Seminar
The Digit Challenge

Dieses Dokument enthält die Seminarausarbeitungen der Teilnehmenden:

1. Alexander Dick, Dominik Vu und Anna Zechner
2. Birgit Hischenhuber, Sonja Höllrigl-Binder und Stefan Schuchnigg
3. Karl Rupp

Seminar: „The Digit Challenge“

Alexander Dick e0425474
Dominik Vu e0425313
Anna Zechner e9071597

31. Juli 2008

1 Aufgabe 1:

Aufgabenstellung: *Finden Sie das globale Minimum* $M = \min f(x,y)$ *der Funktion*

$$f(x,y) = \sin(\cos(x,y))) + \cos(\sin(x+2y)) + \sin(e^{x^2}) + e^{\cos(y)} + x^2 + y^2,$$

wobei $(x,y) \in \mathbb{R}^2$.

1.1 Erste Überlegungen:

Bevor wir uns an die numerische Behandlung des Problems machten, betrachteten wir die Funktion $f(x,y)$:

- $f(x,y) = f(-x,-y)$, d.h. es reicht o.B.d.A. $x \geq 0$ zu betrachten.
- Jeder Summand von $f(x,y)$ ist nach unten beschränkt; alle Summanden außer x^2 und y^2, die für $|x|, |y| \to \infty$ nach $+\infty$ konvergieren, sind nach oben beschränkt. Somit befindet sich das globale Minimum M in einem gewissen Rechteck um 0. Dieses lässt sich folgendermaßen berechnen:

$$\left.\begin{array}{rcl} \sin(\cos(.)) & \in & [-\sin(1), \sin(1)] \\ \cos(\sin(.)) & \in & [\cos(1), 1] \\ \sin(e^{(.)^2}) & \in & [-1, 1] \\ e^{\cos(.)} & \in & \left[\frac{1}{e}, e\right] \end{array}\right\} \implies f(x,y) - x^2 - y^2 \geq -\sin(1) + \cos(1) - 1 + \frac{1}{e} =: m$$

  Durch immer genaueres Zeichnen in Maple erhielten wir als erste Näherung $\tilde{M} = 4.04758\ldots \cong 4.04759 \geq M =: f(x_{min}, y_{min})$ und somit

$$x_{min}^2 + y_{min}^2 \leq \tilde{M} - m < 2.3^2.$$

  Also liegt der Punkt (x_{min}, y_{min}), an dem M angenommen wird, sicher im Rechteck $[-2.3, 2.3]^2$.

- f ist eine glatte Funktion.

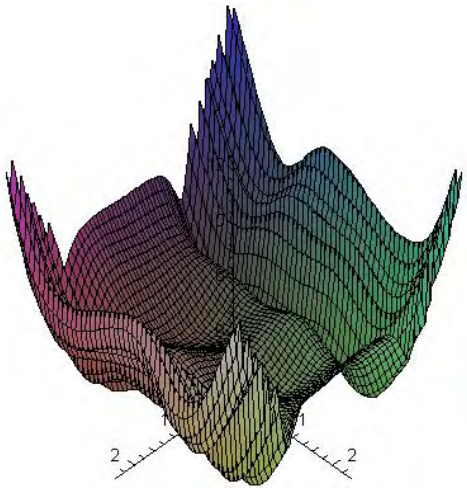


Abbildung 1: Plot von f auf $[-2.3, 2.3] \times [-2.3, 2.3]$. Man sieht, dass, vor allem aufgrund des $\sin(e^{x^2})$ -Termes, die Zeichnengenauigkeit in so einem großen Rechteck viel zu gering ist, als dass man so alle lokalen Minima erhalten könnte.

1.2 Einschränken auf ein Gebiet, in dem es nur ein Minimum gibt - das globale Minimum:

Zuerst betrieben wir eine einfache Gittersuche (Implemetierung in C - siehe Anhang: `A1_Gittersuche.c`): Dabei wird die Funktion f an einem Gitter mit Indices (i, j) ausgewertet und in jedem kleinen Rechteck zwischen vier Gitterpunkten mit Indices (i, j), $(i + 1, j)$, $(i + 1, j + 1)$, und $(i, j + 1)$ das Minimum und das Maximum von den Funktionswerten an den Randpunkten bestimmt. Alle Rechtecke, deren Minimum größer ist, als das Maximum eines anderen Rechtecks, werden im nächsten Schritt bei feinerem Gitter nicht weiter untersucht.
Nach zwei Gitterverfeinerungen erhielten wir für den Gitterabstand $h = 0.001$ $\hat{M} = 4.047598$ als obere Schranke für M, $x_{min} \in [1.234, 1.237]$, $y_{min} \in [-1.339, -1.335]$.
Bei dieser Methode kann es natürlich sein, dass man Minima übersieht. Um diesen Fall auszuschließen, kann man zum Beispiel die größtmögliche Abweichung von Funktionswerten, die innerhalb des untersuchten Rechtecks möglich sind, bezüglich der an den Gitterpunkten angenommenen Funktionswerten mit Hilfe der Gradienten von f abschätzen:

$$|f(u, v) - f(x_{Gitter}, y_{Gitter})| < \frac{|f(x_{Gitter}, y_{Gitter})| + \left\| \frac{df}{d(x,y)} \right\|_{(x,y)\in \text{Rechteck}}}{2},$$

wobei (u, v) im Rechteck liegt und x_{Gitter}, y_{Gitter} so, dass $|u - x_{Gitter}| \leq h/2$, $|v - y_{Gitter}| \leq h/2$.
Wir führten diese Überprüfung zwecks einfacherer Realisierbarkeit zusammen mit einem „Graph von f immer feiner zeichnen“ (was sich auch als Gittersuche interpretieren lässt) durch und erhielten eine Bestätigung des obigen Ergebnisses (Implementierung in Maple - siehe Anhang: `A1Gradient.mw`).

1.3 Berechnung des Minimums auf eine Genauigkeit von 1000 Stellen:

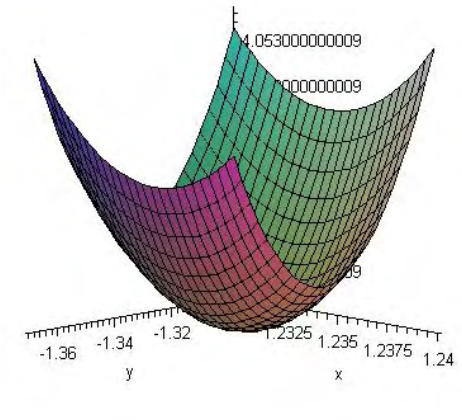


Abbildung 2: Plot von f auf $[1.23, 1.24] \times [-1.37, -1.3]$.

Im Zuge der obigen Überprüfung wurde auch gezeigt, dass f in dem Rechteck $[1.234, 1.237]\times[-1.339, -1.335]$ konvex ist (siehe Anhang: `A1Gradient.mw`), vgl. Abbildung 2. Daher gibt es in diesem Recheck nur ein lokales Minimum. Dieses lässt sich durch ein Newtonverfahren[1] auf den Gradienten $\frac{\partial f}{\partial x}$ und $\frac{\partial f}{\partial y}$ bestimmen.

[1] alle in diesem Unterkapitel verwendeten numerischen Verfahren und Algorithmen sind numerische Standardverfahren und lassen sich deshalb in den meisten Einführungsbüchern finden. Wir haben als Quelle das Skriptum zur Vorlesung „Numerische Mathematik“ von Winfried Auzinger WS 2006/07 verwendet.

Newton-Algorithmus für $F(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)^T$ mit Jacobimatix $DF(x,y)$:

```
for i = 1:n
    A = DF(x_i,y_i)
    b = F(x_i,y_i)
    löse Ad = b
    (x_{i+1},y_{i+1})^T = (x_i,y_i)^T - d
end
```

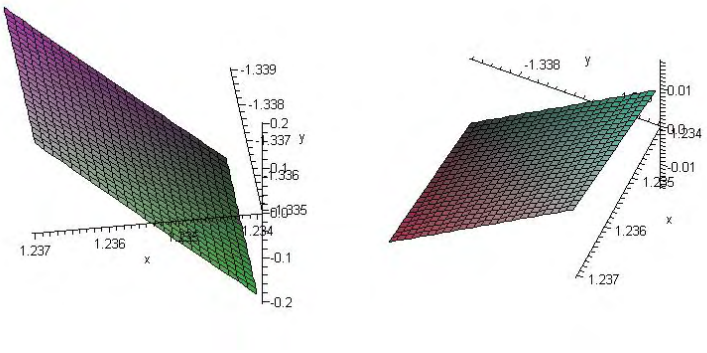


Abbildung 3: Plot der Gradienten $\frac{\partial f}{\partial x}$ und $\frac{\partial f}{\partial y}$ auf $[1.234, 1.237] \times [-1.339, -1.335]$.

Das Newtonverfahren ist quadratisch konvergent und im Fall der Funktion f lassen sich die Gradienten $\frac{\partial f}{\partial x}$ und $\frac{\partial f}{\partial y}$ in obigem Rechteck gut durch Ebenen nähern (vgl. Abbildung 3), daher erwarteten wir, schon nach wenigen Schritten die gewünschte Genauigkeit der Lösung zu erreichen.
Nachdem wir die Funktion in einer Ungebung des Minimums so genau wie möglich gezeichnet hatten, wählten wir als Startwerte $x_0 = 1.2355$, $y_0 = -1.337$, rechneten in Maple mit einer Genauigkeit von 1100 Stellen und erhielten nach 9 Newton-Schritten (vgl. Tabelle 1)für die ersten 1000 Nachkomma-Stellen(siehe Anhang: `A1Newton.mw`):

4.047582689914079035163436263727607019200376712399634318946696854441857011323664389166066148172420037959989500174186350048276540996811902395370580707405584272345755599240897130750710474611124352149960383994943569113585125425676681163677930781779687882312712710767377937847017217113696165509380977827042239019362467665781612819702463848005108595789723065410385892608716785899693628211459076414340229657167994146239479073373667857412617037155233175226604833341831947158482268951408612969384544105627487510541490914979072370210732032329365541647133333468776368977716964671509736172783851389212450693189779905787543674587469198660089143262683240748903026879798060686066752581568013996793061069206876658987783542694129570437330308955365126801224435001791659691600675469455744761789502876609517500967098288214907539064841077532152978022172693403088512845575754646210646135721258387558122131099598579514737090352676453002892687025996692784545400851664339607046526412882158935466013281794685997115521294908957309

Zu guter Letzt begründeten wir, dass die oben erhaltenen Stellen richtig sind:
Prinzipiell kommt man mit dem Newton-Verfahren beliebig nahe an die richtigen Werte x_{min}, y_{min} heran, man muss also nur den Rundungsfehler berücksichtigen. Wenn in einem Newton-Schritt ein Rundungsfehler begangen wird, der nicht allzu groß ist, wird mit dem Newton-Verfahren in den weiteren Schritten

n	$f(x_n, y_n) - f(x_{n-1}, y_{n-1})$
2	$-1.38 \cdot 10^{-16}$
3	$-3.21 \cdot 10^{-33}$
4	$-2.63 \cdot 10^{-66}$
5	$-2.80 \cdot 10^{-132}$
6	$-3.81 \cdot 10^{-264}$
7	$-7.25 \cdot 10^{-528}$
8	$-2.63 \cdot 10^{-1055}$
9	$-1 \cdot 10^{-1099}$

Tabelle 1: Differenz $f(x_n, y_n) - f(x_{n-1}, y_{n-1})$, gerundet: Man erkennt, dass sich die Anzahl der gemeinsamen Stellen in jedem Schritt ungefähr verdoppelt. Im letzten Schritt ist keine Verdoppelung mehr möglich, weil die Rechnengenauigkeit erreicht ist.

immer noch die richtige Lösung angenähert. Wie aus Tabelle 1 ersichlich, wird die Annäherung des Minimums mit jedem Schritt genauer, da die Differenz $f(x_n, y_n) - f(x_{n-1}, y_{n-1})$ immer negativ ist und f in einer hinreichend großen Umgebung des Minimums konvex ist. Also schätzten wir nur den Rundungsfehler im letzten Schritt ab, indem wir die in Maple eingebaute Intervallarithmetik verwendeten (siehe Anhang: `A1Newton.mw`). Wir erhielten, dass der Rundungsfehler im letzten Schritt $< 10^{-1050}$ ist, somit stimmen die ersten 1000 Nachkomma-Stellen.

2 Aufgabe 2:

Aufgabenstellung: *Lösen Sie das Integral:*

$$I = \int_{\mathbb{R}^6} e^{-x_1^4 - \cdots - x_6^4} \sin^2(x_1 \cdot \cdots \cdot x_6) dx,$$

wobei $x = (x_1, \cdots, x_6)$.

2.1 Vorangestellte Überlegungen:

Wir betrachten das gegebene Integral analytisch und werden mithilfe von Umformungen und geeigneteren Interpretationen eine bessere Ausgangsbasis für die numerische Behandlung des Integrals schaffen.
Unmittelbar ersichtlich ist, dass das Integral symmetrisch in jeder Dimension ist. Im eindimensionalen Falle wäre es ausreichend, nur entlang der positiven reellen Halbachse zu integrieren und danach zu verdoppeln. Im vorliegenden Fall werden wir also das Integral im in allen Koordinaten positiven 6-dimensionalen Hyperwürfel $x = (x_1, \ldots, x_6), x_i \geq 0$ behandeln und das dort erhaltene Ergebnis mit $2^6 = 64$ multiplizieren. Während dieser Multiplikationsvorgang den absoluten Fehler im schlechtes Fall in einer Größenordnung von 10^3 vergrößern könnte, gewinnen wir durch die Behandlung von nur einem Bruchteil des zu integrierenden Gebiets viel mehr an Rechenleistung, um diese Fehlerquelle effektiv eindämmen zu können.

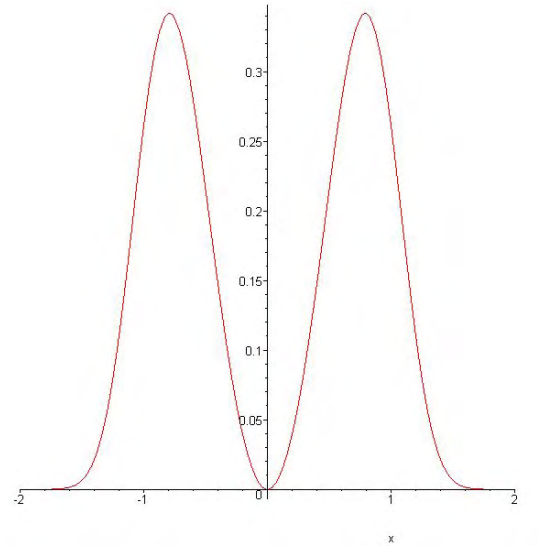


Abbildung 4: $e^{-x^4} \sin^2(x)$

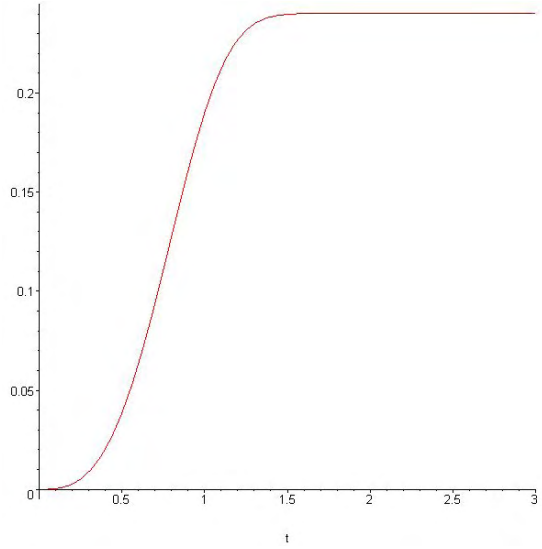


Abbildung 5: $\int_0^t e^{-x^4} \sin^2(x) dx$ nach t

Wir betrachten nun den Wertebereich des Integranden und stellen fest, dass dieser durch den Faktor $g = e^{-x_1^4 - \cdots - x_6^4}$ beschränkt wird. Im eindimensionalen Fall bewegt sich $g(2)$ bereits im Bereich von 10^{-6}, welcher sich im 2-dimensionalen Fall bereits auf 10^{-13} halbiert und im 6-dimensionalen Fall bereits $0.2031092663 \cdot 10^{-41}$ beträgt. Da durch den Faktor $h = \sin^2(x_1 \cdot \cdots \cdot x_6)$ nur mehr eine Modulierung zischen 0 und 1 stattfindet, wäre es naheliegend, das Integral zuerst beschränkt auf dem Hyperwürfel $[0, 2]^6$ zu betrachten und weiterer Folge versuchen die obere Schranke weiter zu erniedrigen, ohne relevaten Rechenfehler zu kumulieren. Auf Grund der letzten Endes gewählten Methode ist dies jedoch hinfällig.
Aus der Betrachtung des Problems für niedrigere Dimensionen, wissen wird, dass ein Ansatz für das Doppelintegral $\int_a^b \int_c^d f(x, y) dy dx$ wäre, eindimensionale Quadratur auf das äußere Integral $\int_a^b F(x) dx$ anzuwenden und dann für jeden x eindimensionale Quadratur über die innere Dimension zu verwenden um $F(x) = \int_c^d f(x, y) dy$ zu approximieren.
Wenn das Integrationsgebiet jedoch kompliziert oder von höherer Dimension ist, so verweist die Literatur auf Monte-Carlo-Methoden. Diese beruht auf dem Zusammenhang zwischen der Integration und der

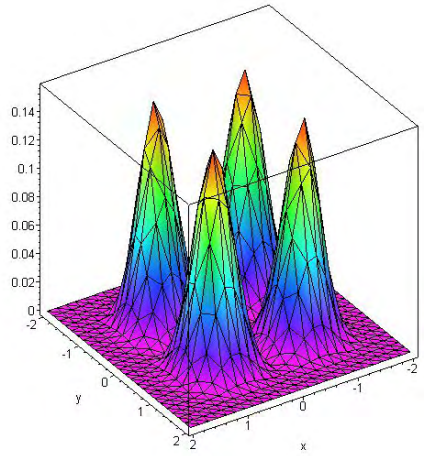


Abbildung 6: $e^{-x^4-y^4}\sin^2(xy)$

möglichen Interpretation als Erwartungswert einer Wahrscheinlichkeitsverteilung. Um eine geeignete Form zu erhalten, formen wir zunächst das Integral um.

2.2 Umformungen und Subsitution:

Wir substituieren $u_i = \sqrt{2}\,x_i^2$, für $i = 1,\ldots,6$ und erhalten folglich für die Funktionaldeterminante das Produkt der $dx_i = \left(\frac{\sqrt{\sqrt{2}}}{2\sqrt{2}} \cdot \frac{1}{\sqrt{u_i}}\right) du_i$. Hierbei sei angemerkt, dass diese Substitution nicht bijektiv ist, jedoch mit der vorher erwähnten Ausnützung der Symmetrieeigenschaften sinnvoll wird. Daher erhalten wir nun das Integral in neuer Gestalt:

$$I = 2^6 \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 x_i^4} \sin^2\left(\prod_{i=1}^6 x_i\right) dx = 2^6 \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} \sin^2\left(\prod_{i=1}^6 \sqrt{\frac{u_i}{\sqrt{2}}}\right) dx$$
$$= 2^6 \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} \sin^2\left(\frac{\prod_{i=1}^6 \sqrt{u_i}}{\left(\sqrt{\sqrt{2}}\right)^6}\right) \left(\frac{\sqrt{\sqrt{2}}}{2\sqrt{2}}\right)^6 \prod_{i=1}^6 \frac{1}{\sqrt{u_i}}\, du$$

Wohlwissend, dass $\left(\sqrt{\sqrt{2}}\right)^6 = 2^{6/4} = 2\sqrt{2}$, vereinfachen wir weiter:

$$I = \frac{2^6}{\left(2\sqrt{2}\right)^6} \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} \sin^2\left(\frac{\prod_{i=1}^6 \sqrt{u_i}}{2\sqrt{2}}\right) \frac{2\sqrt{2}}{\prod_{i=1}^6 \sqrt{u_i}}\, du$$
$$= \frac{1}{8} \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} \cdot \frac{\sin^2\left(\frac{\prod_{i=1}^6 \sqrt{u_i}}{2\sqrt{2}}\right)}{\frac{\prod_{i=1}^6 \sqrt{u_i}}{2\sqrt{2}}}\, du = \frac{1}{8} \cdot \int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} \cdot f\left(\frac{\prod_{i=1}^6 \sqrt{u_i}}{2\sqrt{2}}\right)\, du,$$

wobei wir $f : \mathbb{R}^+ \to [0,1], x \mapsto \frac{\sin^2(x)}{x}$ definieren.

2.3 Das Integral als Erwartungswert

Wie bereits zu Beginn vorangestellt, ist unser Ziel, das Integral als Erwartungswert von normalverteilten, unabhängigen Zufallsvariablen zu interpretieren. Der Faktor $f(x)$ ist im positiven zwischen 0 und 1 beschränkt, also als Wahrscheinlichkeitsmaß geeignet. Da in allen gängigen mathematischen Software-Paketen Generatoren für Zufallszahlen zur Verfügung stehen, wollen wir den Ausdruck vor allem normieren, da der Standard die Normalverteilung mit Erwartungswert $\mu = 0$ und Varianz $\sigma = 1$ darstellt. Wir schreiben

$$\int_{(\mathbb{R}^+)^6} e^{-\sum_{i=1}^6 u_i^2/2} du = \int_{(\mathbb{R}^+)^6} e^{-u_1^2/2} e^{-u_2^2/2} \cdots e^{-u_6^2/2} du,$$

wobei dies als Produkt von 6 voneinander unabhängigen Gauss-Integralen gesehen werden kann. Aus der Wahrscheinlichkeitstheorie wissen wir, dass $\int_{\mathbb{R}} e^{-x^2/2} dx = \sqrt{2\pi}$. Da wir hier nur aber nur den positiven Anteil betrachten, ist der Wert der zugrunde liegenden Norm, die Hälfte, sodass mit $\left(\sqrt{\frac{\pi}{2}}\right)^6$ normieren:

$$\begin{aligned} I &= \left(\sqrt{\frac{\pi}{2}}\right)^6 \cdot \frac{1}{8} \cdot \int_{(\mathbb{R}^+)^6} \frac{e^{-\sum_{i=1}^6 u_i^2/2}}{\left(\sqrt{\frac{\pi}{2}}\right)^6} \cdot f\left(\frac{\prod_{i=1}^6 \sqrt{u_i}}{2\sqrt{2}}\right) \, du \\ &= \frac{\pi^3}{64} \cdot \mathbb{E}\left(f\left(\frac{\sqrt{u_1 u_2 \cdots u_6}}{2\sqrt{2}}\right)\right) \end{aligned}$$

Hierbei sind die u_i gemäß unseren Annahmen als standardnormalverteilte, positive Zufallsvariablen zu sehen.

2.4 Numerische Umsetzung

Wie gesehen, machen wir uns die Eigenschaften des Erwartungswert zunutze, wobei

$$\mathbb{E}\left(f\left(\frac{\sqrt{u_1 u_2 \cdots u_6}}{2\sqrt{2}}\right)\right) = \lim_{j\to\infty} \frac{1}{j} \sum_j f\left(\frac{\sqrt{u_{1_j} u_{2_j} \cdots u_{6_j}}}{2\sqrt{2}}\right).$$

Da wir natürlich nicht unendlich viele Zahlen zur Verfügung haben, müssen wir uns mit einer endlichen Anzahl an "Stichproben" begnügen, die aber erfreulicherweise dem Gesetz der großen Zahl gehorcht. Wir werden bei der Konvergenzbetrachtung und Fehlerabschätzung darauf zurückkommen. Alles was wir benötigen ist ein Programm, dass normalverteilte Zufallszahlen erzeugen kann. In Matlab ist dies der Befehl `randn(`a, b`)`, der eine $a \times b$-Matrix mit unabhängigen standardnormalverteilten Zufallszahlen generiert. In unserem Fall sieht der Code wie folgt aus:

```
Integration von I nach der Monte-Carlo-Methode
while i < (n+1)
    tmp=randn(6,1);
    if tmp(1)>0 && tmp(2)>0 && tmp(3)>0 && tmp(4)>0 && tmp(5)>0 && tmp(6)>0
        for j=1:6
            tmp2= tmp2.*tmp(j);
        end
        tmp2= c.*sqrt(tmp2);
        ell= ell + (((sin(tmp2)).^ 2)./tmp2);
        i= i+1;
        tmp2=1;
    end
end
return pi.^ 3 .*ell ./(64.*n)
```

Hierbei sind `n` die Anzahl der Stichprobe und `ell` die Summe der Funktionswerte von f aller Stichproben.

2.5 Ergebnisse, Interpretation und Effektivität

Um Aussagen über Geschwindigkeit des Algorithmus machen zu können, wurde die Matlab-Befehle `tic` und `toc` verwendet. Auf dem Server `cad.zserv.tuwien.ac.at` haben wir den Algorithmus mit einer Stichprobengröße von 10^4 1000-mal laufen lassen. Dabei merkt man, dass sich die Ergebnisse noch durchaus im Tausendstelbereich unterscheiden, während die Werte von mehreren solchen Testläufen auch im diesem Bereich stabil sind. Wir haben ein paar Ergebnisse aus der Testreihe herausgegriffen um zu illustrieren, wie sich die Lösungen entwickelten. Der Literatur entsprechend, konvergiert diese Monte-Carlo-Methode in der Größenordnung von $\mathcal{O}(1/\sqrt{n})$, wobei n die Anzahl der Stichproben darstellt. Demnach müssten wir für 10 genaue Stellen 10^{20} Stichproben analysieren.

i	Wert	Rechenzeit
1	**0.0**4956062448952	6.38570000000000
2	**0.050**78978654312	6.31948300000000
14	**0.050**94766534634	6.28157000000000
23	**0.050**99712198669	6.37813000000000
43	**0.0503**7482392388	6.37676500000001
44	**0.0**4991297127899	6.21000000000001
50	**0.05**309771252956	6.61430000000000
64	**0.0503**4127959304	5.75800000000000
77	**0.050**57963941133	6.70621300000000
91	**0.05035**847689788	6.03299000000000
95	**0.0**4798604213589	6.71154000000000
all	**0.0503**9320848845	$\sum = 63.7253000000000$

Tabelle 2: 10 exemplarische Werte und Laufzeit aus der Testreihe

Aufgrund des Mangels an verfügbaren Rechnern (bzw. stabiler Remote-Verbindungen) und drohender Überhitzung des eigenen, wurde (da der Programmaufbau dies explizit zulässt) der Algorithmus nach $93616113 \approx 10^8$ Stichproben und ca. 15 Stunden angehalten. Das dabei ermittelte Resultat (0.05035897873862) wurde dann gewichtet mit den anderen Stichproben gemittelt und wir erhielten als vorläufiges Resulat **0.5035 99** 74516882. Dieses ist zumindest auf 4 Stellen gesichert, deckt sich aber auch auf 2 weiteren bereits mit den Ergebnissen anderer Seminarteilnehmer.

