cos(x)$$

$\vdots$ etc.

Einfluß von Parametern in den Eingaben auf Änderung des Werte in den Ausgaben

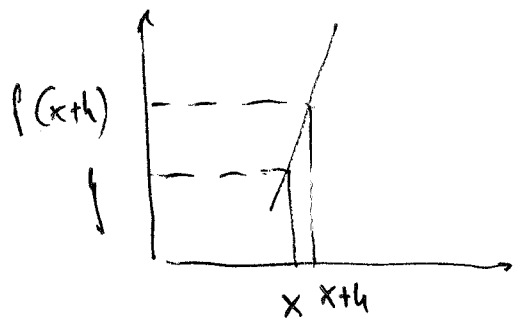
1. kleine positive Ableitung => geringe Vergrößerung



⇓

stabiler Punkt

2. große positive Ableitung => starke Vergrößerung

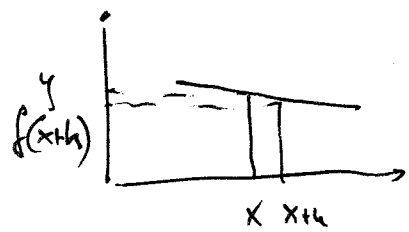


⇓

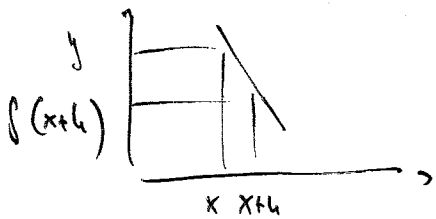
instabile Punkt

z.B. Aussage der Simulation fragwürdig für unsichere Eingabedaten, z.B. Ozean

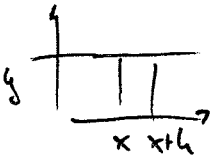
3. kleine negative Ableitung => geringe Verkleinerung



4. große negative Ableitung => starke Verkleinerung



5. Ableitung = 0 => kein Einfluß



Kennt man lokalen Gradienten alle Elementarfunktionen, so kann man linearisierten SAC erzeugen, indem man die SAC Anweisungen mit der Berechnung der lokalen partiellen Ableitungen erweitert, z.B.

$$\frac{\partial v}{\partial x_0} = 1$$

$$\frac{\partial v}{\partial x_1} = 1$$

$$v = x_0 \cdot x_1$$

$$\frac{\partial y_0}{\partial x_0} = v$$

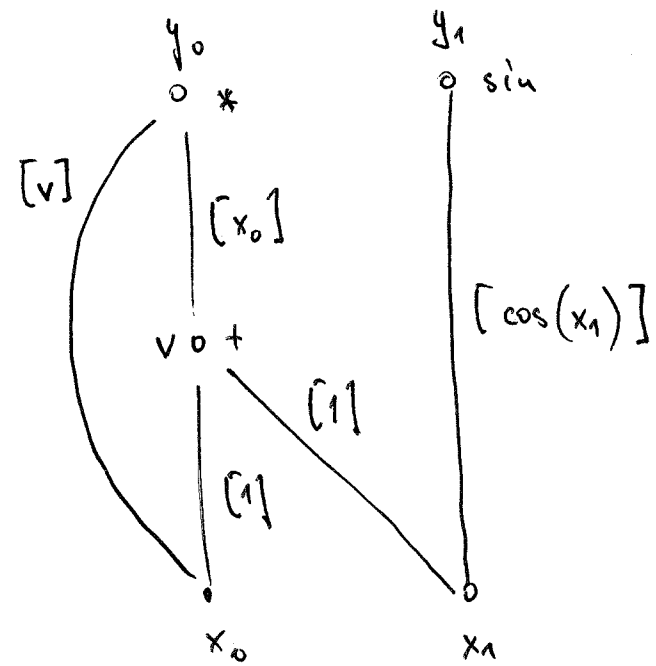
$$\frac{\partial y_0}{\partial v} = x_0$$

$$y_0 = x_0 \cdot v$$

$$\frac{\partial y_1}{\partial x_1} = \cos(x_1)$$

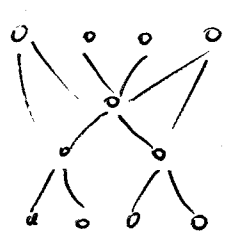
$$y_1 = \sin(x_1)$$

Linearisierte DAG $\Rightarrow$ bzw AST



Aus AST wird DAG, wenn man zu compile-Zeit identische memory locations erkennen kann (i.A. nicht entscheidbar)

