Métodos eficientes para aproximação de curvas e superfícies*

Luiz Henrique de Figueiredo†

1 Introdução

O objetivo deste mini-curso é apresentar técnicas de aproximação de curvas e superfícies *genéricas*, paramétricas e implícitas, com ênfase em métodos numéricos práticos (eficientes e implementáveis), úteis tanto em modelagem geométrica quanto em síntese de imagens.

Curvas e superfícies são objetos geométricos contínuos que frequentemente precisam ser discretizados para serem representados num computador. Por exemplo, as imagens usadas para visualização destes objetos são formadas por uma matriz de *pixels*. Entretanto, existe uma relação geométrica entre o espaço ambiente que contém o objeto contínuo e o espaço que contém a imagem: cada *pixel* corresponde a um retângulo no plano $\mathbf{R}^2$. Isto vale não só para curvas mas também para superfícies, pois as imagens de objetos tridimensionais são geradas a partir de projeções do espaço ambiente em um plano. Por motivos práticos, consideramos os *pixels* centrados nos retângulos correspondentes.

O processo de escolha dos *pixels* na imagem correspondentes ao objeto chama-se *rasterização*. Existem vários critérios para avaliar a qualidade de uma rasterização. O critério mais abrangente é escolher os *pixels* que correspondem aos retângulos que intersectam o objeto; nesse caso, dizemos que a imagem é *exata*. Um outro critério igualmente prático é escolher os *pixels* mais próximos do objeto.

Nem sempre a rasterização é exata: frequentemente, a imagem é criada a partir de uma aproximação do objeto em questão. Essas aproximações ainda são contínuas, como o objeto original; o objetivo de usar uma aproximação é simplificar a geometria do objeto para simplificar a rasterização. Além disso, a imagem da aproximação frequentemente é exata. As aproximações mais usadas são lineares por partes, também chamadas *aproximações poligonais*, pois aproximam curvas com segmentos de reta, e aproximam superfícies com polígonos, tipicamente triângulos e quadriláteros. Aproximações poligonais têm várias vantagens: são simples, fornecem modelos geométricos, e é possível gerar imagens exatas de aproximações poligonais, frequentemente com auxílio de *software* e *hardware* gráficos.

Para construir uma aproximação poligonal de um objeto geométrico, é necessário discretizar o espaço "natural" do objeto (o espaço de parâmetros, no caso de objetos paramétricos, ou o espaço ambiente, no caso de objetos implícitos); chamamos este processo de *amostragem*. Os vários métodos de aproximação linear por partes diferem, é claro, na maneira pela qual os pontos amostrais são escolhidos, pois a eficiência de um método depende da amostragem feita. Um método eficiente deve ser capaz de obter aproximações precisas usando amostras pequenas (naturalmente, esses objetivos são conflitantes).

A maioria dos métodos descritos aqui atua no espaço do objeto, isto é, faz a rasterização diretamente a partir de aproximações convenientes. Alguns métodos atuam no espaço da imagem, calculando uma aproximação adequada à rasterização. Este cálculo é frequentemente implícito: um modelo não é explicitamente criado e posteriormente rasterizado; pelo contrário, os *pixels* selecionados para a imagem induzem uma aproximação. Por outro lado, o grande problema de métodos que atuam no espaço da imagem é que é preciso refazer todos os cálculos se a imagem muda de tamanho.

Em vários pontos nesse texto, mencionamos técnicas clássicas de análise numérica, como aritmética intervalar, métodos para solução de equações não lineares, métodos para quadratura de funções, e métodos para solução de equações diferenciais. Para evitar descontinuidades na apresentação, descrevemos essas técnicas nos apêndices.

---

*Notas de mini-curso ministrado no IMPA no verão de 1996.

†Computer Systems Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. `lhf@csg.uwaterloo.ca`

2 Curvas paramétricas

Uma *curva paramétrica* é definida por uma função $\gamma : U = [a, b] \subset \mathbf{R} \to \mathbf{R}^2$. Em outras palavras, a cada valor do parâmetro t no intervalo $U = [a, b]$ está associado um ponto $\gamma(t) = (x(t), y(t))$ do plano.* Por exemplo, o círculo unitário pode ser descrito parametricamente pela função

$$\gamma : t \in [0, 2\pi] \mapsto (\cos t, \sin t) \in \mathbf{R}^2.$$

Um caso particular importante de curva paramétrica é uma função real de uma variável: $f : [a, b] \to \mathbf{R}$. Neste caso, a curva importante é o *gráfico* de f, dada por $\gamma(t) = (t, f(t))$.

Embora, em última análise, o objeto mais importante seja o conjunto $\gamma([a, b]) \subset \mathbf{R}^2$ definido por γ, é mais conveniente matematicamente considerar a curva como a função (também chamada uma *parametrização* da curva) em vez do conjunto (chamado o *traço* da curva). Várias parametrizações podem ter o mesmo traço, mas têm propriedades diferentes. Por exemplo, o círculo unitário também pode ser descrito por

$$t \in [0, 2\pi] \mapsto (\sin t, \cos t) \in \mathbf{R}^2,$$

e também pela *projeção estereográfica*:

$$t \in \mathbf{R} \mapsto \left(\frac{2t}{t^2+1}, \frac{t^2-1}{t^2+1} \right) \in \mathbf{R}^2,$$

a menos do ponto $(0, 1)$.

Na prática, escolhemos o conjunto ou a função, dependendo do que for mais conveniente no momento. É importante apenas ter em mente essa distinção.

2.1 Aproximação de curvas

Para construir uma aproximação poligonal para uma curva γ definida em U, basta escolher *pontos amostrais* $a = t_0 < t_1 < \ldots < t_n = b$ em U e tomar a linha poligonal com vértices p_0, p_1, ..., p_n, onde $p_i = \gamma(t_i)$. Uma vez calculada essa aproximação, uma imagem da curva pode ser criada simplesmente com o seguinte algoritmo:

```
for i = 1..n
    line(p_{i-1}, p_i)
```

onde `line` é uma rotina gráfica que desenha um segmento de reta entre dois pontos dados.

*Consideraremos somente curvas planas, mas vários dos métodos apresentados aqui se aplicam para curvas no espaço.

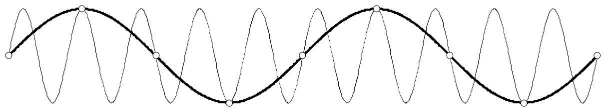

Figura 1: *Aliasing* na amostragem inadequada da senóide $\gamma(t) = (t, \sin 5t)$ em $U = [0, n\pi/2]$.

A eficiência dos métodos de aproximação depende da amostragem feita em U. O número de vezes que a função γ é avaliada é uma boa medida de complexidade para comparação de métodos genéricos, que não dependem de informações adicionais sobre γ. Como γ é avaliada exatamente uma vez em cada ponto amostral, minimizar o número de avaliações de γ é equivalente a minimizar o tamanho da amostra. Portanto, o desafio é escolher amostras pequenas que induzam uma boa aproximação. Como queremos métodos genéricos, essa escolha tem que ser feita baseada somente nos valores de γ.

2.2 Aliasing

É instrutivo ver o problema de aproximação como um problema de reconstrução: queremos reconstruir uma curva contínua a partir de uma amostra discreta. Um problema intrínsico a reconstruções é *aliasing*: o comportamento da curva entre dois pontos amostrais consecutivos não pode ser completamente controlado, e isso pode resultar em uma aproximação de baixa frequência, que ignora oscilações locais de alta frequência na curva original.

Os efeitos de *aliasing* não podem ser completamente evitados [16]. Por exemplo, considere a curva dada por $\gamma(t) = (t, \sin 5t)$ em $U = [0, n\pi/2]$. A amostra t_0, ..., t_n, dada por $t_i = i\pi/2$, não é representativa de γ em U, mas sim representativa de $\alpha(t) = (t, \sin t)$, uma curva de frequência mais baixa que γ (Figura 1). A solução nesse caso, é claro, é aumentar o tamanho da amostra.