3 Aufgabe 3:

Aufgabenstellung: *Die unendliche Matrix* $A = (a_{ij})$*, definiert durch*

$$a_{ij} = \begin{cases} \frac{1}{i+j-1} & : i+j \ \textit{ist Primzahl} \\ 0 & : \ \textit{sonst}, \end{cases} \qquad i,j \geq 1,$$

bilden einen unendlich-dimensionalen Operator auf ℓ^2*. Welchen Wert hat*

$$a = \|A\| = \sup_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2},$$

wobei $\|x\|_2^2 = \sum_{i=1}^{\infty} x_i^2$ *?*

3.1 Zuerst ein paar theoretische Überlegungen:

Bevor wir mit der numerischen Behandlung des Problems beginnen konnten, brauchten wir ein wenig Theorie:

a) Man kann die Norm von A mit Hilfe der Normen der $n \times n$-dimensionalen Teilmatrizen $A_n = (a_{ij})_{i,j \leq n}$ annähern:
für $1 \leq n \leq m$ gilt: $\|A_n\| \leq \|A_m\| \leq \lim_{k\to\infty} \|A_k\| = \|A\|$.

b) $\|A\| \leq \pi$.

c) Der normierte Vektor x_n, für den $\|A_n x_n\|$ maximal ist, hat folgende Gestalt:

Alle Komponenten $(x_n)_i$ von x_n sind $\in \mathbb{R}_0^+$ oder alle $(x_n)_i \in \mathbb{R}_0^-$
und es gilt: $1 \geq |(x_n)_1| \geq |(x_n)_2| \geq \cdots \geq |(x_n)_n| \geq 0$.

d) A ist eine symmetrische Matrix:

$$A = \begin{pmatrix} 1 & \frac{1}{2} & 0 & \frac{1}{4} & 0 & \frac{1}{6} & \cdots \\ \frac{1}{2} & 0 & \frac{1}{4} & 0 & \frac{1}{6} & 0 & \\ 0 & \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & \\ \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & 0 & \\ 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & \\ \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \\ \vdots & & & & & & \ddots \end{pmatrix}$$

Beweis: Im Buch „The Siam 100-Digit Challenge“ von Folkmar Bornemann et al. (Lemma 3.1) wird folgendes gezeigt:

$$\textit{Wenn } \|A\| < \infty, \textit{ dann gilt für } 1 \leq n \leq m: \ \|A_n\| \leq \|A_m\| \leq \lim_{k\to\infty} \|A_k\| = \|A\|.$$

D.h. für uns verbleibt zu zeigen: $\|A\| < \infty$.

Behauptung: $\|A\| \leq \|H\|$, wobei H die Hilbert-Matrix ist $\left(h_{ij} = \frac{1}{i+j-1}\right)$, für die gilt: $\|H\| \leq \pi$.

Beweis:

- Es gilt[2]:
Die Bilinearform $A(x,y) = \sum_i \sum_j \frac{x_i y_j}{i+j-1}$ *ist für alle* $x = (x_1, x_2, \ldots)^T$, $y = (y_1, y_2, \ldots)^T \in \ell^2$ *mit*

[2]siehe “Inequalities, Second Edition“ von G. H. Hardy, J. E. Littlewood, G. Polya, erschienen bei Cambridge University Press: Theorem 294

$\|x\|_2, \|y\|_2 \leq 1$ *mit höchstens* π *beschränkt.*
Diese Aussage lässt sich umformulieren zu:

$$x^T Hy = (x, Hy) \leq \pi,$$

wobei $(.,.)$ das innere Produkt in ℓ^2 bezeichnet. Mit dem folgenden einfachen Satz[3] aus der Funktionalanalysis lässt sich dann leicht auf $\|H\| \leq \pi$ schließen:
Sei T *ein linearer Operator von einem Hilbertraum* H_1 *in einen Hilbertraum* H_2*, dann gilt:*

$$\|T\| = \sup\{|(Tx, y)| : x \in H_1, y \in H_2, \|x\|, \|y\| \leq 1\}.$$

- Wähle $x \in \ell^2$, sodass o.B.d.A. alle Komponenten $x_i \in \mathbb{R}_0^+$ und $\|x\|_2 = 1$. Da alle Einträge von A größer oder gleich 0 sind, gilt sicher

$$\|A\| = \sup_{x \in \{x : x_i \in \mathbb{R}_0^+, \|x\| = 1\}} \|Ax\|_2 .$$

Also folgt aus

$$\begin{aligned}
\|Ax\|_2 &= \sqrt{\left(\sum_{(n+1)\in\mathbb{P}} \frac{1}{n} x_n\right)^2 + \left(\sum_{(n+2)\in\mathbb{P}} \frac{1}{n+1} x_n\right)^2 + \ldots} \quad \leq \\
&\leq \sqrt{\left(\sum_{n\in\mathbb{N}} \frac{1}{n} x_n\right)^2 + \left(\sum_{n\in\mathbb{N}} \frac{1}{n+1} x_n\right)^2 + \ldots} \quad = \|Hx\|_2 \quad \forall x \in \{x : x_i \in \mathbb{R}_0^+, \|x\| = 1\}, \\
\|A\| &\leq \|H\| \quad \square
\end{aligned}$$

3.2 Erste Näherung mit Hilfe des Matlab-Befehls `normest`:

Als allererste Näherung berechneten wir die Normen der Teilmatrizen A_n mit dem Matlab-Befehl `norm`, der die 2-Norm einer Matrix berechnet; die Matrizen A_n stellten wir mit Hilfe der Befehle `isprime` und `hankel` auf. Die Funktion `norm` hat allerdings einen großen Rechneraufwand und ist deswegen für große n sehr langsam. Deshalb verwendeten wir im nächsten Schritt (siehe Anhang - `A3normest.m`) den Befehl `normest`, der die 2-Norm einer Matrix nähert. Wir gaben als Toleranz für den relativen Fehler 10^{-5} an und erhielten folgende Werte:

n	normest(A_n)
5	**1**.26537
10	**1**.29515
20	**1.3**0892
40	**1.3**1540
80	**1.3**1884
160	**1.32**050
320	**1.32**137
640	**1.32**178
1280	**1.32**197
2560	**1.322**07
5120	**1.3221**2
5300	**1.3221**2

$n = 5300$ ist der größte Wert, für den die Matrix A_n noch aufgestellt werden konnte. Man erkennt, dass man für jede weitere Nachkomma-Stelle um einiges mehr an Speicherplatz benötigt.

[3] siehe z.B.: Skriptum zur Vorlesung „Funktionalanalysis“ von Martin Blümlinger, SS 2006, TU Wien: Korollar 5.3.1

Als Nächstes probierten wir als einfaches iteratives Verfahren zur Bestimmung der Norm einer Teilmatrix A_n, den Vektor x_n mit $\|x_n\|_2$ und $\|A_n\| = \|A_n x_n\|_2$ zu bestimmen. Dabei gingen wir folgendermaßen vor: Wir wählten laut Teil d) des ersten Unterkapitels als Startwert $x_{n,0}$ den Vektor $(1, 0, 0, \ldots, 0)^T$ und verschoben im Vektor der Quadrate der einzelnen Komponenten $((x_{n,i})_1^2, (x_{n,i})_2^2, \ldots, (x_{n,i})_n^2)^T$ jeweils solange kleine Einheiten (z.B. 0.01) von einer Komponente in die nächste, bis die Norm $\|A_n x_n\|_2$ maximal war. Bei dieser Methode muss die Matrix A_n zwar nicht explizit augestellt werden, aber leider ist das Verfahren so langsam, dass man auch kein besseres Ergebnis als mit `normest` erhält.

3.3 Extrapolation[4]:

Um höhere Genauigkeit zu erhalten, verwendeten wir den Epsilon-Algorithmus von Wynn (siehe Anhang - `wynn.m`). Dieser besitzt eine ziemlich einfache Rekursionsformel:

$$s_{k,j} = s_{k+1,j-2} + \frac{1}{s_{k+1,j-1} - s_{k,j-1}}$$

mit $s_{k,-1} = 0$ und $s_{k,0} = s_k$. Der Epsilon-Algorithmus liefert nach $2n+1$ Schritten den exakten Grenzwert, falls die Folge als Summe von n geometrischen Folgen dargestellt werden kann.
Die Startwerte s_k sind die mit Vektoriteration berechneten Näherungswerte für die Normen $\|A_{2^{k-1}}\|$. Wir verwendeten zur Berechnung der Näherungswerte die Potenzmethode als Vektoriteration statt dem Matlab-Befehl `normest` - diese liefert dieselben Werte, ist aber um einiges schneller.

n	$\|A_n\|$ mit Vektoriteration	extrapoliert
1	**1**.00000000000000	**1**.00000000000000
2	**1**.20710678118655	**1**.28220799096267
4	**1**.26222205190192	**1**.25854528967668
8	**1**.28495256009120	**1.322**76111174633
16	**1.3**0588442860036	**1.322**31096891827
32	**1.3**1363872898456	**1.32**197663656318
64	**1.3**1782692832248	**1.32217**352307279
128	**1.32**007606106344	**1.32217**248812494
256	**1.32**116798744673	**1.32217**249098306
512	**1.32**167963990059	**1.32217**343242803
1024	**1.32**193027044597	**1.32217**371652841
2048	**1.322**05394898655	**1.32217**353330203
4096	**1.3221**1474747365	1.32211474747365

n	$\|A_n\|$ mit Vektoriteration	extrapoliert
5	**1**.26537731314232	**1**.26537731314232
10	**1**.29515384327312	**1.32**078238809074
20	**1.3**0892747570085	**1.32**063713286271
40	**1.3**1540715195123	**1.322**24707072307
80	**1.3**1884266404561	**1.32**191606237388
160	**1.32**050235443551	**1.32217**501079863
320	**1.32**137618320827	**1.32217**238339273
640	**1.32**178058890543	**1.32217**263079123
1280	**1.32**197970150725	**1.32217**385405483
2560	**1.322**07834230716	**1.32217**218699335
5120	**1.3221**2643380278	**1.3221**2643380278

Wir erhielten somit 5 Nachkommastellen[5]: $\|A\| \doteq 1.32217$.

[4] Die in diesem Kapital verwendeten Verfahren werden auch in dem Buch von Bornemann beschrieben.

[5] Aus den Ergebnis könnte man schließen, dass man 6 Nachkommastellen erhält: 1.322172. Die 6. Selle widerspricht aber dem Ergebnis der anderen Seminarteilnehmer, d.h. die Extrapolation liefert einen kleinen Fehler.

4 Aufgabe 4:

Bestimmen Sie $c > 0$, so dass die Lösung von

$$-\Delta u = e^u \text{ in } \Omega = (0,1)^2,\ u = c \text{ auf } \Gamma = \{0\} \times (0,1),\ u = 0 \text{ auf } \partial\Omega \setminus \Gamma$$

die Beziehung $u(\frac{1}{3}, \frac{1}{3}) = 1$ erfüllt.

4.1 5-Punkt-Differenzenstern

Da es sich hierbei um ein elliptisches Problem handelt, bestehen Grundsätzlich zwei Möglichkeiten es numerisch zu Behandeln. Einerseits die Methode der finiten Elemente und andererseits die Diskretisierung mittels finiter Differenzen. Aufgrund der Nichtlinearität der partiellen Differentialgleichung wird zusätzlich, bei beiden Methoden, noch eine Implementierung des Newton Verfahrens zur Lösung des auftretenden nichtlinearen Gleichungssystems benötigt.

Für unsere erste Implementierung haben wir zweiteres, die Methode der finiten Differenzen, gewählt. Zur Diskretisierung des Laplace Operators wurde ein 5-Punkt-Differenzenstern verwendet. Hierbei werden die partiellen Ableitungen durch zentrale Differenzenquotienten 2. Ordnung ersetzt. In den randnahen Gitterpunkten tritt hier jeweils ein (bzw. zwei) durch die Dirichlet-Randdaten vorgegebener Wert auf. Das heißt, die Randbedingungen sind, aufgrund ihrer einfachen Gestalt, bereits in der Diskretisierung des Problems enthalten. Insgesamt bleibt ein Gleichungssystem mit n^2 Unbekannten zu lösen. Der Parameter n bezeichnet die Anzahl der innern Gitterpunkte in der Diskretisierung des Gebiets. In dem behandelten Beispiel ist unser Gebiet Ω das Einheitsquadrat, es wird also in $n+1 \times n+1$ Quadrate zerlegt.

Um einen ersten Näherungswert zu erhalten, wurde ein c vorgegeben und der Wert der Lösung u an der Stelle $(\frac{1}{3}, \frac{1}{3})$ kontrolliert. Mit dieser Methode konnte in erster Näherung ein Wert von $c = 2,22$ bestimmt werden. Für eine genauere Berechnung ist diese Methode allerdings zu aufwendig.

Wie schon oben erwähnt ensteht bei der Diskretisierung ein Gleichungssystem mit n^2 Unbekannten. Wählt man das Gitter jedoch so, dass der bekannte Wert von u an der Stelle $(\frac{1}{3}, \frac{1}{3})$ auf dem Gitter liegt, gewinnt man einen Freiheitsgrad. Bedenkt man, dass die Randwerte ja auch in diesen Gleichungen vorkommen, ist es naheliegend eine Funktion F folgendermaßen zu definieren:

$$F(u,c) := L_h u - e^u + \text{Randbedingung}$$

wobei L_h hier jene Matrix bezeichnet, welche durch die Diskretisierung des Laplace-Operators entsteht. Nun sucht man mittels des Newtonverfahrens eine Nullstelle dieser Funktion. Dabei wird in der Jacobi Matrix $DF = L_h - \text{diag}(e^u)$, die Spalte welche die Ableitung nach jener Komponente von u welche den Wert an der Stelle $(\frac{1}{3}, \frac{1}{3})$ repräsentiert, durch die Ableitung der Funktion nach dem Wert c ersetzt.

$$DF(u,c) = \begin{pmatrix} \frac{\partial F_1}{\partial u_1} & \cdots & \frac{\partial F_1}{\partial c} & \cdots & \frac{\partial F_1}{\partial u_n} \\ \vdots & & \vdots & & \vdots \\ \frac{\partial F_n}{\partial u_1} & \cdots & \frac{\partial F_n}{\partial c} & \cdots & \frac{\partial F_n}{\partial u_n} \end{pmatrix}$$

Somit liefert die Newtoniteration eine Näherung für den gesuchten Wert c.
In der nachfolgenden Tabelle sind die mit dieser Methode erhaltenen Werte für c in Abhängigkeit von der Anzahl der Unterteilungen des Gebiets angegeben.

Teilungspunkte	Loesung c
30	2.221472661943940
60	2.220531375737992
75	2.220417873967323

Ein Nachteil dieser Rechenmethode ist der große Speicherbedarf, im Fall der 75 Teilungspunkte ist ein $75^2 \times 75^2$ Gleichungssystem zu lösen. Die Matrix ist zwar nur schwach besetzt, sie enthält maximal $5 \cdot 75^2$ Einträge, jedoch auf einem handelsüblichen Desktopsystem gab Matlab die Fehlermeldung aus, dass zu wenig Arbeitsspeicher verfügbar sei. So war es nicht Möglich mit mehr als 75 Teilungspunkten zu rechnen.

4.2 9-Punkt-Stern

Um eine „zweite Meinung“ zu bekommen, wurde der obenstehende Rechenvorgang nochmals mit einem 9-Punkt-Differenzenstern durchgeführt. Untenstehend die erhaltenen Resultate:

Teilungspunkte	Loesung c
30	2.222418392873
45	2.221194758490
60	2.220766475113

Hier war es leider nur möglich bis zu 60 Teilungspunkten zu rechnen, da beim 9-Punktstern die Matrix dichter besetzt ist, sie enthält bei 60 Teilungspunkten $9 \cdot 60^2$ Einträge.
Um eine bessere Konvergenzordnung zu erhalten, haben wir in einem dritten Schritt die Funktion e^u nicht punktweise abgetastet, sondern die Diskretisierung durch eine lokale Mittelung gewonnen. Dies bracht folgende Werte für c:

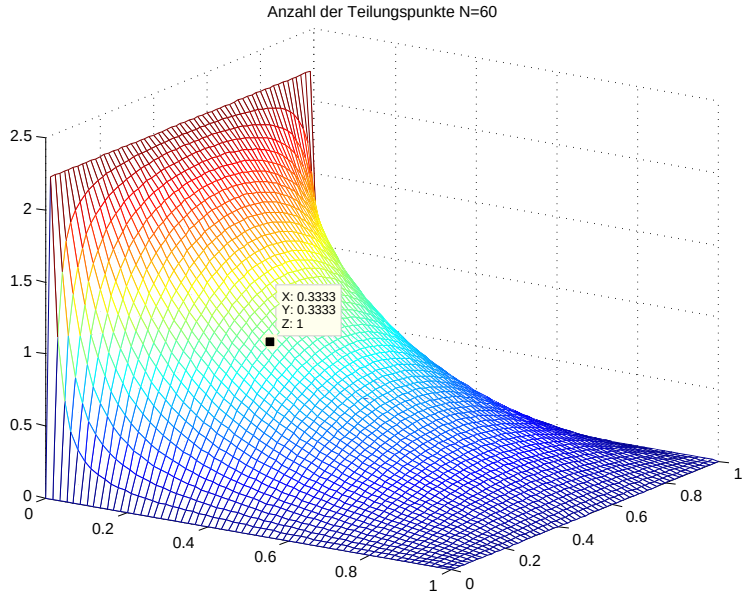
Teilungspunkte	Loesung c
30	2.223090265694
45	2.221493109413
60	2.220934246513

4.3 Ergebnis

Zusammenfassend können wir also davon ausgehen, zumindest 3 richtige Nachkommastellen für unseren gesuchten Wert gefunden zu haben.

$$c \doteq 2.220$$

Nachstehend noch ein Matlab-Plot der Lösung.


Anzahl der Teilungspunkte N=60
X: 0.3333
Y: 0.3333
Z: 1
2.5
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

5 Aufgabe 5:

Aufgabenstellung:
Die Lösung (x, y, z) das Anfangswertproblems

$$\dot{x} = k_1 y - k_2 xy + k_3 x - k_4 x^2, \tag{1}$$
$$\dot{y} = -k_1 y - k_2 xy + k_5 f z, \tag{2}$$
$$\dot{z} = 2k_3 x - k_5 z, \tag{3}$$

mit $(x, y, z)(0) = (0.01, 0.02, 0.2)$ *und* $k_1 = 1.28, k_2 = 2.4 \cdot 10^6, k_3 = 33.6, k_4 = 3000, k_5 = 1$ *und* $f = 0.5$ *ist periodisch. Bestimmen sie das kleinste* $p > 0$*, so dass* $x(t) = x(t + p)$ *für alle* $t > 0$*.*

5.1 Steifheit des Problems

Bei dem obigen AWP handelt es sich um ein steifes Problem. Dies erkennt man, wenn man die Eigenwerte der Jacobimatrix

$$Df = \begin{pmatrix} -k_2 y + k_3 - 2k_4 & k_1 - k_2 x & 0 \\ -k_2 y & -k_1 - k_2 x & k_5 f \\ 2k_3 & 0 & -k_5 \end{pmatrix}$$

betrachtet. An der Stelle $(0, 0, 0)$ ist der kleinste Eigenwert $\lambda_{min} = -0.97626$ und der größte $\lambda_{max} = -5966,39$. Aus der untenstehenden numerischen Simulation erkennt man, dass $y \leq 0.025$ und $x \leq 0.01$ ist. Setzt man diese Werte in die Jacobimatri erhält man $\lambda_{min} = -1,0056$ und $\lambda_{max} = -88345,0096$. Damit hat man ein eindeutiges Indiz für das steife Verhalten des Systems. Aus diesem Grund werden im folgenden Lösunsverfahren verwendet, welche für steife Differentialgleichungen geeignet sind.

5.2 Simulation in Matlab

Wir wollen einen in Matlab standardmäßig vorhandenen Solver verwenden. Für steife Probleme stehen uns hier zwei zur Auswahl: ode23s und ode15s.

Name	Beschreibung
ode23s	Einschrittverfahren, basierend auf einer modifizierten Rosenbrock Formel der Ordnung 2, deswegen häufig effizienter als ode15s, jedoch mit geringerer Genauigkeit
ode15s	Mehrschrittverfahren variabler Ordnung, basierend auf den „numerical differentiation formulas (NDFs)“, für hohe Genauigkeitsanforderungen geeignet

Da wir uns ja nur für die Periode der Lösung interessieren, wählen wir folgende Vorgehensweise:

1. Berechnen der numerischen Lösung
2. Bestimmen der lokalen Maxima der x-Komponente der Lösung
3. Berechnen der Abstände (=Periode) der lokalen Maxima

5.2.1 ode15s

Zu Beginn wählen wir ode15s, von dem wir apriori eine genauere Lösung erwarten, mit absoluter und relativer Toleranz gleich 10^{-15}, also der Maschinengenauigkeit in Matlab. Für den Paramter wählen wir das Intervall $t \in [0, 210]$. Die x- und y- Komponente der erhaltenen Lösung sind in Abbildung 1 dargestellt. Der Abstand von 0 zum ersten Peak ist signifikant kürzer als die Abstände zwischen den anderen Peaks. Hierbei handelt es sich um einen Einschwingvorgang, aus diesem Grund betrachten wir im Weiteren die Lösung immer ab dem erstem lokalen Maximum.
Die Bestimmung der lokalen Maxima ist mittels einer Funktion, die zwei aufeinander folgende Funktionswerte vergleicht, einfach möglich, da der ODE-Solver einen Vektor mit diskreten Werten zurückliefert.

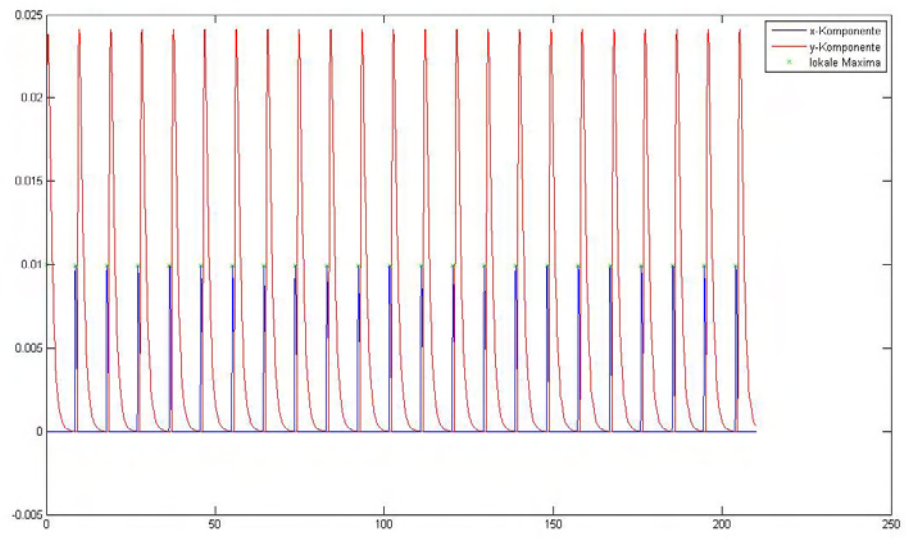


Abbildung 7: Plot der x- und y-Komponente der Lösung für $t \in [0, 210]$

Außerdem ist die Ableitungen der Lösung in der Nähe des lokalen Maximums betragsmäßig groß, so daß immer ein eindeutiger Wert für das Maximum gefunden wird.
Berechnet man nun die Abstände zwischen den Peaks, so erhält man als minimalen Wert $p_{min} = 9.175589$ und als mximalen Wert $p_{max} = 9.176263$. Wir gehen also davon aus, dass wir für den wahren Wert p zwei Nachkommastellen gefunden haben.

$$p \doteq 9.17$$

5.2.2 ode23s

Aufgrund der nicht wirklich befriedigenden Ergebnisse mit ode15s wollen wir den zweiten zur Wahl stehenden Matlab-Solver, ode 23s, verwenden. Wir verwenden die selben Einstellungen wie bei der vorhergehenden Simulation und erhalten $p_{min} = 9.175863$ und $p_{max} = 9.175878$. Dies steht nicht im Widerspruch zu den vorher erhaltenen Resultaten und stellt somit eine Verbesserung der Näherung um zwei Nachkommastellen dar.
Das apriori als weniger genau angenommene Einschrittverfahren liefert hier also bessere Ergebnisse, was in dem Verhalten der Lösung begründet ist. Da diese über weite Strecken nahezu konstant ist und nur sehr schmale Peaks aufweist, dürfte hier die konstante Ordnung 2 des ode23s einen Vorteil gegenüber des Mehrschrittverfahrens mit variabler Ordnung bedeuten.
Wir haben also:

$$p \doteq 9.1758$$

6 Bearbeitung der Aufgaben

Aufgabe 1: Anna Zechner
Aufgabe 2: Dominik Vu
Aufgabe 3:

3.1 Anna Zechner

3.2 Alexander Dick, Anna Zechner

3.3 Anna Zechner

Aufgabe 4:

4.1 Alexander Dick, Anna Zechner

4.2 Alexander Dick

4.3 Alexander Dick

Aufgabe 5: Alexander Dick

AKNUM Seminar:

The Digit Challenge

Birgit Hischenhuber e0426274
Sonja Höllrigl-Binder e0426349
Stefan Schuchnigg e0426505

4. August 2008

1 Aufgabe

Stefan Schuchnigg

1.1 Aufgabenstellung

Finden sie das Minium der Funktion

$$f(x,y) = sin(cos(x \cdot y)) + cos(sin(x + 2 \cdot y)) + sin(e^{x^2}) + e^{cos(y)} + x^2 + y^2$$

Im weiteren sei $f_1(x,y) := sin(cos(x \cdot y))$, $f_2(x,y) := cos(sin(x + 2 * y))$, $f_3(x,y) := sin(e^{x^2})$, $f_4(x,y) := e^{cos(y)}$ und $f_5(x,y) := x^2 + y^2$.

