Projekt 1 - Transformacja potęgowa - Przetwarzanie sygnałów i obrazów

Filip Piękniewski, nr indeksu 146961

2003/11/18

Streszczenie

Projekt przygotowany na zajęcia z przetwarzania sygnałów i obrazów, składa się z niniejszego dokumentu, oraz programu komputerowego napisanego w jezyku java. Opracowanie dotyczy operacji potęgowych postaci αr^{γ} wykonywanych na elementach obrazu cyfrowego. Dokument ten omawia aspekty przekształcenia a także wykorzystane algorymy i metody, oraz wyniki działania programu. Opisany też został po krótce sposób użycia załączonego programu.

1 Opis

Jedną z najbardziej popularnych transformacji obrazu cyfrowego jest tak zwana transformacja potęgowa. Przetwarzając obraz cyfrowy mamy zazwyczaj do czynienia z dyskretnym zkwantyzowaniem rzeczywistego obrazu (który można traktować jako funkcję $f : X \times Y \to V$, gdzie $X, Y, V \in \mathbb{R}$ są domkniętymi odcinkami). Skwantyzowany obraz jest zatem macierzą (tablicą) $n*m$ o wartościach w pewnym podzbiorze liczb naturalnych (zazwyczaj wraz z zerem). Najbradziej typową sytuacją jest obraz próbkowany z rozdzielczościa 256 (w dziedzinie natężenia). Liczba 256 obierana jest z dwóch powodów, po pierwsze oko nie jest w stanie rozdzielić wiele większej ilości odcieni, po drugie liczba z przedziłu $[0, 255]$ mieści się akurat w 8 bitach (tak zwanym bajcie), co jest wygodne, gdyż większosc komputerów ma rejestry o takim rozmiarze (lub jego wielokrotnościach). Z każdą z liczb od 0 do 255 wiązany jest pewien poziom natężenia. Ponieważ oko postrzega natężenie logarytmicznie, zatem odpowiednie poziomy natężenia na sprzęcie wyświetlającym interpretowane są jako potęgowe skoki natężenia. Mając zadany obraz, chcielibyśmy móc poprawić jego charakterystykę tak, aby wyłuskać kontury w dziedzinie niskich lub wysokich wartości kwantyzacji. Można do tego wykorzystać transformację liniową, bądź kawałkami liniową, jednak obrazy w ten sposób powstałe zazwyczaj nie będą wyglądały naturalnie. Wygodna jest tu transformacja nieliniowa postaci:

$$f(r) = \alpha \cdot r^{\gamma}$$

dla pewnych ustalonych parametów α oraz γ. Załóżmy na chwilę, że $\alpha = 1$. Wtedy transformacja w postaci jak wyżej spełnia zależności $f(0) = 0$ oraz $f(1) = 1$. Ponadto dla $r \in [0, 1]$ $f(r) \in [0, 1]$. Dlatego wygodnie byłoby wykonywać transformatę tego typu na liczbach ze zbioru $[0, 1]$. W tym celu należy przeskalować posiomy kwantyzacji (pamiętamy, że są to liczby 0..255) na odcinek $[0, 1]$ (co będzie wymagało operacji na liczbach zmiennoprzecinkowych). Problem ten jednak okaże się nie byc bolesny, dzięki zastosowaniu Look Up Tables, o czym powiemy za chwilę. Możemy teraz napisać pierwszą (wstępną) wersję algorytmu transformacji potęgowej:

```
for i in [1..n]
  for j in [1..m]
    obraz[i,j]:=255*alpha*(obraz[i,j]/255)^gamma
```

Jak widać ten pierwszy, prosty algorytm wymaga dokonania zmiennoprzecinkowej[1] operacji $n*m$ razy. Gdy obraz jest duży (np. 2500x1500) oznacza to potrzebę wykonania milionów potęgowań. Nie jest to oczywiście mądre postępowanie, szczególnie gdy mamy 256 poziomów szarości... Znacznie poprawić wydajność w tym wypadku może zastosowanie tablic LUT[2], czyli stablicować transformate dla każdego poziomu, a następnie wykonać transformacje obrazu przy pomocy powstałej tablicy. Odpowiedni algorytm wyglądał by tak:

```
for i in [0..255]
  LUT[i]:=255*alpha*(255/i)^gamma

for i in [1..n]
  for j in [1..m]
    obraz[i,j]:=LUT[obraz[i,j]]
```

[1] innymi słowy pracochłonnej
[2] Look Up Tables

Jak widać przetworzona wersja algorytmu wykonuje zmiennoprzecinkowe operacje jedynie tyle razy, ile jest poziomów kwantyzacji obrazu, następnie obliczone wartości wykorzystywane są już mechanicznie do przetworzenia obrazu. Nieco więcej kłopotu może być z obrazami kolorowymi. Punkt (pixel) obrazu kolorowego zazwyczaj reprezentowany jest jako wektor $[r, g, b]$ gdzie liczby r, g, b są liczbami całkowitymi z przedziału 0..255 i reprezentują poziomy natężenia kolorów podstawowych (red, green, blue). W systemach komputerowych taki wektor często ma reprezentacje jako liczba z przedzialu 0..16777215. Oczywiście nie ma sensu interpretować tej liczby jako natężnia. Dlatego odpowiednią transformatę potęgową wykonuje się dla każdej składowej osobno. Tutaj oczywiście także można wykorzystać LUT, tak też dzieje się w załączonym programie.

