13 Beispiel: Poisson-Gleichung

Etwas komplizierter wird die Simulation, wenn wir mehr als eine Raumdimension haben – während sich im Eindimensionalen die Unbekannten einfach der Reihe nach in eine Liste fädeln lassen, ist das im Höherdimensionalen mit mehr Aufwand verbunden.

Als Beispiel hierfür sehen wir uns die Poisson-Gleichung $-\Delta u = f$ an – neben der Geometriebehandlung werden wir dabei auch etwas über das iterative Lösen großer linearer Gleichungssysteme lernen.

Poisson-Gleichung

Wir betrachten auf $\Omega = (0,1) \times (0,1)$ die Poisson-Gleichung

$$-\Delta u(x,y) := -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} \stackrel{!}{=} f(x,y)$$

($f : \Omega \to \mathbb{R}$ gegeben) mit Dirichlet-Randbedingungen, also vorgegebenen Funktionswerten $u(x,y)$ für alle Randpunkte $(x,y) \in \partial\Omega$ (hier: $x \in \{0,1\}$ oder $y \in \{0,1\}$).

Zur Diskretisierung stellen wir uns die Ebene mit einem Gitter der Maschenweite $h = 1/M$ überzogen vor und berechnen Näherungen $u_{i,j}$ für $u(i \cdot h, j \cdot h)$, indem wir den Differentialoperator $-\Delta$ durch einen Differenzenoperator ersetzen.

Differenzenoperator

Wie bei der Wärmeleitungsgleichung verwenden wir

$$u''|_{i\cdot h} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

und bekommen

$$-\Delta u|_{i\cdot h, j\cdot h} \approx \frac{4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2}.$$

Setzen wir die rechte Seite gleich $f(i \cdot h, j \cdot h)$, haben wir eine lineare Gleichung für $u_{i,j}$. Allerdings sind die Gleichungen miteinander gekoppelt: ein lineares Gleichungssystem für die $(M-1)^2$ Werte $u_{i,j}$ in Ω.

Die Randpunkte (z.B. $u_{0,j}$ oder $u_{i,M}$) speichern wir diesmal mit ab – die Werte gehen ja in die Differenzenformel der randnahen Punkte ein, wir müssen für die Randpunkte aber natürlich keine Gleichungen aufstellen bzw. lösen.

Verwaltung der $u_{i,j}$

Die u_i aus der Wärmeleitungsgleichung konnten wir bequem in einer Liste speichern und das i als Index verwenden (fast: Die Listenindizes beginnen bei 0 und der erste innere Gitterpunkt ist u_1).

Nun haben wir aber Paare (i, j) als Index – und Glück, dass das in Python kein Problem ist: Wenn wir Dictionaries verwenden, dann können unsere Schlüssel auch Tupel (i, j) sein.

Das ist recht ineffizient, aber zum Ausprobieren für kleine M wird es reichen.

Ein anderes Problem ist, dass unser Rand nicht mehr so handlich ist wie im Eindimensionalen (linker Randpunkt, rechter Randpunkt) sondern nun seinerseits aus vielen ($4M$) Punkten besteht. Daher lohnt es sich hier schon, eine eigene Klasse einzuführen, die sich um die Verwaltung der Geometrieinformationen kümmert. (Welche Punkte gibt es? Welche davon sind Randpunkte?)

Die Klasse `Gitter` wird neben der Maschenweite h eine Liste I aller vorkommenden Indexpaare (Innere Punkte und Randpunkte) speichern.

Weiter werden wir (im Konstruktor) aus der Liste I zwei disjunkte Listen $I1$ mit den Indizes aller inneren Punkte und $I2$ mit den Indizes aller Randpunkte erzeugen.

Dabei sagen wir, dass ein Punkt $(i, j) \in I$ ein innerer Punkt ist, wenn alle vier für die Auswertung der Differenzenformel benötigten Nachbarpunkte $(i \pm 1, j), (i, j \pm 1)$ ebenfalls in I enthalten sind; anderenfalls ist es ein Randpunkt.

