Winfried Auzinger
Klaus Felsenstein
Dirk Praetorius

SS 2010
(19. März 2010)

Übungsaufgaben zur VL Computermathematik

Serie 3

Die Aufgaben mit Stern (*) sind bis zur Übung in der kommenden Woche fix vorzubereiten und werden dort abgeprüft (Minimalerfordernis).

Kopieren Sie Ihre Worksheets auf Ihren Account auf `lva.student.tuwien.ac.at`. *Überprüfen Sie vor der Übung, ob Ihre Codes unter Maple 12 auf* `lva` *einwandfrei funktionieren. Es wird empfohlen, für jedes Beispiel im Verzeichnis* `serie03` *ein eigenes Worksheet mit dem Namen* `aufgabey-y.mw` *anzulegen. Alle zu verfassenden Prozeduren sind zu kommentieren und – je nach Angabe – mit verschiedenen Werten für die Parameter auszutesten.*

Bei Fragen zur Angabe, oder wenn der Tippfehlerteufel zuschlägt: $\longrightarrow$ *TUWIS++ Forum!*

Aufgabe 3.1*. Maple bietet viele Möglichkeiten zur grafischen Visualisierung, z.B. `plot` und `plot3d`, siehe auch die Packages `plots` und `plottools`.

Man präsentiere einen Überblick über die wichtigsten Befehle zur Visualisierung und die wesentlichen Optionen. Unter anderem zeige zeige man, wie man mit Hilfe von `plots[display]` [1] mehrere zuvor erzeugte plot-Datenstrukturen in einem gemeinsamen Bild darstellen kann.

Siehe insbesondere auch `animate` und `animate3d`!

Aufgabe 3.2*. Eine Tridiagonalmatrix[2]

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{pmatrix}$$

repräsentieren wir in naheliegender Weise als kompaktes `Array` der Dimension `(1..n,1..3)`. Jede Zeile des Arrays besteht aus 3 Elementen, die je einem a, b, c - Wert entsprechen. Ausgehend von dieser Darstellung schreibe man eine Maple-Prozedur `dettridiag(A)`, die eine derartige Struktur als Parameter übernimmt und iterativ die Determinante der betreffenden Matrix berechnet. (Die Iteration leitet man sich mit Hilfe des Laplace'schen Entwicklungssatzes her.)

Man verwende diese Prozedur auch dazu, um das charakteristische Polynom $p(\lambda) = \det(A - \lambda\, I)$ explizit als Polynom in der Variablen λ darzustellen. Siehe auch `? LinearAlgebra[Determinant]`, `? LinearAlgebra[CharacteristicPolynomial]`.

Aufgabe 3.3*. Man schreibe eine *rekursive* Maple-Prozedur `bsearch(e,l)`, die zu einer gegebenen, als *absteigend* sortiert angenommenen Liste `l` die Position angibt, an der das gesuchte Element `e` auftritt.

– Falls `e` nicht gefunden wird, ist `NULL` zurückzugeben.

– Mehrfaches Auftreten von `e` soll als unzulässig gelten: Abbruch mit Fehlermeldung (verwende `error`).

Es darf a priori vorausgesetzt werden, dass die Liste korrekt absteigend sortiert ist.

Vorgangsweise: 'Divide and conquer' mittels rekursiver Halbierung des Suchbereiches – verwandt zum Sort/Merge Algorithmus (vgl. VO III).

Die übergebenen Listenelemente müssen sinnvollerweise von verwandtem (d.h. vergleichbarem) Typ sein. Sie müssen aber nicht unbedingt vom Typ `numeric`, sondern können z.B. auch vom Typ `string` sein (Strings gelten als lexikografisch angeordnet, wie im Telefonbuch).