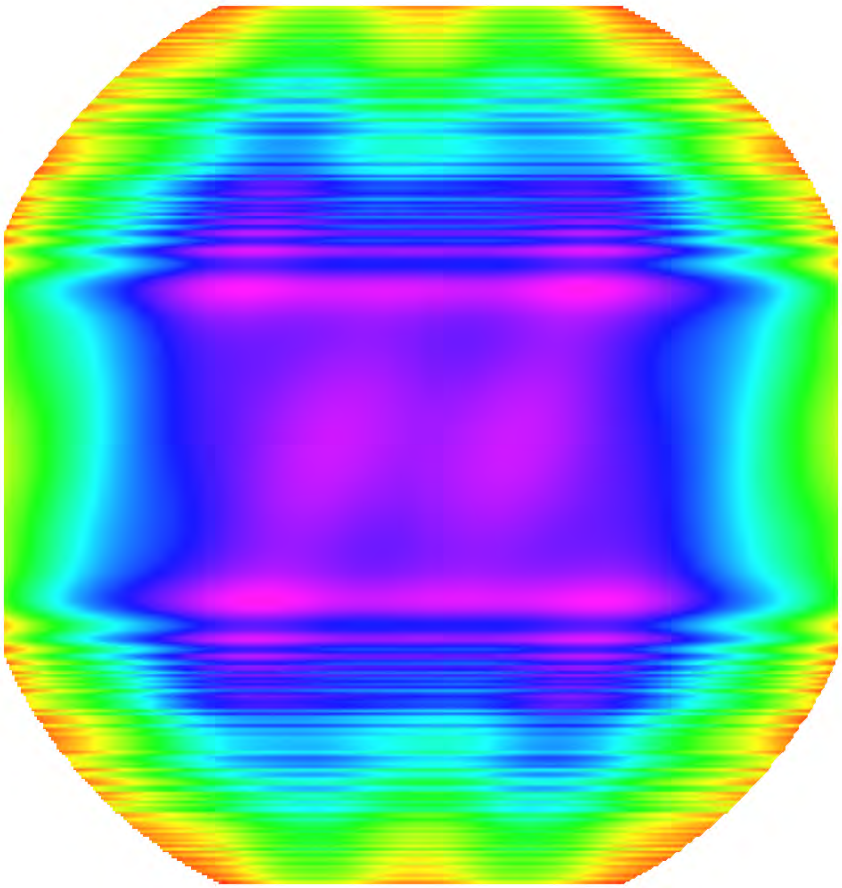


Abbildung 1: Plot der Funktion auf $[-3,3]^2$. Erstellt mit MAPLE

1.2 Eine grobe Abschätzung

Global gesehen dominiert der der parabolische Term f_5. Daher werden im ersten Schritt alle anderen Terme (grob) nach oben bzw. unten abgeschätzt:

$$\begin{array}{rcccl} sin(-1) & \leq & f_1(x,y) & \leq & sin(1) \\ cos(1) & \leq & f_2(x,y) & \leq & cos(0) = 1 \\ -1 & \leq & f_3(x,y) & \leq & 1 \\ e^{-1} & \leq & f_4(x,y) & \leq & e \end{array}$$

$\Rightarrow g_{min}(x,y) := sin(-1) + cos(1) - 1 + e^{-1} + x^2 + y^2 \leq f(x,y)$

$\Rightarrow g_{max}(x,y) := sin(1) + 1 + 1 + e + x^2 + y^2 \geq f(x,y)$

Klarerweise gilt $\min f(x,y) \leq \min g_{max}(x,y) = sin(1) + 2 + e$.

Desweitern gilt für $x^2+y^2 \geq sin(1)+2+e-sin(-1)-cos(1)+1-e^{-1} \geq 6.5$ dass $\min g_{max}(x,y) \leq g_{min}(x,y) \leq f(x,y)$

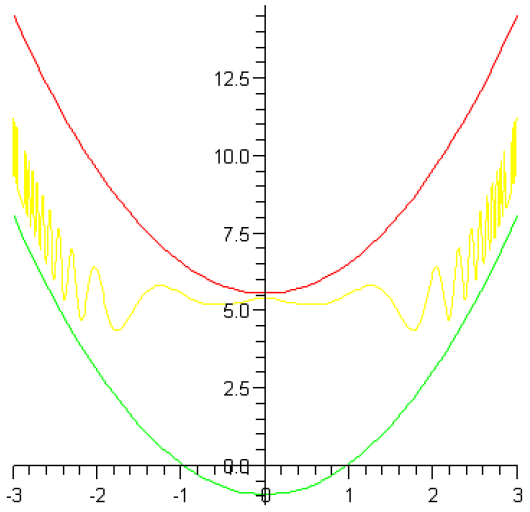


Abbildung 2: Grundidee des Verfahrens. Erstellt mit MAPLE

Das Minimum von $f(x,y)$ liegt daher innerhalb der Kreisscheibe mit Radius $r_0 = 2.55$. Wendet man dieses Wissen für die Abschätzung von f_4 nach unten an so erhält man $f_4(x,y) \geq e^{cos(r_0)}$ und mit obigen Überlegungen einen neuen Radius r_1. Sukzessives einsetzen ergibt für $r_{20} \leq 2.534$. Diese Zahl wurde algorithmisch mit Maple ermittelt, wobei erst ganz am Schluss (auf)gerundet wurde.

1.3 Allgemeine Bemerkungen

Man erkennt sofort dass $f(x,y) = f(-x,-y)$. Daher ist für die weitere Suche nur die Betrachtung von $O := K_{2.534}(0)\backslash\mathbb{R} \times \mathbb{R}^+$ notwendig. Lässt man sich die Funktion über diesem Gebiet plotten so erkennt man dass die bis jetzt gewonnenen Einschränkungen nicht ausreichen werden um Eigenschaften der Funktion herzuleiten. Auch ein Newtonverfahren macht an dieser Stelle noch wenig Sinn da $f(x,y)$ auf O noch sehr viele lokale Minima hat die alle als potentielle Nullstelle des Gradienten in Frage kommen.

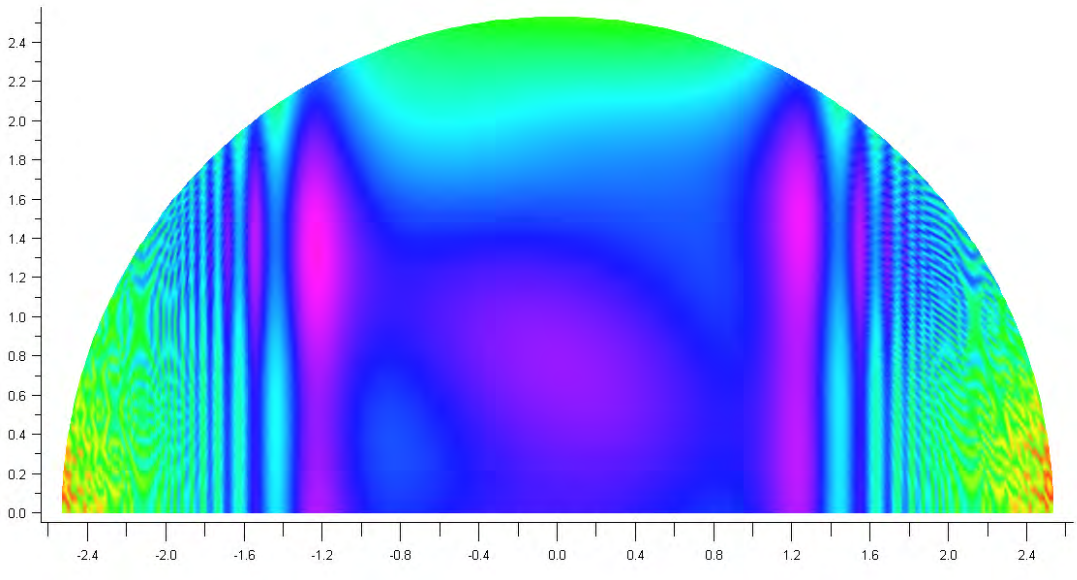


Abbildung 3: $f(x,y)$ auf O. Farben entsprechen der Höhe.

1.4 Einschränkung des Minimums mittels Raster

Im weiteren Verlauf wird die Vorgehensweise nahezu immer die gleiche sein.
Man zerteilt das Gebiet in Rechtecke und schätzt auf denselben das Minimum der Funktion nach unten und oben ab. Anschliessend schränkt man die weitere Suche auf Rechecke ein auf denen die untere Schranke für das Minimum kleiner oder gleich der kleinsten oberen Schranke ist.

1.5 Erster Schritt

Im ersten Schritt wird $\min_{[x0,x1]\times[y0,y1]} f(x,y) = \min_{[x0,x1]\times[y0,y1]} \sum_{i=1}^{5} f_i(x,y)$ durch $\sum_{i=1}^{5} \min_{[x0,x1]\times[y0,y1]} f_i(x,y)$ nach unten, und durch eine beliebige Funktionsauswertung $f(x_a, y_a)$ mit $(x_a, y_a) \in [x0, x1] \times [y0, y1]$ nach oben abgeschätzt. Der Code wurde aus Laufzeitgründen in C++ implementiert.
Es sei an dieser Stelle noch darauf hingewiesen dass der Algorithmus keinen verfahrenseigenen Fehler besitzt!
In diesem Code wurde die 'ganz normale' 8-byte-double Arithmetik verwendet und nach jedem Schritt nach unten bzw oben gerundet (Dh. das Intervall wieder geringfügig vergrössert um dem Rechenfehler entgegenzuwirken). Nach ca. 5 Minuten Rechenzeit ergibt sich eine weitere Einschränkung des zu betrachtenden Gebiets bzw des Minimums.

z0	z1	x0	x1	y0	y1
		-2.53400000	2.53400000	0.00000000	2.53400000
3.9515094500	4.0500546708	-1.26700000	-1.19098000	1.15297000	1.50773000
4.0271889673	4.0475880117	-1.25342500	-1.21677250	1.25433000	1.41270500
4.0380054767	4.0475842355	-1.24731625	-1.22288125	1.28072583	1.38924204
4.0427459667	4.0475831430	-1.24396899	-1.22656324	1.29707759	1.37586333
4.0451490468	4.0475827439	-1.24148245	-1.22921554	1.30908304	1.36460822
4.0463655943	4.0475827021	-1.23982476	-1.23103900	1.31733678	1.35635448
4.0469835924	4.0475826936	-1.23854005	-1.23240660	1.32322624	1.35064906
4.0472834843	4.0475826916	-1.23767561	-1.23331221	1.32736727	1.34669208
4.0474338951	4.0475826903	-1.23703605	-1.23397239	1.33024543	1.34390528
4.0475083812	4.0475826900	-1.23659034	-1.23442323	1.33225558	1.34189514
4.0475455637	4.0475826899	-1.23627489	-1.23474381	1.33367014	1.34049198
4.0475641168	4.0475826899	-1.23605195	-1.23496802	1.33467487	1.33949868
4.0475734069	4.0475826899	-1.23589298	-1.23512763	1.33538517	1.33879408
4.0475780491	4.0475826899	-1.23578122	-1.23524003	1.33588438	1.33829629
4.0475803691	4.0475826899	-1.23570215	-1.23531927	1.33623822	1.33794317
4.0475815303	4.0475826899	-1.23564613	-1.23537553	1.33648804	1.33769370
4.0475821101	4.0475826899	-1.23560657	-1.23541517	1.33666466	1.33751709

Im weiteren wird also das Gebiet $G := [-1.2357, -1.2354] \times [1.3366, 1.3376]$ betrachtet.

1.6 Genauere Betrachtung

Um die Funktion auf G genauer (schneller) abschätzen zu können ist es zweckmässig weitere Eigenschaften derselben herzuleiten. Besonders Konvexität spielt in diesem Zusammenhang eine wichtige Rolle:

$$f \text{ konvex auf } G \Leftrightarrow \text{Die Hessematrix } H(x,y) \text{ von } f(x,y) \text{ ist positiv definit auf } G.$$

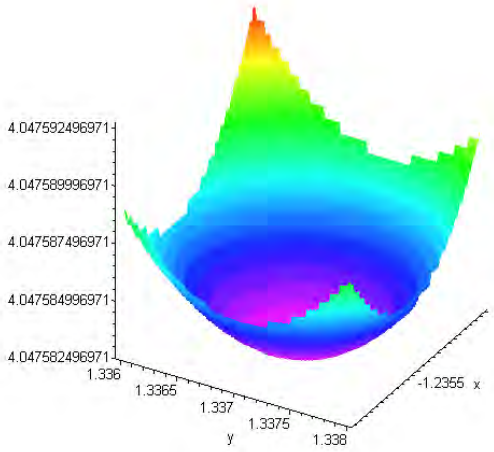


Abbildung 4: $f(x, y)$ auf G.

Das kann man mit Hilfe des Hauptminorenkriteriums ($H_{1,1}(x, y) > 0$ und $|H(x, y)| > 0$) sehr schnell überprüfen.
Nun lässt sich aber eine konvexe Funktion durch die Tangentialebene an einem beliebigen Punkt punktweise nach unten abschätzen. Eine Ebene nimmt ihe Minimum auf G sicher am Rand an. Als Wahl für den Berührpunkt der Tangentialeben erscheint der Mittelpunkt von G sinnvoll. Die Abschätzung des Minimums nach oben erfolgt wie schon zuvor durch Funktionsauswertung an einem beliebigen Punkt. Wieder gibt es hier keinen Verfahrensfehler allerdings muss man sich an dieser Stelle überlegen wie der Rechenfehler in den Griff zu bekommen ist. Abhilfe schaffen hier diverse floating-point-libraries. Für die konkrete Implementierung (C++) wurde das ARPREC-package gewählt. Nach ca. 2,5 Stunden Rechenzeit ergeben sich als die ersten 1000 richtige Stellen (obere und untere Schranke stimmen überein):

4.047582689914079035163436263727607019200376712399634318946696854441857011323664389166066148172420037959989500174186350048276540996811902395370580707405584272345755599240897130750710474611124352149960383994943569113585125425676681163677930781779687882368127107673779378470172171136961655093809778270422390193624676657816128197024638480051085957897230654103858926087167858996936282114590764143402296571679941462394790733736678574126170371552331752266048333418319471584822689514086129693845441056274875105414909149790723702107320323293655416471333334687763689771696467150973617278385138921245069318977990578754367458746919866008914326268324074890302687979806068606752581568013996793061069206876658987783542694129570437330308955365126801224435001791659691600675469455744761789502876609517500967098288214907539064841077532152978022172693403088512845575754646210646135721258387558122131099598579514737090352676453002892687025996692784545400851664339607046526412882158935466013281794685997115521294908957309016643396070465264128821589354660132817946859971155212949089573090

Gerechnet wurde intern mit einem 1500-Nachkommastellen-floating-point-typ. Das ausrechnen weiterer Ziffern mittels obigem Algorithmus wäre prinzipiell kein Problem, allerdings wird die Sache im Vergleich zu anderen Verfahren (zB Newton-Verfahren angewandt auf den Gradienten) recht langsam.

1.7 Newton

Zur Überprüfung des obigen Ergebnisses liegt die Verwendung eines Newton-Verfahrens nahe. Deutlich schneller als mit dem obigen Verfahren bricht der Algorithmus (implementiert in C++ wieder unter Zuhilfenahme des ARPREC-Packets) bereits nach wenigen Minuten ab. Als Abbruchbedingung wurde $|f(x_n, y_n) - f(x_{n+1}, y_{n+1})| < \epsilon = 10^{-1500}$ gewählt. Die Funktion f entspricht hier der Definition von oben und nicht(!) dem Gradienten (auf den das Verfahren angewendet wurde). Gerechnet wurde intern mit 2000 Nachkommastellen. Zum Zeitpunkt des Abbruchs Stimmen die ersten 1000 Stellen mit den zuvor gefundenen überein.
Da das Verfahren nur als Kontrollinstrument verwendet wurde, wurde auf eine exakte Fehlerabschätzung verzichtet. Heuristisch beeindruckend sind vielleicht aber die folgenden (zum Zeitpunkt des Abbruchs berechneten) Werte:

$$
\begin{gathered}
|f(x_n, y_n) - f(x_{n+1}, y_{n+1})| < 1.2 \cdot 10^{-1600} \\
|x_n - x_{n+1}| \approx |x_n - x_{n+1}| < 8.0 \cdot 10^{-847} \\
|\tfrac{\partial f}{\partial x}(x_n, y_n) - \tfrac{\partial f}{\partial x}(x_{n+1}, y_{n+1})| \approx |\tfrac{\partial f}{\partial y}(x_n, y_n) - \tfrac{\partial y}{\partial x}(x_{n+1}, y_{n+1})| < 9.1 \cdot 10^{-845} \\
|\tfrac{\partial^2 f}{\partial (x,y)^2}(x_{n+1}, y_{n+1})| < 9.0 \cdot 10^{2}
\end{gathered}
$$

1.8 Anmerkungen

Dieses erste Beispiel ist wahrscheinlich als die 'leichteste' der fünf Aufgaben zu bezeichnen. Es wurde ohne Zuhilfenahme von weiterführender Literatur gelöst und im Prinzip haben relativ

einfache Überlegungen zum gewünschten Resultat (1000 Nachkommastellen) geführt.
Auch die (streng genommen nicht notwendige) Überprüfung der gefundenen Stellen konnte mittels eines Standardalgorithmus ohne großen Aufwand durchgeführt werden. Nach den Überlegungen in Abschnitt 3 tritt das gefundene Minimum natürlich an zwei voneinander verschiedenen Stellen auf.
Zum eingeschlagenen Lösungsweg: Natürlich mag es einfacher erscheinen als erste Wahl ein Newtonverfahren auf dem Gradienten zu benützen. Zugegebenerweise war das wesentlich schneller zu programmieren und auch die Abarbeitung des kompilierten Programms hat wesentlich kürzer gedauert. Hier muss sich jedoch mit Fehlerabschätzungen des Verfahrens herumgeschlagen werden weswegen von mir die hier vorgeschlagene Methode favourisiert wurde.

2 Aufgabe

Birgit Hischenhuber

2.1 Aufgabenstellung

Lösen Sie das Integral

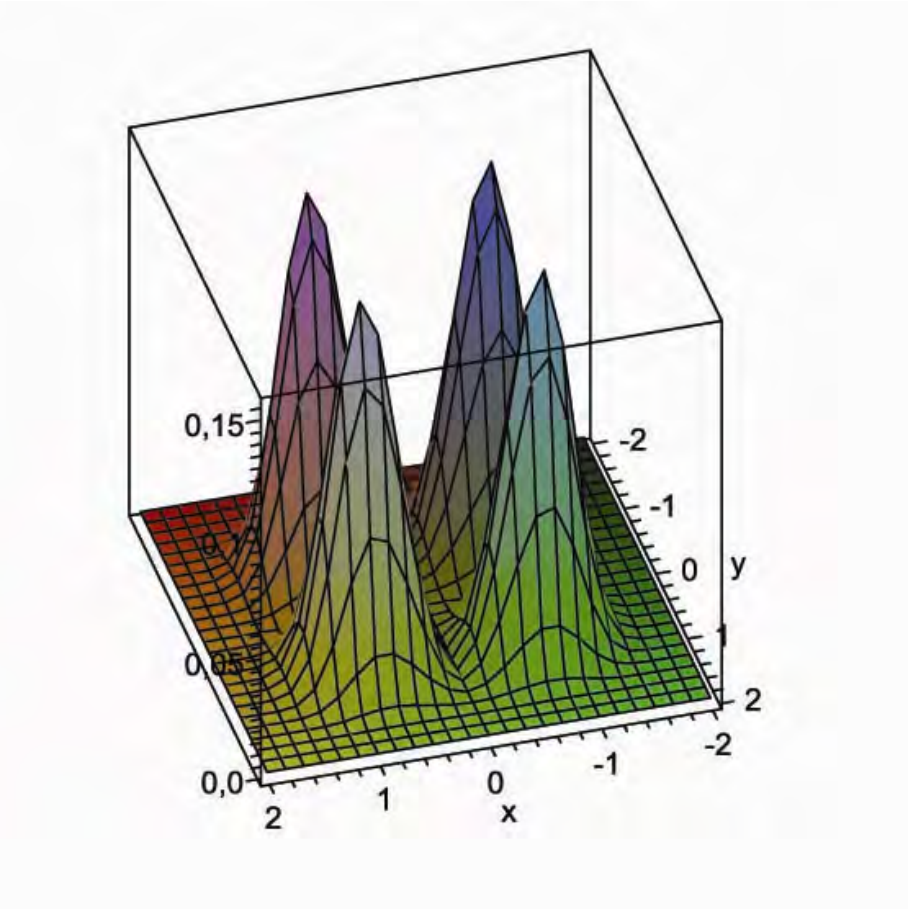
$$I = \int_{R^6} \exp^{-x_1^4 - x_2^4 - x_3^4 - x_4^4 - x_5^4 - x_6^4} \sin^2(x_1 x_2 x_3 x_4 x_5 x_6) dx,$$

wobei $x = (x_1, \ldots, x_6)$.

2.2 Abschätzung

in Zusammenarbeit mit Stefan Schuchnigg

Wenn man den Integranten in zwei Variablen plottet, kann man erkennen, dass man das Integral nicht bis ∞ berechnen muss, sondern durch einen geeigneten Würfel approximieren kann:



Ziel der folgenden Abschätzung ist es einen geeigneten Würfel zu finden, sodass der Fehler genügend klein wird:
$f(x)$ bezeichne im folgenden den Integranten.

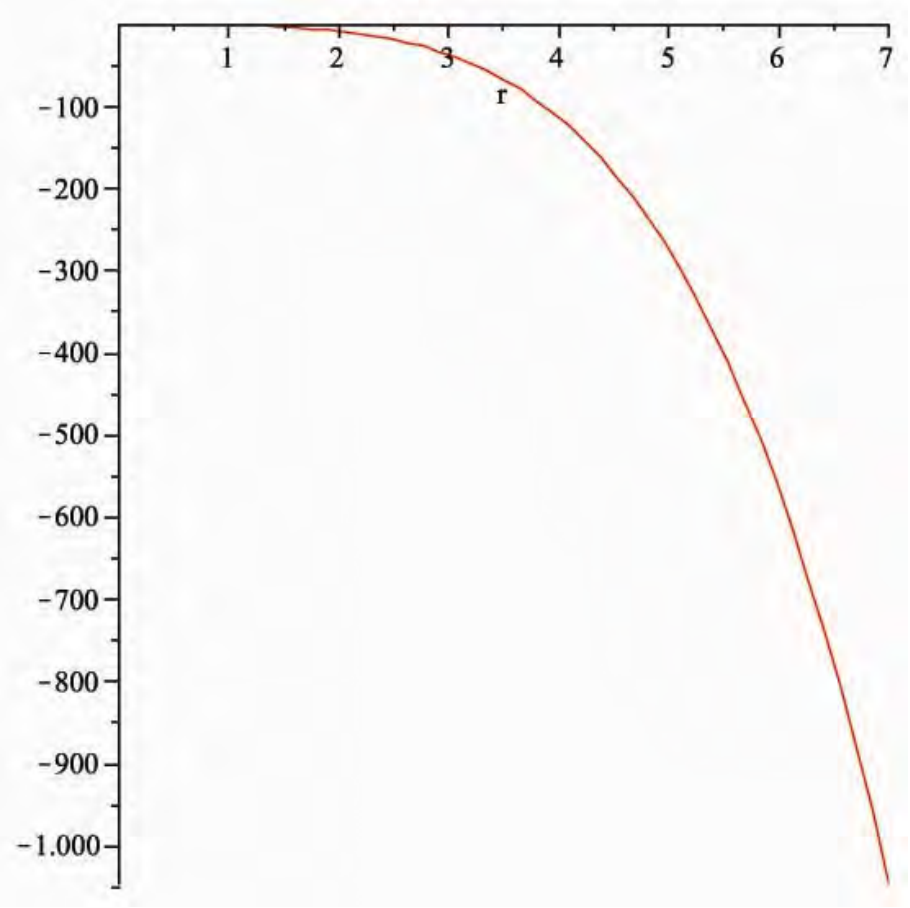
$$\int_{(R^+)^6} f(x)dx - \int_{[0,r]^6} f(x)dx =$$

$$\int_{max(x_i)\geq r} f(x)dx \leq \int_{max(x_i\geq r)} e^{-\sum_{i=1}^{6} x_i^4} dx =$$

$$6\int_r^\infty (\int_0^{x_1} \dots \int_0^{x_1} e^{-\sum_{i=1}^{6} x_i^4} dx_2 ... dx_5) dx_1 = 6\int_r^\infty (\int_0^y e^{-t^4})^5 e^{-y^4} dy =$$

$$6(\int_0^y e^{-t^4} dt)^6 |_r^\infty - 30(\int_0^y e^{-t^4} dt)^5 e^{-y^4}$$

$$\Rightarrow 6\int_r^\infty (\int_0^y e^{-t^4} dt)^5 e^{-y^4} dy = (\int_0^y e^{-t^4} dt)|_r^\infty = (\int_0^\infty e^{-t^4} dt)^6 - (\int_0^r e^{-t^4} dt)^6$$



Aus dieser Abbildung kann man nun erkennen, welchen r man wählen muss, um den Fehler beliebig klein zu machen.
Im folgenden wurde $r = 6$ gewählt und das Integral der Angabe mit folgendem Integral ersetzt:

$$I = \int_{R^6} \exp^{-x_1^4-x_2^4-x_3^4-x_4^4-x_5^4-x_6^4} \sin^2(x_1x_2x_3x_4x_5x_6)dx \approx$$

$$\int_{-6}^{6}\int_{-6}^{6}\int_{-6}^{6}\int_{-6}^{6}\int_{-6}^{6}\int_{-6}^{6} \exp^{-x_1^4-x_2^4-x_3^4-x_4^4-x_5^4-x_6^4} \sin^2(x_1x_2x_3x_4x_5x_6)dx_1dx_2dx_3dx_4dx_5dx_6 =$$

$$64\int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6} \exp^{-x_1^4-x_2^4-x_3^4-x_4^4-x_5^4-x_6^4} \sin^2(x_1x_2x_3x_4x_5x_6)dx_1dx_2dx_3dx_4dx_5dx_6$$

Im folgendem sei $I1$ wie folgt definiert:

$$I_1 = \int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6}\int_{0}^{6} \exp^{-x_1^4-x_2^4-x_3^4-x_4^4-x_5^4-x_6^4} \sin^2(x_1x_2x_3x_4x_5x_6)dx_1dx_2dx_3dx_4dx_5dx_6$$

2.3 Maple

Gibt man das Integral I in Maple ein, liefert Maple folgenden Ausdruck:

$$\int_{-\infty}^{\infty} \frac{1}{128} \exp -x_6^4\pi^{\frac{3}{2}}(x_6^3\text{MeijerG}([[\frac{3}{4}],[1,\frac{5}{4}]],[[\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}],[]],-\frac{16}{x_6^4})+$$

$$+Ix_6^3\text{MeijerG}([[\frac{3}{4},[1,\frac{5}{4}]],[[\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}],[]],-\frac{16}{x_6^4})+$$

$$+2\sqrt{2}\text{MeijerG}([[0],[\frac{1}{2},\frac{3}{4},1]],[[\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},0],[]],-\frac{16}{x_6^4})$$

$$\sqrt{(Ix_6^2)}\text{csgn}(f)(-x_6^4)^{\frac{1}{4}})\text{csgn}(f))dx_6$$

Die Meijer G Funktion ist definiert, als die inverse Laplace Transformation:

$$\text{MeijerG}([as,bs],[cs,ds],z)=\frac{1}{2\pi I}\int_L \frac{\Gamma(1-as+y)\Gamma(cs-y)}{\Gamma(bs-y)\Gamma(1-ds+y)}z^y dy$$

mit $\Gamma(1-as+y)=\Gamma(1-a1+y)\ldots\Gamma(1-am+y)$.
Die *csgn* Funktion ist die Signum Funktion für reelle und komplexe Ausdrücke. Sie wertet aus, in welcher Halbebene sich der Ausdruck befindet:

$$\text{csgn}(x)=\begin{cases}1, \text{ falls } \Re(x)>0 \text{ oder } \Re(x)=0 \text{ und } \Im(x)>0\\ -1, \text{ falls } \Re(x)<0 \text{ oder } \Re(x)=0 \text{ und } \Im(x)<0\end{cases}$$

Wie man nun erkennt ist die ursprüngliche Integraldarstellung besser geeeignet, um das Integral (mit numerischen Verfahren) zu lösen.
In Maple gibt es die Möglichkeit mit dem Befehl *eval* ein Problem numerisch lösen zu lassen. Für mehrdimensionale Integrale sind 2 Methoden implementiert: *_cuhre* und *_MonteCarlo*.

