Das SQP Verfahren

Maximilian Simmoteit

20. Februar 2019

Grundannahmen dieser Arbeit

In dieser Arbeit werden allgemeine Bilder als Funktionen $[0,1]^2 \to [0,1]$ aufgefasst. Diskrete Bilder sind dann zweidimensionale Vektoren mit Werten in $[0,1]$.

1 Das Schärfen von Bildern

In meinem Projekt habe ich mich damit beschäftigt den Algorithmus zum Schärfen von Bildern als Programm umzusetzen. Die Unschärfe eines Bildes wird hier als Faltung aufgefasst. Für einen Faltungskern k und ein Bild u ist das unscharfe Bild dann $k * u$. Für ein gegebenes unscharfe Bild f ergibt sich daraus für die Optimierung der Minimierungsterm:

$$\min_{u \in U} \|k * u - f\| + \alpha TV(u) \tag{1}$$

oder für den diskreten Fall

$$min_{u \in U} \|k * u - f\| + \alpha \left\| \|\nabla_h u\|_2 \right\|_1 \tag{2}$$

Zum Finden der optimalen Lösung u, kann man die Routine von Chambole-Pock nutzen. Als Eingabe benötigt diese einen Term der Form $\min_{x \in X} F(x) + G(Ax)$. Allerdings wird der Term für ein gefaltetes Bild dem nicht ganz gerecht, da dort auch im ersten Term noch eine Funktion auf die Variable angewandt wird. Deswegen setzt man $F(x) = 0$ und setzt den gesamten Term in $G(Ax)$ ein. Hier setzt man $Au = k * u$ und dementsprechend A^* als dessen adjungierte Abbildung. Somit erhält man den folgenden Algorithmus:

$$\begin{aligned}
x^{k+1} &= x^k - \tau(A^* y_1^k - div y_2^k) \\
\bar{x}^{k+1} &= 2 \cdot x^{k+1} - x^k \\
y_1^{k+1} &= \frac{1}{1+\sigma}(y_1^k + \sigma \cdot A\bar{x}^{k+1} - \sigma \cdot f) \\
y2^{k+1} &= \frac{\alpha(y_2^{k+1} + \sigma \nabla \bar{x}^{k+1})}{\max\{\alpha, \left\| y_2^{k+1} + \sigma \nabla \bar{x}^{k+1} \right\|_2\}}
\end{aligned}$$

2 Umsetzung als Programm

Und so wurde dieser Algorithmus als Programm umgesetzt.
Zuerst betrachten wir, wie die Funktionen der diskreten Ableitung implementiert wurden.

2.1 Der diskrete Gradient

Um diesen Algorithmus zu implementieren muss man auch einen Algorithmus für den diskreten Gradienten implementieren. Für den diskreten Fall gilt allgemein, dass $(\nabla_h u)_{ijk} = (\partial_k^{h,+} u)_{ij}$ für die vorwärts-Ableitung. Dieser Zusammenhang lässt sich in Julia sehr einfach umsetzen:

```
function disk_grad(u)
        return cat(disk_ab_v(u,1), disk_ab_v(u,2), dims=3)
end
```

für eine Funktion "disk ab v", die hier die diskrete vorwärts-Ableitung liefert. Das Argument "dims=3"besagt, dass die beiden zweidimensionalen Arrays zu einem dreidimensionalen zusammengefügt werden. Die diskrete vorwärts-Ableitung lässt sich mit einer einfachen Übertragung und Iteration über alle Elemente implementieren:

```
function disk_ab_v(u::Array{Float64,2}, k::Int)
        n = size(u,1)
        m = size(u,2)
        h = 1/sqrt(n*m)
        v = zeros((n,m))
        if k == 1
                for i = 1:n
                        for j = 1:m
                                if i < n
                                        v[i, j] = (u[i+1, j] - u[i,j])/h
                                else
                                        v[i, j] = 0
                                end
                        end
                end
        elseif k == 2
                for i = 1:n
                        for j = 1:m
                                if j < m
                                        v[i, j] = (u[i, j+1] - u[i,j])/h
                                else
                                        v[i, j] = 0
                                end
                        end
                end
```

```
        else

        end
        return v
end
```