Übrigens: Im STCE Logo (des KaFG 12) sieht man so einen DAG, der auch noch ganz besondere inhärente Eigenschaften hat:



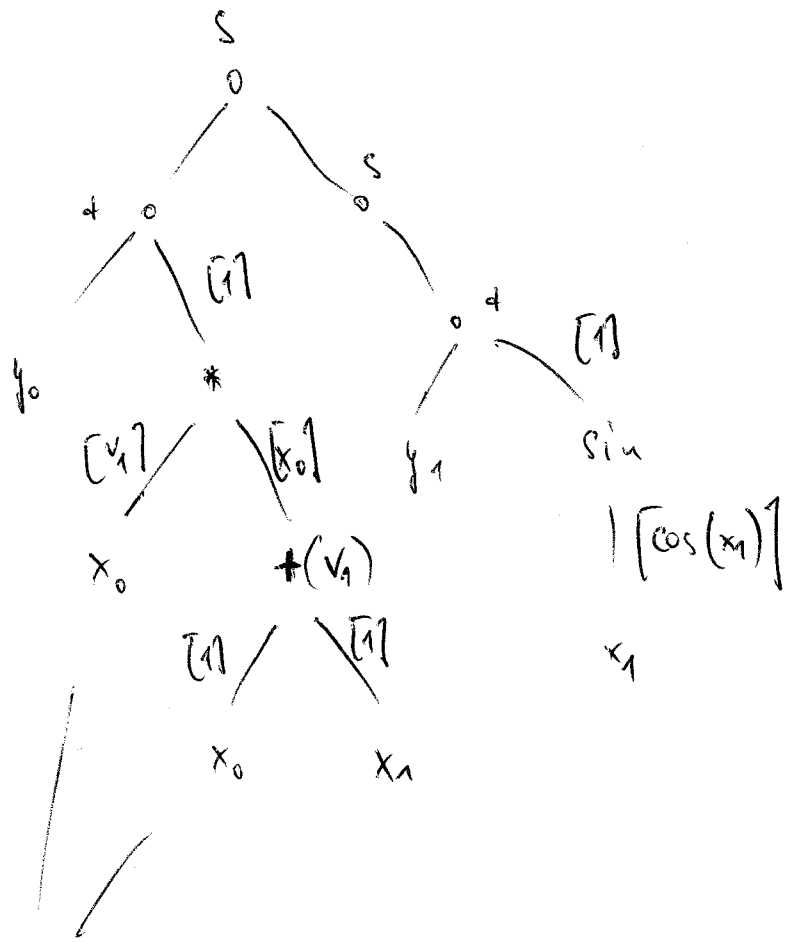
→ Vgl.: Combinatorial Problems in Scientific Computing

Abstrakte Syntaxbäume

```
S : aS
  | a

A :: V = E ;

E :: ...
```



Gleichheit im Allgemeinen
nicht entscheidbar

Die Matrix aller partiellen Ableitungen der Ausgaben bzgl. der Eingaben kann dann mit der Kettenregel wie folgt berechnet werden:

Jacobimatrix

$$\nabla F(x) = A = (a_{ji}) = \sum_{[x_i \to y_j]} \prod_{c_{ek} \in [x_i \to y_j]} c_{ek}$$

d.h. man bildet Produkte entlang der Pfade und addiert Werte paralleler Pfade, z.B.

$$\nabla F(x) = \begin{bmatrix} v + x_0 & x_0 \\ 0 & \cos(x_1) \end{bmatrix} = \begin{bmatrix} 2x_0 + x_1 & x_0 \\ 0 & \cos(x_1) \end{bmatrix}$$