Will man das Integral $I=\int_a^b g(x)dx$ berechnen, so kann man das Integral umformen, sodass das Integral als Erwartungswert der Funktion g der Zufallsvariablen x interpretiert werden kann:

$$I=(b-a)\int f(x)g(x)dx \text{ mit } f(x)=\begin{cases}\frac{1}{b-a}, \text{ wenn } a\leq x\leq b\\ 0, \text{ sonst}\end{cases}$$

Ist das Integral nicht analytisch berechenbar, so wird eine Stichprobe $\xi_1\ldots\xi_n$ der Variablen x erzeugt und der Erwartungswert von g(x) durch den Stichprobenmittelwert approximiert:

$$I\approx(b-a)\frac{1}{n}\sum_{i=1}^{n}g(\xi_i)$$

Dieses Verfahren heißt *naive MC-Integration*. Der Fehler ist proportional zu $1/\sqrt{n}$.
Diese Methode ist für unsere Aufgabenstellung ungünstig, da man für die Sicherheit von 4 Nachkommastellen, 1000^2 zusätzliche Punkte bräuchte. Um mit dieser Methode zu brauchbaren Ergebnissen zu kommen, müsste man lange Rechenzeiten in Kauf nehmen, die ich mir aber ersparen wollte und daher zu anderen Methoden gegriffen habe.

Die Cuhre-Methode ist für 2-15 dimensionale Integrale geeignet. Die Methode beruht auf *adaptiver Kubatur*. Es wird der Integrationsbereich in elementare Dreiecke $P_0P_1P_2$ mittels Triangulierung zerlegt. Das Integral wird nun über einem Dreieckbereich mittels der *Prismenregel*

$$P(h)=\frac{|Dreieck|}{3}(f(P_0)+f(P_1)+f(P_2)), h...maximaleSeitenlänge$$

berechnet. Neben diesem Grobwert wird nun ein Feinwert berechnet, indem man die Seiten des Dreiecks halbiert und die Prismenregel auf die vier entstehenden kongruenten Teildreiecke

anwendet. Damit lässt sich nun der Fehler des Feinwertes mittels der asymptotischen Fehlerabschätzung

$$Error := \frac{1}{3}(P(h/2) - P(h))$$

approximieren. Ist $|Error| < \text{tol}(|P(h/2)|+1)$, so wird $P(h/2)$ als ein geeigneter Näherungswert akzeptiert.[Lit 1]
In Maple kann man die Digits angeben, die genau berechnet werden sollen. Maple kann bei diesem Problem nur bis zu 5 Digits berechnen. Wenn man 6 Digits berechnen will, rechnet Maple einige Zeit lang und gibt dann die Integraldarstellung aus.

Numerische Berechnung des Integrals $I1$ liefert:

$$I_1 = \mathbf{0.0007}8687$$

Wenn man Maple vertraut, so sind die fettgedruckten Ziffern richtig. Ich hole mir allerdings noch mit anderen Methoden, die im Anschluss beschrieben werden, Referenzergebnisse, um die fettgedruckten Ziffern zu verifizieren.
Wenn man nun I_1 mit 64 multipliziert, bekommt man:

$$I = \mathbf{0.05}0368$$

Dies liefert einstweilen eine Vermutung für die ersten 3 Ziffern der reellen Zahl, die gesucht ist.

2.4 Komplexe Integration

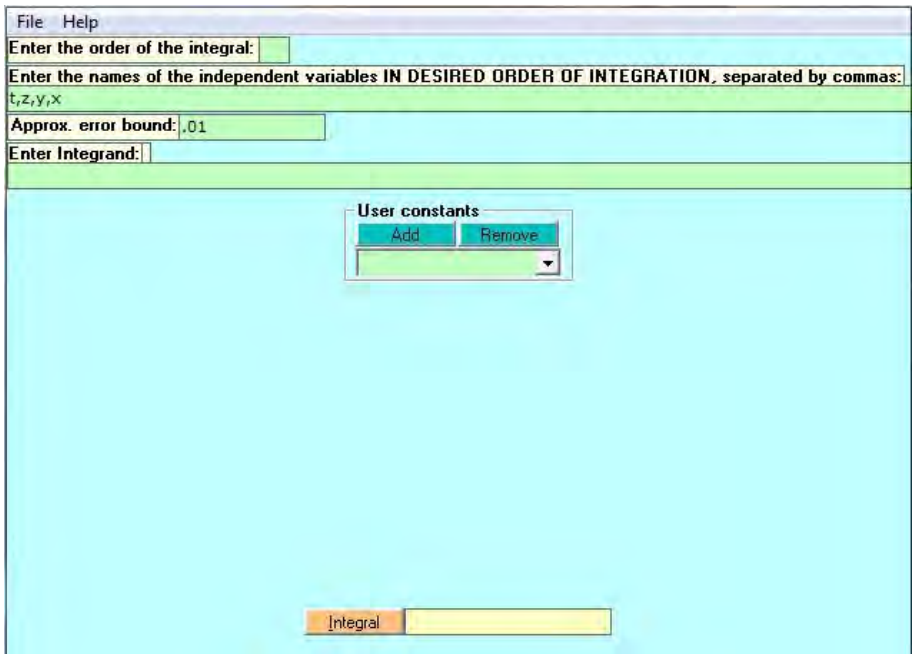
Betrachtet man den Integranten, kann man sich überlegen, ob es eine Möglichkeit gibt das innerste Integral mittels komplexer Integration zu lösen, um dann auf einen Ausdruck zu kommen, der leichter zu bearbeiten ist. Der Ansatz besteht darin die auf ganz C analytische Funktion

$$f(z) = \exp^{-z^4}$$

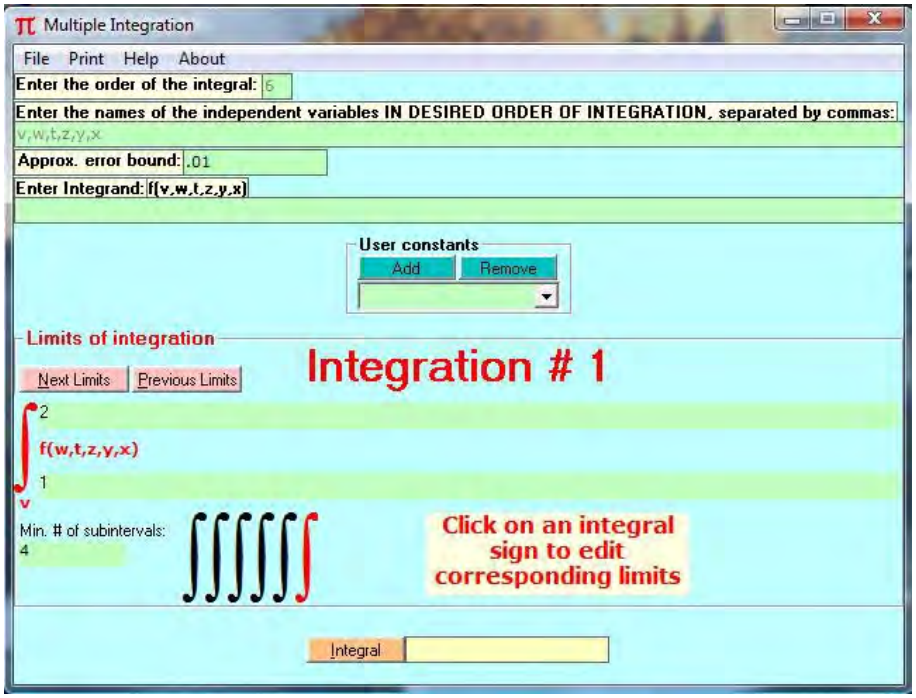
auf einem rektifizierbaren geschlossenen (nullhomologen) Weg in C zu berechnen. Ich habe es zuerst auf einem Rechteck probiert, bei dem der Radius dann gegen ∞ konvergiert (-so wie man die Fresnelschen Integrale berechnet-), doch das hat zu keinem brauchbaren Ergebnis geführt. Auch eine andere Wegwahl hat zu keinen brauchbaren Erkenntnissen geführt.

2.5 Multiple Integration

'Multiple Integration' [Lit 2] ist ein Programm zur numerischen Berechnung von mehrdimensionale Integrale. Die Benutzeroberfläche ist sehr einfach zu bedienen:



Auf der Benutzeroberfläche wird die Ordnung des Integrals, die unabhängigen Variablen in gewünschter Ordnung der Integration, der maximale Fehler, welchen man in der Berechnung haben möchte, der Integrant, die Integrationsgrenzen, eingegeben.



Das Programm generiert eine sukzessive Approximationsfolge, indem es das Integral berechnet, dann nocheinmal mit der doppelten Anzahl von Teilintervallen, u.s.w.
Dieser Prozess wird fortgesetzt bis die aktuelle Approximation in der Folge sich von der vorigen Approximation um weniger als der maximale Fehler unterscheidet.

Die Methode, die verwendet wird, ist die 4-Punkt Newton-Cotes-Formel mit dazupassenden Raumpunkten.
Um $f(x)$ in $[x_0, x_4]$ zu approximieren, wird das Polynom $f(x) = ax^4 + bx^3 + cx^2 + dx + e$ verwendet. Wenn man dieses Polynom analytisch integriert, bekommt man eine Approximation des Integrals von $f(x)$ von x_0 bis x_4:

$$\int_{x_0}^{x_4} f(x)dx \approx 2h/45[7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)], h = (x_4 - x_0)/4$$

Die ausgewählten Punkte sind äquidistant.
Für n Teilintervalle folgt:

$$\int_{x_0}^{x_n} f(x)dx \approx 2h/45[7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + \cdots + 7f(x_n)], h = (x_n - x_0)/4$$

Bei mehrdimensionalen Integralen fasst man die einzelnen Teilintegrale als eindimensionale Integrale auf und geht analog zu oben vor.

Dieses Programm liefert dieselben Erkenntnisse, wie Maple:
Bei einem Toleranzwert von 0.00001 liefert das Programm ein Ergebnis:

$$I_1 = 8.01239224818389 * 10^{-4}$$

Multiplizieren mit 64 liefert:

$$I = \mathbf{0.05}1264$$

2.6 Mathematica

Mathematica hat eine Funktion zum numerischen Lösen von Integralen (sowohl eindimensional, als auch mehrdimensional): *NIntegrate*.
Man kann für diese Funktion verschiedene Methoden wählen, mit welcher das Integral numerisch berechnet werden soll. Es gibt 2 sinnvolle Methoden, die für das 6-dimensionale Integral geeignet sind: *GlobalAdaptive*, *LocalAdaptive*. (Die anderen Methoden funktionieren nur für eindimensionale Integrale.)
Außerdem ist es möglich eine Fehlerschranke vorzugeben, mit Hilfe von *AccuracyGoal* und *PrecisionGoal*.

GlobalAdaptive ist eine Methode, bei der globale Fehlerschätzungen durchgeführt werden. Die Methode liefert schneller ein Ergebnis, als *LocalAdaptive*.
Beim *LocalAdaptive*, wird der Fehler lokal, also bei jeder Teilberechnung abgeschätzt. Dadurch dauert diese Berechnung länger.
AccuracyGoal ist eine Option, mit der man angeben kann, wieviele Stellen im Ergebnis genau sein sollen. *AccuracyGoal* spezifiziert den absoluten Fehler, der in einer numerischen Prozedur erlaubt ist.
PrecisionGoal spezifiziert den relativen Fehler, der in einer numerischen Prozedur erlaubt ist. Das heißt, dass im Ergebnis nicht unbedingt alle n Stellen richtig sein müssen.
Wenn man nun $PrecisionGoal- > p$ und $AccuracyGoal- > a$ setzt, versucht Mathematica den numerischen Fehler in einem Ergebnis der Größe x kleiner zu sein, als $10^{-a} + |x|10^{-p}$.

Normalerweise wird mit *GlobalAdaptive* gearbeitet, da sie für die meisten Probleme schneller zu genauen Ergebnissen führt. Bei unserem Integral gibt es allerdings Probleme mit dieser Methode:
Bei den Berechnungen erscheinen 2 Fehlermeldungen: *NIntegrate::eincr* und *NIntegrate::slwcon*
Diese 2 Fehlermeldungen sagen folgendes aus: Unser Integrand osziliert zu stark für diese Methode. Die numerische Integration konvergiert zu langsam. Bei diesem speziellen Integranden ist also die andere Methode zu bevorzugen:

Versuchsprotokoll:
Es wurde numerisch dasselbe Integral berechnet, wie bereits in Maple und anschließend mit dem Faktor 64 multipliziert.
Im folgenden wurde die Methode *LocalAdaptive* verwendet und jeweils die Fehlerschranken verändert:

- $PrecisionGoal- > 6, AccuracyGoal- > 6$:

$$I1 = \mathbf{0.00078}6597$$

$$I = \mathbf{0.0503}42208$$

- $PrecisionGoal- > 10, AccuracyGoal- > 10$:

$$I1 = \mathbf{0.0007868}72$$

$$I = \mathbf{0.050359}808$$

- $PrecisionGoal- > 11, AccuracyGoal- > 11$:

$$I1 = \mathbf{0.000786874}$$

$$I = \mathbf{0.0503593}6$$

- $PrecisionGoal- > 12, AccuracyGoal- > 12$:

$$I1 = \mathbf{0.000786874247}772$$

$$I = \mathbf{0.0503599}518574$$

- $PrecisionGoal- > 15, AccuracyGoal- > 15$:

$$I1 = \mathbf{0.000786874512}4101048$$

$$I = \mathbf{0.0503599687}9424671$$

An der letzten Berechnung hat der PC bereits 30 Stunden gerechnet.

3 Aufgabe

Sonja Höllrigl-Binder

3.1 Aufgabenstellung

Die unendliche Matrix $A = (a_{ij})$, definiert durch

$$a_{ij} = \begin{cases} \frac{1}{i+j-1}, & i+j \text{ ist Primzahl}, \\ 0, & \text{sonst}, \end{cases} \qquad i,j \geq 1$$

bildet einen unendlich-dimensionalen Operator auf ℓ^2. Welchen Wert hat

$$a = \|A\| = \sup_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2},$$

wobei $\|x\|_2^2 = \sum_{i=1}^{\infty} x_i^2$?

Die Matrix A ist eine Hankel Matrix und hat folgende Gestalt:

$$A = \begin{pmatrix} 1 & \frac{1}{2} & 0 & \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & \dots \\ \frac{1}{2} & 0 & \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \dots \\ 0 & \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & \dots \\ \frac{1}{4} & 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & \dots \\ 0 & \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & 0 & \dots \\ \frac{1}{6} & 0 & 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & 0 & 0 & \frac{1}{16} & \dots \\ 0 & 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & 0 & 0 & \frac{1}{16} & 0 & \dots \\ 0 & \frac{1}{10} & 0 & \frac{1}{12} & 0 & 0 & 0 & \frac{1}{16} & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Im Folgenden werden wir versuchen das Problem mit verschiedenen Ansätzen zu "lösen". Wir werden dabei ähnlich vorgehen wie im Buch "Vom Lösen numerischer Probleme".

3.2 Approximation durch n-dimensionale Untermatrizen

Wie das folgende Lemma zeigt, kann der unendlichdimensionale Operator A durch seine n-dimensionalen Untermatrizen A_n approximiert werden.

Lemma 3.1. *Für $1 \leq n \leq m$ gilt*

$$\|A_n\| \leq \|A_m\| \leq \lim_{k \to \infty} \|A_k\| = \|A\| \leq \pi$$

Beweis. $P_n : \ell^2 \to \text{span}\{e_1, \dots, e_n\} \subset \ell^2$ ist die Orthogonalprojektion von ℓ^2 auf den n-dimensionalen Teilraum, der durch die ersten n Basisfolgen $(e_j)_k = \delta_{jk}$ aufgespannt wird.

Diesen Teilraum können wir mit $\mathbb{R}^n$ identifizieren und erhalten $A_n = P_n A P_n$. Für $n \leq m$ gilt $P_n = P_n P_m = P_m P_n$ und daher $A_n = P_n A_m P_n$. Da ℓ^2 eine normierte Algebra (sogar eine Banachalgebra) ist und eine Orthogonalprojektion $\|P_n\| \leq 1$ erfüllt, kommen wir auf

$$\|A_n\| \leq \|P_n\|^2 \|A_m\| \leq \|A_m\| = \|P_m A P_m\| \leq \|P_m\|^2 \|A\| \leq \|A\|$$

Die Folge $\|A_n\|$ ist also monoton wachsend.
Im Folgenden werden wir beweisen, dass $\|A\| < \infty$ und die Folge daher beschränkt ist.
In diversen Publikationen wird bewiesen oder darauf hingewiesen, dass die Norm der Hilbert-Matrix $H = (h_{ij})$ gleich oder zumindest $\leq \pi$ ist ("An Excursion into the Theory of Hankel Operators" von Vladimir V. Peller, "Treatise on the shift operator" von Nikolski). Diese Tatsache erlaubt es uns, $\|A\|$ abzuschätzen.

Bemerkung Die einzelnen Komponenten der Hilbert-Matrix sind durch $h_{ij} = \frac{1}{i+j-1}$ gegeben.

Sei $x = (x_1, x_2, \ldots, x_n)$. Zuerst werden wir folgende Gleichheit zeigen:

$$\|A_n\| = \sup_{\|x\|=1, x_i \geq 0} \|A_n x\|$$

Berechnung der i-ten Komponente von $A_n(x)$: $(A_n x)_i = \sum_{j=1}^n a_{ij} x_j$
Da $a_{ij} \geq 0$ gilt, kann man die Norm von $A_n(x)$ folgendermaßen abschätzen:
$\sum_{j=1}^n a_{ij} x_j \leq \sum_{j=1}^n a_{ij} |x_j|$ für alle $i = 1 \ldots n$
$\Rightarrow \|A_n x\|^2 = \sum_{i=1}^n |\sum_{j=1}^n a_{ij} x_j|^2 \leq \sum_{i=1}^n |\sum_{j=1}^n a_{ij} |x_j||^2 =: \|A_n y\|$ für $y := (|x_1|, |x_2|, \ldots, |x_n|)$.
x und y haben natürlich die gleiche ℓ^2-Norm: $\|x\|^2 = \|y\|^2 = \sum_{i=1}^n |x_i|^2$.
$\|A_n x\| \leq \|A_n y\| \Rightarrow \sup_{\|x\|=1} \|A_n x\| \leq \sup_{\|y\|=1} \|A_n x\| = \sup_{\|x\|=1, x_i \geq 0} \|A_n x\|$
Da auf der rechten Seite das Supremum über weniger Elemente gebildet wird und somit $\sup_{\|x\|=1} \|A_n x\| \geq \sup_{\|x\|=1, x_i \geq 0} \|A_n x\|$ gelten muss, folgt die Gleichheit $\|A_n\| = \sup_{\|x\|=1, x_i \geq 0} \|A_n x\|$
Die gleiche Aussage gilt natürlich auch für Hilbertmatrizen.

Da $a_{ij} \leq h_{ij}$ für $\quad i, j \geq 1$ kommt man zu folgender Abschätzung:
$\|A_n x\|^2 \leq \|A_n y\|^2 = \sum_{i=1}^n |\sum_{j=1}^n a_{ij} |x_j||^2 \leq \sum_{i=1}^n |\sum_{j=1}^n h_{ij} |x_j||^2 \leq \sum_{i=1}^\infty |\sum_{j=1}^\infty h_{ij} |x_j||^2 = \|H\|^2$
Bildet man nun $\sup_{\|y\|=1}$, erhält man $\|A_n\| \leq \|H\| = \pi$.
Die Folge ist beschränkt und mononton wachsend, somit existiert der Grenzwert $\lim_{n\to\infty} \|A_n\| = \sup_n \|A_n\| \leq \pi$. Wegen der Vollständigkeit der Basisfolgen wissen wir, dass $lim_{n\to\infty} P_n x = x$ für alle $x \in \ell^2$ ist und daher

$$\|Ax\| = \lim_{n\to\infty} \|P_n A P_n x\| \leq \lim_{n\to\infty} \|A_n\| \|x\|.$$

Bildet man nun das Supremum über alle x mit $\|x\| = 1$, so erhält man also

$$\|A\| \leq \lim_{n\to\infty} \|A_n\|.$$

Damit folgt die Aussage des Lemmas. □

Anmerkung Die Abschätzung der Spektralnorm durch die Frobeniusnorm führt in diesem Fall nicht zum Erfolg, sie ist einfach zu grob.
Die Abschätzung sei hier kurz angeführt:
$\|A_n\|_F^2 = \sum_{p\leq n,p\text{prim}} \frac{1}{p-1} + \sum_{p\leq n-1,p\text{prim}} (n-p)\cdot\frac{1}{n+p} \geq \sum_{p\leq n,p\text{prim}} \frac{1}{p-1} \geq \sum_{p\leq n,p\text{prim}} \frac{1}{p}$
$\lim_{n\to\infty} \sum_{p\leq n,p\text{prim}} \frac{1}{p}$ divergiert. Man hat somit eine divergente Minorante für die Frobeniusnorm gefunden.

Wir werden uns also im Folgenden mit der Berechnung des Grenzwertes

$$\lim_{n\to\infty} \|A_n\|$$

beschäftigen.

3.3 Extrapolation

Matlab hat für die Erzeugung von Hankel-Matrizen den Befehl `hankel(c,r)` implementiert. Diese Art der Erzeugung benötigt jedoch sehr viel Speicherplatz, bedenkt man, dass die Hankel-Matrix $A_n \in \mathbb{R}^{n\times n}$ durch $2\cdot n-1$ Elemente festgelegt ist. Wird mit diesem Befehl eine Matrix der Dimension $n\times n$ erzeugt, so müssen n^2 Elemente abgespeichert werden. Bereits bei $n = 6400$ erhalten wir in Matlab die Ausgabe *OUT OF MEMORY*. Diese Art der Implemetierung wird uns also keine hohe Genauigkeit bringen, vor allem weil die Norm der Untermatrizen sehr langsam konvergiert, wie der folgenden Tabelle zu entnehmen ist.
Zur Berechnung der Spektralnorm einer Matrix A stehen in Matlab die Befehle `norm(A)` und `normest(A,tol)` zur Verfügung, der erste basiert auf einer Singulärwertzerlegung und der zweite auf einer Vektoriteration. Da die Laufzeit für den Befehl `norm(A)` im Vergleich zum anderen lang ist und die Ergebnisse bei einer relativen Genauigkeit von `tol=`$5\cdot 10^{-16}$ für unsere Zwecke vollkommen ausreichen, werden wir im folgenden `normest(A,tol)` verwenden.
Wir können nun mittels Extraploation versuchen einen Grenzwert für die Norm der A_n zu erhalten und somit einen Wert für die Norm von A. Wir verwenden hierfür den *Wynn'schen Epsilon-Algorithmus*.
Dieser hat folgende einfache Rekursionsformel:

$$s_{k,j} = s_{k+1,j-2} + \frac{1}{s_{k+1,j-1} - s_{k,j-1}}$$

Um die Rekursion zu starten, setzen wir $s_{k,-1} = 0$ und $s_{k,0} = s_k$. Die s_k sind hierbei unsere Folgenglieder.
Eine einfache Art der Implementierung (in Matlab) findet sich im bereits zuvor erwähnten Buch "Vom Lösen numerischer Probleme":

```
L2=2*L-1; vv=zeros(L2,1); vv(1:2:L2)=vals;
for j=2:L
k=j:2:L2+1-j; vv(k)=vv(k)+1./(vv(k+1)-vv(k-1));
```

```
end;
result=vv(1:2:L2);
```

`L` bezeichnet die Anzahl der Folgenglieder. Der Vektor `vals` beinhaltet die Normen. Die beste Approximation von $\|A\|$ befindet sich im Allgemeinen gerade unterhalb des Elements in der Mitte.

n	`vals`=$\|A_n\|$	Wynn für $L = 11$	Wynn für $L = 13$
1	**1**.00000000000000	**1**.00000000000000	**1**.00000000000000
2	**1**.20710678118655	**1**.28220799096267	**1**.28220799096267
4	**1**.26222205190192	**1**.25854528967671	**1**.25854528967671
8	**1**.28495256009120	**1.322**76111174632	**1.322**76111174632
16	**1.3**0588442860036	**1.322**31096891828	**1.322**31096891828
32	**1.3**1363872898456	**1.32**197663656323	**1.32**197663656323
64	**1.3**1782692832248	**1.32217**823206373	**1.32217**352307281
128	**1.32**007606106344	**1.3221**8219487426	**1.32217**248812495
256	**1.32**116798744673	**1.3221**5531902698	**1.32217**249098307
512	**1.32**167963990059	**1.32217**092331164	**1.32217**343242805
1024	**1.32**193027044597	**1.32**193027044597	**1.32217**371652842
2048	**1.322**05394898655		**1.32217**353330205
4096	**1.3221**1474747365		**1.3221**1474747365
6300	**1.3221**3522695743		

Wegen der langsamen Konvergenz unseres Problems führt auch eine Konvergenzbeschleunigung zu keinem zufriedenstellenden Ergebnis, sie liefert uns lediglich eine weitere Nachkommerstelle. Es liegt nahe, dass

$$\|A\| \doteq 1.32217$$

simmt. Und somit die ersten 6 Ziffern bekannt sind.
Um zu einem genaueren Ergebnis zu kommen, bedienen wir uns der Vektoriteration.