Performance Verbesserungen

Diese Implementierung ist sehr direkt dem Lehrbuch entnommen und noch nicht für die Performance eines Computers optimiert. Als erstes, kann man "Type Annotationsëinführen. Dies sagt der Programmiersrache, welche Datentypen man erwartet und erleichtert das Optimieren und kompilieren dieses Codes:

```
function disk_ab_v(u::Array{Float64,2}, k::Int)
...
function disk_grad(u::Array{Float64,2})
...
```

Als nächstes merkt man, dass die Schleifenschritte voneinander unabhängig sind (von u wird nur gelesen und v wird nur beschrieben) und man kann dieser Schleife das Makro @simd vorstellen. Dies kann bei vielen Prozessoren zu schnellerer Ausführung führen. Allerdings kann man das nur bei der innersten Schleife anführen, da eine einfache Instruktion auf verschiedene Daten angewandt werden kann. Um das zu verbessern, kann man die zwei Schleifen zu einer Schleife zusammen fassen. Da ich diesen Algorithmus schon mehrfach getestet habe, kann Julia darauf verzichten sogenannte "BoundsChecks"durchzuführen, mittels des @inbounds Makros. Außerdem kann mittels des @fastmath Makros auf EEE754 konforme Arithmetik verzichtet werden und die Performance gesteigert werden:

```
M = n*m
@simd for a = 0:M-1
                j = a % m +1
                i = a (ganzzahlige Division) m +1
                if i < n
                        @fastmath @inbounds v[i, j] = (u[i+1, j] - u[i,j])/h
                else
                        @inbounds v[i, j] = 0
                end
end
```

Da für k nur zwei Fälle möglich sind, kann man die if-else Verzweigung sein lassen und für die beiden vorwärts-Ableitungen eigene Funktionen schreiben. Zuletzt kann noch der Speicher verbessert werden. Bisher wurde bei jedem Aufruf, mittels $v = zeros((n, m))$, neuer Speicher vom Betriebssystem angefordert. Da in der Optimierungsschleife diese Funktion nur jeweils einmal aufgerufen

wird und das Ergebnis danach nicht mehr benötigt, kann man ein Array definieren, das man in jedem Schleifendurchlauf neu beschreibt. Dies macht man folgendermaßen:

```
function perf_disk_grad(u::Array{Float64,2}, res_arr::Array{Float64,3})
        perf_disk_ab_v_1(u,res_arr)
        perf_disk_ab_v_2(u,res_arr)
end
...
function perf_disk_ab_v_1(u::Array{Float64,2},v::Array{Float64,3})
...
        if i < n
                @fastmath @inbounds v[i, j, 1] = (u[i+1, j] - u[i,j])/h
        else
                @inbounds v[i, j, 1] = 0
        end
...
end
...
```

hier ist "res arrëin Array in den Dimensionen der Ausgabe. Das Array wird den Funktionen jeweils übergeben "perf disk ab v 1"beschreibt die Werte $v[i, j, 1]$ und "perf disk ab v 2"beschreibt die Werte $v[i, j, 2]$.

2.2 Die diskrete Divergenz

Bei der diskreten Divergenz werden die Ergebnisse diskreten rückwärts-Ableitung miteinander addiert. Programmiert sieht das so aus:

```
function disk_div(u)
        return disk_ab_r(u[:,:,1],1) + disk_ab_r(u[:,:,2],2)
end
```

Die diskrete rückwärts-Ableitung lässt sich mithilfe von Schleifen auch ganz leicht nach Julia übertragen:

```
function disk_ab_r(u::Array{Float64,2}, k::Int)
        n = size(u,1)
        m = size(u,2)
        h = 1/sqrt(n*m)
        v = zeros((n,m))
        if k == 1
                for i = 1:n
                        for j = 1:m
                                if i == 1
                                        v[i,j] = u[i,j]/h
                                elseif i < n
                                        v[i, j] = (u[i, j] - u[i-1,j])/h
```

```
                                else
                                        v[i, j] = -u[n-1,j]/h
                                end
                        end
                end
        elseif k == 2
                for i = 1:n
                        for j = 1:m
                                if j == 1
                                        v[i,j] = u[i,j]/h
                                elseif j < m
                                        v[i, j] = (u[i, j] - u[i,j-1])/h
                                else
                                        v[i, j] = -u[i,m-1]/h
                                end
                        end
                end
        else

        end
        return v
end
```

Performance Verbesserungen

2.3 Die diskrete Faltung

Performance Verbesserungen

2.4 Die Optimierungsschleife