Simulacija premikanja točke po Bezierjevi krivulji

Poročilo o projektni nalogi pri predmetu
Matematično modeliranje

Nik Erzetič

29. september 2020

Kazalo

1 Matematično ozadje

1.1 Bézierjeve krivulje

Bézierjeve krivulje so parametrične krivulje, določene z zaporedjem kontrolnih točk. Ime nosijo po Pierreu Bézierju, ki jih je v drugi polovici dvajsetega stoletja razvil kot orodje za oblikovanje karoserij Renaultjevih avtomobilov.

1.1.1 De Casteljaujev algoritem

Osnovno orodje za delo z Bézierjevimi krivulji je De Casteljaujev algoritem. Le ta vsaki vrednosti t iz intervala $[0, 1]$ (ali $\mathbb{R}$) priredi točko na krivulju. To stori z zaporednim deljenjem stranic in zveznic med v prejšnjem koraku izračunanimi delilnimi točkami, v razmerju določenim s t.

Algoritem 1: De Casteljaujev algoritem

Vhod: $b_0, b_1, \ldots, b_n \in \mathbb{R}^d$, $t \in [0, 1]$ (ali $t \in \mathbb{R}$)
Izhod: točka b_0^n na Bézierjevi krivulji
definiramo $b_j^0(t) = b_j$, $j = 0, 1, \ldots, n$
for $k = 2, 3, \ldots, n$ **do**
  for $i = 0, 1, \ldots, n - k$ **do**
    $b_i^k = (1 - t) \cdot b_i^{k-1} + t \cdot b_{i+1}^{k-1}$

V Matlabu implementacije ne izgleda tako, a o tem bom več napisal v poznejšem razdelku.

1.1.2 Odvod Bézierjeve krivulje

Odvod Bézierjeve krivulje izračunamo po sledeči fomuli:

$$\frac{\mathrm{d}^r b^n}{\mathrm{d}t^r} = n(n-1)\ldots(n-r+1)\sum_{j=0}^{n-r} \Delta^r b_j B_j^{n-r}(t),$$

kjer je $\Delta b_j = b_{j+1} - b_j$ in $\Delta^r b_j = \Delta(\Delta^{r-1} b_j)$.

1.2 Fleksijska ukrivljenost

Fleksijska ukrviljenost κ meri upognjenost krivulje v točki. Definirana je kot drugi odvod krajevnega vektorja pri naravni parametrizaciji in je enaka obratni vrednosti radija pritisnjene krožnice v tej točki. Za poljubno parametrizacijo $r(t)$ jo izračunamo z naslednjo formulo:

$$\kappa(t) = \frac{\mid r'(t) \times r''(t) \mid}{\mid\mid r'(t) \mid\mid^3}.$$

2 Reševanje

Začnimo z De Casteljaujevim algoritmo. V nadaljevanju bom predstavil dva pristopa, a ta algoritem bo ostal enak v obeh primerih.

```
function tocka = deCasteljau(b,t)
  n = size(b,2);
  for i = 2:n
  b = (1-t).*b(:,1:end-1) + t.*b(:,2:end);
  end
  tocka = b(:,end);
end
```

Kot vidimo, ga izvajamo na mestu. Vektor b testnih točk rekurzivno manjšamo, dokler nam ne preostane ena sama točka - to je točka na Bézierjevi krivulji pri vrednosti t.

Prav tako je obema pristopoma skupna sama animacija. V ta namen se uporablja funkcija `plotBezier`. Le ta izračuna točke v vnaprej določenih vrednostih. Z ukazom `scatter` nato izrišemo še posamezno točko in njej pripadajočo ukrivljenost.

```
figure;
for i = 1:m
  % Levi zaslon
  subplot(1,2,1);
  hold off;
  plotBezier(b);
  hold on;
  scatter(S(1,i),S(2,i));
  % Desni zaslon
  subplot(1,2,2);
  hold off;
  plot(s,u,'k');
  hold on;
  scatter(s(i),u(i));
  pause(hitrost);
end
```

Ukazi `figure` in `subfigure` poskrbijo, da se zaslon loči na dva dela, kjer na levi izrisujemo krivuljo, na desni pa ukrviljenost. V vsaki iteraciji sliko pobrišemo z zlorabo parametrov `hold on` in `hold off`.

