Approsimazione con curve di Bézier di curve bitmap assegnate

Giovanni Tocci 421133/IF Nicola Cugola 421592/IF

17 settembre 2001

Indice

1 Introduzione

Le curve di Bézier sono degli oggetti matematici molto interessanti perché allo stesso tempo potenti e semplici. Infatti con poco sforzo è possibile ottenere curve analiticamente "buone" (continue e derivabili). Ogni curva è completamente determinata da un poligono detto *poligono (di controllo) di Bézier*, i cui vertici vengono anche detti *punti di controllo* o *punti di Bézier*.

Infine i metodi per calcolare e disegnare una curva sono computazionalmente molto semplici ed efficienti.

Nella realizzazione del programma abbiamo mantenuto comuni molte delle idee di base; ognuno di noi le ha poi realizzate usando due linguaggi differenti e aggiungendo particolari personali. Globalmente i due programmi sono molto simili, ma presentano comunque delle differenze. Io qui mi limiterò a descrivere la mia implementazione.

Il programma da me realizzato è scritto in C++, ed utilizza le librerie Qt (versione 2.2.0 o superiori) per la gestione dell'interfaccia grafica. Il tutto funziona su piattaforma Linux.

2 Metodi di tracciamento della curva

2.1 Algoritmo di de Casteljau

L'algoritmo di de Casteljau permette di costruire la curva di Bézier associata al vettore di punti di controllo assegnato procedendo mediante successive combinazioni convesse dei punti stessi:

detti: $b_0, b_1, \ldots, b_n$ i *punti di controllo*
posto: $b_i^0 = b_i$
si ha:

$$b_i^r = (1-t)b_i^{r-1}(t) + tb_{i+1}^{r-1}(t) \quad \begin{cases} r = 1, \ldots, n \\ i = 0, \ldots, n-r \end{cases} , \quad t \in \mathbb{R}$$

Il punto della curva di Bézier corrispondente al parametro t assegnato è $b_0^n(t)$.

In particolare, dato che si vuole ottenere una curva di Bézier compresa tra il primo punto di controllo e l'ultimo, si farà in modo che $0 \leq t \leq 1$.

Scrivendo infine i coefficienti intermedi $b_i^r(t)$ secondo lo *schema di de Casteljau*:

$$\begin{array}{cccccc} b_0 & & & & & \\ b_1 & b_0^1 & & & & \\ b_2 & b_1^1 & b_0^2 & & & \\ b_3 & b_2^1 & b_1^2 & b_0^3 & & \\ \vdots & \vdots & & & \ddots & \\ b_n & b_{n-1}^1 & \ldots & \ldots & b_1^{n-1} & b_0^n \end{array}$$

si nota che durante l'elaborazione è possibile usare un vettore colonna per memorizzare i punti mano a mano che vengono calcolati: ogni elemento, per essere calcolato, ha bisogno solo del valore corrente e del valore dell'elemento successivo.

Riporto qui il codice che calcola un punto della curva di Bézier con il metodo sopra esposto; è costituito dalla funzione `Castel(...)`, che accetta come parametri di input rispettivamente il vettore dei punti di controllo (`contr[]`), il numero di punti di controllo (`punti`) ed il valore del parametro t (`sta_t`); gli altri parametri servono per i *sotto-punti di controllo* e li spiegherò più avanti.

```
inline punto Castel (punto contr[], int punti, double sta_t,
                     double end_t=0, int whatcp=0) {
  double t=sta_t;
  // copia i punti di controllo
  punto *tmp_cont=new punto[punti];
  for (int i=0;i<punti;++i)
    tmp_cont[i]=contr[i];

  if (end_t==0){ // Restitisce un un punto della curva
    for (int c=punti-1;c>0;--c){
      for (int i=0;i<c;++i){
        tmp_cont[i].x=(1-t)*(tmp_cont[i].x)+ t* (tmp_cont[i+1].x);
        tmp_cont[i].y=(1-t)*(tmp_cont[i].y)+ t* (tmp_cont[i+1].y);
      }
    }
  }
  else{ //Restituisce un punto di controllo della curva "ridotta"
    for (int c=punti-1;c>0;--c){
      for (int r=0;r<c;++r){
```

```
        if (whatcp>r) // see the blossom notation
          t=end_t;
        else
          t=sta_t;
        tmp_cont[r].x=((1-t)*(tmp_cont[r].x)+
                       (t)*(tmp_cont[r+1].x));
        tmp_cont[r].y=((1-t)*(tmp_cont[r].y)+
                       (t)*(tmp_cont[r+1].y));
      }
    }
  }
  punto tmp=tmp_cont[0];
  delete tmp_cont;
  return tmp;
}
```

Il tipo di output (`punto`) è una semplice struttura che serve per memorizzare punti di $\mathbb{R}^2$ in coordinate cartesiane:

```
struct punto {
  double x;
  double y;

  punto(double _x=0,double _y=0){
    x=_x;
    y=_y;
  }
};
```

La complessità di questo algoritmo è O(n^2) per ogni punto della curva, dato che effettuo due cicli innestati sul vettore dei punti di controllo che viene passato in ingresso.