3.4 Vektoriteration

Zum Aufstellen der Matrix A, wie es im Kapitel zuvor geschehen ist, benötigt man viel Speicherplatz. Um dem zu entgehen, kommen wir zu einer iterativen Methode zur Berechnung der Norm, für die nur das Matrix-Vektor-Produkt benötigt wird.
Dieser wurde folgendermaßen implementiert

```
for v=1 to vmax do
y(v)=A*x(v-1);
normA=(x(v-1))'*y(v);
if normA hinreichend genau
then exit;
x(v)=y(v)/||y(v)||;
```

```
end for;
```

n	vals=$\|A_n\|$	Wynn für $L = 19$
1	**1**.0000000000000000	**1**.00000000000000
2	**1**.2071067811865472	**1**.28220799096267
4	**1**.2622220519019154	**1**.25854528967669
8	**1**.2849525600911980	**1.322**76111174633
16	**1.3**058844286003635	**1.322**31096891827
32	**1.3**136387289845550	**1.32**197663656316
64	**1.3**178269283224799	**1.32217**352307278
128	**1.32**00760610634397	**1.32217143**213951
256	**1.32**11679874467239	**1.32217145**216854
512	**1.32**16796399005866	**1.32217142**792024
1024	**1.32**19302704459643	**1.32217142**421448
2048	**1.322**0539489865497	**1.32217142**563509
4096	**1.3221**147474736465	**1.32217143**051038
8192	**1.3221**437110238849	**1.32217143**482620
16384	**1.3221**580001073130	**1.32217143**670312
32768	**1.3221**648794601844	**1.32217143**339913
65536	**1.3221**682428821693	**1.32217142**569420
131072	**1.3221**698812562113	**1.32217142**960023
262144	**1.32217**06772998514	**1.32217**067729985

Wie man sehen kann, liefert uns die Vektoriteration trotz beachtlicher Erhöhung von n nur eine zusätzliche Nachkommerstelle. Durch anschließende Extrapolation erhalten wir noch zwei Stellen.

Da die Berechnung von $\|A_{524288}\|$ 10 Stunden überschreitet und trotzdem keine neue Stelle zu erwarten ist, wurde die Berechnung bei $n = 262144$ abgebrochen.

Das Ergebnis lautet somit

$$\|A\| \doteq 1.3221714.$$

4 Beispiel

Birgit Hischenhuber

4.1 Angabe

Bestimmen Sie $c > 0$, so daß die Lösung von

$$-\Delta u = e^u \; in \; \Omega = (0,1)^2, \; u = c \; auf \; \Gamma = 0 \times (0,1), \; u = 0 \; auf \; \partial\Omega\Gamma$$

die Beziehung $u(\frac{1}{3}, \frac{1}{3}) = 1$ erfüllt.

4.2 COMSOL Multiphysics

Birgit Hischenhuber

Das ist ein Programm, das partielle Differentialgleichungen numerisch lösen kann, basierend auf FEM.

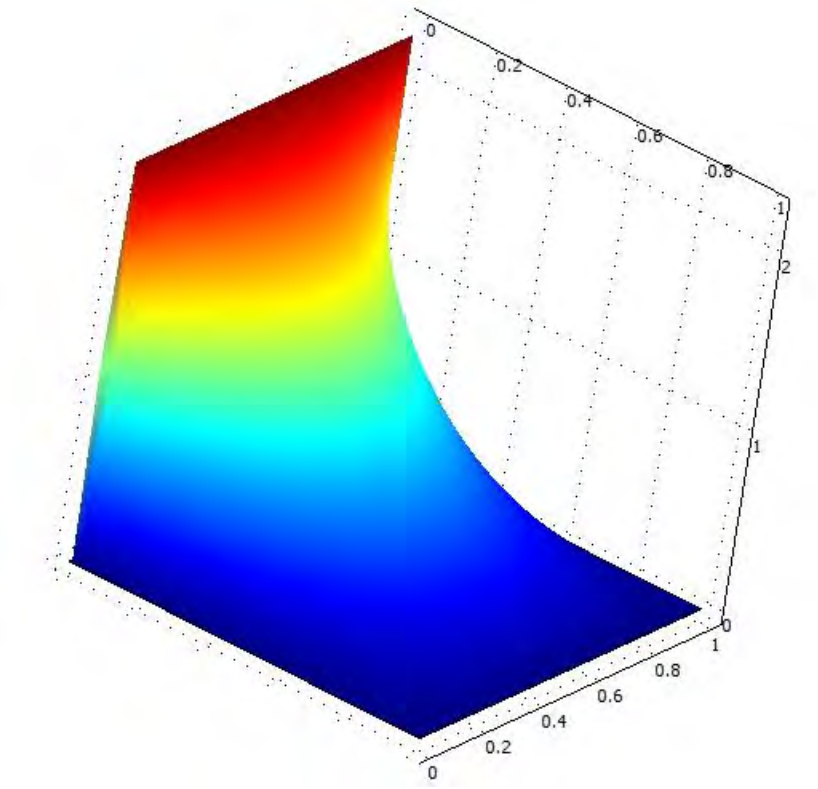
Versuchsprotokoll: Zuerst wird auf der Arbeitsfläche das Gebiet bestimmt: Ein Einheitsquadrat.

Dann habe ich das Achsengitter so gewählt, dass die Schrittweite $\frac{1}{3}$ ist, damit man nach den Berechnungen den Punkt $u(\frac{1}{3}, \frac{1}{3})$ berechnen kann und vergleichen kann, bei welchen Werten c sich dieser Punkt dem Wert 1 nähert. Die relative Genauigkeit habe ich auf $1.0E-15$ gesetzt, die vorgegebene Triangulierung habe ich einmal verfeinert, auf 3816 Elemente. Und nun habe ich Werte für c eingegeben und anschließend den Punkt $u(\frac{1}{3}, \frac{1}{3})$ berechnet:

- $c = 2.2208 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.00027$
- $c = 2.2204 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.000084$
- $c = 2.2202 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 0.999993$
- $c = 2.22028 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.000029$
- $c = 2.22025 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.000016$
- $c = 2.22023 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.000006$
- $c = 2.22022 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1.000002$
- $c = 2.22021 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 0.999997$
- $c = 2.220214 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 0.999999$
- $c = 2.220215 \Rightarrow u(\frac{1}{3}, \frac{1}{3}) = 1$

Für $c = 2.220215$ habe ich nun die Elementanzahl verfeinert auf 244224 Elemente. Auf Grund der Tatsache, dass das Programm nicht mehr als 6 Nachkommaziffern ausgibt, bekommt man mit der Verfeinerung auch nicht mehr Nachkommaziffern. Das heißt man kann jetzt davon ausgehen, dass

$$c \approx 2.2202$$



4.3 Behandlung mittels 5-Punkt-Differenzenstern

Stefan Schuchnigg

Der erste Zugang war, das nichtlineare Problem für ein vorgegebenes c mit einem klassischem 5-Punkte-Differenzenstern und einem Newtonverfahren zu behandeln. Auswertung der approximativen Lösung an der Stelle $(\frac{1}{3}, \frac{1}{3})$ ergibt insgesamt also eine Funktion $h(c) := \hat{f}(\frac{1}{3}, \frac{1}{3})$ von $\mathbb{R} \to \mathbb{R}$.

Sei nun $g(c) := h(c) - 1$, so läuft die Suche nach einem geeigneten c auf die Suche einer Nullstelle von $g(c)$ hinaus.

Dies lässt sich zum Beispiel mittels regula falsi sehr einfach implementieren.

Will man jedoch den Fehler klein halten sind die vielen benötigten Auswertungen von $g(c)$ mit dem lösen sehr grosser LGS verbunden.

Daher war der zweite Ansatz die Bedingung $f(\frac{1}{3}, \frac{1}{3}) = 1$ gleich in das lineare Gleichungssystem aufzunehmen und c als zusätzliche Variable einzuführen. Dadurch verliert man zwar die Symmetrie der Matrix spart jedoch insgesamt Zeit (durch den Wegfall der letzten Iterationsstufe).

Man erhält auf diesem Weg also (bei hinreichender Genauigkeit des Newtonverfahrens) eine Funktion $G : \mathbb{R} \to \mathbb{R}$, die zu einer gewissen Intervallänge h das 'passende' c 'exakt' (im Sinne des Fehlers durch das Newtonverfahren) ermittelt.
Wie sieht es also mit der Konvergenz von $G(h)$ für $h \to 0$ aus?
Betrachtet man $G(h)$ für im kleiner werdendes h so stellt man fest das $G(h)$ scheinbar monoton fallend ist.
Der Fehler eines klassischen 5-Punkte-Sterns schreibt sich als $\epsilon = Kh^x$. Die Vermutung liegt nahe dass sich diese Formel ebenfalls für den Fehler von c_h verwenden lässt.
Also: $|c - c_h| = Kh^x$ bzw (für G monoton fallend) $c_h - c = Kh^x$.
Für geeignete h_i $(i = 1, 2, 3)$ erhält man drei Gleichungen mit drei Unbekannten die explizit lösbar sind und damit eine Approximation von c.
Der oben beschriebene Algorithmus wurde in MATLAB implementiert und ergab folgende Werte:

Intervalle	c_n	x	K	cx_n
5	2.24867455831581			
11	2.22787715728759			
23	2.22217375556034	1.866508	0.81215678	2.22001867455005
47	2.22070847286613	1.960643	1.00226971	2.22020186761107
95	2.22033917337858	1.988315	1.08726359	2.22021473523877
191	2.22024662889441	1.996572	1.12042209	2.22021568281532
383	2.22022347692666	1.999012	1.13232999	2.22021575255794
767	2.22021768781112	1.999725	1.13639497	2.22021575764406

Bemerkenswert ist dabei dass die Werte für K und x auch für eine relativ kleine Anzahl an inneren Punkten nur sehr gering schwanken, was eine gewisses Indiz für die Richtigkeit der Fehlerapproximation mit diesem Ansatz ist. Ausserdem stellt sich als Ordnung des Verfahrens zwei ein was genau den Erwartungen entpricht. Die geringe Schwankung bei den extrapolierten c's lässt zumindest die ersten 7 Nachkommastellen als richtig erscheinen $\Rightarrow c = 2.2202157...$

5 Beispiel

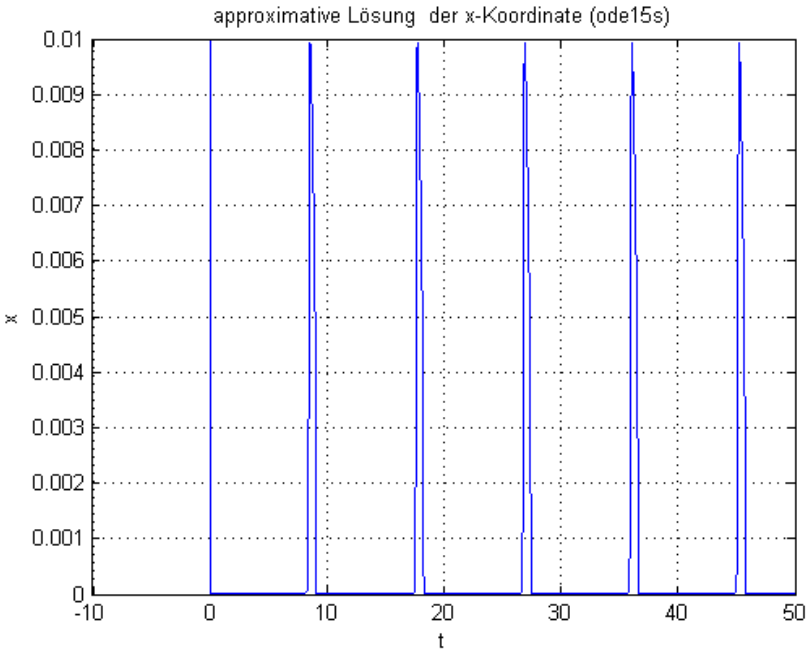
Sonja Höllrigl-Binder

5.1 Aufgabenstellung

Die Lösung (x, y, z) des Anfangswertproblem

$$\begin{aligned} \dot{x} &= k_1 y - k_2 xy + k_3 x - k_4 x^2, \\ \dot{y} &= -k_1 y - k_2 xy + k_5 f z, \\ \dot{z} &= 2k_3 x - k_5 z, \end{aligned}$$

mit $(x, y, z)(0) = (0.01, 0.02, 0.2)$ und $k_1 = 1.28$, $k_2 = 2.4 \cdot 10^6$, $k_3 = 33.6$, $k_4 = 3000$, $k_5 = 1$ und $f = 0.5$ ist periodisch. Bestimmen Sie das kleinste $p \geq 0$, so dass $x(t) = x(t+p)$ für alle $t \geq 0$.



Wie man in diesem Bild gut erkennen kann, weicht die 'erste Periode' etwas von den anderen ab. Das kommt deshalb zustande, weil sich das System erst einschwingen muss. Dieser Umstand macht eine analytische Berechnung der Lösung schwierig.

5.2 Beweis der Nichtexistenz periodischer Lösungen und Folgerungen

Stefan Schuchnigg

Ziel dieses Abschnitts ist es zu zeigen, dass zu gegebenen Startwerten keine periodische Lösung des Systems von Differentialgleichungen existiert. Zunächst folgt aus der Periodizität von $x(t)$ die Periodizität von $z(t)$:

$$x(t) = x(t + \omega) \Rightarrow \dot{x}(t) = \dot{x}(t + \omega)$$

$$\Rightarrow \dot{x}(t) = k_1 y(t) - k_2 x(t) y(t) + k_3 x(t) - k_4 x(t)^2 = k_1 y(t) - k_2 x(t+\omega) y(t) + k_3 x(t+\omega) - k_4 x(t+\omega)^2$$
$$= \dot{x}(t+\omega) = k_1 y(t+\omega) - k_2 x(t+\omega) y(t+\omega) + k_3 x(t+\omega) - k_4 x(t+\omega)^2$$
$$\Rightarrow y(t) = y(t+\omega) \Rightarrow \dot{y}(t) = \dot{y}(t+\omega)$$

jeweils für alle $t > 0$. Mit dem selben Argument folgt

$$\Rightarrow z(t) = z(t+\omega)$$

Klarerweise folgt aus der Nichtexistenz einer periodischen Lösung von $z(t)$ die Nichtexistenz Derselben von $x(t)$. Aus der letzten Gleichung des Systems folgt unmittelbar:

$$z(t) = e^{-t} \cdot (\int_0^t e^u x(u) du + z_0)$$

Ist $x(t)$ periodisch so lässt sich die Funktion als absolut konvergente trigonometrische Reihe aufschreiben. Vertauschung von Integration und Reihenbildung ergeben direkt dass $e^{-t} \cdot \int_0^t e^u x(u) du$ eine periodische Funktion ist. Daher muss für die Periodizität von $z(t)$ $z_0 = 0$ gelten und das steht klarerweise im Widerspruch zu den Voraussetzungen.
Die Vermutung liegt Nahe, dass sich $x(t)$,$y(t)$ und $z(t)$ jeweils aus einer periodischen Funktion und einer, im Unendlichen Verschwindenden, zusammensetzen.

5.3 Beweis der Nichtexistenz nichttrivialer periodischer Lösungen

Stefan Schuchnigg
Angenommen $\exists\ t_0$ mit $z(t_0) \neq 0$.
Dann sei $\tilde{x}(t) := x(t+t_0)$, $\tilde{y}(t) := y(t+t_0)$ und $\tilde{z}(t) := z(t+t_0)$.
Klarerweise gilt $\dot{\tilde{x}}(t) = \dot{x}(t+t_0)$ und $\tilde{x}(t)$ ist periodisch. (Genauso für $\tilde{y}$, $\tilde{z}$)
Daraus folgt in einfacher Weise dass $(\tilde{x}\ \tilde{y}\ \tilde{z})$ periodische Lösung des autonomen(!) Systems ist.
$\Rightarrow 0 = \tilde{z}(0) = z(t_0) \neq 0$ Widerspruch!
$\Rightarrow z = \dot{z} = 0 \Rightarrow x = 0 \Rightarrow \dot{x} = 0 \Rightarrow y = 0$
Nehmen wir nun wie oben an dass sich unsere Lösung jeweils aus einer periodischen Funktion und einer, im Unendlichen Verschwindenden zusammensetzt. Sei also:
$(x, y, z)(t) := (x_p, y_p, z_p)(t) + (x_0, y_0, z_0)(t)$. Dann gilt:

$$\dot{x_p} + \dot{x_0} = \dot{x} = k_1 y - k_2 xy + k_3 x - k_4 x^2 =$$

$$= k_1(y_p + y_0) - k_2(x_p + x_0)(y_p + y_0) + k_3(x_p + x_0) - k_4(x_p + x_0)^2 =$$
$$= k_1 y_p - k_2 x_p y_p + k_3 x_p - k_4 {x_p}^2 + r_0(t)$$

Wobei $r_0(t)$ im Unendlichen verschwindet.
Da die Ableitung einer periodischen Funktion periodisch ist und die Ableitung einer im Unendlichen Verschwindenden Funktion im Unendlichen verschwindet muss der periodische Teil ebenfalls das ODE lösen (wenn Gleichheit für alle drei Gleichungen des Systems gelten soll).
$\Rightarrow x_p = y_p = z_p = 0$.

Daraus ist leider ersichtlich dass das System zwar bei numerischer Betrachtung ein gewisses Schwingungsverhalten erkennbar macht, dieses jedoch mit dem Begriff der Periodizität mathematisch exakt nicht in den Griff zu bekommen ist.

Der weitere numerische Zugang vernachlässigt diesen Umstand!

5.4 Behandlung mittels implizietem Eulerverfahren

Stefan Schuchnigg

Die Grundidee war, die Problemstellung mit einem, für steife ODEs's geeignetem, Verfahren in einer sehr schnellen und hinreichend genauen Umgebung numerisch zu lösen. Das implizierte Eulerverfahren hat (abgesehen von der unglaublichen Stabilität) den Vorteil, dass die auftretenden Gleichungen bei dem vorliegenden Problem explizit lösbar sind und daher in jedem Schritt nur die Grundrechnugsarten benötigt werden und das macht eine Implementierung in einer nicht-mathematisch-spezifischen Umgebung wie zum Beispiel C++ natürlich wesentlich schmackhafter.
Da die Laufzeit keine Rolle spielt wurde auf eine Schrittweitensteuerung verzichtet, was ausserdem den Vorteil hat dass sehr viele Rechnungen nur einmal (und nicht in jedem Schritt) durchgeführt werden müssen. Um hinreichende Genauigkeit geährleisten zu können wurde bei der Implementierung in C++ eine floating-point-library verwendet.
Die Periode wurde an den Peaks gemessen. Ein Peak wurde dabei wie folgt ermittelt:
Für $\hat{x}(t_0) <= \hat{x}(t_1) >= \hat{x}(t_2)$ wurde $x(t)$ durch diese Punkte quadratisch interpoliert (und davon das Maximum berechnet). Um die Anzahl an Messungen zu erhöhen wurden die Abstände der Peaks an $x(t)$,$y(t)$ und $z(t)$ aufgezeichnet.
Dabei ergaben sich für $0 <= t <= 100$ folgende Werte:

Schrittweite	Periode
10^{-4}	9.1757220
$5 \cdot 10^{-5}$	9.1758000
10^{-5}	9.1758547
10^{-6}	9.1756090

Leider dauert die Berechnung mit der Schrittweite $h = 10^{-6}$ schon sehr lange (mehr als 20 Stunden) weshalb eine weitere Verkleinerung der Schrittweite nicht praktikabel erscheint. Die Vermutung liegt nahe dass zumindest die ersten 3 Nachkommastellen richtig sind. Damit ergibt sich für die Periode $\omega = 9.175...$

5.5 Lösen mittels Solver ode15s

Sonja Höllrigl-Binder

`ode15s` ist einer der Solver, den Matlab zur Lösung von steifen Differentialgleichungen anbietet. Er basiert auf einem Mehrschrittverfahren.
In diesem Ansatz zur Berechnung der Periode wird zuerst die Differentialgleichung mittels Solver 'gelöst' und anschließend jene t gesucht, an denen ein bestimmter (geeigneter) Wert überschrit-

ten bzw. unterschritten wird.
Wir betrachten nun im Zeitintervall 0-250 jene x-Werte bei denen 0.001 überschritten bzw. unterschritten wird. Die Werte in der folgenden Tabelle kommen zustande, indem aufeinanderfolgende Werte voneinander subtrahiert werden, man erhält somit die Periodendauer.

Periode	überschritten	unterschritten
1	**9.175**63039191073	9.04631403794224
2	**9.17**627275768151	**9.175**80623518502
3	**9.175**71579714728	**9.175**91314020510
4	**9.175**75822855669	**9.175**86387035596
5	**9.175**90912274995	**9.175**84116456117
6	**9.17**602914033645	**9.175**84302554613
7	**9.17**601645724078	**9.175**90535438276
8	**9.1758**0832504919	**9.175**83078389298
9	**9.17**603706883831	**9.175**87646147807
10	**9.175**39603170671	**9.175**96806476250
11	**9.17**629347771414	**9.175**80604085845
12	**9.175**58608514328	**9.175**81057497256
13	**9.175**81566084183	**9.175**99780387265
14	**9.175**71494278535	**9.175**86214246809
15	**9.17**610266895142	**9.175**77966861455
16	**9.17**619018501244	**9.175**89522271109
17	**9.175**88030541825	**9.175**90537014209
18	**9.175**31648605484	**9.175**84789076014
19	**9.17**632224600467	**9.175**97186042812
20	**9.175**62608971193	**9.175**70385631271
21	**9.17**612358687197	**9.175**90431194691
22	**9.175**65884249441	**9.175**94703331449
23	**9.175**64428524608	**9.175**84999871178
24	**9.17**603527159065	**9.175**94272305635
25	**9.17**606247451980	**9.175**83245312039
26	**9.175**53668727936	**9.175**87596572676
27		**9.175**88729108419

Matlab berechnet im Intervall $[0, 200]$ die Lösung an 56255 Stellen. Zusätzlich zu der Ungenauigkeit des Solvers kommen daher auch noch Diskretisierungsfehler bei dieser Methode hinzu. Insgesamt erhalten wir somit folgendes Ergebnis

$$p \doteq 9.175$$

5.6 Lösen mittels Solver ode23s

Sonja Höllrigl-Binder

Obwohl der Solver ode23s auf einem Einschrittverfahren basiert, liefert er genauere Werte als ode15s. Der Grund hierfür ist wahrscheinlich, dass die Lösung lange beinahe konstant bleibt und dann sehr schnell ansteigt. Das Einschrittverfahren kann auf eine solche Änderung schneller 'reagieren'.
ode23s wertet im Intervall $[0, 200]$ an über 130000 Stellen aus. Im Vergleich zu ode15s sind das mehr als doppelt so viele. Die Rechenzeit verlängert sich dementsprechend.

Periode	überschritten	unterschritten
1	**9.1758**8384625042	9.04633676418764
2	**9.1758**6519410715	**9.1758**7001907126
3	**9.1758**6391326072	**9.1758**7543826383
4	**9.1758**8786684307	**9.1758**7078276447
5	**9.1758**6452183036	**9.1758**7052627833
6	**9.1758**6413345149	**9.1758**7520618125
7	**9.1758**8393871252	**9.1758**7008753835
8	**9.1758**6502941242	**9.1758**7041429192
9	**9.1758**6617030548	**9.1758**7567427400
10	**9.1758**8462236981	**9.1758**7069204690
11	**9.1758**6490262408	**9.1758**7011887395
12	**9.1758**8450735845	**9.1758**7559962136
13	**9.1758**6644888732	**9.1758**7051839276
14	**9.1758**6744975934	**9.1758**7079941487
15	**9.1758**8311278246	**9.1758**7603364314
16	**9.1758**6538425513	**9.1758**6988924415
17	**9.1758**6403769829	**9.1758**7539174431
18	**9.1758**8582543630	**9.1758**7055219491
19	**9.1758**6744312692	**9.1758**7097697395
20	**9.1758**6509847672	**9.1758**7580857571
21		**9.1758**6989267940

Um zu überprüfen, ob die Periodendauer abhängig vom Punkt ist, der überschritten wird, wurde das Programm auch für den Wert 0.009 getestet. Die Resulate sind sehr ähnlich. Mit dem Befehl ode23s kommt man somit zu folgendem Ergebnis

$$p \doteq 9.1758$$

Endbericht Digit Challenge

Karl Rupp, 0325941

3. August 2008

Aufgabe 1

Finden Sie das globale Minimum $M = \min f(x, y)$ der Funktion

$$f(x, y) = \sin(\cos(xy)) + \cos(\sin(x + 2y)) + \sin(e^{x^2}) + e^{\cos(y)} + x^2 + y^2, \tag{1}$$

wobei $(x, y) \in \mathbb{R}^2$.

Zuerst einmal bemerkt man die Symmetrie $f(-x, -y) = f(x, y)$. Damit kann man sich o.B.d.A. auf den Fall $x > 0$ konzentrieren. Weiters zeigt eine grobe Abschätzung für $f(0, 0)$, dass das globale Minimum sicher kleiner als $3 + e \approx 5.72$ und größer als $-3 + \frac{1}{e} \approx -2.63$ ist.

Aufgrund des paraboloidischen Anteils $x^2 + y^2$ kann man sich daher auf eine Minimumssuche im Bereich $(x, y) \in [0, 3] \times [-3, 3]$ beschränken. Wenn man noch die feineren Abschätzungen $\cos(\sin(x+2y)) > 0.5$ sowie $|\sin(\cos(xy))| < 0.85$ berücksichtigt, kann man den Bereich noch um ein paar Zentel auf beiden Achsen einschränken.

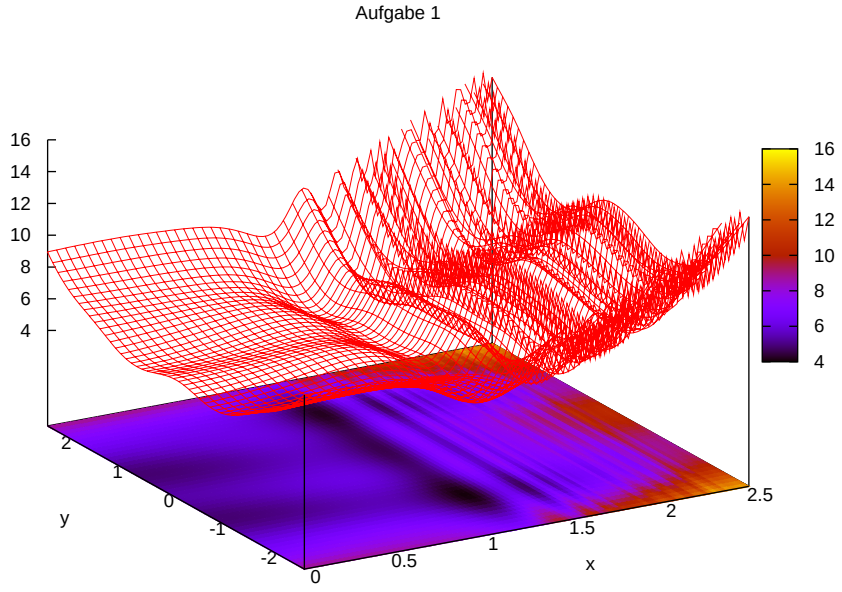
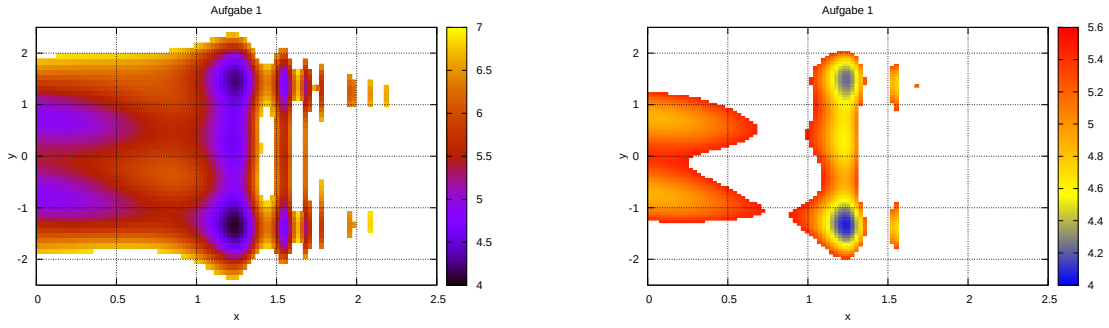


Abbildung 1: Erste Übersicht über die Funktion im interessanten Bereich

Aus Abb. 1 ist zu erkennen, dass das globale Minimum ungefähr 4 ist. Dabei ist aber noch sicherzustellen, dass durch das Raster, mit dem der Plot erzeugt wurde, eventuelle weitere Minima nicht übersehen wurden: Für $(x, y) \in [0, 2] \times [-2, 2]$ kann man sehr leicht

$$\left|\frac{\partial f(x, y)}{\partial x}\right| \leq 250, \quad \left|\frac{\partial f(x, y)}{\partial y}\right| \leq 20 \tag{2}$$

abschätzen. Damit haben wir bei einem Raster von 0.001 in x-Richtung bereits die Gewissheit, dass zwischen zwei Punkten höchstens eine Differenz von 0.25 auftritt bzw. der Fehler durch lineare Interpolation in x-Richtung kleiner als 0.125 ist. Dadurch kann ausreichend Information über die Funktion abgelesen werden und ohne Weiteres die Lage des Minimums auf das Intervall $(x, y) \in [1.1, 1.4] \times [-1.2, -1.5]$ eingeschränkt werden, siehe dazu auch Abb. 2.