Das ist für den Benutzer bequem, weil er dann auch kompliziertere Gebiete als das Quadrat rechnen kann; er muss dann nur die Liste I erzeugen und sich nicht weiter um Rand oder Inneres kümmern.

Parameter für den Konstruktor sind also h und eine Liste I mit Indizes.

```
class Gitter(object):
    def __init__(self, h, I):
        self.h = float(h)
        self.I = I
        # Aufteilen in innere Punkte I1 und Randpunkte I2
        self.I1 = []
        self.I2 = []
        for p in I:
            i,j = p
            if (i-1,j) in I and (i+1,j) in I\
               and (i,j-1) in I and (i,j+1) in I:
                self.I1.append(p)
            else:
                self.I2.append(p)
```

Fügen wir der Klasse gleich noch eine Methode hinzu, mit der die Funktionswerte für Gnuplot (`splot 'poisson.txt'`) ausgegeben werden können. Parameter ist dabei (außer dem Dateinamen) ein Dictionary *u*, das für jeden Schlüssel aus *I* einen Wert enthalten muss (wenn man keins angibt, wird jeder Gitterpunkt in Höhe 0 gezeichnet).

```
def zeichnen(self, u=None, fnam='poisson.txt'):
    f = open(fnam, 'w')
    for p in self.I:
        x = p[0]*self.h
        y = p[1]*self.h
        if u==None:
            z = 0.0
        else:
            z = u[p]
        f.write('%g %g %g\n' % (x, y, z))
    f.close()
```

Später werden wir Funktionen messen wollen, als Norm tut es für uns einfach die euklidische Norm des Vektors aus allen $u_{i,j}$, deren Berechnung wir auch noch in die Klasse `Gitter` einbauen.

```
def norm(self, u):
    sum = 0.0
    for p in self.I:
        sum = sum + u[p]**2
    return math.sqrt(sum)
```

Mehr muss unsere Gitterverwaltung gar nicht tun – es ist ja auch der einfachst mögliche Fall eines Gitters; für realistische Simulationen ist die Erzeugung und Organisation des Rechengitters eine wesentlich kompliziertere Aufgabe.

Ach ja: in Wirklichkeit haben wir es ja mit $\Omega = [0, 1]^2$ zu tun, dafür schreiben wir schnell eine spezialisierte Klasse:

```
class Rechteck(Gitter):
    def __init__(self, M):
        self.M = M
        IRechteck = [(i,j) for i in range(0,M+1) for j in range(0,M+1)]
        Gitter.__init__(self, 1./M, IRechteck)
```

Um was zu sehen, speichern wir einfach mal die x-Koordinaten der Gitterpunkte:

```
def RechteckTest(M):
    R = Rechteck(M)
    xkoord = {}
    for p in R.I:
        xkoord[p] = p[0]*R.h
    R.zeichnen(xkoord)
```

Poisson-Gleichung: Randwerte und Startlösung

Jetzt folgt eine Klasse, deren Objekte die Poisson-Gleichung numerisch lösen können. Der Konstruktor bekommt

- ein Gitter (das gespeichert wird)
- eine Funktion f für die rechte Seite (wird auch nur gespeichert)
- eine weitere Funktion u_{Rand}, die in den Randpunkten ausgewertet wird, um die Dirichlet-Randwerte zu setzen.

Es gibt ein Attribut u, das ein Dictionary mit Schlüsseln aus der Indexmenge des Gitters ist, das soll zum Schluss die $u_{i,j}$ enthalten.
Für die Randwerte können wir schon den Konstruktor die richtigen Werte einfüllen lassen.
Für die inneren Punkte werden wir nachher eine Folge von Werten konstruieren, die gegen die Lösung des LGS konvergiert. Zum Studium dieser Konvergenz wird es hilfreich sein, mit Zufallswerten (hier im Bereich $-1 \ldots 1$) anzufangen (im wirklichen Leben würde man z.B. mit 0 anfangen).

```
class Poisson(object):
    def __init__(self, Gitter, f, u_Rand):
        self.Gitter = Gitter
        self.f = f
        self.u = {}
        # Randwerte setzen
        for p in Gitter.I2:
            x = p[0]*Gitter.h
            y = p[1]*Gitter.h
            self.u[p] = u_Rand(x, y)
        # Uebrige Werte zufaellig
        for p in Gitter.I1:
            self.u[p] = 2.*random.random()-1
```

LGS lösen

Nun müssen wir ein lineares Gleichungssystem in den $(M-1)^2$ Werten der inneren Punkte lösen.