O teorema de Shannon-Whittaker garante que aumentar o tamanho da amostra funciona no caso geral, e dá uma estimativa do tamanho de uma amostra adequada: qualquer sinal pode ser reconstruído corretamente a partir de uma amostra uniforme de frequência pelo menos duas vezes maior do que a maior frequência que ocorre no sinal (essa frequência de amostragem miníma é chamada a *frequência de Nyquist* do sinal) [18]. Entretanto, essa estimativa

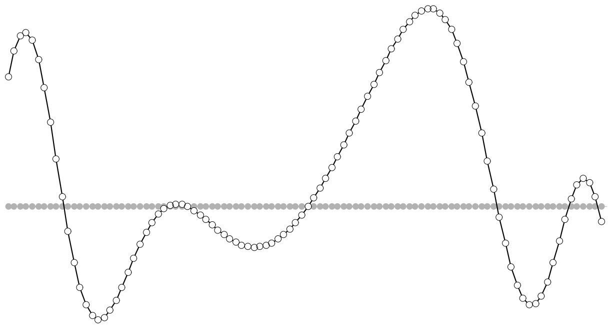

Figura 2: Amostragem uniforme da função $f(t) = \sin(t^2) + \sin(\sqrt{2}\,t)$ em $U = [-3, 3]$ com 100 pontos amostrais.

não é útil na prática pois calcular a maior frequência que ocorre num sinal é muito mais difícil do que calcular uma boa aproximação poligonal (é preciso usar Análise de Fourier para calcular a maior frequência). Além disso, esse resultado só se aplica a sinais de banda limitada, que não ocorrem na prática. Portanto, continuamos com o problema de calcular o tamanho e a distribuição de amostras representativas e adequadas para aproximação poligonal de curvas.

2.3 Amostragem uniforme

O método de amostragem mais popular é *amostragem uniforme*: escolha um *tamanho de malha* $\delta > 0$, divida o intervalo U uniformemente em pedaços de tamanho δ, e tome os extremos dos sub-intervalos como pontos amostrais, isto é, $t_i = a + i\,\delta$, para $i = 0..n$, onde $n = \lceil (b-a)/\delta \rceil$ (Figura 2). Como é pouco provável que $b - a$ seja um múltiplo inteiro de δ, é necessária uma correção: ou fazer t_n explicitamente igual a b, encolhendo o último sub-intervalo, ou usar $\delta' = (b-a)/n$ em vez do δ original, onde $n = \lceil (b-a)/\delta \rceil$. Alternativamente, em vez de escolher o tamanho da malha δ, podemos escolher o número n de divisões de U e fazer $\delta = (b-a)/n$.

O método de amostragem uniforme é muito simples e merecidamente popular pela sua simplicidade, mas a sua eficiência depende da escolha de δ (ou n), pois a função γ é sempre avaliada $n+1$ vezes, qualquer que seja a curva. Além disso, para que a aproximação seja boa, é necessário escolher um δ pequeno, embora tentativa e erro seja o único método geral para decidir quão pequeno δ deve ser. Entretanto, se δ for muito pequeno, então a amostragem vai ser muito ineficiente. Por exemplo, numa imagem construída a partir de uma amostragem uniforme muito fina, vários pontos amostrais vão contribuir somente um *pixel* na imagem. No caso do gráfico de uma função, se a função variar muito entre os pontos amostrais, então a imagem conterá segmentos verticais.

Amostragem uniforme com muitos pontos amostrais também é ineficiente para outras aplicações, como modelagem, porque os objetos construídos a partir de amostras uniformes necessitam de estruturas de dados muito grandes. Em outras palavras, uma amostragem uniforme muito fina implica em grande consumo tanto de tempo (pois a função é avaliada muitas vezes) quanto de memória (pois é preciso armazenar a amostra e possivelmente outras informações para cada ponto amostral). Portanto, embora amostragem uniforme seja tentadoramente simples, ela pode ser muito ineficiente quando alta precisão é importante.

2.4 Amostragem uniforme inversa

Se o objetivo é simplesmente gerar uma imagem do gráfico de uma função, então uma variação do método de amostragem uniforme fornece uma imagem quase exata: basta escolher δ tal que os pontos amostrais em U correspondam a *pixels* adjacentes na imagem. Equivalentemente, bastar tomar $n+1$ igual à resolução horizontal da imagem. Este método é chamado *amostragem uniforme inversa* pois usamos a inversa da transformação de visualização para obter uma amostra em U a partir de uma reta horizontal amostrada unitariamente na imagem. O δ escolhido por amostragem uniforme inversa é o menor para o qual não há coincidência de amostras na imagem.

Neste método, o número de vezes que a função é avaliada só depende da resolução horizontal da imagem. Embora não seja o método mais eficiente, pois ainda faz uma amostragem uniforme, este método é, sem dúvida, o método mais simples para calcular uma imagem (quase) exata. O seu grande mérito é ter uma boa escolha de δ. Na ausência de informações adicionais sobre a função, esta escolha de δ pelo menos garante uma boa imagem.

Infelizmente, amostragem uniforme inversa só se aplica a gráficos, e não a curvas paramétricas mais gerais. De fato, gráficos de funções são curvas especiais: cada reta vertical na imagem intersecta a curva em exatamente um ponto. Por essa razão, é possível aproximar gráficos de funções usando uma amostra horizontal. Para curvas paramétricas gerais, uma reta vertical pode não cortar a curva ou então cortá-la em mais de um ponto. Uma tentativa de extender o método de amostragem uniforme inversa para curvas gerais cai portanto no seguinte problema: para cada abscissa x_0, achar *todas* as soluções da equação $x(t) = x_0$. Esse problema é muito mais difícil do que amostragem (Apêndice C). Além disso, mesmo que fosse simples amostrar uma curva paramétrica desta maneira, restaria o problema de reconstrução da curva a partir dessas "fatias". Este problema também não tem uma solução imediata (Seção 4.1).

Note que a imagem gerada por amostragem uniforme inversa não é exata, mas é quase exata. Embora a amostra nas abscissas seja a mais fina possível, é necessário ligar os pontos correspondentes no gráfico, ainda que por segmentos verticais. Se a função varia muito rapidamente entre duas abscissas consecutivas, então é possível que a imagem exata neste trecho não corresponda ao segmento de reta ligando os dois pontos correspondentes no gráfico, pois a rasterização do segmento de reta muda de vertical exatamente a meio caminho.* Em outras palavras, amostragem uniforme inversa faz uma boa amostragem horizontal, mas não uma perfeita amostragem vertical.

2.5 Amostragem adaptativa

A aproximação poligonal construída a partir de uma amostra uniforme não tem a mesma precisão em todas as partes: a aproximação é melhor nos trechos onde a curva é quase plana, e se aproxima de um segmento de reta, isto é, nos trechos de curvatura baixa. Nestes trechos, uma amostragem fina não é necessária pois podemos substituir vários pequenos segmentos de reta quase colineares por um único segmento de reta, evitando assim ter que avaliar γ em vários pontos intermediários. Portanto, deve ser possível substituir amostragem uniforme por uma amostragem não uniforme, adaptada à geometria da curva, usando a curvatura como critério de amostragem (Figura 3).

Baseados no paradigma de *dividir para conquistar*, adotaremos a seguinte estratégia geral para obter métodos de amostragem adaptativa de um intervalo:

*Exercício: verifique essa afirmativa no clássico algoritmo de Bresenham para rasterização de segmentos de reta.

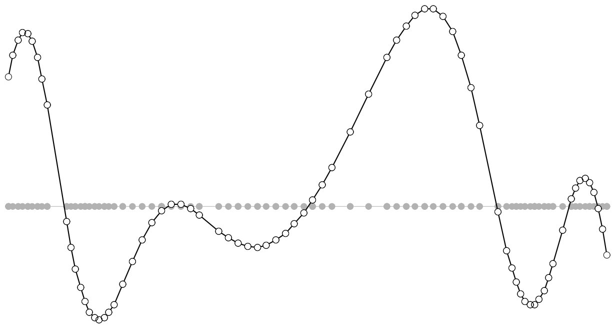

Figura 3: Amostragem adaptativa à curvatura para a função $f(t) = \sin(t^2) + \sin(\sqrt{2}\,t)$ em $U = [-3, 3]$.