(a) Plot jener Regionen, wo $f(x, y) < 7$

(b) Plot jener Regionen, wo $f(x, y) < 5.5$

Abbildung 2: Kontur-Plots von $f(x, y)$ für ausgewählte Wertebereiche.

Sobald diese erste Schätzung über die Lage des Minimas vorliegt, liefert eine Newton-Iteration die genaue Lage des Minimas: Es wird nach der Nullstelle des Gradienten gesucht:

$$(x_{n+1}, y_{n+1}) = (x_n, y_n) - \left\{ \frac{\partial f_i(x, y)}{\partial (x, y)} \right\}^{-1} f(x, y). \tag{3}$$

Das Newton-Verfahren konvergiert quadratisch gegen die tatsächliche Lösung, das heißt, dass sich mit jedem Schritt die Anzahl der Ziffern verdoppelt. Gehen wir von einer richtigen Ziffer als Startwert aus, so können wir nach 10 Schritten eine Genauigkeit von 1024 Ziffern erwarten. Tatsächlich benötigen wir nicht die Lage des Minimums, sondern den Funktionswert an dieser Stelle. Da die zweiten Ableitungen der Funktion in der Nähe des globalen Minimas ähnlich gut abgeschätzt werden können wie die ersten Ableitungen, gibt es hier keine weiteren Probleme von Seiten der Mathematik.

Die einzige kleine Hürde bleibt damit die Verwendung hochgenauer Arithmetik. In `Maple` kann allerdings mit einem einzigen Befehl die Genauigkeit des Rechenwerks festgelegt werden, wobei auch sehr viele Stellen (mehrere Zehntausend) möglich sind, auch wenn natürlich die Laufzeit darunter leidet.

Eine Implementierung in `Maple` findet sich im Anhang, in Tab. 1 sind die Ergebnisse für die ersten Iterationsschritte eingetragen. Daraus ist die quadratische Konvergenz gut ablesbar.

Schrittanzahl	Funktionswert
1	4.049125350380187348199094599813312206807 01
2	4.047583405207768646444020391555192718284 63
3	4.047582689914333898415399583268108465141 28
4	4.047582689914079035163436296213513547459 35
5	4.047582689914079035163436263727607019200 38

Tabelle 1: Funktionswerte, die durch die ersten Newtonschritte erhalten werden. Man beachte die quadratische Konvergenz.

Da die Berechnung numerisch sehr günstig ist, wählen wir mit gutem Gewissen eine Rechengenauigkeit rund 5000 Stellen und führen 15 Newton-Iterationsschritte durch, womit wir das Funktionsminimum zu

```
4.047582689914079035163436263727607019200376712399634318946696854441857011323664380
  916606614817242003795998950017418635004827654099681190239537058070740558427234570
  555992408971307507104746111243521499603839949435691135851254256766811636779307810
  779687882368127107673779378470172171136961655093809778270422390193624676657816120
  819702463848005108595789723065410385892608716785899693628211459076414340229657160
  799414623947907337366785741261703715523317522660483334183194715848226895140861290
  693845441056274875105414909149790723702107320323293655416471333334687763689777160
  964671509736172783851389212450693189779905787543674587469198660089143262683240740
  890302687979806068606752581568013996793061069206876658987783542694129570437330300
  895536512680122443500179165969160067546945574476178950287660951750096709828821490

  075390648410775321529780221726934030885128455757546462106461357212583875581221310
  099598579514737090352676453002892687025996692784545400851664339607046526412882150
  893546601328179468599711552129490895730903796643396913446555395279612149648583350
  277269249406049252361745962826930805756939449368692229928389924485011231919798670
  882160228737602869061464956631531895855765516569108044143000939327110144401972720
  151335433703027024491124329139437638281743728391289219430569435947658390246599610
  959192880134407827248375923326120152536826872503676135512480910367387624018111810
  507585682355277865373992361897352939487750640675882316373981458831764884351134820
  579357079522584899994983696307744339472397411817464893932725980904499807907590720
  690208021989297525686498801376565188542122015885098197137286636562775101699670300

  539013418518603509313229192165112827470341252921092113479135164718440645370724350
  935109095572891313692396219599718954745391909743683106584255023875519725754313640
  794506993249603696161867535317442468926759837475783100267871894981536764692402310
  318716810742830601822287566773193041611345855192874285101966049167609332503964290
  251828526552930412079074217743197588002042637139323073634150970864037975437954210
  244389683383109271632147649646042012810578861535072281768684168081763841321498040
  537422080968765538971355574444785264826949825031873922698241437655352243099963760
  702082087476585818369669754099640609314999430136188482168462813333226795723754510
  018153702451324480352701953268227729395730828255091417517867208644521214378204160
  337988171770006781045834065113442528761722109924064011113362505452328244463471750

  423778539239421520176295564099407207220030890778812170166898431524585013540840080
  952717378134605527180030620833306245531439849035253887636312414691390409838191600
  067967360355119236046255159945385846997727930448966477262065843573011390062214870
  467218538499292778881186752260384469449544322403417102503489279064409078520921770
  443012772888830546310461263010733681943667453087676187695711692378842614229928520
  309366767905706767039944798778411622820789112354555258141918192306319530018652200
  361035003733965765883094266072329121203149035621745785309820793479520951644830830
  574001719901722251770510876387138568581557534531830886865037917260639322665036440
  875234924749835348218177592706024749795910363636260090459040767533942277077907020
  479619198125660288268786941183025766062603304658101960026318754187894865980676370

  508467257272377177292307056091045319341773927302328139251151379726501102121160270
  272316597057699859946630085455556539930493397620658254367500620240803347595256912
  268475482159178600449519512539959769781327510363810376898956114963858196459748800
  111531670924592187467139419912645299817765555647533529474322776464158202347305740
  684336103082008047063044583825154695049184837087125037424572370345084642722239200
  350852685137909673729736397357374900572267998060423745453236355873291913177487120
  520909998094257632037322001938877235583878888476534872216045413585817378428711500
  862070126992128944364955686835059151076299044244837449376502887504263959216232480
  822040541725365886314405573857145642269739374386779675075518269902571450521268540
  656838588611050416637422028555909001134980058362928081089558157253978310695448030
```

bestimmt haben (Jeder Block enthält 10x80 Ziffern nach dem Komma). Weitere Ziffern stellen kein Problem dar, verschwenden aber nur (virtuelles) Papier. ;-)

Aufgabe 2

Lösen Sie das Integral

$$I = \int_{\mathbb{R}^6} e^{-x_1^4 - \cdots - x_6^4} \sin^2(x_1 \cdots x_6) dx, \tag{4}$$

wobei $x = (x_1, \ldots, x_6)$.

Diese Aufgabe hat meine Kreativität am längsten beschäftigt und mich durch ein Wellental der Gefühle laufen lassen. Ich möchte meine Ideen hier in chronologischer Ordnung wiedergeben, da sie sich (glücklicherweise) sukzessive in Richtung des Endergebnisses entwickeln, ohne dabei auf grobe Holzwege zu stoßen.

Erste Betrachtungen

Noch bevor ein Gedanken an die Implementierung verschwendet wird, kann man die Symmetrie des Integranden bezüglich Null in jeder Raumdimension feststellen. Damit gilt

$$I = \int_{\mathbb{R}^6} e^{-x_1^4 - \cdots - x_6^4} \sin^2(x_1 \cdots x_6) dx = 2^6 \cdot \int_{[0,\infty]^6} e^{-x_1^4 - \cdots - x_6^4} \sin^2(x_1 \cdots x_6) dx. \tag{5}$$

Definieren wir

$$I_r := \int_{[-r,r]^6} e^{-x_1^4 - \cdots - x_6^4} \sin^2(x_1 \cdots x_6) dx, \tag{6}$$

so ist wegen der stark fallenen Exponentialfunktion zu erwarten, dass bereits für $r \approx 5$ eine sehr gute Näherung an I vorliegt.

Mit dieser Beobachtung scheint die Berechnung von I auf wenige Zeilen Code in `MATLAB` hinauszulaufen, jedoch entpuppt sich sehr schnell die Gemeinheit im Detail: Bei Verwendung von numerischer Quadratur explodiert sehr schnell die Rechenzeit, da bei einer Quadraturformel mit k Stützstellen in einer Raumdimension k^6 Stützstellen für das gesamte Integral notwendig sind. Bei 10 Stützstellen sind damit bereits eine Million Funktionsauswertungen notwendig, für hochgenaue Resultate sind natürlich ungleich mehr Stützstellen erforderlich.

Numerische Integration

Dennoch wurde in `MATLAB` ein derartiger Brute-Force-Versuch gestartet, eine Verschachtelung der Funktion `tripelquad` erledigte die Integration über sechs Raumrichtungen. Der erhaltene Wert des Integrals lag je nach Versuch bei

$$I \doteq 0.05 \begin{matrix} 3 \\ 1 \end{matrix} . \tag{7}$$

Offensichtlich ist ein simpler Brute-Force-Versuch zu wenig, um dieses Beispiel zu knacken; ein cleverer Zugang wird wohl nötig sein. Stecken wir also etwas mehr Information über den Integranden in die numerische Quadratur: Da eine Vertauschung der Variablen $x_i \leftrightarrow x_j$ das Integral in sich selbst überführt, können wir die Symmetrie ausnützen, um auf einem Schlag sehr viele Integrationspunkte abgehandelt zu haben. Die Auswertung an einem Punkt $(x_1, x_2, \ldots, x_6)$ liefert auf einem Schlag auch den Werte des Integranden am Punkt $(x_2, x_1, \ldots, x_6)$ sowie allen weiteren Permutationen, was immerhin $6! = 720$ Möglichkeiten entspricht und beispielsweise eine Monte-Carlo-Integration enorm beschleunigen würde. Trotzdem würde dieser Ansatz wohl nur wenige Stellen sicher angeben können.

Eine weitere Erhöhung der Effizienz bringt eine Aufspaltung des Integrals. Für die Berechnung

von I_2 kann beispielsweise zu

$$I_2 = \int_0^2 \int_0^2 \cdots \int_0^2 e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 \cdots x_6) dx \tag{8}$$

$$= \int_0^1 \int_0^2 \cdots \int_0^2 e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 \cdots x_6) dx + \int_1^2 \int_0^2 \cdots \int_0^2 e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 \cdots x_6) dx \tag{9}$$

$$\vdots \tag{10}$$

$$= \int_0^1 \int_0^1 \cdots \int_0^1 + 6 \int_1^2 \int_0^1 \cdots \int_0^1 + \binom{6}{2} \int_1^2 \int_1^2 \int_0^1 \cdots \int_0^1 + \ldots + \int_1^2 \int_1^2 \cdots \int_1^2 \tag{11}$$

aufgespalten werden, wobei jetzt sieben Integrale berechnet werden müssen, die allerdings nur je $1/2^6$ des ursprünglichen Integrationsbereichs überstreichen. Eine Ausnutzung der Symmetrie für die gemischen Integralgrenzen wird schwieriger, ist aber immer noch möglich.

Nützt man die Beziehung $\sin^2(x) + \cos^2(x) = 1$, so kann man mehr "Masse" zum Ursprung hin verlagern:

$$I = \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 \cdots x_6) dx \tag{12}$$

$$= \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4} dx - \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4} \cos^2(x_1 \cdots x_6) dx \tag{13}$$

$$= \left[\frac{1}{2}\Gamma\left(\frac{1}{4}\right)\right]^6 - \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4} \cos^2(x_1 \cdots x_6) dx. \tag{14}$$

Die Gammafunktion steht in nahezu beliebiger Genauigkeit zur Verfügung, weshalb die Auswertung des Integrals mit dem Cosinus ebenso für eine hohe Anzahl an Ziffern ausreicht bzw. ausreichen würde.

Erste Schritte mit Analysis enden mit einer Bruchlandung

Um die ersten numerischen Integrationsergebnisse abschätzen zu können, wurde $\sin^2(x) \approx x^2$ gesetzt:

$$I \leq \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4} x_1^2 \cdots x_6^2 dx \tag{15}$$

$$= \left(2 \int_0^\infty e^{-x^4} x^2 dx\right)^6 = \left[\frac{1}{2}\Gamma\left(\frac{3}{4}\right)\right]^6 \tag{16}$$

$$= 0.0529 \tag{17}$$

Diese obere Schranke ist verblüffend gut, obwohl wir den Sinus sehr grob approximiert haben. Der Grund liegt darin, dass der Integrand in der Nähe des Ursprungs seine größten Werte annimmt, wo der Exponentialterm noch nicht zu stark drückt. Aber gerade in der Nähe des Ursprungs ist wiederum die Linearisierung relativ gut.

Dadurch motiviert sich auch gleich der Versuch, das Sinus-Quadrat über ein Polynom vom Grad vier, das sich aus der Taylorentwicklung im Ursprung ergibt, anzunähern. Wir erhalten diesmal eine untere Schranke, da einerseits der dominierende Term x^4 negatives Vorzeichen hat und andererseits über die Taylorentwicklung

$$\sin^2(x) = \frac{1 - \cos(2x)}{2} = \frac{1}{2}\left(x^2 - \frac{1}{3}x^4 + \frac{2}{45}\sin(\xi)\right), \quad \xi \in (0, 2x) \tag{18}$$

$$\geq \frac{1}{2}\left(x^2 - \frac{1}{3}x^4\right), \quad 0 \leq x \leq \pi/2 \tag{19}$$

die untere Abschätzung für kleine x folgt. Da das Näherungspolynom bereits vor $\pi/2$ hinreichend weit von $\sin^2(x)$ entfernt liegt (vlg. Abb. 3) und kurz darauf nur noch negative Funktionswerte annimmt, folgt die Abschätzung nach unten für alle positiven x.

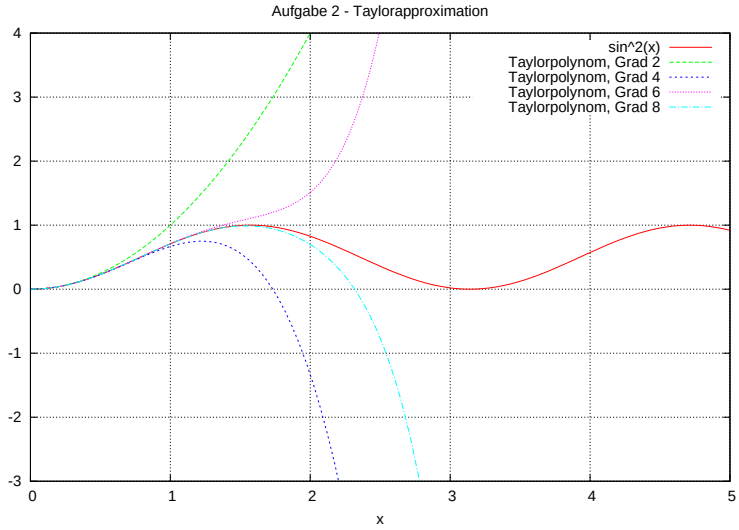


Abbildung 3: Taylorapproximation von $\sin^2(x)$

Durch diese bessere Abschätzung des Sinus erhalten wir

$$I \leq \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4}(x_1^2\cdots x_6^2 - \frac{1}{3}x_1^4\cdots x_6^4)dx \tag{20}$$

$$= \left(\frac{1}{2}\int_0^\infty e^{-x^4}x^2dx\right)^6 - \frac{1}{3}\left(\frac{1}{2}\int_0^\infty e^{-x^4}x^4dx\right)^6 \tag{21}$$

$$= \left[\frac{1}{2}\Gamma\left(\frac{3}{4}\right)\right]^6 - \frac{1}{3}\left[\frac{1}{2}\Gamma\left(\frac{5}{4}\right)\right]^6 \tag{22}$$

$$= 0.05002, \tag{23}$$

womit wir bereits analytisch die erste Ziffer 5 bewiesen hätten.

Von dieser Idee getrieben folgt natürlich auch gleich die Idee, weitere Taylorpolynome einzusetzen. Warum also nicht die gesamte Taylorentwicklung einsetzen? Die Vorzeichen stehen denkbar günstig: Der Konvergenzradius für Sinus-Quadrat ist unendlich, daher konvergiert die Taylorentwicklung überall gleichmäßig. Setzen wir also ein, so erhalten wir

$$I = \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4}\sin^2(x_1x_2\cdots x_6)dx \tag{24}$$

$$= \int_{\mathbb{R}^6} e^{-x_1^4-\cdots-x_6^4}\frac{1}{2}\sum_{k=1}^\infty(-1)^{k+1}\frac{2^{2k}}{(2k)!}x_1^{2k}\cdots x_6^{2k}dx. \tag{25}$$

Nun geht es an die Vertauschung von Summation und Integration: Die Konvergenz der Reihe ist gleichmäßig, der Integrand stetig (ja sogar C^∞), es sollte also alles glatt (im wahrsten Sinne des Wortes) laufen:

$$I = 2^6\int_{[0,\infty]^6} e^{-x_1^4-\cdots-x_6^4}\frac{1}{2}\sum_{k=1}^\infty(-1)^{k+1}\frac{2^{2k}}{(2k)!}x_1^{2k}\cdots x_6^{2k}dx \tag{26}$$

$$\stackrel{?}{=} \frac{2^6}{2}\sum_{k=1}^\infty(-1)^{k+1}\frac{2^{2k}}{(2k)!}\int_{[0,\infty]^6} e^{-x_1^4-\cdots-x_6^4}x_1^{2k}\cdots x_6^{2k}dx \tag{27}$$

$$= \frac{1}{2}\sum_{k=1}^\infty(-1)^{k+1}\frac{2^{2k}}{(2k)!}\left(2\int_0^\infty e^{-x^4}x^{2k}dx\right)^6 \tag{28}$$

$$= \frac{1}{2}\sum_{k=1}^\infty(-1)^{k+1}\frac{2^{2k}}{(2k)!}\left[\frac{1}{2}\Gamma\left(\frac{2k+1}{4}\right)\right]^6 \tag{29}$$

Die so entstandene alternierende Reihe liefert nun abwechselnd obere und untere Schranken für

den tatsächlichen Wert I. Wir erhalten damit für die Folge der Partialsummen

$$
\begin{array}{cc}
 & 0.0529081 \\
0.0500199 & \\
 & 0.0504385 \\
0.0503335 & \\
 & 0.0503717 \\
0.0503533 & \\
 & 0.0503643 \\
0.0503564 & \\
 & \mathbf{0.0503631} \\
\mathbf{0.0503567} & \\
 & 0.0503636 \\
0.0503552 & \\
 & 0.0503664 \\
0.0503501 & \\
 & 0.0503759 \\
\vdots & \ddots \\
-4224.4702 & \\
 & 17299.5958
\end{array}
$$

Nachdem anfänglich alles bestens begonnen hat, divergiert die Reihe schlussendlich! Immerhin bleiben drei Ziffern für das Endergebnis übrig, was aber nicht über den Schock der Divergenz hinwegtröstet. Was ist schief gelaufen?

Ein Blick auf die entstandene alternierende Reihe zeigt recht bald den Übeltäter: Die Reihenglieder müssen gegen Null gehen, damit die Reihe konvergiert. Die Fakultät dominiert offensichtlich die Zweierpotenz, weshalb nur noch das Wachstum der Gammafunktion der Fakultät gegenüberzustellen ist: Mit Stirlings Formel ergibt sich

$$(2k)! \sim (2k)^{2k}, \quad \Gamma\left(\frac{2k+1}{4}\right) \sim \left(\frac{2k+1}{4}\right)^{\frac{2k+1}{4}} \tag{30}$$

Da wir aber die sechste Potenz der Gamma-Funktion benötigen, erhalten wir insgesamt

$$\Gamma\left(\frac{2k+1}{4}\right)^6 / (2k)! \sim \left(\frac{2k+1}{4}\right)^{\frac{2k+1}{2}} \to \infty \tag{31}$$

Offensichtlich ist das Problem nicht die einmalige Vertauschung von Integration und Summation, sondern die sechsmalige: Jede Vertauschung erzeugt einen Gamma-Funktions-Term, der die Konvergenz der Reihe verschlechtert und nach viermaliger Wiederholung schließlich zerstört[1]. Unser Ausflug in die Analysis wurde offensichtlich zur ernüchternden Bruchlandung…

Wie ein Phoenix aus der Asche

Der Schock über die Divergenz des Taylorreihen-Ansatzes saß tief: Obwohl wir drei Ziffern erhielten, scheint eine höhere Genauigkeit auf diesem Wege nicht möglich.

Erinnern wir uns aber zurück die Definition von $I_r := \int_{[-r,r]^6} e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 \cdots x_6) dx$: Es reicht doch für unsere Zwecke in gewissem Maße aus, wenn wir nicht das unbestimmte Integral betrachten, sondern nur das Integral über einen sechsdimensionalen Kubus von ausreichender Größe! Die Gammafunktionsterme übertragen sich auf das beschränkte Integrationsgebiet wie folgt:

$$\int_0^\infty e^{-x} x^\alpha dx \sim \Gamma(\alpha+1) \sim \left(\frac{\alpha+1}{e}\right)^{\alpha+1}, \quad \text{aber} \int_0^r e^{-x} x^\alpha dx \le \frac{r^{\alpha+1}}{\alpha+1}. \tag{32}$$

[1] Eventuell ergibt sich für viermalige Vertauschung immer noch Konvergenz, da dann die Zweierpotenzen entscheiden. Für unser Problem ist das aber irrelevant, da bei sechsmaliger Vertauschung Divergenz garantiert ist.

Die *unvollständige Gammafunktion* $\gamma(x;r) := \int_0^r e^{-x}x^{\alpha-1}dx$, wie sie nun für die Integrale über den Kubus auftreten, wächst "nur" noch exponentiell mit α, während die gewöhnliche Gammafunktion wie eine Fakultät wächst. Das ist aber genau das, was wir für die Vertauschung von Summation und Integration benötigen: Die Fakultät im Nenner wird nun immer dominieren.

Für das Integral über einen Kubus erhalten wir nun

$$I_r = 2^6 \int_{[0,r]^6} e^{-x_1^4-\cdots-x_6^4} \frac{1}{2} \sum_{k=1}^{\infty} (-1)^{k+1} \frac{2^{2k}}{(2k)!} x_1^{2k} \cdots x_6^{2k} dx \tag{33}$$

$$= \frac{2^6}{2} \sum_{k=1}^{\infty} (-1)^{k+1} \frac{2^{2k}}{(2k)!} \int_{[0,r]^6} e^{-x_1^4-\cdots-x_6^4} x_1^{2k} \cdots x_6^{2k} dx \tag{34}$$

$$= \frac{1}{2} \sum_{k=1}^{\infty} (-1)^{k+1} \frac{2^{2k}}{(2k)!} \left(2 \int_0^r e^{-x^4} x^{2k} dx\right)^6 \tag{35}$$

$$= \frac{1}{2} \sum_{k=1}^{\infty} (-1)^{k+1} \frac{2^{2k}}{(2k)!} \left[\frac{1}{2}\gamma\left(\frac{2k+1}{4}; r^4\right)\right]^6, \tag{36}$$

was wir nun numerisch auswerten wollen.

Auswertungen mit `MATLAB` oder `Octave` machen aber schnell Probleme: Da die Reihe alternierend ist, müssen Terme von annähernd gleicher Größe voneinanderer subtrahiert werden, was starke Auslöschungseffekte mit sich bringt. Was aber noch viel stärker wiegt: Zuerst wachsen die Terme dem Betrag nach stark, die Reihenglieder werden erst sehr spät wirklich vernachlässigbar klein, sodass die Partialsumme ausreichend nahe am Ergebnis liegt. Um das zu verdeutlichen, betrachten wir

$$\frac{2^{2k}\Gamma\left(\frac{2k+1}{4}; r^4\right)^6}{(2k)!} \sim \frac{2^{2k}\left(r^{2k+5}\right)^6}{(2k)^{2k}} \sim \frac{(r^6)^{2k+5}}{k^{2k}} \tag{37}$$

Damit sind bei $k \approx r^6$ die Reihenglieder noch immer in der Größenordnung des Ergebnisses. Für $r = 3$ tritt dies dementsprechend erst bei $k \approx 729$ auf, weshalb wir sehr viele Reihenglieder berechnen müssen. Weiters ist das Maximum auch dementsprechend groß, sodass hochgenaue Arithmetik notwendig ist: Aus den numerischen Ergebnissen ergibt sich, dass bei $r = 3$ bereits fast 1000 Ziffern an Rechengenauigkeit notwendig sind, da Auslöschungseffekte in den relevanten Stellen auf jeden Fall vermieden werden müssen.