Eine Möglichkeit wäre, die Koeffizientenmatrix aufzustellen und das System mittels Gauß-Elimination zu lösen (letzteres müssten wir gar nicht selber programmieren, das kann Numpy schon).

Problem dabei: Der Speicherplatz der Koeffizientenmatrix wächst quadratisch mit der Zahl der Unbekannten, die Zahl der Rechenschritte sogar kubisch (durch Ausnutzen der speziellen Struktur der Matrix kann man etwas sparen, aber es bleibt unhandlich). Für Probleme mit einigen 1000 Unbekannten geht das noch, aber für sehr große Probleme sind **Iterationsverfahren** eine beliebte Alternative.

Wir verwenden ein hier ein Iterationsverfahren – in unserem Fall ist allerdings ausschlaggebend, dass es sich viel einfacher programmieren lässt . . .

Ein Iterationsverfahren für ein LGS $Ax = b$ konstruiert eine Folge $x^{(i)}$, die für $i \longrightarrow \infty$ gegen die Lösung $x = A^{-1}b$ konvergiert. Man führt die ersten i Schritte (das können viele sein) aus und verwendet das $x^{(i)}$ als Näherung für x.

Somit haben wir verschiedene Fehlerquellen in unserem Verfahren:

- Den Abbruchfehler der Iteration $x^{(i)} - x$
- Den Diskretisierungsfehler (durch die Diskretisierung ist $u_{i,j}$ ja nur eine Näherung für $u(i \cdot h, j \cdot h)$)
- Den Rundungsfehler durch die Benutzung von Fließkommazahlen (der in unserem Beispiel aber so gering ist, dass wir ihn vernachlässigen können).

Eine wichtige Größe bei Iterationsverfahren ist das **Residuum**

$$r^{(i)} := b - Ax^{(i)},$$

für das wir also die Iterierte $x^{(i)}$ in das LGS einsetzten.

Ein einfaches Iterationsverfahren ist das **Jacobi-Verfahren**, das

$$x^{(i+1)} = x^{(i)} + D^{-1} r^{(i)}$$

wählt, wobei D eine Matrix mit den Diagonalelementen von A (in unserem Beispiel: $4/h^2$) und allen Nicht-Diagonalelementen 0 ist. In unserem Beispiel wird mithin $r^{(i)}$ einfach mit $h^2/4$ multipliziert und zur bisherigen Näherung addiert.

Für unser LGS konvergiert das schon, das muss aber nicht so sein – bei vielen Problemen muss man mit einem Faktor $\alpha \in (0, 1]$ dämpfen, um Konvergenz herzustellen:

$$x^{(i+1)} = x^{(i)} + \alpha D^{-1} r^{(i)}$$

Das Programm (hier ohne Dämpfung) ist geradlinig und folgt auf der nächsten Seite – `residuum` berechnet zur derzeitigen Iterierten u (im Konstruktor angelegt) das Residuum, `jacobi` führt auf u einen Iterationsschritt aus (beide Methoden gehören natürlich in die Klasse `Poisson`).

```
def residuum(self):
    r = {}
    h = self.Gitter.h
    for p in self.Gitter.I2:
        r[p] = 0.0
    for p in self.Gitter.I1:
        i,j = p
        rtmp = 4.*self.u[i,j]\
               - self.u[i-1,j] - self.u[i+1,j]\
               - self.u[i,j-1] - self.u[i,j+1]
        r[p] = self.f(i*h, j*h) - rtmp/h**2
    return r

def jacobi(self):
    diag = 4./self.Gitter.h**2
    r = self.residuum()
    for p in self.Gitter.I1:
        self.u[p] = self.u[p] + r[p]/diag
```

Um einen Anhaltspunkt für die notwendige Zahl von Iterationen zu bekommen, verwenden wir $f \equiv 0$ und $u_{\text{Rand}} \equiv 0$, denn dann kennen wir die exakte Lösung $u \equiv 0$ und haben mit der Norm von $u^{(i)}$ ein Maß für den noch abzubauenden Fehler.

