Einführung in die Programmierung mit C++

Modern Family: Sensitivitätsanalyse

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

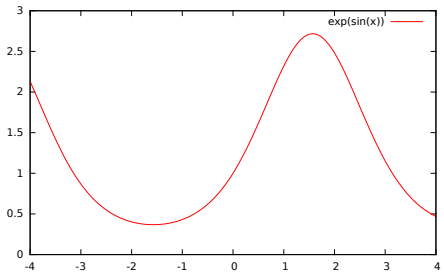
RWTH Aachen

Inhalt

Inhalt

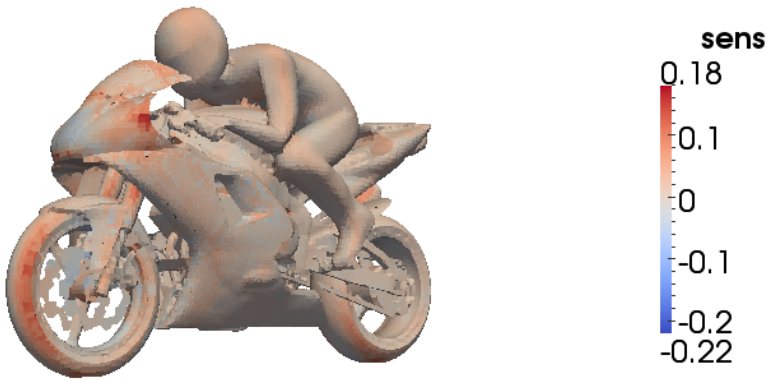
Wir wollen wissen, wie sensitiv das Resultat einer differenzierbaren Funktion $f : \boldsymbol{R} \to \boldsymbol{R}$, $y = f(x)$, an einer Stelle $x^* \in \boldsymbol{R}$ bezüglich (infinitesimal) kleiner Variationen (Fehler) in x^* ist, d.h. wir suchen die erste Ableitung $\frac{df}{dx}(x^*)$.

z.B. $y = e^{\sin(x)}$, $x \in \boldsymbol{R}$




```
1 float f(float x) {
2     return exp(sin(x));
3 }
```

z.B. Formoptimierung



Klassik: $y = p \cdot x \Rightarrow$ Einfluß von Fehlern in p verstärken sich mit wachsendem x.

Numerische Differentiation

Beispiel: $y = f(x) = 3x^2$

- symbolisch

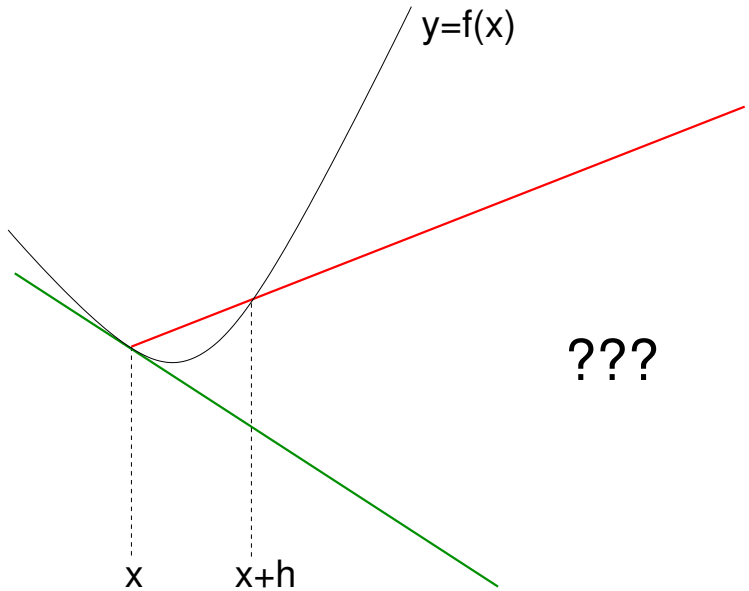
$$f' = \frac{\partial f}{\partial x} = 6x$$

- numerisch (Vorwärtsdifferenzen)

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

- numerisch (zentrale Differenzen)

$$f' = \frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2 \cdot h}$$



Man kann zeigen, dass zentrale Differenzen für $h \to 0$ eine bessere Approximation der ersten Ableitung liefert.