2.2 Forma polinomiale di Bernstein

Un altro modo per tracciare le curve di Bézier si ottiene usando i polinomi di Bernstein. Questi sono polinomi del tipo:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

Infatti un punto della curva di Bézier corrispondente ad un certo valore del parametro t si può calcolare considerando una proprietà di questo tipo di polinomi:

$$b^n(t) = b_0^n(t) = \sum_{j=0}^{n} b_j B_j^n(t).$$

Computazionalmente questo metodo, essendo O(n), è più rapido rispetto a quello di de Casteljau, ma richiede alcuni calcoli "pesanti" come ad esempio il calcolo del coefficiente binomiale. Io ho aggirato questo problema tabulando proprio quest'ultimo: quando il programma viene lanciato, costruisce una tabella di coefficienti binomiali, così il calcolo non viene ripetuto ogni volta che si vuole disegnare la curva.

La parte di codice relativa a questo metodo si divide in due parti: la funzione `BernPoly(...)` che restituisce il valore del polinomio di Bernstein di grado `n` di ordine `i` calcolato in `t` e la funzione `Bern(...)` che come la funzione `Castel(...)` riportata sopra restituisce il `punto` della curva corrispondente al vettore di controllo `contr[]` con un numero di punti pari a `punti`, con un valore del parametro `t`. Le due funzioni si basano sulle formule sopra riportate.

```
inline double BernPoly(int i, int n,double t){
  return binom[n][i]*(pow_i(t,i)) * pow_i((1-t),(n-i));
}

inline punto Bern(punto contr[],int punti,double t){
  punto tmp;
  tmp.x=0;
  tmp.y=0;
    for(int j=0;j<punti;++j){
      double coef=BernPoly(j,punti-1,t);
      tmp.x+=contr[j].x*coef;
      tmp.y+=contr[j].y*coef;
    }
  return tmp;
}
```

3 Tracciamento sullo schermo

Il tracciamento della curva sullo schermo presenta un problema: la curva è parametrica e in teoria verrebbe completamente disegnata facendo variare, come ho già detto, il parametro t tra 0 e 1; però una variazione costante del parametro non provoca un corrispondente tracciamento "continuo" della curva (che viene disegnata "per punti" sullo schermo). Si potrebbe risolvere il problema considerando una variazione molto piccola del parametro (per esempio $\Delta t = 10^{-6}$), ma in pratica questo provoca un aumento inutile del tempo di calcolo perché vengono generati molti punti che corrispondono allo stesso pixel sullo schermo e comunque questo non dà sufficiente garanzia che la curva non presenterà discontinuità.

Per risolvere questo problema, ho usato un piccolo trucco: prima di tracciare un punto, controllo se questo dista sullo schermo più di 1 pixel dal precedente (questo è il motivo delle discontinuità); se così non è ricalcolo un altro punto considerando una variazione del parametro più piccola (per la precisione la metà della precedente); se invece i due punti coincidono, proseguo ma nello stesso tempo raddoppio la variazione del parametro, così da evitare inutili calcoli di punti già tracciati.

La "distanza" che uso nella routine non è calcolata in maniera precisa: questa infatti e abbastanza pesante computazionalmente. Io uso una approssimazione (considerando x_i, y_i le coordinate dei due punti espresse in pixel sullo schermo):