Wir wollen eine semantische Quellcode Transformation betrachten, die aus einem gegebenen numerischen Programm für $y = F(x)$ eines generiert mit dessen Hilfe Produkte der Jacobimatrix mit Vektoren der Länge n berechnet werden können.
(zusätzlich zum Funktionswert)

$\Longrightarrow$ Tangenten-linearer Code

$$\begin{aligned} \dot{y} &= \nabla F(x) \cdot \dot{x} \\ y &= F(x) \end{aligned} \qquad (1)$$

Sei $y = F(x)$ implementiert als

```
void f( int n, double * x, int m, double * y ) {
        ⋮
}
```

Dann wird der tangenten-lineare Code (die tangenten-lineare Funktion / Routine) folgende Signatur haben:

```
void df ( int n, double * x, double * dx,
          int m, double * y, double * dy ) {
        ⋮
}
```

wobei $y = F(x)$

$dy = \nabla F(x) \cdot dx$

Bemerke: Der tangenten-lineare Code berechnet <u>nicht</u> die Jacobimatrix, sondern lediglich Produkte dieser mit einem Vektor ohne sie explizit zu berechnen (Matrix-frei)

Was kann ich mit tangenten-linearem Code anfangen?

1. Indem ich $\left[\dot{x} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow \text{i-te Stelle} \right]$ setze
   kann ich die i-te Spalte der Jacobimatrix berechnen, d.h. ich kann quantifizieren, wie stark alle Ausgaben auf Perturbationen in x_i reagieren

2. Indem ich $\dot{x}$ über die kanonischen Einheitsvektoren im $\mathbb{R}^n$ laufen lasse kann ich die gesamte Jacobimatrix berechnen

Wie sieht tangenten-linearer Code aus?

Gleichung (1) wird ganz einfach auf jede Zuweisung im SAC angewandt, z.B.

$$
\begin{aligned}
dv &= dx_0 + dx_1 \\
v &= x_0 + x_1 \\
dy_0 &= v \cdot dx_0 + x_0 \cdot dx_1 \\
y_0 &= x_0 \cdot v \\
dy_1 &= \cos(x_1) \cdot dx_1 \\
y_1 &= \sin(x_1)
\end{aligned}
$$

z.B. $y = F(x) = x_0 \cdot x_1$

$$\Downarrow$$

$$
\begin{aligned}
\dot{y} &= DF(x) \cdot \begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \end{pmatrix} \\
&= \begin{pmatrix} x_1 & x_0 \end{pmatrix} \begin{pmatrix} \dot{x}_0 \\ \dot{x}_1 \end{pmatrix} \\
&= x_1 \cdot \dot{x}_0 + x_0 \cdot \dot{x}_1
\end{aligned}
$$

Setze $dx_0 = 1$, $dx_1 = 0$ und verfolge tangenten-linearen Code:

$$dv = 1 + 0 = 1$$
$$v = x_0 + x_1$$
$$dy_0 = v \cdot 1 + x_0 \cdot 1 = v + x_0 = 2 \cdot x_0 + x_1$$
$$y_0 = x_0 \cdot v$$
$$dy_1 = \cos(x_1) \cdot dx_1 = 0$$
$$y_1 = \sin(x_1)$$

$\Downarrow$

$$\left.\begin{aligned} dy_0 &= 2 \cdot x_0 + x_1 \\ dy_1 &= 0 \end{aligned}\right\}$$ erste Spalte der Jacobimatrix

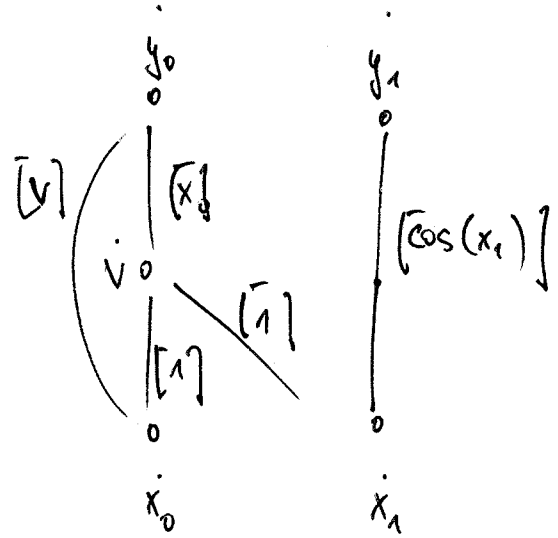
Analog: $dx_0 = 0$ und $dx_1 = 1$

Das funktioniert so schön aufgrund der Kettenregel!