2 Wyniki działania

Do oceny działania programu wykorzystamy obraz transf.jpg:

Rysunek 1: Obraz oryginalny.

Obraz nie jest zbyt wyrażny, kontrast jest zbyt wysoki, fragmenty ciemniejsze są zbyt ciemne aby można było na nich coś zobaczyć, jasne plamy odcinają się dość ostro od tła. Obraz taki należałoby lekko zmiękczyć, wydobyć różnice w niskich pozimach natężenia. Z pomocą może przyjść tutaj transformacja potęgowa. Zauważmy, że dla $\gamma \in (0, 1)$ funkcja r^γ jest wypukła. W takim wypadku odcinek $[0, 1]$ deformowany jest w ten sposób, że pierwsza jego częśc (bliższa zeru) jest rozciągana, zaś reszta jest ściskana. Pytanie gdzie przebiega granica pomędzy częścią ściskaną a rozciąganą? Odpowiedź jest niezwykle prosta, tam gdzie funkcja r^γ jest lokalnie podobna do funckji liniowej o współczynniku kierunkowym 1, zatem w miejscu r_1 takim, że $f'(r_1) = 1$. Oczywiście

$$f'(r) = \gamma r^{\gamma-1}$$

Czyli:

$$\gamma r^{\gamma-1} = 1 \Leftrightarrow r^{\gamma-1} = \frac{1}{\gamma} \Leftrightarrow r = \left(\frac{1}{\gamma}\right)^{\frac{1}{\gamma-1}}$$

$$\left(\frac{1}{\gamma}\right)^{\frac{1}{\gamma-1}} = \left(1+\frac{1}{\frac{\gamma}{1-\gamma}}\right)^{\frac{1}{\gamma-1}}$$

Funkcja $\left(1+\frac{1}{\frac{\gamma}{1-\gamma}}\right)^{\frac{1}{\gamma-1}}$ jest rosnąca [3] w przedziale $[0,1]$, zatem wraz z maleniem γ rozciągany odcinek się zmniejsza. Zobaczmy jednak:

$$f\left(\left(1+\frac{1}{\frac{\gamma}{1-\gamma}}\right)^{\frac{1}{\gamma-1}}\right) = \left(1+\frac{1}{\frac{\gamma}{1-\gamma}}\right)^{\frac{\gamma}{\gamma-1}} = \left(1+\frac{1}{\frac{\gamma}{1-\gamma}}\right)^{-\left(\frac{\gamma}{1-\gamma}\right)}$$

Oczywiście funkcja $(1+\frac{1}{x})^x$ jest rosnąca w przedziale $(0,+\infty)$, zatem funkcja $(1+\frac{1}{x})^{-x}$ jest malejąca. Zatem chociaż rozciągany odcinek maleje wraz z maleniem γ, to jednak jego obraz przez przekształcenie potęgowe rośnie! Zatem zmiejszenie gamma powoduje większe rozciągnięcie pewnego początkowego odcinka $[0,r_1] \subset [0,1]$, a tym samym większe zwężenie pozostałej części. W języku obrazów oznacza to, że małe γ wydobywa ciemne szczegóły, zmniejszając kontrast na szczegółach jasnych. Żeby nasza analiza była kompletna trzeba jeszcze powiedzieć co dzieje się dla innych wartości γ. Dla $\gamma = 1$ przekształcenie jest identycznościowe. Dla $\gamma > 1$ nietrudno zauważyć, że powtarza się poprzednia sytuacja, tym razem jednak rozciągany jest pewien końcowy odcinek $[r_1,1] \subset [0,1]$, zwężana jest natomiast reszta. Zatem w języku obrazów, duże γ wydobywa jasne szczegóły zmniejszając kontrast na szczegółach ciemnych. Po tej małej dygresji matematycznej, wiadomo już, że do poprawienia jakości naszego obrazka, przyda się przekształcenie potęgowe, w którym $\gamma < 1$. Zobaczmy zatem co dzieje się dla kilku wartości γ: Widać zatem, że zgodnie z

Rysunek 2: $\gamma = 0.6$

Rysunek 3: $\gamma = 0.4$

Rysunek 4: $\gamma = 0.21$

rozważaniami przeprowadzonymi wyżej, zmniejszanie γ powoduje ekstrakcję szczegółow ciemnych (na kolejnych rysunkach widać coraz więcej szczegółów wyłaniających się z ciemności). Ostatni obraz (4) posiada dodatkowo pewien nieporządany efekt. Tło obrazka nie jest do końca czarne (warość 0), lecz ma pewna niewielką wartość natężenia, co dla małego γ owocuje pojawieniem się szarości. Stosowanie mniejszego γ nie miałoby także sensu z jeszcze jednego powodu: rozważany obraz jest w formacie jpg, a to oznacza, że jest jedynie rekonstrukcją obrazu powstałą z bardziej znaczących współczynników DCT[4]. Przy skrajnych wartościach γ wyostrzenie szczegółow

[3] Co na pierwszy rzuk oka wcale nie jest oczywiste...