Tragen wir $\|u^{(i)}\|$ in einem Diagramm mit logarithmischer Ordinate (Gnuplot: `set logscale y`, rückgängig mit `unset logscale y`) auf, sehen wir, dass sich für große i der Fehler in jedem weiteren Schritt um einen konstanten Faktor abgebaut wird: $\|u^{(i+1)}\| \approx \rho \|u^{(i)}\|$.

Man kann zeigen, dass dieser Faktor ρ unabhängig von der rechten Seite, von den Randwerten und von der Startlösung $u^{(0)}$ ist, daher ist die Messung am Spezialfall nicht abwegig.

Die folgende Funktion führt n_{it} Iterationen aus, Ausgabedateien für Gnuplot sind `poisson.txt` (finale Iterierte), `jacobi`i`.txt` (Iterierte $i = 0 \ldots 9$) und `konvergenz.txt` für $\|u^{(i)}\|$.

```
def PoissonTest(M, n_it):
    def f(x,y):
        return 0.0
    def u_rand(x,y):
        return 0.0
    R = Rechteck(M)
    P = Poisson(R, f, u_rand)
    f_konv = open('konvergenz.txt', 'w')
    f_konv.write('%g\n' % P.Gitter.norm(P.u))
    for i in range(0, n_it):
        if i<10:
            P.Gitter.zeichnen(P.u, 'jacobi%d.txt' % i)
        P.jacobi()
        f_konv.write('%g\n' % P.Gitter.norm(P.u))
    f_konv.close()
    P.Gitter.zeichnen(P.u)
```

Selber machen!

Anfangen mit dem Selbermachen kann man, indem man interessantere Probleme rechnen lässt, z.B. mit $f = 1$ oder mit anderen Randwerten, jedenfalls was, wo die Lösung nicht gerade konstant 0 ist.

Dann sollte man (wieder mit $f = u_{\text{Rand}} = 0$) das Programm so erweitern, dass es aus z.B. $\left\|u^{(n_{it})}\right\|$ und $\left\|u^{(n_{it}-10)}\right\|$ die Konvergenzrate ρ schätzt (indem man ausnutzt, dass für hinreichend große i_1, i_2 gilt: $\left\|u^{(i_1)}\right\| / \left\|u^{(i_2)}\right\| \approx \rho^{i_1 - i_2}$).

(Zur Kontrolle: in diesem Beispielproblem kann man ρ auch exakt ausrechnen, es ist $\rho = \cos(\pi/M)$)

Berechnen Sie aus dem Schätzwert für ρ noch die Zahl der Schritte, die man braucht, um den Fehler bei Konvergenzrate ρ um einen Faktor 10 zu reduzieren, d.h. das n aus $\rho^n = 1/10$.

Wie verändert sich dieser Wert, wenn Sie M verdoppeln?

Im Paket `time` finden Sien ein Funktion `time.clock()`, die die Systemzeit in Sekunden angibt – der Startwert ist für uns irrelevant, weil wir nur Differenzen zwischen verschiedenen Aufrufen brauchen: damit können wir messen, wie lange unser Programm rechnet.

Bestimmen Sie damit für einige Werte von M und n_{it}, wie viel Zeit die Vorbereitung (Gitter anlegen, Konstruktor ausführen, ...) und wie viel Zeit die Jacobi-Iterationen brauchen.

*

Ausblick: Das Jacobi-Verfahren ist ein sehr einfaches Iterationsverfahren, es gibt auch schnellere, insbesondere welche, bei denen sich die Konvergenzrate für $h \longrightarrow 0$ nicht so schnell verschlechtert (z.B. das Verfahren der Konjugierten Gradienten) oder im Idealfall gar nicht verschlechtert (Mehrgitterverfahren).