0. escolha um critério de refinamento;
1. avalie o critério no intervalo dado;
2. se o intervalo é "bom" segundo o critério, então a amostra é dada pelos dois extremos do intervalo;
3. senão, divida o intervalo em dois e use recursão nas duas partes, voltando ao passo 1.

Essa estratégia é semelhante a usar o algoritmo de de Casteljau para curvas de Bézier, terminando a recursão quando o polígono de controle for quase plano [16]. Discutiremos a seguir vários critérios de refinamento, que, embora heurísticos, são aplicáveis a curvas mais gerais.

Critérios de refinamento

Uma implementação de um critério de refinamento baseado na curvatura é escolher um ponto w no interior do intervalo $[u, v]$ sendo considerado e verificar se os três pontos correspondentes na curva, $p = \gamma(u)$, $q = \gamma(v)$, $r = \gamma(w)$, são aproximadamente colineares. Chamamos isso de *verificação de planaridade local por sondagem*. Existem vários critérios numéricos naturais para verificar colinearidade de três pontos p, q, r:

- a área do triângulo pqr é pequena;
- o ângulo $\angle prq$ é grande, próximo de π;
- r está próximo da corda pq;
- $|p-r|+|r-q|$ é aproximadamente igual a $|p-q|$;
- as retas tangentes a γ em p, q, e r são aproximadamente paralelas.

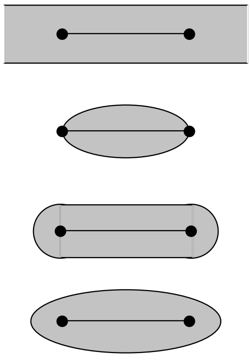

Figura 4: Zonas de tolerância para planaridade local.

Note que é necessário definir uma tolerância para avaliação destes critérios, para dar sentido às palavras "pequena", "grande", "próximo", "aproximadamente". Uma vez escolhida a tolerância, os pontos p, q, r satisfazem um desses critérios se e somente se o ponto r está contido em uma certa região em volta do segmento pq. Esta região é uma faixa infinita, para o critério da área; a união de duas calotas circulares, para o critério do ângulo; um retângulo arredondado, para o critério da corda; e uma elipse, para o critério da distância (Figura 4).

Somente os critérios de área e tangentes são invariantes por escala. Para os outros critérios, se a escala muda, então a tolerância deve ser ajustada de acordo com a nova escala de modo a obter os mesmos resultados. Na prática, esses critérios parecem ser igualmente efetivos na localização de trechos de baixa curvatura. Entretanto, o critério das tangentes só é útil se a derivada de γ está disponível (e.g., se γ é dada por uma fórmula), porque uma amostra adequada para diferenciação numérica é bem mais fina do que uma amostra adequada para geração de imagens, por exemplo. (Por outro lado, se γ é dada por um algoritmo, então é possível usar diferenciação automática, calculando as tangentes junto com os pontos da curva, com pouco custo adicional [?].)

O critério da área é particularmente simples e eficiente para programar, porque, ao contrário dos outros, esse critério não necessita extrair raízes quadradas. Por outro lado, o critério da área é teoricamente o mais fraco porque a sua zona de tolerância é infinita e aumenta de diâmetro quando o segmento pq diminiu de comprimento (i.e, se a corda pq é curta, então a área do triângulo pqr pode ser pequena, mesmo quando o ponto r está longe de pq; como consequência, o critério da área não é adequado para curvas fechadas). Desse ponto de vista, o critério do ângulo é o mais forte, porque tem a menor zona de tolerância. Entretanto, o critério da área apresenta bons resultados na prática.

A escolha do ponto interior

A escolha natural para o ponto interior de sondagem w é o ponto médio do intervalo. Entretanto, sondagem determinística é vulnerável a *aliasing*. Por exemplo, considere $[u, v] = [-\pi, \pi]$ e $\gamma(t) = (t, \sin t)$. Se tomarmos w como o ponto médio, então p, q, e r são colineares, mas γ certamente não é plana nesse intervalo. Para evitar *aliasing* devido a esse tipo de simetria em γ, é mais seguro escolher um ponto de sondagem aleatório, possivelmente perto do ponto médio; chamamos isso de *sondagem aleatória*. Uma escolha que dá bons resultados na prática é $w = u + \lambda(v - u)$, onde λ é um número aleatório no intervalo $[0.45, 0.55]$. O custo desse método depende do gerador de números pseudo-aleatórios usado. Uma maneira de evitar este custo "escondido" é sondar primeiro o ponto médio, só usando sondagem aleatória como confirmação.

Uma outra possibilidade para a escolha do ponto de sondagem, especialmente adequada para o critério de proximidade da corda, é escolher w tal que $r = \gamma(w)$ seja o ponto no trecho entre p e q na curva que está mais longe da corda pq [38]. Pelo teorema do valor médio, $\gamma'(w)$ é paralelo à corda pq. Esse método é exato, não heurístico, mas requer a solução de uma equação não linear a cada passo para achar w. Além disso, requer informações sobre a derivada de γ.

O algoritmo

Por razões de eficiência, é desejável organizar a amostragem de modo que o ponto interior usado para sondagem seja também usado para subdividir o intervalo, se necessário. Assim, a função γ é avaliada exatamente uma vez em cada ponto amostral. Por exemplo, um algoritmo para síntese de imagens é:

```
sample(u, v, p, q):
    w, r ← probe(u, v)
    if flat(p, q, r)
        line(p, q)
    else
        sample(u, w, p, r)
        sample(w, v, r, q)
```

A amostragem do domínio $U = [a, b]$ com esse algoritmo começa com `sample`$(a, b, \gamma(a), \gamma(b))$. A função `probe` escolhe um ponto interior w e calcula $r = \gamma(w)$. A função `flat` implementa o critério de refinamento. A função `line` desenha o segmento de reta entre dois pontos; essa função dever ser substituída em outras aplicações que não síntese de imagens.

Uma implementação em C pode ser escrita diretamente da descrição acima. Por eficiência, agrupamos as coordenadas de um ponto na curva e o valor correspondente do parâmetro em uma `struct`, com macros para acesso legível aos seus membros:

```
typedef struct { double t,x,y; } Point;

#define T(p)    ((p)->t)
#define X(p)    ((p)->x)
#define Y(p)    ((p)->y)
#define urand() (rand()/(double) RAND_MAX)

void probe(Point* p, Point* q, Point* r)
{
 double t=0.45+0.1*urand();
 T(r)=T(p)+t*(T(q)-T(p));
 gamma(r);
}

void sample(Point* p, Point* q)
{
 Point rr, *r=&rr;
 probe(p,q,r);
 if (flat(p,q,r))
  line(p,q);
 else {
  sample(p,r);
  sample(r,q);
 }
}
```

A função `gamma` preenche os campos `x` e `y` correspondentes ao valor do parâmetro no campo `t`. Para o círculo teríamos:

```
void gamma(Point* p)
{
 X(p)=cos(T(p));
 Y(p)=sin(T(p));
}
```

A função `flat` calcula um dos critérios de refinamento, por exemplo, comparando a área do triângulo pqr com uma tolerância `tol`:

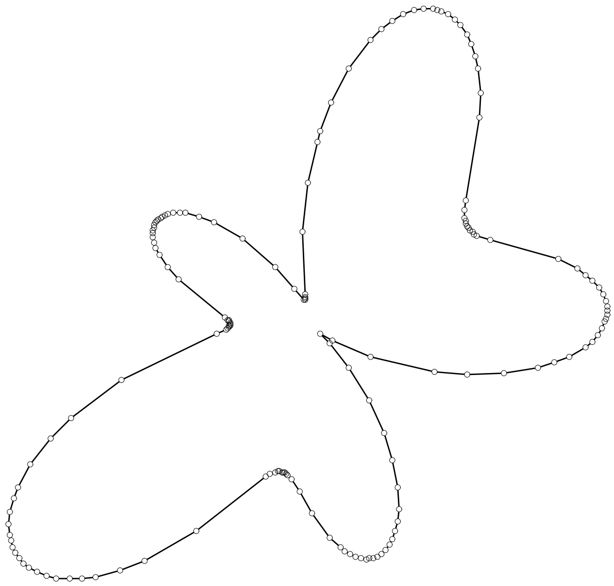

Figura 5: Amostragem adaptativa da curva dada em coordenadas polares por $r = 2 + \sin(2\theta) + \sin(5\theta)$.

```
int flat(Point* p, Point* q, Point* r)
{
 double a=X(p)*Y(q)+X(q)*Y(r)+X(r)*Y(p)
         -X(q)*Y(p)-X(r)*Y(q)-X(p)*Y(r);
 return (a<tol);
}
```

A função que inicia a amostragem no intervalo $[a, b]$ é:

```
void start(double a, double b)
{
 Point pp, *p=&pp;
 Point qq, *q=&qq;
 srand(time(0));
 T(p)=a;   gamma(p);
 T(q)=b;   gamma(q);
 sample(p,q);
}
```

Com definições adequadas de `gamma`, `line`, e `tol`, as rotinas acima formam um pacote completo para desenhar uma curva paramétrica qualquer usando amostragem adaptativa. A Figura 5 mostra uma imagem, criada com essas rotinas, da curva dada em coordenadas polares por $r = 2 + \sin(2\theta) + \sin(5\theta)$.

Complexidade

Se a amostra final tem n pontos, então o algoritmo `sample` avalia γ exatamente $2n-1$ vezes: uma vez para cada ponto amostral, e uma vez em um ponto interior em cada sub-intervalo final. Esse número de avaliações não deve ser comparado apressadamente com as n avaliações necessárias para obter uma amostra uniforme com n pontos amostrais. Uma comparação mais justa é comparar $2n-1$ com o número de avaliações em uma amostra uniforme com tamanho da malha δ igual ao tamanho do menor sub-intervalo na amostragem adaptativa. Essa comparação depende, é claro, da função γ.

Sondagem múltipla

É provavelmente mais seguro avaliar o critério de refinamento em mais de um ponto interior, separadamente ou simultaneamente. Por exemplo, Chandler [5] usa $k-1$ pontos interiores amostrados uniformemente, um critério de refinamento baseado na área do polígono definido por esses pontos e pelos extremos do intervalo, e recursão de ordem k caso seja necessário refinar a amostra. Para $k=2$, esse método é o mesmo descrito antes, tomando o ponto médio como ponto de sondagem. Chandler dá exemplos, parecidos com o dado na página 5, que mostram que recursão binária ($k=2$) pode ter problemas de *aliasing*, que são resolvidos usando recursão ternária ($k=3$). Chandler não fornece critérios para a escolha de k; na prática, é provavelmente suficiente usar $k=2$ ou $k=3$.

Como toda amostragem determinística, sondagem múltipla com pontos interiores amostrados uniformemente é vulnerável a *aliasing*. Uma possível melhoria no método de Chandler é usar sondagem aleatória múltipla. De qualquer modo, o método de recursão binária com sondagem aleatória em um único ponto é simples, fácil de implementar, e parece ser adequado na prática.

2.6 Outras aplicações

Amostragem adaptativa não é útil somente para aproximação de curvas, mas também para rasterização e problemas de integração numérica como: quadratura de funções (cálculo de áreas), retificação de curvas (cálculo de comprimentos), e parametrização por comprimento de arco.

Rasterização direta

Uma propriedade interessante do algoritmo `sample` descrito na Seção 2.5 é que os pontos amostrais são gerados na ordem exata em que eles aparecem na curva, apesar (ou por causa) da recursão. De fato, um algoritmo de amostragem pura, que simplesmente armazena os pontos amostrais, pode ser escrito:

```
sample(u, v, p, q):
    w, r ← probe(u, v)
    if flat(p, q, r)
        p_n ← p
        n ← n + 1
    else
        sample(u, w, p, r)
        sample(w, v, r, q)
```

tomando cuidado para fazer $p_n \leftarrow \gamma(b)$ como último passo.

É fácil adaptar este algoritmo para fazer rasterização direta da curva, sem antes criar uma aproximação poligonal. Primeiro, use $\bar{\gamma}$ em vez de γ, onde $\bar{\gamma}(t)$ é o *pixel* correspondente ao ponto $\gamma(t)$. Depois, verifique se p e q são *pixels* vizinhos, em vez de verificar se a curva é localmente plana. Finalmente, plote o *pixel* p em vez de desenhar o segmento de reta pq. Com essas modificações, temos o seguinte algoritmo, já simplificado para a implementação em C:

```
raster(p, q):
    probe(p, q, r)
    if neighbours(p, q)
        plot(p)
    else
        raster(p, r)
        raster(r, q)
```

Como o ponto r não participa do teste, podemos mover a sondagem para perto da recursão. Embora a versão original seja possivelmente mais segura, esta variante é mais rápida, e depende somente do fator de escala e da resolução da imagem; não é preciso escolher uma tolerância pois não há mais teste de planaridade local.

```
raster(p, q):
    if neighbours(p, q)
        plot(p)
    else
        probe(p, q, r)
        raster(p, r)
        raster(r, q)
```

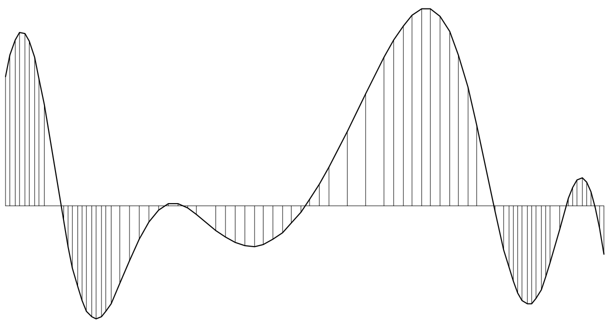

Figura 6: Quadratura adaptativa da função $f(t) = \sin(t^2) + \sin(\sqrt{2}t)$ em $[-3, 3]$.

Integração numérica

O critério de refinamento geralmente usado nos métodos clássicos de integração numérica adaptativa é avaliar e comparar dois métodos numéricos diferentes, ou um único método em diferentes resoluções. Se a discrepância nos resultados é maior que a tolerância desejada, então o intervalo é refinado [8, 19].

O método de amostragem adaptativa descrito na Seção 2.5 implicitamente constrói uma boa aproximação linear por partes da curva. Para essa aproximação, o problema de integração é muito simples e pode ser resolvido exatamente. Assim, obtemos uma classe de métodos simples para integração numérica adaptativa.

Por exemplo, considere quadratura de funções. Como a regra do trapézio é exata para funções lineares por partes, podemos usá-la nos intervalos finais de uma amostragem adaptativa (Figura 6). Temos assim o seguinte algoritmo de quadratura adaptativa:

```
area(p, q):
    probe(p, q, r)
    if flat(p, q, r)
        return area_trap(p, q)
    else
        return area(p, r) + area(r, q)
```

Analogamente, um algoritmo adaptativo para retificação de curvas é:

```
length(p, q):
    probe(p, q, r)
    if flat(p, q, r)
        return |p − q|
    else
        return length(p, r) + length(r, q)
```

Parametrização por comprimento de arco

Parametrizações de uma curva por comprimento de arco, isto é, tal que a curva seja percorrida com velocidade constante unitária, são muito úteis para controle de trajetórias em animação [19]. Teoricamente, é simples calcular uma parametrização α por comprimento de arco a partir de uma parametrização γ qualquer: se a curva tem comprimento total L, então basta fazer

$$\alpha(s) = \int_0^s \|\gamma'(t)\|\, dt.$$

O método de integração adaptativa descrito na seção anterior pode ser usado para calcular $\alpha(s)$ para qualquer s usando a fórmula acima (desde que se saiba avaliar γ'). Entretanto, é muito mais eficiente fazer uma única amostragem, adequada para calcular $\alpha(s)$ eficientemente para qualquer s.