Interpretation auf dem linearisierten DAG:

- Werte an den Knoten werden zu (Richtungs-) Ableitungen
  $v \rightarrow \dot{v}$
- Werte für $\dot{x}_i$ sind Eingaben
- Zwischen- und Ausgabewerte werden berechnet als Summe über alle Vorgänger des Produkts der Vorgängerwerte mit den entsprechenden Kantenwerten

Beispiel:

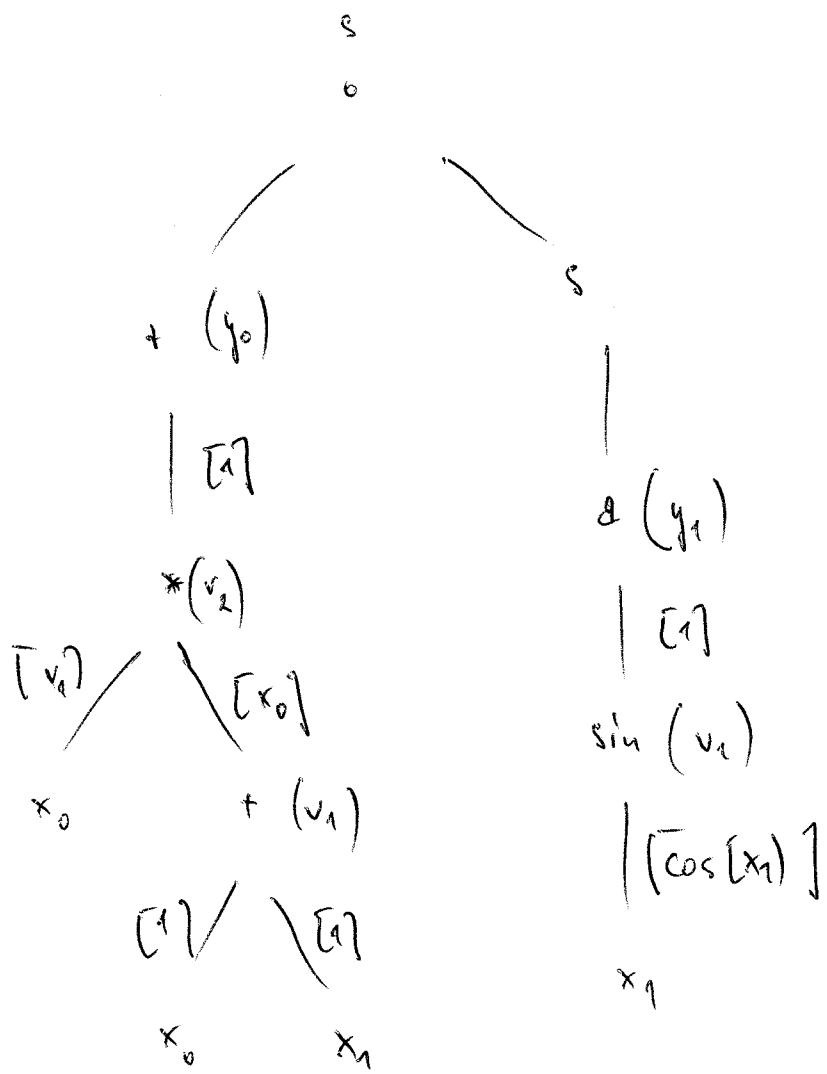


$$\dot{v} = 1 \cdot \dot{x}_0 + 1 \cdot \dot{x}_1$$
$$\dot{y}_0 = v \cdot \dot{x}_0 + x_0 \cdot \dot{v}$$
$$\dot{y}_1 = \cos(x_1) \cdot \dot{x}_1$$

bzw. auf linearisiertem AST

Funktionswerte müssen parallel dazu berechnet werden, da sie in die Berechnung der (Richtungs-) Ableitungen eingehen.

