Das Schärfen von Bildern in Julia

Maximilian Simmoteit

27. 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
```

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
```

2.3 Die diskrete Faltung

Bei der diskreten Faltung wird ein Bild der Dimension $n \times m$ mit einem Vektor der Dimension $(2r+1) \times (2s+1)$ gefaltet. Da Arrays in Julia mit Index 1 anfangen, habe ich das Array zum Falten als Funktion modelliert:

```
function kreis_k(h,p,q,r,s)
        if p^2/r^2 + q^2/s^2 <= 1
                return h
        else
                return 0
        end
end
```

Damit lässt sich die diskrete Faltung wie folgt programmieren:

```
function disk_falt(u::Array{Float64,2}, r::Int, s::Int, k::Function)
        n_d = size(u,1)
        m_d = size(u,2)
        h = 1/((2*r+1)*(2*s+1))
        A = zeros((n_d-2*r, m_d-2*s))
        for i = 1:n_d-2*r
                for j = 1:m_d-2*s
                        a = 0
                        for n = i-r:i+r
                        for m = j-s:j+s
                                a = a + u[n+r, m+s]*k(h,i-n,j-m,r,s)
                        end
                        end
                        A[i,j] = a
                end
        end
        return A
end
```

Zuletzt fehlt noch die adjungierte Faltung. Die akzeptiert ein Array der Dimension $n \times m$ und gibt ein Array der Dimension $(n+2r) \times (m+2s)$ aus. Die adjungierte Faltung lässt sich wie folgt programmieren:

```
function disk_falt_adj(w::Array{Float64,2}, r::Int, s::Int, k::Function)
        n_a = size(w,1)
        m_a = size(w,2)
```

```
        h = 1/((2*r+1)* (2*s+1))
        A = zeros((n_a + 2*r, m_a + 2*s))
        for i = 1:n_a+2*r
        for j = 1:m_a+2*s
                a = 0
                for n = max(1,i-2*r):min(i, n_a)
                for m = max(1,j-2*s):min(j, m_a)
                        a = a + w[n, m]*k(h,n-i+r,m-j+s,r,s)
                end
                end
                A[i,j] = a
        end
        end
        return A
end
```

Dass das wirklich die adjungierte Faltung ist lässt sich mit folgendem Programm überprüfen:

```
function dual_paarung(u::Array{Float64,2}, w::Array{Float64,2})
        n1 = size(u,1)
        n2 = size(u,2)
        a = 0

        for i=1:n1
                for j=1:n2
                        a = a + u[i, j]*w[i, j]
                end
        end

        return a

end

function teste_adj_faltung(n_wert,m_wert,r_wert,s_wert, k)
        f = 0
        for r=1:r_wert
                println("r: ",r)
                for s=1:s_wert
                        for n = (n_wert-10):n_wert
                                for m = (m_wert-10):m_wert
                                u = rand(Float64,(n,m))
                                w = rand(Float64,(n-2*r,m-2*s))
                                res1 = dual_paarung(w,
                                 disk_falt(u,r,s,k))
                                res2 = dual_paarung(
                                 disk_falt_adj(w,r,s,k),u)
```

```
                                        if abs(res1-res2) > 1e-4
                                                println("!Fehler: ",
                                                 res1, ", ", res2)
                                                f = f+1
                                        end
                                        end
                                end
                        end
                end
                return f
end
```

2.4 Die Optimierungsschleife

Zuerst müssen verschiedene Werte initialisiert werden:

```
function bild_schaerfer(bild::Array{Float64,2}, alpha::Float64, r::Int,
                        s::Int,          k::Function, it=10000::Int)
        n_a = size(bild,1)
        m_a = size(bild,2)

        n = n_a + 2*r
        m = m_a + 2*s

        xk = embed_image(bild,r,s)
        y1k = zeros((n_a,m_a))
        y2k = zeros((n,m,2))

        tau = 1/(n*m*sqrt(8)+1)
        sigma = (1/(n*m*sqrt(8)+1))
```

xk steht hier für x^k und $y1k$ für y_1^k sowie $y2k$ für y_2^k. y wird als 0 initialisiert und x als das Eingabebild, bei dem die Ränder mit Nullen aufgefüllt sind. σ und τ sind so gewählt, dass $\sigma \cdot \tau < \frac{1}{h^2 \cdot 8}$. Damit lässt sich nun die Optimierungsschleife umsetzen. Hierbei wird jeweils $xk2, y1k2, y2k2$ für x^{k+1} geschrieben. Der erste Schritt des Algorithmus

$$x^{k+1} = x^k - \tau(A^* y_1^k - div y_2^k)$$

lässt sich somit als

```
        xk2 = xk - tau*(disk_falt_adj(y1k,r,s,k) - disk_div(y2k))
```

schreiben. Der zweite Schritt

$$\bar{x}^{k+1} = 2 \cdot x^{k+1} - x^k$$

ist ganz einfach

```
xk3 = 2*xk2 - xk
```

und die letzten beiden Schritte

$$y_1^{k+1} = \frac{1}{1+\sigma}(y_1^k + \sigma \cdot A\bar{x}^{k+1} - \sigma \cdot f)$$
$$y2^{k+1} = \frac{\alpha(y_2^{k+1} + \sigma\nabla\bar{x}^{k+1})}{\max\{\alpha, \|y_2^{k+1} + \sigma\nabla\bar{x}^{k+1}\|_2\}}$$

lassen sich als

```
y1k2 = (1/(1+sigma))*(y1k + sigma*disk_falt(xk3,r,s,k) - sigma*bild)
y2k2 = alpha*(y2k + sigma*disk_grad(xk3))/max(alpha,
                  m_norm2_3(y2k + sigma*disk_grad(xk3)))
```

programmieren. Zuletzt müssen die neuen Variablen die alten überschreiben und die Schleife ist fertig:

```
xk = xk3
y1k = y1k2
y2k = y2k2)
```

Aufgrund der kleinen Schrittgrößen läuft dieses Programm eine fest vorgegebene Schrittzahl durch.
Von der Konsole kann man dieses Programm wie folgt aufrufen:

```
julia src/run_script_perf.jl --i "exp_bilder/test_bild1_falt.png" --vr 3
--hr 3 --a 5.0 --it 15000
```