Numerische Differentiation

Beispiel: $y = f(x) = 3x^2$

Auf der Suche nach dem besten Wert für h in doppelter Genauigkeit ...

```
1  #include<iostream>
2
3  double f(double x) { return 3*x*x; }
4
5  int main() {
6    double x=1;
7    for (double h=1;h>1e-20;h/=10) {
8      std::cout << x << ", "
9                << f(x+h) << ", "
10               << f(x-h) << ", "
11               << f(x+h)-f(x-h) << ", "
12               << h << ", "
13               << (f(x+h)-f(x-h))/(2*h)
14               << std::endl;
15   }
16 }
```

```
1, 12, 0, 12, 1, 6
1, 3.63, 2.43, 1.2, 0.1, 6
1, 3.0603, 2.9403, 0.12, 0.01, 6
1, 3.006, 2.994, 0.012, 0.001, 6
1, 3.0006, 2.9994, 0.0012, 0.0001, 6
1, 3.00006, 2.99994, 0.00012, 1e-05, 6
1, 3.00001, 2.99999, 1.2e-05, 1e-06, 6
1, 3, 3, 1.2e-06, 1e-07, 6
1, 3, 3, 1.2e-07, 1e-08, 6
1, 3, 3, 1.2e-08, 1e-09, 6
1, 3, 3, 1.2e-09, 1e-10, 6
1, 3, 3, 1.2e-10, 1e-11, 6
1, 3, 3, 1.20002e-11, 1e-12, 6.00009
1, 3, 3, 1.19993e-12, 1e-13, 5.99965
1, 3, 3, 1.20348e-13, 1e-14, 6.01741
1, 3, 3, 1.33227e-14, 1e-15, 6.66134
1, 3, 3, 8.88178e-16, 1e-16, 4.44089
1, 3, 3, 0, 1e-17, 0
```

Numerische Differentiation

Beispiel: $y = f(x) = 3x^2$

Den besten Kompromiss zwischen numerischer Stabilität und Genauigkeit der Approximation erhält man typischerweise durch Perturbation (Störung) der Hälfte der Mantisse der Eingabeparameter, z.B.

```
1  template<typename T>
2  T f(T x) { return 3*x*x; }
3
4  template<typename T>
5  T dfdx(T x) {
6    const T eps=std::numeric_limits<T>::epsilon();
7    const T h= (x==0) ? sqrt(eps) : sqrt(eps)*fabs(x);
8    return (f(x+h)-f(x-h))/(2*h);
9  }
```

Die systemabhängige Konstante std::numeric_limits<**float**>::epsilon() bezeichnet die kleinste positive Gleitkommazahl einfacher Genauigkeit, deren gerundete Summe mit 1.0 größer als 1.0 ist, d.h. 1.0 + std::numeric_limits<**float**>::epsilon() > 1.0. Das Produkt von deren Quadratwurzel mit dem nichtverschwindenden Betrag des Eingabeparameters resultiert in einer Perturbation im Zentrum der Mantisse.

Inhalt

Für das gegebene Modell $y = p \cdot x$ interessieren wir uns für die Sensitivität des mittels

```
1 T estimate(T x1, T y1, T x2, T y2) {
2   T d=(pow(x1,2)+pow(x2,2));
3   assert(d!=0);
4   return (x1*y1+x2*y2)/d;
5 }
```

geschätzten Parameters $p \in \boldsymbol{R}$ bezüglich (infinitesimaler) Variationen der Eingabedaten, d.h. wir suchen Approximationen der vier ersten Ableitungen

$$\frac{dp}{dx_1}, \quad \frac{dp}{dx_2}, \quad \frac{dp}{dy_1}, \quad \text{und} \quad \frac{dp}{dy_2}$$

als (zentrale) finite Differenzenquotienten.