Comece com uma amostragem adaptativa $t_0, \ldots, t_n$ do domínio U e faça

$$\begin{aligned} s_0 &= 0 \\ s_i &= s_{i-1} + |\gamma(t_i) - \gamma(t_{i-1})|, \quad i = 1..n. \end{aligned}$$

O domínio da parametrização α é o intervalo $[0, s_n]$, pois s_n é uma boa aproximação para L. Para calcular o ponto $\alpha(s)$ na curva correspondente a um valor $s \in [0, s_n]$, primeiro encontre o intervalo $[s_i, s_{i+1}]$ contendo s. Isso pode ser feito eficientemente em tempo $O(\log n)$ com busca binária:

```
locate(s, i, j):
    if i = j then return i
    k ← ⌊(i + j)/2⌋
    if s < s_k then
        return locate(s, i, k)
    else
        return locate(s, k, j)
```

Uma vez localizado o trecho contendo s, calcule o ponto t correspondente a s em $[t_i, t_{i+1}]$ por interpolação linear, resolvendo a equação:

$$\frac{s - s_i}{s_{i+1} - s_i} = \frac{t - t_i}{t_{i+1} - t_i},$$

isto é:

$$t = t_i + (t_{i+1} - t_i)\frac{s - s_i}{s_{i+1} - s_i}.$$

Então temos $\alpha(s) = \gamma(t)$.

2.7 Multi-resolução

Considere novamente o método de amostragem adaptativa, usando o critério de distância da corda. Muito provavelmente, a primeira sondagem vai concluir que o trecho não é plano. Entretanto, mantendo a tolerância fixa, podemos mudar a escala até que o critério seja satisfeito. Em outras palavras, qualquer curva é plana em alguma escala. Esta observação é a base de representações em multi-resolução de curvas.

Uma representação em multi-resolução de uma curva é uma representação hierárquica da curva, que pode ser percorrida somente até o nível adequado para a escala corrente [21]. Essas representações são claramente úteis em síntese de imagens, pois reduzem o volume de dados a somente o necessário para gerar uma imagem numa dada escala. Entretanto, representações em multi-resolução também são muito úteis para processamento geométrico adaptativo, como cálculo de interseções. Consideraremos somente algoritmos simples para linhas poligonais, embora recentemente haja grande interesses em métodos de multi-resolução usando *wavelets* [15].

Strip trees

Uma *strip tree* é uma representação hierárquica de uma curva por meio de retângulos contendo trechos da curva [2]. A curva é representada por uma árvore binária. Na raiz, temos um retângulo com dois lados paralelos ao segmento ab, onde a e b são os extremos da curva. Esses lados passam pelos pontos c e d mais distantes de ab em cada lado. No próximo nível, a curva é subdivida em c ou d, dependendo do que estiver mais longe de ab (digamos, c), e representada com retângulos contendo os dois trechos ac e cb. Esse processo de subdivisão é repetido recurssivamente nos dois trechos, até que os retângulos tenham largura menor que uma tolerância.

Esta estrutura de dados está intimamente ligada com o algoritmo de Douglas-Peuker para redução de pontos numa linha poligonal [13, 28]. As poligonais obtidas por digitalização tipicamente têm muito mais pontos do que é necessário tanto para gerar imagens quanto para processamento geométrico. Uma maneira de reduzir o número de pontos em uma poligonal $p_0, \ldots, p_n$ é comecar com o segmento p_0p_n e calcular o ponto p_k mais distante deste segmento. Se a distância de p_k a p_0p_n for menor que a tolerância, então todos os pontos estão perto do segmento p_0p_n, e a linha poligonal se reduz a esse único segmento. Caso contrário, processe recursivamente os trechos p_0p_k e p_kp_n. Ao final do processo, os pontos visitados formam uma linha poligonal que aproxima bem a linha poligonal original, mas que geralmente tem muito menos pontos.

Armazenando os pontos visitados e as distâncias desses pontos ao segmento corrente, e encerrando a recursão somente quando o trecho não puder ser subdivido por não ter pontos interiores, em vez de usar tolerância,* obtemos uma estrutura de dados que representa a linha poligonal completamente, e é útil para processamento em qualquer resolução. Esta estrutura de dados é na verdade uma *strip tree* simplificada.

O algoritmo de Douglas-Peuker tem complexidade $\Theta(n^2)$ no pior caso, mas na prática tem complexidade $\Theta(n \log n)$. É possível obter $\Theta(n \log n)$ no pior caso, com uma implementação mais sofisticada [24].

Arc trees

Uma alternativa para a escolha do ponto de subdivisão é o ponto médio do trecho da curva sendo considerado. Este é o pont que está a meio caminho entre os extremos, onde a distância é medida *sobre* a curva. Portanto, o mais conveniente é que a curva esteja parametrizada por comprimento de arco. Como vimos, isso é relativamente simples de garantir, especialmente para linhas poligonais. Entretanto, para linhas poligonais é mais simples usar o ponto cujo *índice* está a meio caminho, e armazenar o comprimento de arco até esse ponto.

Uma *arc tree* é a árvore binária correspondente a uma subdivisão nos pontos médios [20, 22]. Uma característica importante de representações hierárquicas é a existência de algum conjunto simples que contenha um trecho da curva em cada nível. Numa *strip tree*, esse conjunto é um retângulo. Numa *arc tree*, esse conjunto é uma elipse. Entretanto é mais simples usar círculos, ainda que sejam maiores do que necessários.

3 Superfícies paramétricas

Uma *superfície paramétrica* é definida por uma função $f : U \subset \mathbf{R}^2 \to \mathbf{R}^3$. Em outras palavras, a cada ponto p do domínio U está associado um ponto

*Ou equivalentemente, usando tolerância zero.

$f(p)$ do espaço. Tipicamente, o domínio U é um retângulo $[a,b]\times[c,d]$. Neste caso, escrevemos $f(u,v)$ para o ponto em $\mathbf{R}^3$ correspondente ao ponto (u,v) de U. Por exemplo, a esfera unitária pode ser descrita parametricamente pela função

$$f(u,v) = (\cos u \sin v, \sin u \sin v, \cos v)$$

definida em $U = [0,2\pi]\times[0,2\pi]$. Como para curvas, um caso particular importante de superfície paramétrica é uma função real de duas variáveis: $g : [a,b]\times[c,d] \to \mathbf{R}$. Neste caso, a superfície importante é o *gráfico* de f, dada por $f(x,y) = (x,y,g(x,y))$.

3.1 Amostragem uniforme

O método de amostragem uniforme para curvas descrito na Seção 2.3 funciona também para superfícies; claramente, é necessário fazer a amostragem nas duas variáveis independentes. O modo mais simples de fazer esta amostragem é usar uma malha regular (p_{ij}), obtida como produto cartesiano de malhas $a = u_0 < u_1 < \ldots < u_n = b$ e $c = v_0 < v_1 < \ldots < v_n = d$, em cada eixo coordenado (Figura 7). Note que, se estas malhas têm n pontos cada uma, então a malha final tem n^2 pontos. Portanto, a complexidade deste método é quadrática na resolução da malha.

Uma implementação direta do método de amostragem uniforme é:

for $i = 0..n$
  for $j = 0..n$
    $p_{ij} \leftarrow f(u_i, v_j)$

Para construir uma aproximação de uma superfícies paramétrica não é suficiente somente amostrar o domínio; é preciso estruturar as amostras em polígonos, tipicamente, triângulos ou quadriláteros (Figura 7). Uma estruturação em quadriláteros é:

for $i = 1..n$
  for $j = 1..n$
    $Q_{ij} \leftarrow (p_{i-1,j-1}, p_{i,j-1}, p_{ij}, p_{i-1,j})$

Uma estruturação em triângulos é:

for $i = 1..n$
  for $j = 1..n$
    $T^1_{ij} \leftarrow (p_{i-1,j-1}, p_{i,j-1}, p_{ij})$
    $T^2_{ij} \leftarrow (p_{i-1,j-1}, p_{ij}, p_{i-1,j})$

Estes algoritmos gastam tempo e espaço $\Theta(n^2)$. Para síntese de imagens, não é necessário armazenar todos os polígonos (por exemplo, se usamos um algoritmo de z-*buffer*), mas ainda é necessário conhecer os seus vértices. Veremos agora como evitar armazenar todos os vértices.