$$dist = |x_1 - x_2| + |y_1 - y_2|$$

che nel mio caso è molto valida in quanto se vale 0 i punti sono coincidenti, se vale 1 i punti sono "attaccati", e nei restanti casi sono troppo "lontani".

```
  points=1;
  punto pu=trova_punto(cp,ncp,sta_t);
  double delta_t=.001; // esagerato...
  int old_x=int(pu.x*xscale+.5);// first point x
  int old_y=int(pu.y*yscale+.5);// first point y
  p->setPoint(0,old_x,old_y);
```

```
t_vec[0]=sta_t;
double t=sta_t+delta_t;

while (t<=end_t){
  bool draw=false;
  int p_x,p_y;
  int mean=0;
  while (!draw){
    pu=trova_punto(cp,ncp,t);
    p_x=int (pu.x*xscale+.5);
    p_y=int (pu.y*yscale+.5);
    mean=( abs(old_x-p_x)+abs(old_y-p_y));
    switch(mean){
    case 0:          // punti coincidenti
      delta_t*=2;
      draw=true;
      break;
    case 1:
      draw=true;
      break;
    case 2:
      delta_t/=2; //diminusce il delta
      // Finds the point that closes the "hole" ...
      if ( (old_x-p_x)==-2){
        p->setPoint(points,p_x-1,p_y);
        t_vec[points]=t-(delta_t);
      }
      else if ( (old_x-p_x)==2){
        p->setPoint(points,p_x+1,p_y);
        t_vec[points]=t-(delta_t);
      }
      else if ( (old_y-p_y)==-2){
        p->setPoint(points,p_x,p_y-1);
        t_vec[points]=t-(delta_t);
      }
      else if ( (old_y-p_y)==2){
        p->setPoint(points,p_x,p_y+1);
        t_vec[points]=t-(delta_t);
      }
      else {
        pu=trova_punto(cp,ncp,t-delta_t);
        p->setPoint(points,int(pu.x*xscale+.5),
                    int(pu.y*yscale+.5));
        t_vec[points]=t-(delta_t);
      }
      ++points;
      draw=true;
      break;

    default:
      delta_t/=2;
      t-=delta_t;
      break;
    } //switch
```

```
    } //while(!draw)

    // I found a point to draw: draw it !
    if (mean){
      p->setPoint(points,p_x,p_y);
      t_vec[points]=t;
      points++;
      if ( (max_points)&&(max_points-points<10)){
        max_points+=500;
        p->resize(max_points);
        t_vec.resize(max_points);
        cerr<<"Resizing vector!"<<endl;
      }
      old_x=p_x;
      old_y=p_y;
    }
    t+=delta_t;
  }//while(t<=endt)
```

4 Proprietà varie

Oltre all'algoritmo di approssimazione di una curva assegnata con una curva di Bézier (che spiegherò più avanti), ho pensato di realizzare delle funzioni che risolvono alcuni interessanti problemi.

4.1 Suddivisione di una curva

Una curva di Bézier di solito è definita con $0 \leq t \leq 1$. È possibile tracciare solo una parte di curva se cambio l'intervallo di definizione del parametro: in generale posso considerare $l \leq t \leq m$, con $0 \leq l \leq m \leq 1$.

Inoltre, utilizzando la tecnica del *Blossom (fioritura di punti)* in un caso particolare, è possibile risalire ai punti di controllo del frammento di curva (punti che ho chiamato *sotto-punti di controllo*).

La tecnica del *Blossom* consiste nell'eseguire l'algoritmo di de Casteljau variando ad ogni iterazione il valore del parametro t: la curva viene così a dipendere da un numero maggiore di variabili (in pratica diventa una superficie di $\mathbb{R}^k$ con k pari al numero di parametri diversi usati.

Se voglio calcolare un punto di controllo c_j della curva ridotta con con $0 \leq l \leq t \leq m \leq 1$, se i punti di controllo sono n, basta che eseguo un Blossom considerando l come valore del parametro nelle prime $n - j$ iterazioni e m nelle ultime i. In simboli:

$$c_j = b[l^{<n-i>}, m^{<i>}].$$

A questo servono gli ulteriori parametri della funzione `Castel(...)`: l corrisponde a `sta_t`, m a `end_t` e l'ultimo parametro indica j (cioè quale punto di controllo si vuole calcolare).

```
void paper::findSubCP(punto* cp,int ncp,double sta_t, double end_t,
                      punto* out){
  for (int i=0;i<ncp;++i)
    out[i]=Castel(cp,ncp,sta_t,end_t,i);
}
```

4.2 Intersezione di una curva con una retta

Una proprietà importante delle curve di Bézier è che il *Minimax*, ossia quel rettangolo che circoscrive il poligono di controllo, contiene sempre la curva di Bézier relativa a quel poligono. Dunque calcolarlo è molto semplice (basta un ciclo sui punti di controllo alla ricerca di quelli "estremi").

Questa proprietà si può sfruttare per costruire un algoritmo che cerca l'intersezione della curva assegnata con un segmento che può essere introdotto dall'utente:

- ⋄ si esegue un semplice test se il segmento è contenuto o interseca il Minimax;
- ⋄ se il test fallisce sicuramente il segmento non interseca la curva;
- ⋄ altrimenti si divide la curva in due parti rispetto al parametro t e si richiama ricorsivamente l'algoritmo sulle due metà;
- ⋄ l'algoritmo termina quando il test è vero e il Minimax è molto piccolo (tipicamente un rettangolo di pochi pixel di lato); questo rettangolo individua con buona precisione il punto di intersezione del segmento con la retta.

Questo algoritmo trova tutti i punti di intersezione del segmento con la curva, in quanto nella divisione riesegue il test su entrambe le metà della curva, scartandone una solo se non ci sono intersezioni.

La funzione che effettua il test tra il segmento e il minimax è riportata qui sotto: i parametri di input sono le coordinate dell'angolo inferiore destro e superiore sinistro del minimax. I vertici del segmento sono contenuti nella variabile `p_line[]`.