Modern Family

Sensitivitätsanalyse: Lösung


```
1  using T=double;
2
3  T estimate(T x1, T y1, T x2, T y2) { ... }
4
5  int main() {
6    using namespace std;
7    T x1=0,x2=0,y1=0,y2=0,eps=std::numeric_limits<T>::epsilon(),h;
8    cout << "x1="; cin >> x1; cout << "y1="; cin >> y1;
9    cout << "x2="; cin >> x2; cout << "y2="; cin >> y2;
10   h= (x1==0) ? sqrt(eps) : sqrt(eps)*fabs(x1); // dp/dx1
11   cout << (estimate(x1+h,y1,x2,y2)-estimate(x1-h,y1,x2,y2))/(2*h) << endl;
12   h= (x2==0) ? sqrt(eps) : sqrt(eps)*fabs(x2); // dp/dx2
13   cout << (estimate(x1,y1,x2+h,y2)-estimate(x1,y1,x2-h,y2))/(2*h) << endl;
14   h= (y1==0) ? sqrt(eps) : sqrt(eps)*fabs(y1); // dp/dy1
15   cout << (estimate(x1,y1+h,x2,y2)-estimate(x1,y1-h,x2,y2))/(2*h) << endl;
16   h= (y2==0) ? sqrt(eps) : sqrt(eps)*fabs(y2); // dp/dy2
17   cout << (estimate(x1,y1,x2,y2+h)-estimate(x1,y1,x2,y2-h))/(2*h) << endl;
18   return 0;
19 }
```

Aus

$$p = \frac{x_1 \cdot y_1 + x_2 \cdot y_2}{x_1^2 + x_2^2}$$

folgt

$$\frac{dp}{dx_i} = \frac{y_i}{x_1^2 + x_2^2} - \frac{2 \cdot x_i \cdot (x_1 \cdot y_1 + x_2 \cdot y_2)}{(x_1^2 + x_2^2)^2} \quad \text{und} \quad \frac{dp}{dy_i} = \frac{x_i}{x_1^2 + x_2^2}$$

für $i = 1, 2$ und korrekt approximiert für z.B.

$$\mathrm{x} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \ \mathrm{y} = \begin{pmatrix} 2 \\ 4 \end{pmatrix} \quad \Rightarrow \quad \frac{dp}{d\mathrm{x}} = \begin{pmatrix} -0.08 \\ -0.44 \end{pmatrix} \quad \text{und} \quad \frac{dp}{d\mathrm{y}} = \begin{pmatrix} 0.1 \\ 0.3 \end{pmatrix}$$

oder

$$\mathrm{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \ \mathrm{y} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \Rightarrow \quad \frac{dp}{d\mathrm{x}} = \begin{pmatrix} 0.28 \\ -0.04 \end{pmatrix} \quad \text{und} \quad \frac{dp}{d\mathrm{y}} = \begin{pmatrix} 0.2 \\ 0.4 \end{pmatrix} .$$

AUFWACHEN!

Validieren sie Korrektheit der Approximation.

Wach?

Validieren sie Korrektheit der Approximation.

Symbolische Differentiation ergibt

```
1  using T=double;
2
3  int main() {
4    using namespace std;
5    T x1=0,x2=0,y1=0,y2=0;
6    cout << "x1="; cin >> x1; cout << "y1="; cin >> y1;
7    cout << "x2="; cin >> x2; cout << "y2="; cin >> y2;
8    T aux1=x1*y1+x2*y2, aux2=pow(x1,2)+pow(x2,2), aux3=pow(aux2,2);
9    cout << "dp/dx1=" << y1/aux2-2*x1*aux1/aux3 << endl;
10   cout << "dp/dx2=" << y2/aux2-2*x2*aux1/aux3 << endl;
11   cout << "dp/dx1=" << x1/aux2 << endl;
12   cout << "dp/dx1=" << x2/aux2 << endl;
13   return 0;
14 }
```

Inhalt

Die Approxmation höherer Ableitungen mittels finiter Differenzenquotienten eignet sich hervorrragend zur Illustration der Grenzen der Gleitkommaarithmetik.