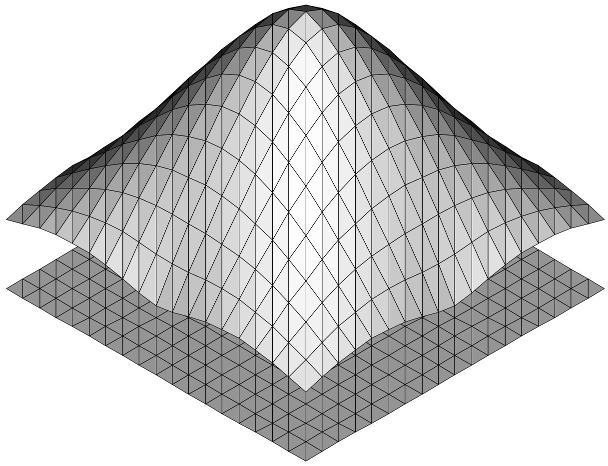

Figura 7: Amostragem uniforme de uma superfície usando uma malha de triângulos.

3.2 Amostragem uniforme eficiente

Consideremos o problema geral. Devemos visitar todas as células de uma malha retangular. Em cada visita, devemos fazer algo que depende de valores $A_{ij} = f(a_{ij})$ calculados nos vértices a_{ij} da célula. Queremos percorrer a completamente malha, mas só queremos calcular os valores dos vértices uma única vez, e não queremos pré-calcular todos os valores. Uma solução é armazenar somente os valores de duas linhas consecutivas:

for $i = 0..n$
  $A_i \leftarrow f(a_{i0})$
for $j = 1..n$
  $B_0 \leftarrow f(a_{0j})$
  for $i = 1..n$
    $B_i \leftarrow f(a_{ij})$
    $\texttt{visit}(i, j, (A_{i-1}, A_i, B_{i-1}, B_i))$
  $A \leftarrow B$

Este algoritmo gasta portanto espaço $\Theta(n)$, o que já é uma grande melhoria em relação ao algoritmo anterior, que gastava espaço $\Theta(n^2)$. Entretanto, é possível gastar ainda menos espaço. De fato, o algoritmo abaixo percorre a malha usando uma *quadtree* e gasta espaço somente $\Theta(\log n)$. Além disso, esse espaço não precisa ser pré-alocado, pois é alocado dinamicamente na pilha de execução.

```
explore(a, b, c, d):
    if iscell(a, b, c, d)
        visit(a, b, c, d)
    else
        M, A, B, C, D ← split(a, b, c, d)
        explore(a, A, M, D)
        explore(A, b, B, c)
        explore(M, B, c, C)
        explore(D, M, C, d)
```

3.3 Amostragem adaptativa

O método de amostragem uniforme inversa não pode ser usado diretamente para superfícies pois a escolha dos *pixels* depende da projeção. Na verdade, a generalização do método de amostragem uniforme para superfícies seria *ray-casting*, mas isso não é simples para superfícies paramétricas, pois temos que achar os zeros simultâneos de duas funções.

Por outro lado, o método de amostragem adaptativa pode ser usado facilmente. Para superfícies, não é necessário escolher um ponto interior a cada retângulo para avaliar o critério de refinamento: basta testar se os quatro pontos extremos estão num mesmo plano. Adicionalmente, pode-se escolher um ponto interior e verificar se este ponto não está muito longe do plano definido pelos pontos extremos. Note que este método decompõe o domínio em uma *quadtree*:

```
explore(r):
    if flat(f(r))
        use r
    else
        r1, r2, r3, r4 ← split(r)
        explore(r1)
        explore(r2)
        explore(r3)
        explore(r4)
```

O principal problema deste algoritmo é a falta de coerência na subdivisão: Se um retângulo é subdivido mas um retângulo vizinho não é, então a superfície terá uma fissura ao longo da aresta comum aos dois retângulos (Figura 8). Em outras palavras, a aproximação poligonal construída com esse algoritmo não é contínua porque a decomposição do domínio feita pela *quadtree* não é uma decomposição celular.

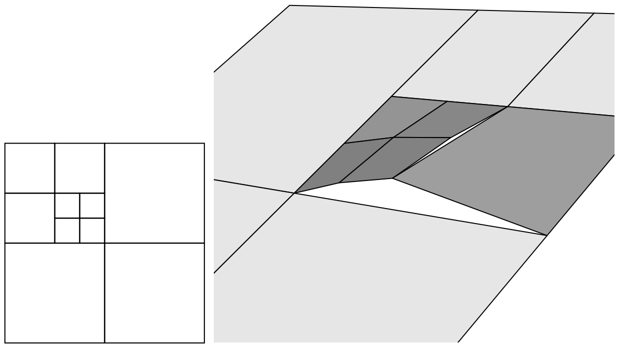

Figura 8: Fissura decorrente de inconsistência topológica em decomposição por *quadtree* [37].

Uma solução para evitar fissuras é completar a decomposição do domínio, tornando-a uma decomposição celular válida. Von Herzen e Barr propuseram usar somente *quadtrees restritas*, nas quais retângulos vizinhos diferem de no máximo um nível, pois essas *quadtrees* são facilmente completadas para obter uma triangulação do domínio [37].

3.4 Amostragem adaptativa usando caminhos

Um outro modo de evitar fissuras e não ter que corrigir a decomposição do domínio é criá-la topologicamente consistente desde o início, focalizando a decomposição nas arestas e não nas células. Um algoritmo que faz isso é [36]:

1. Comece com uma decomposição simplicial do domínio, que pode ser simplesmente a decomposição do retângulo em dois triângulos, ao longo da diagonal.
2. Amostre adaptativamente as curvas na superfície correspondentes às arestas de todas as células da decomposição inicial.
3. Subdivida cada célula recursivamente construindo arestas internas, baseado no número de pontos das arestas externas.

A amostragem adaptativa do passo 2 é uma variação dos métodos descritos na Seção 2.5 que leva em conta que a curva está numa superfície. A subdivisão das células no passo 3 é determinada pela complexidade das arestas externas. Um aresta simples é composta de um único segmento de reta. As arestas

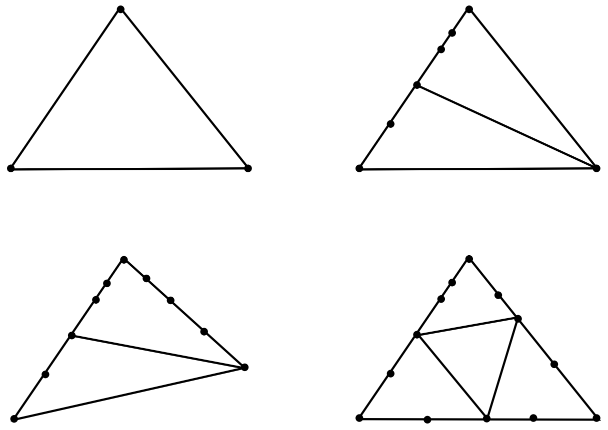

Figura 9: Casos de subdivisão para amostragem adaptativa usando caminhos.

complexas são subdivididas no ponto de maior curvatura; as novas células são criadas ligando os pontos de subdivisão ao vértice oposto ou a outros pontos de subdivisão. Temos quatro casos possíveis, dependendo do número de arestas simples na célula (Figura 9):

- *três arestas simples*: A subdivisão recursiva é encerrada e a célula é a amostra.
- *duas arestas simples*: A célula é dividida em duas ligando a aresta complexa ao vértice oposto.
- *uma aresta simples*: A célula é dividida em duas ligando a mais longa aresta complexa ao vértice oposto.
- *nenhuma aresta simples*: A célula é dividida em quatro ligando pares de arestas complexas adjacentes.