[4] Discrete Cosine Transform - dyskretna transformata cosinusowa

w niewielkim przedziale natężeń jest tak duże, że mogą pojawiać się ślady megabloków kompresji, normalnie niezauważalnych. Dla rozważanego obrazka najodpowiedniejszą wartością wydaje się γ w okolicach 0.4.

3 Instrukcja obsługi programu

Program został napisany w języku Java, do jego uruchomienia potrzeby jest dowolny komputer wyposażony w maszynę wirtualną javy zgodną z wersją 1.3 lub nowszą. Uruchomienie programu polega na wpisaniu polecenia:

```
$ java PSiO1
```

W przypadku braku skompilowanych plików (pliki `*.class`) można wykonać kompilację samodzielnie korzystając z załączonych źródeł wpisując polecenie:

```
$ javac PSiO1.java
```

4 Źródła programu

Poniżej przedstawiamy najważniejsze (kompletne źródła wraz z dokumentacja javadoc dostępne są w formie elektronicznej) metody klasy PSiOFrame odpowiedzialej za przetwarzanie obrazu:

```
public class PSiOFrame extends JFrame implements ActionListener {

    public int[] fetchPixels(Image image, int width, int height){
       int pixMap[] = new int[width*height];
       PixelGrabber pg = new PixelGrabber(image, 0,0,width,height, pixMap, 0, width);
       try {
     pg.grabPixels();
       }  catch (InterruptedException e){return null;}
       if((pg.status()  & ImageObserver.ABORT)!=0) return null;
       return pixMap;
    }


   public byte transformPixel(int b)
   {
      int x=(int)b;
      if (x<0) x+=255;
      double b1=x;
      b1/=255;

      double w=alphav*Math.pow(b1,gammav);
      w*=255;
      return (byte)Math.round(w);
   }

   public byte[] extractData(int[] pixmap, int numbands) {
       byte data[] = new byte[pixmap.length*numbands];

       for(int i=0;i<pixmap.length;i++){
   int pixel = pixmap[i];
   byte a = (byte)((pixel >> 24) & 0xff);
   byte r = (byte)((pixel >> 16) & 0xff);
   byte g = (byte)((pixel >>  8) & 0xff);
   byte b = (byte)((pixel      ) & 0xff);
           data[i*numbands+0] = LUT[((r<0)?((int)r+255):r)];
   data[i*numbands+1] =  LUT[((g<0)?((int)g+255):g)];
   data[i*numbands+2]=  LUT[((b<0)?((int)b+255):b)];
       }
       return data;
   }

   public void transformImage()
   {
      for(int i=0;i<256;i++) LUT[i]=transformPixel(i);
      int[] pix = fetchPixels(orgimg, orgimg.getWidth(this), orgimg.getHeight(this));
      byte[] data = extractData(pix, 3);
      imagepanel.setImage(createInterleavedRGBImage(orgimg.getWidth(this), orgimg.getHeight(thi
      img.repaint();
   }


   public static BufferedImage createInterleavedRGBImage(int imageWidth,int imageHeight,int ima
                                                          byte data[],boolean hasAlpha)
   {
```

```
        int pixelStride,transparency ;
        if(hasAlpha) {
      pixelStride = 4;
    transparency = Transparency.BITMASK;
        }
        else {
    pixelStride = 3;
    transparency = Transparency.OPAQUE;
        }
        int[] numBits = new int[pixelStride];
        int[] bandoffsets = new int[pixelStride];

        for(int i=0;i<pixelStride;i++){
     numBits[i] = imageDepth;
     bandoffsets[i] =i;
        }

        ComponentColorModel ccm = new ComponentColorModel(ColorSpace.getInstance(ColorSpace.CS_sR
                                                          numBits,
                                                          hasAlpha,
          false,
          transparency,
          DataBuffer.TYPE_BYTE);

        PixelInterleavedSampleModel csm = new PixelInterleavedSampleModel(
          DataBuffer.TYPE_BYTE,
          imageWidth, imageHeight,
          pixelStride,
          imageWidth*pixelStride,
          bandoffsets);

        DataBuffer dataBuf = new DataBufferByte(data, imageWidth*imageHeight*pixelStride);
        WritableRaster wr = Raster.createWritableRaster(csm, dataBuf, new Point(0,0));
        return  new BufferedImage(ccm, wr, false, null);
     }

  }
```