Il funzionamento è molto semplice: si controlla dapprima se uno dei vertici del segmento è interno al Minimax, altrimenti si cerca una eventuale intersezione del segmento con uno dei lati del rettangolo.

```
bool paper::lineAndMinimax(double min_x, double min_y, double max_x,
                           double max_y){
  double x1=p_line[0].x;
  double x2=p_line[1].x;
  double y1=p_line[0].y;
  double y2=p_line[1].y;
  // Primo vertice dentro
  if ( (x1>min_x) && (x1<max_x) && (y1>min_y) && (y1<max_y) )
     return true;
  // Secondo vertice dentro
  if ( (x2>min_x) && (x2<max_x) && (y2>min_y) && (y2<max_y) )
     return true;
  // I vertici sono fuori...
  // La retta la vedo con una rappresentazione parametrica :
  //         x=(1-t)*x1+t*x2
  //         y=(1-t)*y1+t*y2
  // Valore del parametro che da' l'intersezione
  // con x=min_x (per x)
  double t=(min_x - x1) / (x2-x1);
  for (int i=0;i<2;++i){
    // Calcola la y dell'intersezione
    double inter_y=t*(y2-y1)+y1;
    // Il SEGMENTO interseca la retta verticale
```

```
    // x=min_x (cioe' 0<t<1) e rimane
    // all'interno del lato del rettangolo?
    if ((t>=0) && (t<=1) && (inter_y <=max_y) && (inter_y >=min_y))
      return true;
    // Ripeti tutto per x=max_x
    t=(max_x - x1) / (x2-x1);
  }

  // Valore del parametro che da' l'intersezione con y=min_y (per y)
  t=(min_y - y1) / (y2-y1);
  for (int i=0;i<2;++i){
    // Calcola la x dell'intersezione
    double inter_x=t*(x2-x1)+x1;
    // Il SEGMENTO interseca la retta verticale
    // x=min_x (cioe' 0<t<1) e rimane
    // all'interno del lato del rettangolo?
    if ((t>=0) && (t<=1) && (inter_x <=max_x) && (inter_x >=min_x))
      return true;
    // Ripeti tutto per y=max_y
    t=(max_y - y1) / (y2-y1);
  }
  return false;
}
```

La funzione che realizza l'algoritmo completo è riportata qui. Si nota che per la suddivisione della curva si usa la funzione `findSubCP(...)` definita prima per trovare i sotto-punti di controllo. La funzione richiama se stessa ricorsivamente ed usa il vettore `int_p` dichiarato all'esterno per memorizzare i punti di intersezione trovati, e l'indice `int_p_i` che memorizza sempre la prima posizione disponibile per l'inserimento:

```
void paper::xintersect(double st_t, double en_t, punto* int_p,
                       int& int_p_i){
  double min_x,max_x;
  double max_y,min_y;
  punto *subcp=new punto[ncp];
  findSubCP(cp,ncp,st_t,en_t,subcp);
  findMinimax(subcp,ncp,min_x,min_y,max_x,max_y);
  delete subcp;
  if(lineAndMinimax(min_x,min_y,max_x,max_y)){
    // If Minimax is little enough ...( 4x4 )
    if ( ((int(  (max_x-min_x)*xscale)) < 4) &&
         ((int( (-min_y+max_y)*yscale)) < 4) ) {
        // ... then add this point ...
      displayMinimax(min_x,min_y,max_x,max_y);
      //... but if there is another point near this?
      for (int i=0;i<int_p_i;++i)

        if((abs(((max_x+min_x)/2 - int_p[i].x)*xscale)+
             abs(((max_y+min_y)/2 - int_p[i].y)*yscale) )<6)
          return; // Don't add it !

      int_p[int_p_i].x=(max_x+min_x)/2;
      int_p[int_p_i++].y=(min_y+max_y)/2;
    }
```

```
    else{ // reduce minimax
      double eps=1e-6;
      //first half
      xintersect(st_t,(st_t+en_t+eps)/2, int_p,int_p_i);
      //second half
      xintersect((st_t+en_t-eps)/2,en_t, int_p,int_p_i);
    }
  }
  return;// does not intersects
}
```

4.3 Riduzione del grado

Nel programma è presente una funzione per ridurre il numero di punti di controllo di una curva. In generale la curva ottenuta dopo la riduzione non sarà identica all'originale, dato che la riduzione procede per estrapolazioni: la riduzione anche di un solo punto fa perdere alcune "informazioni" sulla curva, ma se è effettuata con un certo criterio, la curva "ridotta" non differirà molto dall'originale.