Beispiel:

```
> l := ["und","Müller","Kuh","ich","du"]:
> bsearch("Kuh",l);
                    3
```

Aufgabe 3.4*. In Maple gibt es indizierte Variablen, z.B. `a[0], a[1]` für a_0, a_1 (intern repräsentiert mittels tables; nicht zu verwechseln mit Elementen einer Liste, bei denen dieselbe Syntax verwendet wird). Man kann alles mögliche indizieren, z.B. auch Funktionen. Definieren Sie konkret die *Bernstein-Polynome* vom vorgegebenen festen Grad n,

$$B_{i,n}(t) := \binom{n}{i} t^i\,(1-t)^{n-i}$$

[1] Die Syntax *packagename[funktionsname]* aktiviert eine einzelne Funktion aus einem package. Dabei muss letzteres nicht mittels `with` komplett geladen werden.

[2] Das ist nur eine Übung. Die flexiblen *indexing functions* in Maple ersparen einem diese Arbeit in der Praxis. Es würde genügen, automatische Speicherung des $n \times n$-Arrays als tridiagonale Bandmatrix anzufordern (`? indexfcn`, `? band`).

als Funktionen mit dem Namen `B[i]`, $i = 0, 1, \ldots, n$. Diese Definition erfolgt individuell für jeden konkreten Wert des Index `i`.[3]

Alternativ dazu ist es praktischer und flexibler, eine derartige Deklaration in der Form

`B := (t,i) -> ...` bzw. gleich allgemeiner: `B := (t,i,n) -> ...` (auch `n` variabel)

durchzuführen; dies ist nur *eine einzige* Definition, und rein formal ist dadurch eine Funktion von 2 bzw. 3 Variablen erklärt. Das Problem dabei: Eigentlich sind die $B_{i,n}(t)$ ja Funktionen einer einzigen Variablen t. Was ist also zu tun, wenn man eine so definierte Funktion an eine Prozedur übergeben will, die eine Funktion einer einzigen Variablen erwartet?

Dafür kann man z.B. `unapply` einsetzen.[4] Verwenden Sie diesen Befehl, um für konkrete feste Werte von i und n eine Funktion $q(t)$ zu basteln, die zu $B_{i,n}(t)$ äquivalent (aber eben nur eine Funktion der einzigen Variablen t) ist.

Siehe auch Aufgabe 3.7.

Aufgabe 3.5. Verwenden Sie – für einen festen Wert von n (z.B $n = 5$) – die Bernstein-Polynome $B_{i,n}(t)$ aus Aufgabe 3.4 dazu, um eine sogenannte *Bézier-Kurve* zu konstruieren. Dazu gibt man $n+1$ Punkte $P_i := (x_i, y_i) \in \mathbb{R}^2$ vor $(i = 0 \ldots n)$ und definiert die Kurve über die Parameterdarstellung

$$B(t) := \sum_{i=0}^{n} B_{i,n}(t) P_i, \quad t \in [0, 1].$$

Diese glatte Kurve folgt in etwa dem Verlauf des Polygonzuges, der durch die 'Kontrollpunkte' P_i vorgegeben ist. Stellen Sie den Polygonzug und die Kurve $B(t)$ in einem gemeinsamen Plot dar (`? plots[pointplot]`, `? plot/parametric`). Spielen Sie ein bisschen mit den plot-Parametern, um ein optisch ansprechendes Resultat zu erhalten.

Aufgabe 3.6. Man verallgemeinere die Prozedur `diffq` aus VO III (Berechnung eines Differential- bzw. Differenzenquotienten) in folgender Weise:

Der Aufruf `diffq(f,x)` (2 Parameter übergeben) berechnet die Ableitung f'. Dabei kann `x` kann als Variablenname übergeben werden; falls jedoch `x` vom Typ `numeric` ist, wird f' an der Stelle `x` ausgewertet. Falls ein weiterer Parameter `y<>x` übergeben wird, wird der Differenzenquotient ausgewertet, wobei der Typ von `x` und `y` hier keine Rolle spielt. Der Aufruf `diffq(f,x,x)` soll äquivalent sein zu `diffq(f,x)`.

Hinweis: Dies erfordert eine variable Parameterliste; vgl. VO III (Stichworte: `_passed`, `_npassed`).