Mapping Tangent-Linear Code



DFS Mapping

$$\dot{v}_1 = \dot{x}_0 + \dot{x}_1$$
$$v_1 = x_0 + x_1$$
$$\dot{v}_2 = v_1 \dot{x}_0 + \dot{v}_1 \cdot x_0$$
$$v_2 = x_0 * v_1$$
$$\dot{y}_0 = \dot{v}_2$$
$$y_0 = v_2$$

etc. ...

Es gibt sich folgende C++-Implementierung

```
void f ( int n , double * x , int m , double * y ) {
    y[0] = x[0] * ( x[0] + x[1] );
    y[1] = sin(x[0]);
}
```

↓

```
void df ( int n , double * x , double * dx ,
          int m , double * y , double * dy ) {
    double v, dv;
    dv = dx[0] + dx[1];
    v = x[0] + x[1];
    dy[0] = v · dx[0] + x[0] · dv;
     y[0] = x[0] · v;
    dy[1] = cos(x[1]) · dx[1];
     y[1] = sin(x[1]);
}
```

Beispielsweise wird ein Testprogramm benötigt, um z.B. die gesamte Jacobi-Matrix zu berechnen.

→ Quellcodelisting

Wie kann ich überprüfen, ob mein Implementierter Code korrekt ist?

1. Gar nicht!
2. Ich kann eine Approximation der Jacobimatrix berechnen und damit vergleichen. ~> Finite Differenzen

Zur Erinnerung: Definition von Ableitungen

$$\frac{\partial f}{\partial x}(x_0) = \lim_{h \to 0} \frac{f(x_0+h) - f(x_0)}{h}$$

Falls auch der linksseitige / Grenzwert existiert und linksseitige und rechtsseitige denselben Wert haben.

→ Quellcodelisting

$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h}$$

Vorwärts differenzenquotient

Kann auf Gleitkommazahlen leider selbst ungenau werden. Funktioniert trotzdem oft recht gut.

```
#include<iostream>
#include<cmath>
using namespace std;

void f(int n, double* x, int m, double* y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}

void df(int n, double* x, double* dx,
        int m, double* y, double* dy) {
  double v,dv;
  dv=dx[0]+dx[1];
  v=x[0]+x[1];
  dy[0]=dx[0]*v+x[0]*dv;
  y[0]=x[0]*v;
  dy[1]=cos(x[1])*dx[1];
  y[1]=sin(x[1]);
}

int main() {
  const int n=2, m=2;
  double x[n], y[m];
  double dx[n], dy[m];
  for (int i=0;i<n;i++) { x[i]=.5; dx[i]=0.; }
  for (int i=0;i<n;i++) {
    dx[i]=1;
    df(n,x,dx,m,y,dy);
    cout << i+1 << ". Spalte der Jacobimatrix:" << endl;
    for (int j=0;j<m;j++)
      cout << dy[j] << endl;
    dx[i]=0;
  }
  return 0;
}
```

Template - Newton Code + Treiber

Siehe mdr!

```
#include<iostream>
#include<cmath>
using namespace std;

void f(int n, double* x, int m, double* y) {
  y[0]=x[0]*(x[0]+x[1]);
  y[1]=sin(x[1]);
}

int main() {
  const int n=2, m=2;
  double x[n], y[m];
  const double h=1e-8;
  double xph[n], yph[m];
  for (int i=0;i<n;i++) { x[i]=xph[i]=.5; }
  f(n,x,m,y);
  for (int i=0;i<n;i++) {
    xph[i]+=h;
    f(n,xph,m,yph);
    cout << i+1 << ". Spalte der Jacobimatrix:" << endl;
    for (int j=0;j<m;j++)
      cout << (yph[j]-y[j])/h << endl;
    xph[i]-=h;
  }
  return 0;
}
```

Approximation der Jacobimatrix mittels Finite Differenzen

siehe auch: Rückwärts- und Zentrale Differenzenquotienten

Weitere Anwendungen:

Finanzwesen

Flugzeugbau

Automobilbau

⋮

Optimierung

Template-Library Code can automatisch generiert werden,
z.B. dcc 0.9:

1. Runterladen von www.stce.rwth-aachen.de/TheArt.html
2. unpack tar -xzvf dcc-0.9.tar.gz
3. cd dcc-0.9
4. ./configure --prefix=[Wo der Compiler gestellt werden soll]
5. make
6. make install
7. Compiler dcc in [Wo der Compiler gestellt werden soll]/bin
8. void f(...) in f.cpp
9. ~/.../bin/dcc f.cpp 1 1 (template-library Code 1. Ordnung)
   => tl-f.cpp mit template-library funktion tl-f(...)
10. in Treiber: #include "tl-f.cpp" + Änderung des Namens der template-library funktion df -> tl-f
11. g++ main.cpp
12. ./a.out

By the way: Das ganze funktioniert auch auf nicht-straight-line Code, z.B.

```
void f (int n, double * x, double & y)
{
  int i = 0;
  y = 0
  while (i<n) {
      y = y + x[i] * x[i];
      i = i+1;
  }
  y = y * y;
}
```

Live Demo:
- tangentiell-linearer Code + Treiber
- finite Differenzen
- dco 0.9

... und auch auf interprozeduralem Code

↗ Vorlesung: Computationelle Differentiation

```
#include<iostream>
#include<cstdlib>
#include<cmath>
using namespace std;

void f(int n, double *x, double &y) {
  int i=0;
  while (i<n) {
    if (i==0) { y=x[i]*x[i]; }
    else { y=y+x[i]*x[i]; }
    i=i+1;
  }
  y=y*y;
}

void df(int n, double *x, double* dx, double &y, double &dy) {
  int i=0;
  while (i<n) {
    if (i==0) {
      dy=2*x[i]*dx[i];
      y=x[i]*x[i];
    }
    else {
      double v, dv;
      dv=2*x[i]*dx[i];
      v=x[i]*x[i];
      dy=dy+dv;
      y=y+v;
    }
    i=i+1;
  }
  dy=2*y*dy;
  y=y*y;
}

void fg(int n, double *x, double &y, double *g) {
  double *dx=new double[n], dy;
  for (int i=0;i<n;i++) dx[i]=0;
  for (int i=0;i<n;i++) {
    dx[i]=1;
    df(n,x,dx,y,dy);
    dx[i]=0;
    g[i]=dy;
  }
  delete [] dx;
}

int main(int argc, char* argv[]) {
  int n=atoi(argv[1]);
  double *x=new double[n];
  double y;
  double *g=new double[n];
  for (int i=0;i<n;i++) x[i]=cos(i+ );
  fg(n,x,y,g);
  cout << "y=" << y << endl;
  for (int i=0;i<n;i++) cout << "g[" << i << "]=" << g[i] << endl;
  delete [] g; delete [] x;
  return 0;
}
```

```
#include<iostream>
#include<cstdlib>
#include<cmath>
using namespace std;

void f(int n, double *x, double &y) {
  int i=0;
  while (i<n) {
    if (i==0) { y=x[i]*x[i]; }
    else { y=y+x[i]*x[i]; }
    i=i+1;
  }
  y=y*y;
}

void fg_ffd(int n, double *x, double &y, double *g) {
  const double h=1e-8;
  double *x_ph=new double[n], y_ph;
  f(n,x,y);
  for (int i=0;i<n;i++) x_ph[i]=x[i];
  for (int i=0;i<n;i++) {
    x_ph[i]+=h;
    f(n,x_ph,y_ph);
    x_ph[i]-=h;
    g[i]=(y_ph-y)/h;
  }
  delete [] x_ph;
}

int main(int argc, char* argv[]) {
  int n=atoi(argv[1]);
  double *x=new double[n];
  double y;
  double *g=new double[n];
  for (int i=0;i<n;i++) x[i]=cos(i+1);
  fg_ffd(n,x,y,g);
  cout << "y=" << y << endl;
  for (int i=0;i<n;i++) cout << "g[" << i << "]=" << g[i] << endl;
  delete [] g; delete [] x;
  return 0;
}
```

Finite Differenzen