La tecnica di base della riduzione consiste nel considerare una curva come proveniente da una processo contrario di *elevazione del grado* (parto da $n-1$ punti di controllo e costruisco la stessa curva che ha n punti di controllo), processo che al contrario non muta la forma della curva. Nel processo di elevazione, detti $\tilde{b}_i$ i vecchi punti di controllo e b_i i nuovi si ha:

$$b_i = \frac{i}{n}\tilde{b}_{i-1} + \frac{n-i}{n}\tilde{b}_i \qquad i = 0, 1, \ldots, n$$

e questo dà origine a due formule per la riduzione:

$$\tilde{b}_i = \frac{nb_i - i\tilde{b}_{i-1}}{n-i} \qquad i = 0, 1, \ldots, n-1$$

e

$$\tilde{b}_{i-1} = \frac{nb_i - (n-i)\tilde{b}_i}{i} \qquad i = n, n-1, \ldots, 1$$

La prima delle due formule è più precisa nella prima parte del poligono di controllo, mentre la seconda e più precisa nell'ultima. Per questo nel mio programma io le uso entrambe, applicando la prima nella prima metà dei punti di controllo e la seconda nella seconda metà; se poi i punti sono dispari, il punto centrale viene calcolato come la media dei due punti che si ottengono applicando prima la prima formula e poi la seconda. Riporto qui il codice di questa funzione che in input accetta il vettore di punti di controllo `old` il numero di punti `n` ed un (eventuale) vettore (`ret`) dove memorizzare i punti che risultano dalla riduzione.

```
inline punto* degreeReduction(punto *old, int n, punto *ret=NULL){
  if (ret==NULL)
    ret=new punto[n-1];
  ret[0]=old[0];              //primo
  ret[n-2]=old[n-1];          //ultimo
  for (int i=1;i<n/2;++i){
    // parte iniziale
    ret[i].x=(n*old[i].x-i*ret[i-1].x)/(n-i);
    ret[i].y=(n*old[i].y-i*ret[i-1].y)/(n-i);
    // parte finale
```

```
    ret[n-i-2].x=(n*old[n-i-1].x-(i)*ret[n-i-1].x)/(n-i);
    ret[n-i-2].y=(n*old[n-i-1].y-(i)*ret[n-i-1].y)/(n-i);
  }
  // n dispari; fai la media dei punti centrali
  if (dispari(n-1)){
    int i=n/2-1;
  ret[i].x=( ((n*old[i].x-i*ret[i-1].x)/(n-i)) +
             ((n*old[n-i-1].x-(i)*ret[n-i-1].x)/(n-i)) )/2;
  ret[i].y=( ((n*old[i].y-i*ret[i-1].y)/(n-i)) +
             ((n*old[n-i-1].y-(i)*ret[n-i-1].y)/(n-i)) )/2;
  }
  return ret;
}
```

Se però si prova ad effettuare più di una riduzione applicando più volte questo metodo, si nota che ben presto la curva comincia ad allontanarsi molto da quella originale.

Per questo ho effettuato una piccola aggiunta che funziona bene quando si hanno molti punti di controllo: divido l'insieme dei punti in sotto-gruppi da 4 punti ed applico la riduzione a questi quattro, come se costituissero gli unici punti di una piccola curva; alla fine riunisco tutti i sottoinsiemi da 3 punti ottenuti. Così facendo uso l'algoritmo di riduzione in modo da perdere meno precisione possibile: passando da 4 punti a 3, il primo e l'ultimo punto rimangono fissi, mentre il secondo è ottenuto come media delle due formule di riduzione. La Fig. 1 mostra come l'algoritmo riesca a ridurre in un colpo circa un quarto dei punti di controllo, mantenendo la curva molto vicina all'originale.

Riporto qui il codice:

```
void paper::degRe(){
  int punti_gruppo=4;
  if (ncp<punti_gruppo+1) return; // pochi punti!
  punto *tmp=new punto[(ncp/punti_gruppo)*(punti_gruppo-1)];
  for (int i=0;i<ncp/punti_gruppo;++i)
    degreeReduction(&cp[i*(punti_gruppo)], punti_gruppo,
                    &tmp[(i*(punti_gruppo-1))]);

  int i=0;
  for (;i<(ncp/punti_gruppo)*(punti_gruppo-1);++i)
    cp[i]=tmp[i];
  // qui i e' gia' stato incrementato...
  for (int j=0;j<ncp%(punti_gruppo);++j)
     // gli ultmi [1 2 3..n-1] punti vengono ricopiati
    cp[i++]=cp[ncp-(ncp%punti_gruppo)+j];
  delete tmp;
  ncp=i;
  repaint(TRUE);
  ++riduzioni;
}
```

5 Il programma

Il programma si presenta con una finestra divisa in tre parti, come mostra la Fig.2:

- a sinistra si trova il "foglio" in cui è possibile impostare i punti di controllo della curva ed è possibile vedere la curva disegnata

- a destra c'è una colonna con pulsanti, caselle di controllo e aree di testo che servono per interagire con il programma
- in basso c'è una barra di stato dove di solito compare il numero di punti disegnati e il tempo impiegato per calcolarli.

Per inserire i punti di controllo della curva, basta cliccare con il mouse nel "foglio". Appare così un punto blu al di sopra del quale viene collocato il numero del punto stesso.