Außerdem muss der Einfluss der Wahl von r noch näher beleuchtet werden: Es zeigt sich, dass die Wahl $r = 3$ schon über 20 Stellen an Genauigkeit liefert, was wir analytisch abschätzen möchten:

$$I - I_r \approx 6 \cdot 2 \int_r^{\infty} \int_{[-r,r]^6} e^{-x_1^4-\cdots-x_6^4} \sin^2(x_1 x_2 \cdots x_6) dx \tag{38}$$

$$= 6 \cdot \left(2 \int_r^{\infty} e^{-x^4} dx\right)\left(2 \int_0^r e^{-x^4} dx\right)^5 \tag{39}$$

$$= \frac{6}{2}\left(\Gamma\left(\frac{1}{4}\right) - \gamma\left(\frac{1}{4}; r^4\right)\right)\left(\frac{1}{2}\gamma\left(\frac{1}{4}; r^4\right)\right)^5 \tag{40}$$

Diese Abschätzung benützt, dass der Integrand durch den Exponentialterm für $x > r$ dominiert wird, weshalb jener Fehlerterm dominiert, der nur eine Integrationsvariable außerhalb des Kubus um den Ursprung besitzt.

Mit vielen Optimierungen zu vielen Ziffern

In `Maple` sind nun die im vorigen Abschnitt präsentierten Kniffe zusammengetragen und in eine Funktion gegossen worden. Erste Experimente zeigten eine starke Abhängigkeit der Laufzeit von der gewählten Rechengenauigkeit. Daher wurde in einem Preprocessing-Schritt zuerst die benötigte Genauigkeit näherungsweise ermittelt: Mit Standard-Genauigkeit wurden die Terme der Summe ausgewertet und das betragsmäßig größte Term ermittelt. Dieser muss mindestens noch alle relevanten Ziffern der Lösung exakt beinhalten. Einige zusätzliche Stellen an Sicherheit ergeben die benötigte Rechengenauigkeit.

Die Anzahl der Reihenglieder wurde derart bestimmt, dass in niedriger Genauigkeit in sehr groben Raster nach jenem Index gesucht wird, für den der Beitrag zur Reihe kleiner als 10^{-100} ist. Damit ist man auf jeden Fall auf der sicheren Seite, da asymptotisch nach wie vor die Reihenglieder wie die Inverse einer Fakultät abfallen.

Zur Erhöhung der Laufzeiteffizienz wurde die Rekursionsformel $\gamma(\alpha;z) = \frac{1}{\alpha}\left[\gamma(\alpha+1;z) + e^{-z}z^{\alpha}\right]$ benutzt. Die Reihe wurde dabei von hinten nach vorne ausgewertet und auf die Summanden mit jeweils geraden bzw. ungeraden Indices die obige Rekursionsformel angewandt. Zur Absicherung gegen Rundungsfehler wurde außerdem noch nach (zumeist) 100 Gliedern die unvollständige Gammafunktion neu berechnet. Weiters wurde die Rechengenauigkeit adaptiv angepasst: Für große Summanden wurde die Genauigkeit erhöht, während sie für kleine Terme verringert wurde, sodass immer eine feste Anzahl an Stellen nach dem Komma (meist 200) garantiert sind, aber nicht verschwendet werden.

Die unvollständige Gammafunktion ist in `Maple` übrigens als Spezialfall einer *konfluenten hypergeometrischen Funktion* implementiert. Nähere Details verrät die Dokumentation des Programmpakets.

Nach der Ausgabe des Ergebnisses für I_r wird auch noch der Fehlerschätzer berechnet und ausgegeben, der sehr gute Abschätzungen für den tatsächlichen Fehler liefert, siehe Tab. 2. Der absolute Fehler wird dabei im Allgemeinen um etwa einen Faktor 10 bis 100 überschätzt, was für unsere Zwecke aber ausgezeichnet ist.

r	I_r	$\lvert I-I_r\rvert$	$\lvert I-I_r\rvert$ gesch.	Glieder	RGen.
1.5	0.049460838314670856863222403051 5	8.99e-4	9.74e-2	600	200
1.6	0.050170398594892853905216767980 8	1.89e-4	1.85e-2	600	200
1.7	0.050330653521524839038299022542 7	2.93e-5	2.61e-3	600	200
1.8	0.050356762829288808055695587228 7	3.21e-6	2.61e-4	600	200
1.9	0.050359730986017463386430469571 5	2.38e-7	1.78e-5	600	200
2.0	0.050359957497114763810834497274 0	1.15e-8	7.91e-7	600	200
2.1	0.050359968624360638179983279137 9	3.43e-10	2.19e-8	600	200
2.2	0.050359968960853250572209820393 2	6.03e-12	3.58e-10	600	227
2.3	0.050359968966827314752537966686 8	5.95e-14	3.31e-12	600	244
2.4	0.050359968966886538046762698873 4	3.12e-16	1.62e-14	700	267
2.5	0.050359968966886848978772702327 4	8.19e-19	4.00e-17	800	297
2.6	0.050359968966886849796527118043 4	1.02e-21	4.69e-20	900	336
2.7	0.050359968966886849797543453090 6	5.62e-25	2.44e-23	1100	384
2.8	0.050359968966886849797544014716 6	1.30e-28	5.35e-27	1400	444
2.9	0.050359968966886849797544014846 1	1.16e-32	4.57e-31	1600	517

Tabelle 2: Werte von I_r für ausgewählte Werte von r. Tatsächlicher Fehler und geschätzter Fehler sowie benötigte Anzahl an Reihentermen (mindestens 600) und Rechengenauigkeit (RGen, mindestens 200 Ziffern)

Damit haben wir eine genaue Kontrolle über das benötigte r für eine geforderte Anzahl an Ziffern. Wir sind nun nur mehr durch die Auswertbarkeit der Reihe für hohe Genauigkeit limitiert: Die Auswertung von beispielsweise 5000 Reihengliedern mit je 1000 Ziffern Genauigkeit erfordert eben einen nicht unwesentlichen Rechenaufwand, vor allem wenn man bedenkt, dass die hohe Genauigkeit in Software realisiert werden muss und keine Hardwareunterstützung erfährt.

Für $r = 4.2$ ergeben sich ca. 140 Ziffern, auf eine tabellarische Auflistung von Ergebnissen für $r = 3.0 \ldots 4.2$ sei verzichtet, jedoch findet sich das `Maple`-Skript im Anhang, womit der geneigte Leser selbst weitere Stellen berechnen kann.

Nach ca. einer Stunde Rechenzeit auf einem nicht allzu modernen Laptop erhält man die ersten 120 Stellen von I zu

```
0.0503599689 6688684979 7544014846 1897520242 8924981218 3776641975
  3134568451 1099131276 6212096387 6770935154 8156133243 0075084237.
```

Beispiel 3

Die unendliche Matrix $A = (a_{ij})$, definiert durch

$$a_{ij} = \begin{cases} \frac{1}{i+j-1} & : i+j \text{ ist Primzahl} \\ 0 & : \text{sonst,} \end{cases} \qquad i, j \geq 1, \tag{41}$$

bildet einen unendlich-dimensionalen Operator auf l^2. Welchen Wert hat

$$a = \|A\| = \sup_{x \neq 0} \frac{\|A\|_2}{\|x\|_2}, \tag{42}$$

wobei $\|x\|_2 = \sum_{i=1}^{\infty} x_i^2$?

Zuerst einmal stellt sich die Frage nach der Beschränktheit dieses Operators. Dies ist aber leicht zu beantworten, da A aus dem Hilbert-Operator (mit Norm π) durch Nullsetzen entsprechender Matrixeinträge hervorgeht, weshalb $\|A\| \leq \pi$.

Wir werden in weiterer Folge Projektionen A_n des Operators A auf endlichdimensionale Unterräume des l^2 betrachten, deren Normen mit Vektoriteration bestimmen und dann auf die Norm von A extrapolieren. Wegen

$$\|A_n\| = \|P_n A P_n\| \leq \|P_n\| \cdot \|A\| \cdot \|P_n\| \leq \|A\|, \tag{43}$$

wobei P_n der Projektor von l^2 in den endlichdimensionalen Teilraum ist, sind die Normen der A_n alle kleiner oder gleich π. Auf entsprechendem Wege folgt weiters, dass $\|A_n\| \leq \|A_m\|$ für $n < m$.

Vektoriteration

Erste Implementierungen mit vollen Matrizen A_n liefern die Normen

n	$\|A_n\|$
64	1.31782
128	1.32008
256	1.32117
512	1.32168
1024	1.32193
2048	1.32205

Tabelle 3: Erste Schätzungen der Norm mittels Vektoriteration bei voller Speicherung der Matrix.

Der enorme Speicherbedarf bei Speicherung der vollen Matrix verhindert das Berechnen der Norm für größere n, weshalb nur noch die Diagonale der Matrix gespeichert wird. Damit wird der Speicherbedarf für die Matrix von n^2 auf $2n - 1$ reduziert, was sehr große Werte von n ermöglicht. Die Vektoriteration benötigt weiterhin $O(n^2)$ Rechenschritte, weshalb zu erwarten ist, dass wir nun durch die Rechenleistung limitiert sind.

$\log_2(n)$	$\|\|A_n\|\|$
6	1.31782
7	1.32008
8	1.32117
9	1.32168
10	1.32193
11	1.32205
12	1.3221147
13	1.3221437
14	1.3221580
15	1.3221649
16	1.3221682

Tabelle 4: Durch Vektoriteration berechnete Normen mit gewöhnlichem Matrix-Vektor-Produkt und Speicherung der Matrix-Diagonale.

Bei $n = 2^16 = 65536$ betragen die Rechenzeiten schon mehrere Minuten, was sich mit jeder weiteren Verdopplung von n mit einem Faktor 4 in der Laufzeit niederschlägt. Eine Verbesserung des Laufzeitverhaltens ist also gefragt, wenn mehr als sechs Ziffern herausspringen sollen.

Geschwindigkeit mittels Fast Fourier Transformation

Bis jetzt haben wir zwar die Hankel-Matrix-Struktur der A_n zur Reduktion des Speicherbedarfs benutzt, jedoch noch nicht zur Verbesserung des Laufzeitverhaltens. Dies soll nun der nächste Schritt sein: Bezeichne $\mathbf{d}$ den Vektor der Diagonaleinträge von A_n, sei $\mathbf{x}$ ein Vektor der Länge n und bezeichne $\mathbf{y} = A_n\mathbf{x}$, wobei der Index jeweils bei Null startet. Dann gilt

$$\mathbf{y}[i] = (A_n \cdot \mathbf{x})[i] = \sum_{j=0}^{n-1} d[i+j]\mathbf{x}[j]. \tag{44}$$

Momentan ist die obere Summe lediglich ein Korrelationsprodukt von $\mathbf{d}$ und $\mathbf{x}$. Sei nun $\tilde{\mathbf{x}}[i] = \mathbf{x}[n-1-i]$ für $i = 0, \ldots, n-1$. Dann können wir

$$\mathbf{y}[i] = (A_n \cdot \mathbf{x})[i] = \sum_{j=0}^{n-1} d[i+j]\mathbf{x}[j] = \sum_{j=0}^{n-1} d[i+n-1-j]\tilde{\mathbf{x}}[j] = \sum_{j=0}^{n-1} d[n-1-j+i]\tilde{\mathbf{x}}[j] = (A_n * \tilde{\mathbf{x}})[i] \tag{45}$$

schreiben, wobei $*$ die Faltung bedeutet. Diese kann mittels *Fast Fourier Transformation* aber mit Aufwand $O(n\log(n))$ berechnet werden:

$$A_n * x = \mathcal{F}^{-1}\left(\mathcal{F}(A_n).*\mathcal{F}(\tilde{\mathbf{x}})\right), \tag{46}$$

wobei $.*$ die punktweise Multiplikation der beiden Vektoren bedeutet.

Die Implementierung erfolgte mit Hilfe der `fftw`-Bibliothek in C, um minimalen Speicherbedarf bei maximaler Laufzeiteffizient zu erzielen. Es zeigte sich nämlich, dass nun wieder der Speicher der limitierende Faktor ist, selbst zwei Gigabyte RAM sind mit $n = 2^{25}$ erschöpft.

ld(n)	$\|\|A_n\|\|$
6	1.31782
7	1.32008
8	1.32117
9	1.32168
10	1.32193
11	1.32205
12	1.3221147
13	1.3221437
14	1.3221580
15	1.3221649
16	1.3221682
17	1.322169881256
18	1.322170677300
19	1.322171066454
20	1.322171255802
21	1.322171348429
22	1.322171393652
23	1.322171415762
24	1.322171426584
25	1.322171431885

Tabelle 5: Approximation der Norm mit Hilfe der Fourier-Transformation für die Matrix-Vektor-Multiplikation

Konvergenzbeschleunigung

Die bisherigen Zahlen erlauben ohne weiteres das Ablesen von neun Ziffern. Wenn man jedoch das Wachstumsverhalten von $\|\|A_n\|\|$ mit n beobachtet, so ist zu erhoffen, dass noch die eine oder andere zusätzliche Ziffer bestimmt werden kann.

Mit Hilfe des *Wynnschen Epsilon-Algorithmus* wurde in Tab. 6 die Konvergenz beschleunigt.

$\log_2(n)$	$\|\|A_n\|\|$	Wynn	$\|\|A_n\|\|$ - $\|\|A_{n/2}\|\|$	$(\|\|A_{n/2}\| - \|\|A_{n/4}\|\|)/(\|\|A_n\| - \|\|A_{n/2}\|\|)$
6	1.31782	1.32219831641198	-	
7	1.32008	1.32213078585147	0.002249132741	
8	1.32117	1.32217092331046	0.001091926384	2.059784225344190
9	1.32168	1.32217443847139	0.000511652454	2.134117359280210
10	1.32193	1.32217353330172	0.000250630545	2.041460884188680
11	1.32205	1.32217006217489	0.000123678541	2.026467509828960
12	1.3221147	1.32217191392704	0.000060798487	2.034237151330750
13	1.3221437	1.32217126639956	0.000028963550	2.099137950982410
14	1.3221580	1.32217146041032	0.000014289083	2.026970520053470
15	1.3221649	1.32217143731168	0.000006879353	2.077096930528850
16	1.3221682	1.32217142959992	0.000003363422	2.045343403333370
17	1.322169881256	1.32217143864726	0.00000163837421	2.052902187493930
18	1.322170677300	1.32217143523981	0.00000079604360	2.058146324864560
19	1.322171066454	1.32217143713406	0.00000038915452	2.045572026935260
20	1.322171255802	1.32217143679345	0.00000018934815	2.055232754283740
21	1.322171348429	1.32217143691325	0.00000009262659	2.044209445071910
22	1.322171393652	1.32217143696078	0.00000004522265	2.048234453578140
23	1.322171415762	1.32217143697501	0.00000002211018	2.045331618960740
24	1.322171426584	-	0.00000001082247	2.042988323858040
25	1.322171431885	-	0.00000000530108	2.041559454236760

Tabelle 6: Konvergenzbeschleunigung und Abnahmerate der Differenzen zwischen aufeinanderfolgenden Normen.

In Tab. 6 liefert der Wynnsche Epsilon-Algorithmus $a = 1.3221714369$, wobei die letzten zwei Ziffern

nicht ganz sicher sind. Der Neuner am Ende ist in diesem Fall sichtlich ungünstig, da er die Ziffer davor auch noch kippen könnte. Daher habe ich versucht, mittels einer alternativen Methode die letzten beiden Ziffern zu bestätigen oder zu widerlegen: Ich habe die Differenzen zwischen in der Tabelle aufeinanderfolgende Normen betrachtet und festgestellt, dass die Differenzen in sehr guter Näherung eine geometrische Folge bilden. Danach habe ich $n = 8$ als Startwert für die Folge

$$a_8 = 1.32117, \quad a_{k+1} = a_k + cq^k, \quad k > 7 \tag{47}$$

verwendet. Die beiden Parameter c und q habe ich über Interpolationsbedingungen für a_9 und a_{25} aus der Tabelle bestimmt und schließlich den Grenzwert für $k \to \infty$ bestimmt. Hiermit ergibt sich der (gerundete) Grenzwert $a_\infty = 1.322171436979$, welcher sogar eine zusätzliche Stelle mit dem letzten Wert des Wynnschen Epsilon-Algorithmus gemein hat.

Da wir nun die Stellen zehn und elf mit zwei unterschiedlichen Methoden bestimmt haben, können wir somit ruhigen Gewissens $a = 1.3221714369$ angeben.

Beispiel 4

Bestimmen Sie $c > 0$, so daß die Lösung von

$$-\Delta u = e^u \quad \text{in } \Omega = (0,1)^2, \quad u = c \quad \text{auf } \Gamma = \{0\} \times (0,1), \quad u = 0 \text{auf } \partial\Omega \setminus \Gamma, \tag{48}$$

die Beziehung $u\left(\frac{1}{3}, \frac{1}{3}\right) = 1$ erfüllt.

Zuerst fällt auf, dass es sich um eine nichtlineare partielle Differentialgleichung handelt. Damit sind leider viele analytische Zugänge wie Fourier-Synthese oder Ausnutzung der Linearität zur Vereinfachung nicht möglich. Daher habe ich mich auf rein numerische Methoden beschränkt.

Die partielle Differentialgleichung wurde mittels `SGFramework` und einem Finite-Differenzen-Verfahren implementiert und gelöst, wobei c händisch so gewählt wurde, dass $u\left(\frac{1}{3}, \frac{1}{3}\right) = 1$ erfüllt ist. Damit erhält man $c \doteq 2.2202$, wobei weitere Ziffern zum Einen durch die Laufzeit des Programms und zum Anderen durch die verwendete Diskretisierung (5-Punkt oder 9-Punkt) limitiert sind, siehe Tab. 7.

DIM	5-Stern	9-Stern
120	2.22030	2.22036
150	2.22027	2.22023
180	2.22025	2.22023
240	2.22024	2.22023
300	2.22023	2.22022
330	2.22023	2.22022

Tabelle 7: Werte für c für eine Fünf-Stern- und eine Neun-Stern-Diskretisierung mit Finiten Differenzen. DIM gibt die Anzahl der Gitterpunkte pro Achse an.

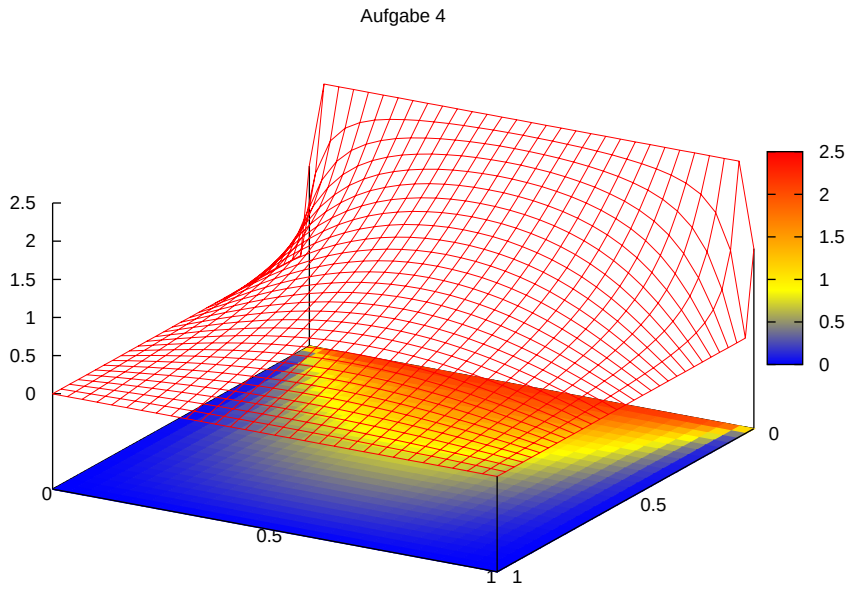


Abbildung 4: Die mittels Finiten Differenzen approximierte Lösung $u(x, y)$

Leider war es mir nicht möglich, die obigen Ergebnisse mit anderer Software zu überprüfen, weshalb bei diesem Beispiel die etwas magere und unsichere Ausbeute $c \doteq 2.2202$ bleibt.

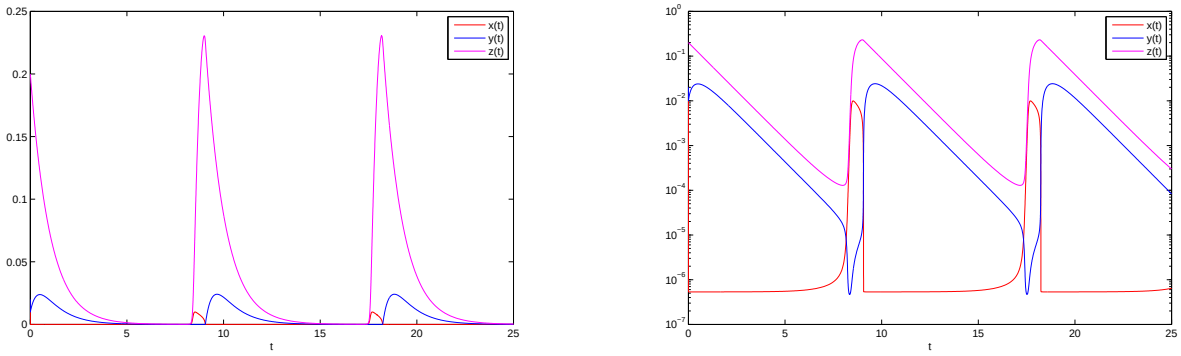
Beispiel 5

Die Lösung (x, y, z) des Anfangswertproblems

$$\dot{x} = k_1 y - k_2 xy + k_3 x - k_4 x^2 \tag{49}$$
$$\dot{y} = -k_1 y - k_2 xy + k_5 fz \tag{50}$$
$$\dot{z} = 2k_3 x - k_5 z \tag{51}$$

mit $(x, y, z)(0) = (0.01, 0.02, 0.2)$ und $k_1 = 1.28$, $k_2 = 2.4 \cdot 10^6, k_3 = 33.6, k_4 = 3000, k_5 = 1.0$ und $f = 0.5$ ist periodisch. Bestimmen Sie das kleinste $p > 0$, sodass $x(t) = x(t + p)$ für alle $t > 0$.

Das vorliegende Differentialgleichungssystem ist aus der Klasse der *steifen* Differentialgleichungen, was insbesondere Probleme mit numerischen Methoden mit sich bringt. In Abb. 5 ist der Zeitverlauf über ein paar Perioden zu sehen, wobei deutlich ersichtlich ist, dass über einen Großteil der Periode wenig "passiert".



(a) Lineare Skala

(b) Logarithmische Skala

Abbildung 5: Zeitverlauf der Systemvariablen.

Erste Untersuchungen und Fragen der Periodizität

Obwohl das System steif ist, wurden erste Untersuchungen mit einem expliziten Euler-Verfahren unternommen. Dabei stellte sich wenig überraschend heraus, dass sehr kleine Schrittweiten (kleiner 0.00005) notwendig sind, um überhaupt sinnvolle Lösungskurven zu erhalten. Weiters zeigte sich, dass die Periodenlänge bei ca. 9.2 liegt (vgl. Abb. 5), was bereits gut 200.000 Auswertungen erfordert. Da aber moderne Rechner mit sehr viel Rechenpower ausgestattet sind und außerdem der Auswertungsaufwand für das vorliegende System sehr gering ist, bleibt noch genug Raum für Verbesserungen der Genauigkeit.

Wie ein Kollege im Seminar richtigerweise bemerkte, stellt sich überhaupt erst einmal die Frage nach der Periodizität des gegebenen Systems. Für (51) lässt sich nämlich die Lösung für $z(t)$ angeben als

$$z(t) = 2k_3 \int_{t_0}^{t} e^{s-t} x(s)\, \mathrm{d}s, \tag{52}$$

für ein passendes t_0, wobei $x(t)$ nach Angabe periodisch ist. Damit $z(t)$ auch wirklich die selbe Periode wie $x(t)$ hat, müssen die Anfangsbedingungen tatsächlich so sein, dass $t_0 = -\infty$ ist - dies ist aber höchst unwahrscheinlich, weshalb wir o.B.d.A.[2] die Angabe so verstehen, dass nach der Periode p des Grenzzykels gefragt ist. Für die numerische Behandlung hat das dementsprechend zur Folge, dass wir die anfänglichen Einschwingvorgänge nicht in unser System miteinbeziehen dürfen.

[2] ohne Beleidigung des Aufgabenstellers

Um das Abklingverhalten der transienten Vorgänge abzuschätzen, betrachten wir $z(t;t_0) := 2k_3 \int_{t_0}^{t} e^{s-t}x(s)\,\mathrm{d}s$ und vergleichen mit $z(t+p;t_0)$:

$$z(t+p;t_0) = 2k_3 \int_{t_0}^{t+p} e^{s-t-p}x(s)\,\mathrm{d}s \overset{r=s-p}{=} 2k_3 \int_{t_0-p}^{t} e^{r-t} \underbrace{x(r-p)}_{=x(r)}\,\mathrm{d}r = z(t;t_0-p) = z(t;t_0) + 2k_3 \int_{t_0-p}^{t_0} e^{s-t}x(s)\,\mathrm{d}s. \quad (53)$$

Klarerweise gilt für den Grenzzykel $z(t;-\infty)$ wegen der Periodizität $z(t;-\infty) = z(t+p;-\infty)$. Im Vergleich dazu tritt in (53) der zusätzliche Term $2k_3 \int_{t_0-p}^{t_0} e^{s-t}x(s)\,\mathrm{d}s = 2k_3 e^{-t} \int_{t_0-p}^{t_0} e^{s}x(s)\,\mathrm{d}s$ auf, der als Maß für das Abklingen der transienten Vorgänge angesehen werden kann. Ein Voranschreiten um eine Periode in der Zeit hat damit den Effekt, dass t_0 für die nächste Periode um p kleiner erscheint, was den Fehlerterm um e^{-p} verkleinert. Wir können also erwarten, dass bei $p \approx 9.2$ mit jeder Periode die transienten Vorgänge um ca. einen Faktor 10.000 kleiner werden. Nach ein paar Perioden wäre damit der Grenzzykel im Rahmen der Rechengenauigkeit erreicht, jedoch muss noch berücksichtigt werden, dass auch $x(t)$ und $y(t)$ erst einschwingen müssen, weshalb auch eine numerische Überprüfung der Einschwingvorgänge notwendig ist.

Resultate mit Standardalgorithmen

Nach den ersten Betrachtungen hat sich das folgende Simulationssetting herauskristallisiert: Eine Simulation soll von $t = 0$ bis $t = 1000$ durchgeführt werden. Im Zeitrahmen $t = 200 \ldots 1000$ werden die Perioden durch jene Zeitpunkte $t_i, i = 0 \ldots n$ detektiert, für die $z(t_i) = 0.1, z(t_i - \varepsilon) < 0.1$ für kleine ε gilt. Damit ist $p \doteq \frac{t_n - t_0}{n}$.

Der Grund für die Nichtberücksichtigung der Simulationsergebnisse im Bereich $t = 0 \ldots 200$ liegt im Verwerfen der transienten Zustände, während für den Bereich $t = 200 \ldots 1000$ knapp 90 Perioden auftreten, die Ablesefehler und numerisches Rauschen mildern.

Die eben beschriebene Simulation wurde schließlich in `MATLAB` implementiert. Als Differentialgleichungslöser wurde zuerst der Standardlöser `ode45` versucht, der aber aufgrund der Steifigkeit des Systems viel zu langsam arbeitet. Für steife Systeme werden `ode15s` und `ode23s` empfohlen, davon blieb aber ersterer bei rund $t = 300$ mit einem Fehler stehen und war auch mit anderen Optionen nicht zur Zusammenarbeit zu überreden.