Aufgabe 3.7. Gegeben sei irgendeine Funktion $f(x, y)$ in zwei Variablen, $f : \mathbb{R}^2 \to \mathbb{R}$. Daraus sollen unter Verwendung von `? curry` zwei Funktionen[5] $g(x)$ und $h(y)$ gebastelt werden, und zwar in folgender Weise: Man schreibt eine Prozedur `generate_g(f,y_fixed)`, die die Funktion `g` als Ergebnis zurückliefert (!), mit der Eigenschaft dass $g(x) \equiv f(x, y_{fixed})$, d.h. der Wert von y wird zu $y = y_{fixed}$ 'eingefroren'. Analog für `h`.

Der Sinn besteht darin, dass man mit eine so entstehende Funktion als normale eindimensionale Funktion weiterverwenden kann. Testen Sie aus, dass das funktioniert, z.B. durch Verwendung von `g` als Parameter in einer Prozedur, die eine Funktion einer einzigen Variablen erwartet. Vgl. auch Aufgabe 3.4.

Aufgabe 3.8. Implementieren Sie die einfachste Variante eines iterativen Verfahren zum Aufsuchen einer Minimalstelle einer nichtlinearen reellwertigen Funktion $f(x, y, z)$ (*Gradientenverfahren*):

Für einen gegebenen Startwert (x_0, y_0, z_0) geht man iterativ wie folgt vor:
Man bestimmt den *Gradienten* $\nabla f(x_i, y_i, z_i) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})(x_i, y_i, z_i)$ (das ist die Richtung des lokal steilsten Anstiegs von f) und setzt

$$(x_{i+1}, y_{i+1}, z_{i+1}) := (x_i, y_i, z_i) - \alpha_i \nabla f(x_i, y_i, z_i). \tag{1}$$

Dies ist ein simpler und naheliegender Ansatz ($-\nabla f$ ist die lokal steilste Abstiegsrichtung); das Problem besteht in der Wahl des Parameters α_i (Schrittlänge). Eine eher naive Variante lautet wie folgt: Probieren Sie $\alpha_i := 1, \frac{1}{2}, \frac{1}{4}, \ldots$ und testen Sie auf Abstieg, d.h.

$$¿\ f((x_i, y_i, z_i) - \alpha_i \nabla f(x_i, y_i, z_i)) < f(x_i, y_i, z_i)\ ? \tag{2}$$

Sobald das erfüllt ist, akzeptiert man den neuen Wert gemäß (1) und macht iterativ weiter.

- Erfolgreiche Beendigung sobald $\|\nabla f(x_i, y_i, z_i)\|_2 \leq$ `tol`, (warum ?); (x_i, y_i, z_i) wird zurückgegeben; es sei denn:
- Fehler-Exit (`error ...`) falls
  - eine vorgegebene Maximalanzahl an Iterationsschritten überschritten wird (Parameter `maxiter`) – divergentes Verhalten;
  - die Abstiegsbedingung (2) nicht erfüllbar ist, weil eines der α_i einen vorgegebenen Minimalwert `minstep` unterschreitet – Iteration stagniert.

Dies ist in einer Prozedur `graditer(f,x0,y0,z0,tol,maxiter,minstep)` zu implementieren. Überlegen Sie sich selbst ein Beispiel.

Die Aufgabe ist beliebig ausbaufähig (z.B. optionale Protokollierung des Konvergenzverlaufes oder Implementierung für beliebig viele Variablen.)

---

[3] Vorsicht: In einer Schleife über i funktioniert das nicht – versuchen Sie zu verstehen, warum. Weiß jemand eine Lösung? (?)

[4] Generell dient `unapply` dazu, einen beliebigen Ausdruck in irgendwelchen Variablen (typischerweise das Ergebnis einer zuvor durchgeführten Berechnung) in eine Funktion in einer oder mehrerer dieser Variablen umzuwandeln.

[5] $g, h : \mathbb{R} \to \mathbb{R}$.