È anche possibile spostare i punti di controllo dopo averli inseriti: per fare questo basta spostare il puntatore sul punto da spostare; in prossimità del punto il cursore cambia forma ed è quindi possibile cliccare, trascinare il punto e rilasciare il mouse nella nuova posizione desiderata.

Infine per eliminare un punto di controllo basta posizionarci il mouse sopra e cliccare col in tasto destro.

È possibile scegliere se disegnare la curva usando il metodo di de Casteljau o con la forma polinomiale di Bernstein, semplicemente selezionandolo dalla lista in alto.

La casella "Auto Refresh", quando selezionata, fa sì che la curva venga automaticamente ridisegnata quando subisce delle variazioni, altrimenti quando non attivata e possibile fare la stessa operazione cliccando sul tasto "Refresh".

Gli altri pulsanti svolgono le seguenti funzioni:

Intersect trova le intersezioni di un segmento con la curva (dopo che entrambi sono stati impostati dall'utente);

Deg. Reduction applica l'algoritmo di riduzione del numero di punti di controllo;

Guess applica (quando il programma viene lanciato con una immagine) l'algoritmo di approssimazione della curva;

Reset riporta il programma nello stato iniziale (cancella i punti di controllo e l'eventuale segmento)

Show subcp visualizza (in viola) i sotto-punti di controllo che corrispondono alla curva calcolata quando il parametro varia tra il valore indicato nella casella "Start t" e il valore indicato in "End t"

Add line permette di introdurre con un click del mouse gli estremi del segmento che poi verrà usato per eseguire l'algoritmo di intersezione; nella barra di stato compariranno in ordine le richieste per inserire il primo estremo e il secondo.

Infine c'è una casella ("Zoom") dove è possibile impostare un fattore di scala per l'immagine. I calcoli sulle curve infatti vengono eseguiti in virgola mobile, con precisione `double`; le coordinate (intere) dei pixel sullo schermo vengono sempre scalate per il fattore di scala specificato in questa casella, mentre i punti calcolati prima di essere visualizzati vengono moltiplicati per lo stesso fattore di scala e arrotondati all'intero più vicino (un semplice troncamento farebbe perdere precisione).

6 L'algoritmo di approssimazione

6.1 Campionamento della curva

La curva che si vuole approssimare con una curva di Bézier, può essere fornita al programma sotto forma di immagine in vari formati (BMP, PNG). Ci sono però alcune semplificazioni che ho introdotto per non complicare l'algoritmo che trasforma la curva fornita in punti elaborabili dall'algoritmo di approssimazione:

1. La curva deve essere un'immagine nera o grigio-scura a 256 tonalità di grigio su fondo bianco;

2. La curva deve cominciare a sinistra e finire a destra: deve cioè avere almeno una colonna a sinistra in cui c'è un solo pixel nero (punto di inizio) e una colonna a destra con la stessa proprietà (punto di fine);

3. La curva non può incrociare se stessa;

4. Lo spessore della curva deve essere abbastanza piccolo (max 2-3 pixel).

La curva quindi deve essere abbastanza "buona".

L'algoritmo di campionamento è molto semplice. Partendo da una certa posizione iniziale cerca di "camminare" sulla curva, guardando sempre i pixel adiacenti e scegliendo il "più nero".

Ovviamente ci sono dei controlli per fare in modo che l'algoritmo vada "sempre avanti" e non percorra più volte lo stesso tratto di curva: quando si controllano i pixel adiacenti, vengono esclusi quelli che non sono lungo la direzione di movimento della precedente scelta; quando ciò non è possibile, si cambia direzione, escludendo sempre quella contraria alla direzione attuale (se questa fosse l'unica scelta possibile, vuol dire che si è arrivati alla fine della curva).