Erste Resultate zeigten schon bald, dass die eingestellten Standardtoleranzen zu groß waren. Kleinere Toleranzen führten dagegen schnell zu langen Rechenzeiten, wie in Tab. 8 zu sehen ist.

Rel.Tol.	Abs.Tol	p	RZ (sek)
1e-8	1e-8	9.1765157	62
1e-10	1e-8	9.1765157	62
1e-9	1e-9	9.1758975	152
1e-10	5e-10	9.1758791	200
1e-11	5e-10	9.1758791	200
1e-12	5e-10	9.1758797	200
1e-12	2e-10	9.1758633	292
1e-12	1e-10	9.1758683	400
1e-12	5e-11	9.1758691	540
1e-13	2e-11	9.1758681	833
1e-13	1e-11	9.1758707	1174

Tabelle 8: Ermittelte Werte für p mittels `ode23s` für verschiedene Toleranzen. Für kleine Toleranzen wächst die Rechenzeit (RZ) merklich an. Die Ablesetoleranzen liegen bei etwa $5 \cdot 10^{-6}$, weshalb nur sieben Nachkommastellen angegeben werden.

Aus den Ergebnissen aus Tab. 8 sind die Ziffern `9.1758` mit guten Abschätzungen über die fünfte Nachkommaziffer ablesbar, (wesentlich) höhere Genauigkeiten sind aber durch die Laufzeit beschränkt, außerdem sind aufgrund der eingebauten Schrittweitensteuerung keine Extrapolationsmethoden anwendbar.

Triumph der Holzhammermethode(n)

Im Rahmen der Analyse mit Standardalgorithmen stellte sich heraus, dass die Ablesegenauigkeit (das ist die Genauigkeit, mit der die Periodenlänge durch überschreiten einer Schwelle bestimmt wird) zu gering für mehr als sechs oder sieben Ziffern ist. Um beispielsweise acht Ziffern bestimmen zu können, ist bei rund 100 simulierten Perioden eine Schrittweite von ca. 10^{-6} notwendig. Für $t = 0 \ldots 1000$ bedeutet das wiederum 10^9 Schritte! Ein einzelner Schritt muss daher numerisch so billig wie möglich sein, womit implizite Methoden praktisch ausscheiden, da das gegebenen System sogar nichtlinear ist, was zusätzlichen Aufwand in Form von Newton-Iterationen erfordern würde.

Explizite Einschrittverfahren gibt es zur Genüge, wir beginnen daher mit einem expliziten Euler-Verfahren. Wie eingangs erwähnt, muss die Schrittweite ausreichend klein sein, um überhaupt sinnvolle Lösungen zu erhalten. Der globale Fehler verhält sich bei diesem Verfahren wie $O(h)$, weshalb wir auf die Anwendbarkeit von Konvergenzbeschleunigern hoffen können.

$\log_2(\frac{1}{h})$	p	RZ (sek)
15	9.17591801396	1
16	9.17589512577	2
17	9.17588375232	4
18	9.17587803028	8
19	9.17587516925	16
20	9.17587372991	33
21	9.17587301907	67
22	9.17587265924	130
23	9.17587248153	260
24	9.17587239212	520
25	9.17587234769	1039
26	9.17587232534	2080
27	9.17587231410	4159

Tabelle 9: Ermittelte Werte für p mittels explizitem Euler-Verfahren. Die Rechenzeit (RZ) hängt wenig überraschend linear von der Schrittanzahl ab.

In Tab. 9 sind die ermittelten Periodendauern im Bereich $t = 1000 \ldots 2000$ zu sehen, wobei insbesondere die asymptotische Näherung an einen Grenzwert von oben zu beobachten ist. Insbesondere fällt auf, dass trotz der hohen Schrittanzahl dennoch weniger Rechenzeit notwendig ist als für `ode23s`.

Die naheliegende Verbesserung ist die Verwendung von Einschrittverfahren mit höherer Ordnung. In Tab. 10 sind die Ergebnisse bei der Verwendung des Heun-Verfahrens gezeigt. Die Genauigkeit der Ergebnisse verglichen mit dem expliziten Euler-Verfahren ist beeindruckend, jedoch sind Extrapolationsmethoden nicht mehr anwendbar. Außerdem ist zu beachten, dass zu große Schrittweiten aufgrund der Steifigkeit des Systems nach wie vor keine sinnvollen Lösungen ergeben. Dennoch haben wir mit guter Gewissheit eine zusätzliche Ziffer gefunden.

$\log_2(\frac{1}{h})$	p	RZ (sek)
15	9.1758722375	7
16	9.1758723788	14
17	9.1758723082	28
18	9.1758723082	55
19	9.1758723082	110
20	9.1758722994	220
21	9.1758723038	440
22	9.1758723016	880
23	9.1758723027	1740
24	9.1758723033	2500

Tabelle 10: Ermittelte Werte für p mittels explizitem Heun-Verfahren. Bereits ein Minimum an Rechenzeit (RZ) liefert sieben korrekte Stellen.

Es wäre nun naheliegend, Verfahren mit noch höheren Ordnungen zu verwenden. Das klassische Runge-Kutta-Verfahren lieferte aber exakt die selben Ziffern wie das Heun-Verfahren, benötigt jedoch die dreifache Laufzeit, wodurch die mögliche Schrittweite zunehmend beeinträchtigt wird. Aus diesem Grund wurden keine weiteren Verfahren mit höheren Ordnungen untersucht.

Zu guter Letzt: Konvergenzbeschleunigung

Die Werte für p aus dem expliziten Euler-Verfahren erlauben (wie schon bei Beispiel 3) die Anwendung des *Wynnschen Epsilon-Algorithmus*.

$\log_2(\frac{1}{h})$	p_h	Wynn	$p_h - p_{h/2}$	$\frac{p_{h/2}-p_{h/4}}{p_h-p_{h/2}}$
15	9.17591801396	9.175872518430650	-	-
16	9.17589512577	9.175872236711010	-0.000022888183594	-
17	9.17588375232	9.175872308236580	-0.000011373449254	2.012422360341580
18	9.17587803028	9.175872272695920	-0.000005722045898	1.987654320895670
19	9.17587516925	9.175872325469080	-0.000002861022949	2.000000000000000
20	9.17587372991	9.175872290353800	-0.000001439341792	1.987730061258840
21	9.17587301907	9.175872308129550	-0.000000710840578	2.024844723213150
22	9.17587265924	9.175872301600050	-0.000000359835449	1.975460117611380
23	9.17587248153	9.175872303814650	-0.000000177710145	2.024844715592120
24	9.17587239212	9.175872302714180	-0.000000089406967	1.987654328346250
25	9.17587234769	9.175872302715940	-0.000000044427535	2.012422393029740
26	9.17587232534	-	-0.000000022351742	1.987654288609820
27	9.17587231410	-	-0.000000011244852	1.987731049877820

Tabelle 11: Konvergenzbeschleunigung und Abnahmerate der Differenzen zwischen aufeinanderfolgenden Werten für p.

In Tab. 11 liefert der Wynnsche Epsilon-Algorithmus $p \doteq 9.175872302$, wobei über die letzte Ziffer (neunte Nachkommastelle) noch etwas gesagt werden muss. Einerseits liefert der Epsilon-Algorithmus als neunte und zehnte Nachkommastellen 27 bei der Beschleunigung der genauesten Werte, andererseits liefert die Interpolation mit einer geometrischen Reihe wie in Beispiel 3 mit den Abnahmeraten 1.98 und 2.01 die Werte 9.1758723029 bzw. 9.1758723026, wobei die interpolierte Abnahmerate 1.993 beträgt. Überdies bestätigt die Mittelung der letzten drei aus dem Heun-Verfahren gewonnen Werte für p die neunte Nachkommastelle 2.

Damit können wir mit guter Gewissheit zehn Ziffern angeben: $p \doteq 9.175872302$.

Anhang

Listing 1: Maple-Code zu Aufgabe 1

```
> restart;
> with(linalg):
> Digits := 5042;
> f := (x,y) -> sin(cos(x*y)) + cos(sin(x+2*y)) +sin(exp(x*x)) + exp(cos(y)) + x*x + y*
    y;
> fx := D[1](f);
> fy := D[2](f);
> fxx := D[1](fx);
> fxy := D[2](fx);
> fyx := D[1](fy);
> fyy := D[2](fy);
> newtonsearch := proc (iternum) local i,x,y, deltavec, jacobimat;
    x := 1.24;
    y := -1.33;
    for i from 1 to iternum do
       jacobimat := array([[evalf(fxx(x,y)), evalf(fxy(x,y))], [evalf(fxy(x,y)), evalf(
           fyy(x,y))]]);
       deltavec := multiply(inverse(jacobimat), vector([ -1.0 * evalf(fx(x,y)), -1.0 *
           evalf(fy(x,y))]) );
       x := deltavec[1] + x;
       y := deltavec[2] + y;
       #print(vector([x,y,evalf(f(x,y))]));
    end do;
    evalf(f(x,y));
end proc;
> newtonsearch(15);
```

Listing 2: Maple-Code zu Aufgabe 2

```
> restart;
> computeIntegral := proc (ubound) local iternum, i, ival, alph, lastprefactor,
    summandvalue, maxval, summand, summandplus, summandminus, errorest, incGamma1,
    incGamma2, uboundup4, maxindex;

        ival := 0.0;

    #estimate number of terms necessary in series:
    Digits := 10;
    i := 500;
    summandvalue := 1.0;
    maxval := 0.0;
    maxindex := 10;
    while summandvalue > evalf(10^(-100)) do
      alph := (2 * i + 1) / 4.0;
      summandvalue := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph *
          hypergeom([alph],[1+alph],-ubound^4))^6);
      if maxval < summandvalue then
         maxindex := i;
         maxval := summandvalue;
      end if;
      i := i + 100;
    end do;
    iternum := i;
    print("Estimated terms in summation:");
    print(iternum);

    #estimate number of necessary digits:
    Digits := 10;
    maxval := 0.0;
    summand := 0.0;
    i := max(20, maxindex);

        alph := (2 * i + 1) / 4.0;
        summand := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph *
            hypergeom([alph],[1+alph],-ubound^4))^6);
        i := i + 10;
        alph := (2 * i + 1) / 4.0;
        summandplus := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph *
            hypergeom([alph],[1+alph],-ubound^4))^6);
        i := i - 20;
        alph := (2 * i + 1) / 4.0;
        summandminus := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph *
            hypergeom([alph],[1+alph],-ubound^4))^6);
        i := maxindex;

        while true do
            if (summand > summandplus and summand > summandminus) or i < 20 then
            maxval := summand;
            break;
        elif summandplus > summandminus then
            i := i + 20;
            summandminus := summand;
                summand := summandplus;
            alph := (2 * i + 1) / 4.0;
            summandplus := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph *
                 hypergeom([alph],[1+alph],-ubound^4))^6);
            i := i - 10;
        else
            i := i - 20;
            summandplus := summand;
            summand := summandminus;
            alph := (2 * i + 1) / 4.0;
```

```
            summandminus := evalf(2^(2*i-7) / factorial(2*i) * ((ubound^4)^alph / alph
                * hypergeom([alph],[1+alph],-ubound^4))^6);
            i := i + 10;
        end if;
        end do;

    Digits := 200 + max(0, round(log10(maxval)));   # 200 digits after comma should be
        correct
    print("Estimated number of digits:");
    print(Digits);

    #evaluate series from back to front
    lastprefactor := (-1)^(iternum-1) * 2^(2*iternum-7) / factorial(2*iternum);
    alph := (2 * iternum + 5) / 4.0;
    incGamma1 := (ubound^4)^alph / alph * hypergeom([alph],[1+alph],-ubound^4);
    alph := (2 * iternum + 3) / 4.0;
    incGamma2 := (ubound^4)^alph / alph * hypergeom([alph],[1+alph],-ubound^4);
    uboundup4 := ubound^4;

        for i from iternum by -2 to 2 do

        if i mod 300 = 0 then
           # recompute:
           alph := (2 * i + 1) / 4.0;
           incGamma1 := (uboundup4)^alph / alph * hypergeom([alph],[1+alph],-uboundup4)
               ;
           alph := (2 * i - 1) / 4.0;
           incGamma2 := (uboundup4)^alph / alph * hypergeom([alph],[1+alph],-uboundup4)
               ;
        else
           # use recursion formula:
           alph := (2 * i + 1) / 4.0;
           incGamma1 := evalf((incGamma1 + exp(-uboundup4) * (uboundup4)^alph) / alph);
           alph := (2 * i - 1) / 4.0;
           incGamma2 := evalf((incGamma2 + exp(-uboundup4) * (uboundup4)^alph) / alph);
        end if;

        summand := lastprefactor * incGamma1^6 - lastprefactor * (2*i) * (2*i-1) *
            incGamma2^6 / 4.0;
        ival := ival + evalf(summand);
        lastprefactor := evalf(lastprefactor * (2*i) * (2*i-1) * (2*i-2) * (2*i-3) /
            16.0);

        #readjust number of digits:
        Digits := max(200, 200 + round(log10(abs(summand))));
        end do;
    print(evalf(ival));

    #compute error estimation:
    Digits := 20;
    errorest := evalf(6 * (0.5 * GAMMA(0.25, ubound^4)) * (0.5 * ((ubound^4)^0.25 /
        0.25 * hypergeom([0.25],[1.25],-ubound^4)) )^5);
    print("Estimated error:");
    print(errorest);

    end proc;
> computeIntegral(4.2);
```

Listing 3: C-Programm zu Aufgabe 3

```c
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <math.h>
#include <complex.h>
#include <fftw3.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  long long N = 1;
  char *primesarray;    //primesarray[i] represents prime-state of i+1
  long i, j;
  long temp;
  long double op_norm = 0.0;
  long iternum = 20;

  fftw_complex *matrix_elements, *xvector;
  fftw_plan p_matrix, p_vector_forward, p_vector_backward;

  // STEP 1: Read power of 2:
  printf("Enter power of 2 for upper bound: ");
  scanf("%lld", &temp);
  getchar();

  //create power of 2:
  for (i=0; i<temp; ++i)
        N *= 2;

  // STEP 2: Generate all needed primes:
  primesarray = (char *) malloc(sizeof(char) * N);

  for (i = 0; i<N; ++i)
  {
    primesarray[i] = 1;
  }

  for (i = 1; i<N; ++i)
  {
        if (primesarray[i] == 1)
        {
                //remove all higher primes:
                for (j=2*i+1; j < N; j = j + i + 1)
                {
                        primesarray[j] = 0;
                }
        }
  }

  // STEP 3: Set up the matrix vector:
  matrix_elements = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  for (i=1; i<N; ++i)
  {
    if (primesarray[i] == 1)
        matrix_elements[i-1] = 1.0 / i;
    else
        matrix_elements[i] = 0.0;
  }
  matrix_elements[N-1] = 0.0;
  free(primesarray);

```

```
64  // STEP 4: Compute DFT of matrix:
65  p_matrix = fftw_plan_dft_1d(N, matrix_elements, matrix_elements, FFTW_FORWARD,
        FFTW_ESTIMATE);
66  fftw_execute(p_matrix);
67
68  // STEP 5: Set up the x-vector:
69  xvector = (fftw_complex *) fftw_malloc(sizeof(fftw_complex) * N);
70  for (i=0; i<N/2; ++i)
71  {
72    xvector[i] = i+1;
73    //xvector[i][1] = 0.0;
74  }
75  for (i=N/2; i<N; ++i)
76  {
77    xvector[i] = 0.0;
78    //xvector[i][1] = 0.0;
79  }
80
81  printf("Starting power iteration...\n");
82
83  // STEP 6: Run power iteration:
84  p_vector_forward = fftw_plan_dft_1d(N, xvector, xvector, FFTW_FORWARD, FFTW_ESTIMATE)
        ;
85  p_vector_backward = fftw_plan_dft_1d(N, xvector, xvector, FFTW_BACKWARD,
        FFTW_ESTIMATE);
86
87  for (i = 0; i<iternum; ++i)
88  {
89        //Status:
90        if (i % 5 == 0)
91                printf("Iteration %ld...\n", i);
92
93        //x-vector and remove complex contributions:
94        fftw_execute(p_vector_forward);
95
96        //do a pointwise multiplication:
97        for (j=0; j<N; ++j)
98                xvector[j] *= matrix_elements[j] / (double) N;
99
100       fftw_execute(p_vector_backward);
101
102       //set up new xvector:
103       for (j=0; j<N/2; ++j)
104               xvector[j] = creal(xvector[N - j - 2]);
105       for (j=N/2; j<N; ++j)
106               xvector[j] = 0.0;
107
108       if (i == iternum-2)
109       {
110               //normalise:
111               op_norm = 0.0;
112               for (j=0; j<N/2; ++j)
113                       op_norm += ( (long double) creal(xvector[j])) * ((long double)
                            creal(xvector[j]));
114
115               op_norm = sqrtl(op_norm);               //long double sqrt
116
117               for (j=0; j<N/2; ++j)
118                       xvector[j] = xvector[j] / op_norm;
119       }
120 }
121
122 // Step 7: compute final norm:
123 op_norm = 0.0;
```

```
124   for (j=0; j<N/2; ++j)
125         op_norm += ( (long double) creal(xvector[j])) * ((long double) creal(xvector[j
              ]));
126   op_norm = sqrtl(op_norm);             //long double sqrt
127
128   // FINAL STEP: Clean up:
129   fftw_destroy_plan(p_matrix);
130   fftw_destroy_plan(p_vector_forward);
131   fftw_destroy_plan(p_vector_backward);
132   fftw_free(xvector); fftw_free(matrix_elements);
133
134   printf("Computed norm: %.12Lf\n", op_norm);
135
136   return EXIT_SUCCESS;
137 }
```

Listing 4: SG-Framework-Skript zu Aufgabe 4

```
// Karl Rupp
// 0325941
const MAGICPOINT = 100;
const DIM = 3 * MAGICPOINT  + 1;
const DX = 1.0/(DIM-1);
const C = 2.22022;

SET NEWTON ACCURACY = 1e-12;

// quantities
var U[DIM, DIM];
var OUT[1];

unknown U[1..DIM-2, 1..DIM-2];
unknown OUT[0];
known U[0..DIM-1, 0];

//equ U[i=1..DIM-2, j=1..DIM-2] -> -1.0*exp(U[i,j]) = (U[i+1,j] + U[i,j+1] - 4.0*U[i,j]
     + U[i-1,j] + U[i,j-1])/sq(DX); //5-point
equ U[i=1..DIM-2, j=1..DIM-2] -> -1.0*exp(U[i,j]) = (U[i+1,j-1] + 4.0*U[i+1, j] + U[i
    +1,j+1] + 4.0*U[i, j-1] - 20.0*U[i,j] + 4.0*U[i, j+1] + U[i-1, j-1] + 4.0*U[i-1, j]
     + U[i-1,j+1])/(6.0*sq(DX));  //9-point

equ OUT[i=0] -> OUT[i] = U[MAGICPOINT, MAGICPOINT];

begin main
// boundary conditions:
    assign U[i=0, j=1..DIM-1] = 0.0;
    assign U[i=DIM-1, j=1..DIM-1] = 0.0;
    assign U[i=0..DIM-1, j=DIM-1] = 0.0;
    assign U[i=1..DIM-2, j=0] = C;
    assign U[i=0, j=0] = C/2.0;
    assign U[i=DIM-1, j=0] = C/2.0;

    solve; write;
end
```

Listing 5: C-Programm zu Aufgabe 5

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// use 'long double' precision (80 bit)
#define _LONG_ long
// OR use 'double' only (64 bit)

//choose between FORWARDEULER, HEUN and RK4
#define HEUN

int main(int argc, char *argv[])
{
  _LONG_ double xnew, ynew, znew;
  _LONG_ double xold = 0.01;
  _LONG_ double yold = 0.02;
  _LONG_ double zold = 0.20;
  _LONG_ double xinter, yinter, zinter;
  _LONG_ double xk1, xk2, xk3, xk4;
  _LONG_ double yk1, yk2, yk3, yk4;
  _LONG_ double zk1, zk2, zk3, zk4;

  _LONG_ double stepsize = 1e-6;
  long stepsizeinput = 0;
  _LONG_ double aktpos = 0.0;

  _LONG_ double k1 = 1.28;
  _LONG_ double k2 = 2.4e6;
  _LONG_ double k3 = 33.6;
  _LONG_ double k4 = 3000.0;
  _LONG_ double k6 = 0.5;

  _LONG_ double J11, J12;
  _LONG_ double J21, J22, J23;
  _LONG_ double J31, J33;
  _LONG_ double detJ;

  _LONG_ double k2xy, k1y, k3x;

  long periodcount = -1;
  _LONG_ double startpos = 0.0;
  _LONG_ double endpos = 0.0;

  printf("Enter log2(1/stepsize): ");
  scanf("%ld", &stepsizeinput);

  stepsize = powl(2, -1.0 * stepsizeinput);
  printf("Step size: %Lg\n", stepsize);

  J23 = stepsize * k6;
  J31 = -2.0*k3*stepsize;
  J33 = stepsize;

  while (aktpos < 1000.0)
  {
    #ifdef FORWARDEULER
    xnew = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold);
    yold += stepsize * (-k1*yold - k2*xold*yold + k6*zold);
    zold += stepsize * (2.0*k3*xold - zold);

    xold = xnew;
    #endif

```

```
64      #ifdef HEUN
65      xinter = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold);
66      yinter = yold + stepsize * (-k1*yold - k2*xold*yold + k6*zold);
67      zinter = zold + stepsize * (2.0*k3*xold - zold);
68
69      xnew = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold + k1*
            yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter) / 2.0;
70      yold += stepsize * (-k1*yold - k2*xold*yold + k6*zold - k1*yinter - k2*xinter*
            yinter + k6*zinter) / 2.0;
71      zold += stepsize * (2.0*k3*xold - zold + 2.0*k3*xinter - zinter) / 2.0;
72
73      xold = xnew;
74      #endif
75
76      #ifdef RK4
77      xk1 = k1*yold - k2*xold*yold + k3*xold - k4*xold*xold;
78      yk1 = -k1*yold - k2*xold*yold + k6*zold;
79      zk1 = 2.0*k3*xold - zold;
80
81      xinter = xold + stepsize * xk1 / 2.0;
82      yinter = yold + stepsize * yk1 / 2.0;
83      zinter = zold + stepsize * zk1 / 2.0;
84
85      xk2 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
86      yk2 = -k1*yinter - k2*xinter*yinter + k6*zinter;
87      zk2 = 2.0*k3*xinter - zinter;
88
89      xinter = xold + stepsize * xk2 / 2.0;
90      yinter = yold + stepsize * yk2 / 2.0;
91      zinter = zold + stepsize * zk2 / 2.0;
92
93      xk3 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
94      yk3 = -k1*yinter - k2*xinter*yinter + k6*zinter;
95      zk3 = 2.0*k3*xinter - zinter;
96
97      xinter = xold + stepsize * xk3;
98      yinter = yold + stepsize * yk3;
99      zinter = zold + stepsize * zk3;
100
101     xk4 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
102     yk4 = -k1*yinter - k2*xinter*yinter + k6*zinter;
103     zk4 = 2.0*k3*xinter - zinter;
104
105     xold += stepsize * (xk1 + 2.0*xk2 + 2.0*xk3 + xk4) / 6.0;
106     yold += stepsize * (yk1 + 2.0*yk2 + 2.0*yk3 + yk4) / 6.0;
107     zold += stepsize * (zk1 + 2.0*zk2 + 2.0*zk3 + zk4) / 6.0;
108     #endif
109
110     aktpos = aktpos + stepsize;
111   }
112
113   printf("Reached 1000...\n");
114
115   while (aktpos < 2000.0)
116   {
117     #ifdef FORWARDEULER
118     xnew = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold);
119     yold += stepsize * (-k1*yold - k2*xold*yold + k6*zold);
120     znew = zold + stepsize * (2.0*k3*xold - zold);
121
122     xold = xnew;
123     #endif
124
125     #ifdef HEUN
```

```
126     xinter = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold);
127     yinter = yold + stepsize * (-k1*yold - k2*xold*yold + k6*zold);
128     zinter = zold + stepsize * (2.0*k3*xold - zold);
129
130     xnew = xold + stepsize * (k1*yold - k2*xold*yold + k3*xold - k4*xold*xold + k1*
            yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter) / 2.0;
131     yold += stepsize * (-k1*yold - k2*xold*yold + k6*zold - k1*yinter - k2*xinter*
            yinter + k6*zinter) / 2.0;
132     znew = zold + stepsize * (2.0*k3*xold - zold + 2.0*k3*xinter - zinter) / 2.0;
133
134     xold = xnew;
135     #endif
136
137     #ifdef RK4
138     xk1 = k1*yold - k2*xold*yold + k3*xold - k4*xold*xold;
139     yk1 = -k1*yold - k2*xold*yold + k6*zold;
140     zk1 = 2.0*k3*xold - zold;
141
142     xinter = xold + stepsize * xk1 / 2.0;
143     yinter = yold + stepsize * yk1 / 2.0;
144     zinter = zold + stepsize * zk1 / 2.0;
145
146     xk2 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
147     yk2 = -k1*yinter - k2*xinter*yinter + k6*zinter;
148     zk2 = 2.0*k3*xinter - zinter;
149
150     xinter = xold + stepsize * xk2 / 2.0;
151     yinter = yold + stepsize * yk2 / 2.0;
152     zinter = zold + stepsize * zk2 / 2.0;
153
154     xk3 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
155     yk3 = -k1*yinter - k2*xinter*yinter + k6*zinter;
156     zk3 = 2.0*k3*xinter - zinter;
157
158     xinter = xold + stepsize * xk3;
159     yinter = yold + stepsize * yk3;
160     zinter = zold + stepsize * zk3;
161
162     xk4 = k1*yinter - k2*xinter*yinter + k3*xinter - k4*xinter*xinter;
163     yk4 = -k1*yinter - k2*xinter*yinter + k6*zinter;
164     zk4 = 2.0*k3*xinter - zinter;
165
166     xold += stepsize * (xk1 + 2.0*xk2 + 2.0*xk3 + xk4) / 6.0;
167     yold += stepsize * (yk1 + 2.0*yk2 + 2.0*yk3 + yk4) / 6.0;
168     znew = zold + stepsize * (zk1 + 2.0*zk2 + 2.0*zk3 + zk4)  / 6.0;
169     #endif
170
171     if (zold < 0.01 && znew >= 0.01)
172     {
173       if (startpos == 0.0)
174         startpos = aktpos;
175
176       endpos = aktpos;
177       ++periodcount;
178     }
179
180     zold = znew;
181     aktpos = aktpos + stepsize;
182   }
183
184   printf("Period length: %1.20Lf \n", (endpos - startpos)/periodcount );
185
186   return EXIT_SUCCESS;
187 }
```