Für $y = e^x$ approximieren wird die Ableitungen bis zur vierten Ordnung, d.h.

$$\frac{d^i y}{dx^i}$$

für $i = 1, 2, 3, 4$. Dabei ergibt sich die $i + 1$-the Ableitung als Approximation der Ableitung der approximierten i-ten Ableitung für $i = 1, 2, 3$.

Numerische Differentiation Höherer Ordnung

Beispiel I

```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<cmath>
4  #include<limits>
5
6  template<typename T>
7  T f(T x) { return exp(x); }
8
9  template<typename T>
10 T dfdx(T x, T h) {
11   return (f(x+h)-f(x-h))/(2*h);
12 }
13
14 template<typename T>
15 T ddfdxx(T x, T h) {
16   return (dfdx(x+h,h)-dfdx(x-h,h))/(2*h);
17 }
18
19 template<typename T>
20 T dddfdxxx(T x, T h) {
```

Numerische Differentiation Höherer Ordnung

Beispiel II


```
21    return (ddfdxx(x+h,h)-ddfdxx(x-h,h))/(2*h);
22  }
23
24  template<typename T>
25  T ddddfdxxxx(T x, T h) {
26    return (dddfdxxx(x+h,h)-dddfdxxx(x-h,h))/(2*h);
27  }
28
29  int main() {
30    using namespace std;
31    using T=float;
32    T x=0;
33    for (T h=1;h>std::numeric_limits<T>::epsilon();h/=10)
34      std::cout << h << "\t"
35                << dfdx(x,h) << "\t"
36          << ddfdxx(x,h) << "\t"
37          << dddfdxxx(x,h) << "\t"
38          << ddddfdxxxx(x,h) << std::endl;
39    return 0;
40  }
```

Für $h = 10^i$, $i = 0, \ldots, -6$ wird folgende Ausgabe generiert.

```
1 1.1752 1.3811 1.62307 1.90743
0.1 1.00167 1.00334 1.00501 1.00669
0.01 1.00002 1.00002 0.998378 1.49012
0.001 1.00002 0.998408 −7.46551 3710.4
0.0001 1.00017 1.49012 −14901.2 −7.45058e+07
1e−05 1.00136 0 −7.45058e+06 3.72529e+11
1e−06 0.983477 0 7.45058e+09 3.72529e+15
```

Alle Ableitungen (2.-4. Spalte) sollten den Wert 1 haben. Der beste Kompromiss scheint $h \approx 10^{-1} \approx \sqrt{\sqrt{\sqrt{\sqrt{\epsilon}}}}$ zu sein.

AUFWACHEN!

Wiederholen sie die Fallstudie für $y = e^{\sin(x)}$ and der Stelle $x = 0$ und analysieren sie die Ergebnisse.

Wach?

1 0.94435 0.221346 −0.672866 −0.183808
0.1 0.999993 0.989987 −0.0393763 −3.01577
0.01 0.999996 1.00002 0.0223517 −4.47035
0.001 1.00002 0.998408 −7.46551 3710.4
0.0001 1.00017 1.49012 −14901.2 −7.45058e+07
1e−05 1.00136 0 −7.45058e+06 3.72529e+11
1e−06 0.983477 0 7.45058e+09 3.72529e+15

Zusammenfassung

Numerische Differentiation

- Motivation
- Finite Differenzen

Modern Family

- Sensitivitätsanalyse
- Finite Differenzen

Numerische Differentiation Höherer Ordnung