Riporto qui il codice:

```
const int max_points=300; // max numero di punti di campionamento
static int passo=25;      // in pixel
const int soglia_MAX=100;

punto *camp; // vettore di punti di campionamento

int campiona(QImage *q){
  int soglia=soglia_MAX;
  punto destro;
  camp=new punto[max_points];

  // Cerca il pixel nero piu' a sx
  for(int c=0;c<q->width();++c){
    bool row=false;
    for (int r=0;r<q->height();++r)
      if ( qGray(q->pixel(c,r))<soglia){
        soglia=qGray(q->pixel(c,r));
        camp[0].x=c;
        camp[0].y=r;
        row=true;
      }
    if (row) break; // rompe il for sulle colonne
  }
  // Cerca il pixel nero piu' a dx
  for(int c=q->width()-1;c>=0;--c){
    bool row=false;
    for (int r=q->height()-1;r>=0;--r)
      if ( qGray(q->pixel(c,r))<soglia){
        soglia=qGray(q->pixel(c,r));
        destro.x=c;
```

```
      destro.y=r;
      row=true;
    }
  if (row) break; // rompe il for sulle colonne
}
/* Adesso cammino sulla curva: guardo gli 8 pixel intorno
   al mio e scelgo il piu' nero. Questo mi dira' la direzione
   da seguire (ascisse crescenti o decrescenti.) Quando ad
   un certo punto le ascisse cercheranno di decrescere, avro'
   finito di percorrere una catena. */
int x_cur=int(camp[0].x);
int y_cur=int(camp[0].y);
int x_old1=0, y_old1=0;
int tmp_gray,i;
int direzione=0;
int cambi=0; // cambi di direzione
//  3 0
//   X        <-- direzioni
//  2 1
//
bool a_destra, in_alto;
for (i=passo;i<max_points*passo;++i){
  a_destra = (direzione==0 || direzione==1);
  in_alto  = (direzione==0 || direzione==3);
  soglia=soglia_MAX;
  if ( a_destra)
    for (int j=0;j<3;++j){  // colonna dx
      if ((tmp_gray=qGray(q->pixel(x_cur+1,y_cur-1+j)))
                                    <soglia){
        soglia=tmp_gray;
        camp[i/passo].x=x_cur+1;
        camp[i/passo].y=y_cur-1+j;
      }
    }  //for j
  if (!a_destra)
  for (int j=0;j<3;++j){  // colonna sx
    if ((tmp_gray=qGray(q->pixel(x_cur-1,y_cur-1+j)))
                                  <soglia){
      soglia=tmp_gray;
      camp[i/passo].x=x_cur-1;
      camp[i/passo].y=y_cur-1+j;
    }
  }  //for j

  if ((tmp_gray=qGray(q->pixel(x_cur,y_cur-1)))
                              <soglia)  // in alto
    if ( (y_cur-1!=y_old1) && (in_alto) ){
      soglia=tmp_gray;
      camp[i/passo].x=x_cur;
      camp[i/passo].y=y_cur-1;
    }

  if ((tmp_gray=qGray(q->pixel(x_cur,y_cur+1)))
                             <soglia)  // in basso
```

```
      if ( (y_cur+1!=y_old1) && (!in_alto) ){
        soglia=tmp_gray;
        camp[i/passo].x=x_cur;
        camp[i/passo].y=y_cur+1;
      }

    if (soglia==soglia_MAX){
      if(++cambi==1)
        if (direzione==1)
          direzione=0;
        else
          direzione=(direzione+1)%4;
      if (cambi==2)
        if (direzione==0)
          direzione=2;
        else
          direzione=(direzione+2)%4; //no -1 altrimenti...!
      if (cambi>2){
        return i/passo+1;
      }
      --i; // ripeti il ciclo
      continue;
    }
    cambi=0;
    x_old1=x_cur;
    y_old1=y_cur;
    x_cur=int(camp[i/passo].x);
    y_cur=int(camp[i/passo].y);
    if ((x_cur==destro.x)&&(y_cur==destro.y)){
      return i/passo+1;  // raggiunta la fine della curva
    }
  }
  return i/passo+1;
}
```

6.2 Approssimazione della curva campionata

L'idea alla base dell'algoritmo che approssima la curva campionata parte da una proprietà delle curve di Bézier: quando i punti di controllo tendono a ∞ la curva coincide con i punti di controllo stessi. Questo in pratica vuol dire che la curva segue molto fedelmente i suoi punti di controllo se questi sono "abastanza numerosi". Dunque i punti che provengono dal campionamento sono una buona prima approssimazione della curva data in ingresso.

A questo punto ho elaborato un algoritmo che sposta i punti in modo da modificare la relativa curva di Bézier in modo che si avvicini sempre di più ai punti che provengono dal campionamento. L'algoritmo è *Greedy*, cioè cerca sempre il punto di controllo che effettua lo spostamento maggiore e lo sposta.

Dopo aver scelto il punto da spostare, ho pensato che la direzione migliore di spostamento, quella che modifica di più la curva, è quella perpendicolare alla curva stessa, perpendicolare calcolata in un punto della curva prossimo al punto di controllo che deve essere spostato.

Per calcolare la perpendicolare in un punto di una curva di Bézier, è sufficiente trovare la derivata in un punto della curva(che mi dà la direzione della tangente): sfruttando la forma polinomiale di Bernstein e posto $\Delta b_j = b_{j+1} - b_j$ si ha:

$$\frac{d}{dt} b^n(t) = n \sum_{j=0}^{n-1} \Delta b_j B_j^{n-1}(t)$$

Questa formula è facilmente realizzabile:

```
inline punto Bern_der(punto contr[],int punti,double t){
  punto tmp;
  tmp.x=0;
  tmp.y=0;
    for(int j=0;j<=punti-1;++j){
      double coef=BernPoly(j,punti-2,t);
      punto del=delta(contr,1,j);
      tmp.x+=(del.x)*coef;
      tmp.y+=(del.y)*coef;
    }
    tmp.x*=punti-1;
    tmp.y*=punti-1;
  return tmp;
}
```

mentre la funzione `delta(...)` usa una formula più generica per restituire il $\Delta^r b_i = \sum_{j=0}^{r} \binom{r}{j} (-1)^{r-j} b_{i+j}$.

```
inline punto delta (punto contr[],int r, int i){
  punto ret;
  ret.x=0;
  ret.y=0;
  for (int j=0;j<=r;j++){
    ret.x+=binom[r][j]*pow_i(-1.0,r-j)*contr[i+j].x;
    ret.y+=binom[r][j]*pow_i(-1.0,r-j)*contr[i+j].y;
  }
  return ret;
}
```