4 Objetos implícitos

Um dos principais objetivos da matemática é resolver equações. Pode-se mesmo dizer que grande parte do desenvolvimento da matemática teve origem na procura de métodos de solução de equações (e também na definição do que é uma solução).

Objetos implícitos são dados por equações em várias variáveis, e ocorrem naturalmente em várias áreas da matemática e em aplicações. Mais precisamente, um objeto V é dado implicitamente quando ele é o conjunto de zeros de uma função $h : \Omega \subset \mathbf{R}^d \to \mathbf{R}^m$, i.e., $V = \{x \in \Omega : h(x) = 0\}$. Consideraremos aqui somente $m = 1$, e $d = 2$ ou $d = 3$, isto é, uma só equação em duas ou três variáveis. O teorema da função implícita diz que o conjunto de soluções de uma tal equação é uma curva no plano ($d = 2$) ou uma superfície no espaço ($d = 3$), se a equação não for singular, no sentido que as derivadas parciais de h não se anulam simultaneamente em nenhum ponto da solução. Por outro lado, o teorema de Sard garante que esta condição vale genericamente, isto é, após uma perturbação da equação, que pode ser escolhida bem pequena e aleatoriamente, com probabilidade 1. Por exemplo, uma perturbação pode ser considerar $h - \varepsilon$ em vez de h.

Uma *curva implícita* é o conjunto de zeros de uma função $h : \Omega \subset \mathbf{R}^2 \to \mathbf{R}$. Em outras palavras, temos uma função real h (também chamada um *campo escalar*) definida num subconjunto Ω do plano e estamos interessados nos pontos aonde h se anula: $\{(x, y) \in \Omega : h(x, y) = 0\}$. Novamente, o objeto mais importante é esse conjunto, embora várias funções possam ter o mesmo conjunto de zeros (por exemplo, qualquer potência de h). Um mesmo conjunto pode ser descrito tanto parametricamente quanto implicitamente. Por exemplo, o círculo unitário pode ser descrito implicitamente pela função $h(x, y) = x^2 + y^2 - 1$. (Este fato é um teorema importante sobre as funções seno e cosseno.) Analogamente, uma superfície implícita é conjunto de zeros de uma função $h : \Omega \subset \mathbf{R}^3 \to \mathbf{R}$.

Como vimos na Seção 3, o cálculo de aproximações de objetos paramétricos se reduz a uma amostragem estruturada do domínio dos parâmetros e avaliação das funções de parametrização na amostra. Por outro lado, o cálculo de aproximações de objetos implícitos requer a solução de vários sistemas de equações não-lineares. É importante ter em mente que não é possível resolver exatamente uma equação $h(x) = 0$ em um computador. Os métodos numéricos para solução de equações só são capazes de obter x tal que $|h(x)| \leq \varepsilon$, ou tal que $|x - \hat{x}| < \delta$, onde $\hat{x}$ é uma solução exata de $h(x) = 0$, e ε e δ são tolerâncias pré-definidas. Estas tolerâncias não podem ser arbitrariamente pequenas por limitações de *hardware*, já que existe um menor número real positivo representável em cada computador.

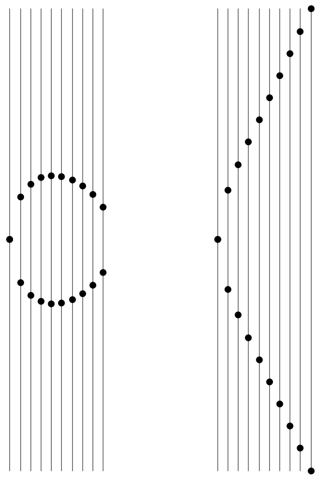

Figura 10: *Ray-casting* vertical.

4.1 Amostragem por *ray-casting*

Amostragem por *ray-casting* é feita reduzindo a equação $h(x) = 0$, que tem d variáveis, a várias equações de uma variável, fixando $d - 1$ das d variáveis. Geometricamente, isto corresponde a intersectar V com uma família de retas (os raios!).

Para curvas planas ($d = 2$), um critério simples para escolher os raios é escolhê-los paralelos a um dos eixos coordenados (Figura 10). Se os raios correspondem a linhas da imagem, então o método passa a ser um método de *scan-line*. De qualquer modo, é necessário tomar cuidado com a estruturação das amostras em cada raio, de modo a obter uma aproximação poligonal correta. Um maneira simples de fazer isso é manter as amostras ordenadas em cada raio e ligar as amostras correspondentes. Entretanto, é necessário tomar cuidado com bifurcações, não só para curvas singulares, pois uma amostragem inadequada pode produzir bifurcações, mesmo para curvas regulares.

Para usar *ray-casting* em modelagem, é claramente necessário ter disponível um bom método numérico para resolver equações de uma variável (Apêndice C). Entretanto, isto não é suficiente para garantir uma boa aproximação, pois é necessário que *todas* as soluções sejam encontradas em cada raio. Além disso, é muito difícil escolher a família de retas e controlar a densidade final de pontos.

Para visualização (e não modelagem) de superfícies ($d = 3$), *ray-casting* é uma técnica clássica. Sem perda de generalidade, podemos supor que estamos criando uma imagem no plano $z = 0$ por projeção ortogonal, isto é, com ponto de visão em $z = -\infty$. Tomando raios verticais em cada *pixel*, basta agora calcular o primeiro ponto aonde o raio encontra a superfície, pois os outros pontos de interseção não aparecem na imagem. Em outras palavras, para cada raio temos que achar somente a menor solução de uma equação, e não mais todas as soluções, o que simplifica bastante o problema (Apêndice C).

Calcular os pontos de interseção dos raios com a superfície não é suficiente para criar uma imagem: é necessário calcular uma cor para cada *pixel*. Este cálculo depende do modelo de iluminação usado, mas certamente precisa do valor do gradiente de h no ponto de interseção, pois esse é o vetor normal à superfície.

4.2 Amostragem inversa

Se o objetivo é simplesmente gerar uma imagem de uma curva implícita, então um método de amostragem inversa fornece uma imagem quase exata: para cada *pixel* p da imagem, avalie a função h no ponto x correspondente ao centro de p no domínio de h. Se $|h(x)| \leq \varepsilon$, então acenda p. O custo deste método de *scan-line* é proporcional à resolução da imagem.

Note que a imagem gerada não é exata, pois a solução pode passar "entre" os *pixels*. Em uma imagem exata, os *pixels* escolhidos são aqueles que estão a menos de 1 *pixel* de V; esta distância é a distância Euclidiana. O método descrito acima usa uma distância induzida pela função h dada por $d_h(x, y) = |h(x) - h(y)|$.* Assim, a imagem não é exata porque é possível que alguns *pixels* sejam ignorados (quando a curva passa pelo *pixel*, mas o valor de h no centro é alto), e que alguns *pixels* sejam escolhidos erroneamente (quando a curva não passa pelo *pixel*, mas o valor de h no centro é baixo).

Uma variante deste método gera um "mapa de contornos" de h: atribui-se cores a faixas de valores e escolhemos para p a cor correspondente ao valor de h em x. Este mapa de contorno é uma representação bi-dimensional alternativa para o gráfico de h, muito útil quando se está escolhendo o nível de h a aproximar, e também para obter informações qualitativas sobre o comportamento global de h.

*Exercício: mostre que d_h é realmente uma métrica.

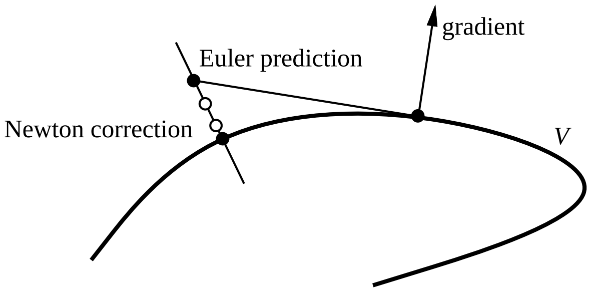

Figura 11: Método de continuação de Euler-Newton.