2.1 Splošna rešitev

2.1.1 Implementacija v Matlabu

V prvem primeru Bézierjevo krivuljo izrišemo tako, da interval $[0, 1]$ razdelimo na m delov. V vsaki izmed teh vrednosti potem izračunamo točko po zgoraj zapisanem De Casteljaujevem algoritmu. Na naslednji način izračunamo odvod Bézierjeve krivulje:

```
function db = bezier_der(b,r)
  db = b;
  for i = 1:r
  db = (length(db)-1) * diff(db,1,2);
  end
end
```

Tu gre ravno za prej opisano formulo, skrajšano spričo orodji razpoložljivih v Matlabu.

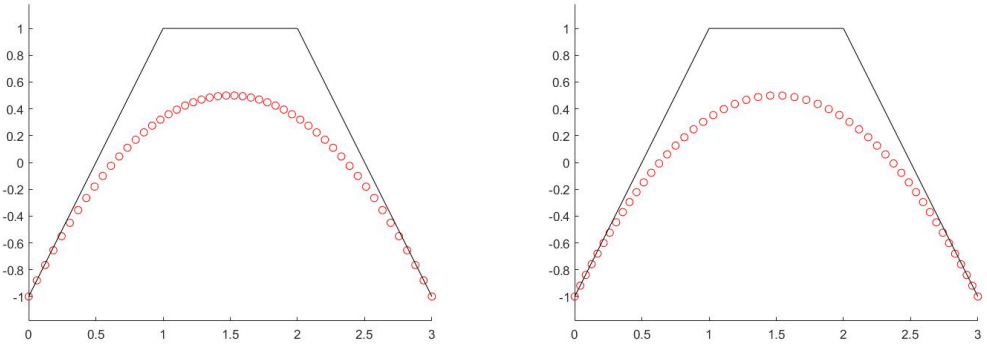
S pomočjo De Casteljaujevega algoritma nato izračunamo še vrednosti odvoda v vrednostih iz intervala $[0, 1]$. Celoten postopek izgleda takole:

```
s = 0:(1/(m-1)):1;
S = [b(:,1) zeros(2,m-2) b(:,end)];
u = zeros(1,m);
db = bezier_der(b,1);
ddb = bezier_der(b,2);
for i = 2:(m-1)
  S(:,i) = deCasteljau(b,s(i));
  dbi = [deCasteljau(db,s(i));0];
  ddbi = [deCasteljau(ddb,s(i));0];
  u(i) = norm(cross(dbi,ddbi),1)/(norm(dbi)^3);
end
```

2.2 Premikanje z enakomerno hitrostjo

Če interval $[0, 1]$ razdelimo na m enakih delov, se delilne točke ne bodo nujno preslikale v točke na Bézierjevi krivulji, ki so med sabo enako oddaljene. Pravzaprav se to, razen v izjemnih primerih, ne bo zgodilo.

To pomeni, da bo na zgornji način implementirana animacija na delih krivulje pospešila - natančneje na prevojih; izkaže se, da je gostota točk na teh mestih največja.



(a) Poljubna (b) Ekvidistančna

Slika 1: Bézierjeva krivulja pri poljubni in ekvidistančni parametrizaciji

Le to nam ne najbolj ustreza. Kako hitro se točka premika je izven našega vpliva. Želeli bi, da hitrost lahko nadziramo. Prvi korak do tega pa je enakomerno gibanje.

2.2.1 Ekvidistančna parametrizacija

Na misel nam takoj pade naravna parametrizacija. Vendar je računanje le te nerodno. Prav tako bi to pomenilo, da moramo razširiti interval $[0, 1]$ na $[0, L]$, kjer je L dolžina Bézierjeve krivulje. To zadnje ne bi bilo preveč zahtevno, a presege namen te projektne naloge.

Zato se bomo zadovoljili z ekvidistančno parametrizacijo. Pri tej so točke na krivulji nanizane enako daleč ena od druge, kar pomeni, da se bo točka na animaciji premikala z enako veliko hitrostjo.