A questo punto non resta che descrivere come l'algoritmo cerca il punto da spostare:

- ▷ viene eseguito un ciclo su un certo numero di punti della curva di Bézier (saltando ad intervalli di 10 punti) e per ogni punto considerato viene "tracciata" una perpendicolare che è lunga circa 20 pixel sullo schermo;
- ▷ la perpendicolare viene intersecata con i segmenti che costituiscono la spezzata che si ottiene unendo i punti che provengono dal campionamento;
- ▷ se vi è intersezione tra i due segmenti, (espressi in forma parametrica con una combinazione convessa dei vertici) viene preso il parametro (quello che descrive il segmento) della perpendicolare e confrontato con i precedenti;
- ▷ alla fine del ciclo si sceglie il massimo tra i parametri descrittori del segmento e dopo averlo scalato per il solito fattore di zoom, viene usato come quantità di spostamento del punto di controllo che corrispondeva al segmento che intersecava la perpendicolare.

L'algoritmo tiene traccia anche di eventuali riduzioni del numero di punti effettuate: è quindi possibile rieseguire l'algoritmo di approssimazione dopo aver effettuato una o più riduzioni (che portano ad un allontanamento della curva di Bézier da quella originaria). Ovviamente meno punti di controllo si hanno, meno precisa potrà essere l'approssimazione.

```
void paper::guess(){
  for (int volte=0;volte<20;volte++){
    double l1,l2,maxl1=0,maxl,maxm;
    int maxj,j;

    disegna_curva(curva,*t_val,curva->size());
    for (unsigned int i=10;i<points;i+=10){
      // Trovo la perpendicolare alla curva
      punto x0y0=Bern(cp,ncp,t_val->at(i));
      punto tmp2=Bern_der(cp,ncp,t_val->at(i));
      double m=-tmp2.x/tmp2.y; // m_perp=-1/m_tg
      // Trova i due punti di controllo che
      // intersecano la perpendicolare
      // do' una certa lunghezza alla perp (~20 px)
      double l=20.0/(xscale*m);
      if (abs(m)<1){
        // cout <<"m piccolo!"<<endl;
        l=30.0/xscale;
      }
      l1=0;j=1; // il primo e l'ultimo punto NON SI MUOVONO
      while (l1==0&&j<num_camp-2){
        for (int h=0;h<2;++h){
          double p[8];
          //test intersezione
          p[0]=x0y0.x;
          p[1]=x0y0.y;   //A0

          p[2]=x0y0.x+l;
          p[3]= m*(l)+x0y0.y; //A1

          p[4]=campioni[j].x/xscale; //B0
          p[5]=campioni[j].y/yscale;

          p[6]=campioni[j+1].x/xscale;  //B1
          p[7]=campioni[j+1].y/yscale;
          if (interseca(p,l1,l2)) break;
          else {l=-l;l1=0;} //prova dalla parte opposta...
        }
        ++j;
      }
      if (maxl1<l1){
        maxl1=l1;
        maxj=j;
        maxm=m;
        maxl=l;
      }
    }
    int y=maxj,x=maxj;
```

```
    // trovo il corrispondente punto "ridotto"
    for (int h=0;h<riduzioni;++h){
      y=(x/4)*3; //blocco di appartenenza
      if ((x%4==1)||(x%4==2)) ++y;
      if (x%4==3) y+=2;
      x=y;
    }
    QString t("Moving"+QString::number(y)+"("+QString::number(maxj)+
              ") with l1= "+ QString::number(maxl));
    emit(message(t));
    moveControlPoint(y,int((cp[y].x+maxl)*xscale+.5),
                     int((cp[y].y+maxm*maxl)*yscale+.5) );
  }
}
```

Non riporto qui il codice relativo alla gestione dell'interfaccia grafica, che per altro non interessa per lo studio vero e proprio delle curve di Bézier.

Nelle prossime pagine riporto le figure citate nel testo e altre che mostrano alcune caratteristiche del programma: intersezione e divisione di una curva).

Infine riporto alcuni esempi di prova dell'algoritmo di approssimazione.

Alcuni commenti finali: le curve di prova le ho disegnate con il mouse, quindi sono un po' irregolari. Le caratteristiche di continuità delle curve di Bézier alcune volte impediscono una corretta approssimazione, soprattutto nei casi in cui c'è una brusca variazione di direzione nella curva (discontinuità nelle derivate).