Tudi ta naloga je vse prej kot enostavna, oziroma zanjo nisem našel nobene elegantne rešitve. Zato sem to dosegel tako, da sem na intervavlu $[0, 1]$ minimiziral funkcijo absolutne vrednosti razlike med razdaljo točke od začetka krivulje in željene oddaljenosti:

$$f(t) = \mid dolzina(b_0, t) - d_i \mid, \quad i = 1, 2, \ldots, m-1$$

Pri tem sem dolzino med b_0 in t aproksimiral s polinomom. Na enak nacin sem izracunal dolžino celotne krivulje L in d, ki predstavlja povprečno razdaljo, torej: $d = \frac{L}{m}$. Konstanta na i-tem koraku je potem enaka $d_i = i \cdot d$.

2.2.2 Aproksimacija drugega odvoda

Druga težava, ki se je pojavila pri takšni parametrizaciji krivulje je, da nimamo direktnega načina za računanje odvodov. To pomeni, da smo odvode primorani aproksimirati. To storimo s kvocientom iz definicije odvoda in izbiro dovolj majnega h. Prvi odvod Bézierjeve krivulje b je potem:

$$b'(t) = \frac{b(t+h) - b(t-h)}{2h}$$

Isti trik uporabimo za izračun drugega odvoda, kjer v formulo vstavimo prva odvoda z levo in desno točko:

$$b''(t) = \frac{\frac{b(t+h)-b(t)}{h} - \frac{b(t)-b(t-h)}{h}}{h} = \frac{b(t+h) - 2b(t) + b(t-h)}{h^2}$$

Omenimo še krajišča. V njih nimamo vseh treh potrebnih točk za izračun približka odvodov. Zato privzamemo, da je enaka kot v sosednji točki. Pri zadostnem številu točk napaka ni prevelika.

2.2.3 Implementacija v Matlabu

Najprej poiščemo ekvidistančno parametrizacijo. V ta namen definiramo naslednjo funkcijo:

```
function [s,d] = ekvidistancni_parameter(b,m)
  L = dolzinaBezier(b,10*m);
  d = L/(m-1);
  s = zeros(m,1);
  for i = 2:(m-1)
    f = @(t) abs((i-1)*d - dolzinaBezier(b,m,t));
    s(i) = fminbnd(f,0,1);
  end
  s(m) = 1;
end
```

Funkcija `fminbnd` poišče lokalni minimum na intervalu $[0, 1]$. Dolžino same krivulje aproksimiramo s polinomi. Temu služi funkcija `dolzinaBezier`

```
function L = dolzinaBezier(b,N,t0)
  t = 0:(t0/(N-1)):t0;
  for i = 2:N-1
    p(:,i) = deCasteljau(b,t(i));
  end
  p = diff(p,1,2); %[dx; dy]
```

```
7     L = 0;
8     for i = 1:N-1
9       L = L + norm(p(:,i));
10    end
11  end
```

Ločena funkcija nato iz delilnih točk s izračuna vektor ukrivljenosti. To stori na sledeč način:

```
1   function u = ukrivljenosti(s,b,d)
2     m = length(s);
3     u = zeros(m,1);
4     h = d/100;
5     for i = 2:m-1
6       dbi = [(deCasteljau(b,s(i)+h) - deCasteljau(b,s(
          i)-h))/2/h;0];
7       ddbi = [(deCasteljau(b,s(i)+h) - 2*deCasteljau(b
          ,s(i)) + deCasteljau(b,s(i)-h))/h/h;0];
8       u(i) = norm(cross(dbi,ddbi),1)/(norm(dbi)^3);
9     end
10    u(1) = u(2);
11    u(m) =  u(m-1);
12  end
```

Oba vektorja z zgoraj opisanim postopkom izrišemo na vzporednih platnih.

3 Zaključek

V projektni nalogi sem implementiral potovanje točke po Bézierjevi krivulji. To sem storil na dva načina. Čeprav nam drugi da premikanje z enakomerno hitrostjo, je mnogo počasnejši od prvega. Vsakokratna minimizacija funkcije vzame veliko časa - natančneje vsaj $n \cdot O(n)$.

Z zmanjšanjem natančnosti bi se morda lahko povečalo hitrost. Prav tako se znotraj različnih funkcij nekateri koraki po nepotrebne ponavljajo. Vendar se, vsaj to drugo, ne da dosti izboljšati.

4 Viri

Vsi podatki, navedeni v tem poročilu, izhaja iz mojih zapiskov s predavanj iz Matematičnega modeliranja in Analize 2b na dodiplomskem študiju Matematike, na Fakulteti za matematiko in fiziko Univerze v Ljubljani.