4.3 Aproximação por continuação

Uma técnica muito eficiente para aproximação de curvas planas dadas implicitamente por $h(x) = 0$ é a integração do campo Hamiltoniano $\mathcal{H}(h) = (-\partial h/\partial y, \partial h/\partial x)$. Como o campo gradiente $\nabla h = (\partial h/\partial x, \partial h/\partial y)$ é ortogonal às curvas de nível de h, e $\mathcal{H}(h)$ é ortogonal a ∇h, temos que $\mathcal{H}(h)$ é tangente às curvas de nível de h. Portanto, as curvas de nível de h são curvas integrais do campo Hamiltoniano, e logo, podemos aproximar V resolvendo um problema de Cauchy (Apêndice E):

$$
\begin{aligned}
\frac{dx}{dt} &= -\frac{\partial h}{\partial y} \qquad x(0) = x_0, \\
\frac{dy}{dt} &= \frac{\partial h}{\partial x} \qquad y(0) = y_0,
\end{aligned}
$$

para um ponto inicial (x_0, y_0) de cada componente conexa de V.

Qualquer método de integração numérica de equações diferenciais pode ser usado para resolver este problema de Cauchy; os métodos de Euler e Runge-Kutta são os mais populares por serem fáceis de implementar. Entretanto, um método de integração que leva em consideração que a solução é de fato uma curva de nível combina o método de Euler com o método de Newton para solução de equações não lineares. Neste método combinado, o método de Euler é usado como preditor e o método de Newton é usado como corretor (Figura 11) [1]:

$$
\begin{array}{ll}
p \leftarrow p + \delta\mathcal{H}(h)(p) & \textit{preditor de Euler} \\
u \leftarrow \nabla h(p) & \textit{direção de correção} \\
\text{while } |h(p)| > \varepsilon & \\
\qquad p \leftarrow p - \dfrac{h(p)}{\langle \nabla h(p), u \rangle} u & \textit{corretor de Newton}
\end{array}
$$

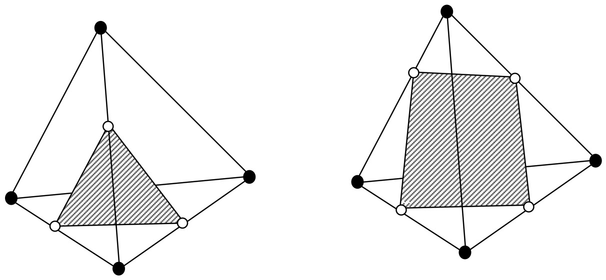

Figura 12: Interseção de um plano e um tetraedro.

Este método é muito robusto e, com cuidado, pode ser extendido para dimensões superiores. Entretanto, ele necessita de pontos iniciais em cada componente conexa, e cuidado especial com órbitas fechadas.

4.4 Aproximação por enumeração

Ao invés de calcular uma aproximação de V, podemos calcular V *exatamente* para uma aproximação de h [12]. As aproximações mais comuns são lineares por partes, baseadas em decomposições celulares do domínio Ω de h. Para estas aproximações, é simples calcular V exatamente desde que as células tenham uma geometria simples (Figura 12). A princípio, a geometria das células pode ser arbitrária, mas na prática, as células mais usadas são triângulos e retângulos, em dimensão 2, ou tetraedros e paralelepípedos, em dimensão 3, porque é mais simples decidir se V intersecta uma célula deste tipo, e também por que é mais fácil calcular a interseção.* Além disso, é conveniente que a decomposição celular tenha uma topologia simples, de modo a permitir uma organização simples da enumeração (Figuras 13 e 14).

Um método geral para aproximar V é então:

1. decompor Ω em células pequenas;
2. enumerar as células que intersectam V;
3. aproximar V dentro de cada célula.

O método de enumeração mais simples usa uma malha uniforme composta de triângulos ou retângulos (Figura 15), e decide se V intersecta uma célula baseado no sinal de h nos vértices da célula: se os sinais não forem todos iguais, então V intersecta a célula. Claramente, a recíproca não é verdadeira: se os sinais forem todos os mesmos, não podemos concluir que a

*Mesmo assim, isso não é trivial (Figura 12).

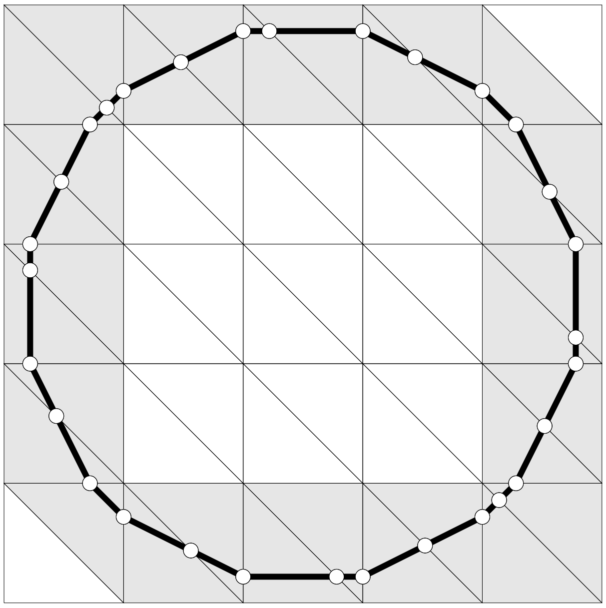

Figura 13: Enumeração de um círculo com decomposição triangular uniforme ($h(x, y) = x^2 + y^2 - 1$, $\Omega = [-1, 1] \times [-1, 1]$).

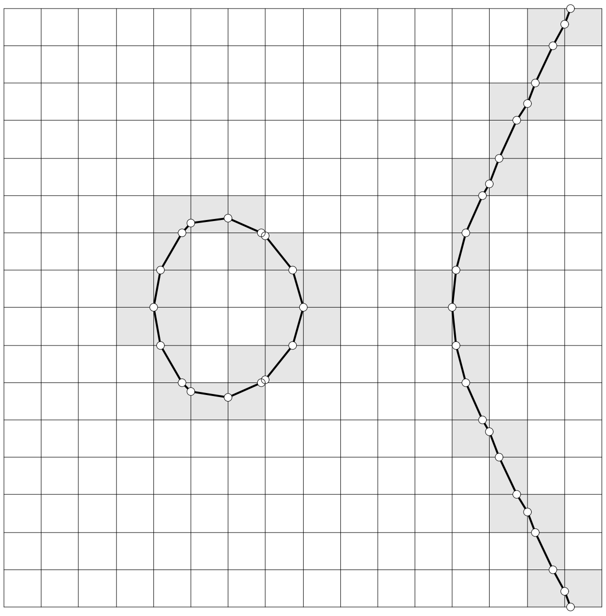

Figura 14: Enumeração de uma cúbica com decomposição retangular uniforme ($h(x, y) = y^2 - x^3 + x$, $\Omega = [-2, 2] \times [-2, 2]$).

célula não intersecta V (Figura 16). Para evitar este tipo de *aliasing*, é necessário escolher o tamanho das células com cuidado. Por outro lado, malhas uniformes com células muito pequenas são muito ineficientes para enumeração completa, que tem que visitar todas as células da decomposição celular. Isto é muito ineficiente porque só algumas células intersectam V (Figura 14), especialmente em co-dimensão alta (por exemplo, curvas em $\mathbf{R}^3$).

Um modo de evitar enumeração completa é visitar somente as células que realmente intersectam V, passando de uma célula para outra usando pivoteamento [1, 12, 17]. Este método híbrido de enumeração com continuação tem, é claro, as desvantagens dos métodos de continuação, mas é um método muito popular na prática, pois permite aproximação concorrente com amostragem. Além disso, não é necessário construir explicitamente a decomposição celular do domínio, pois as decomposições clássicas são definidas por regras algorítmicas. Por exemplo, mas triangulações clássicas, dois triângulos adjacentes são simétricos em relação à aresta comum. Assim, se uma curva “sai” de um triângulo cruzando uma aresta, ela entra no triângulo definido por essa aresta e pela reflexão do vértice oposto (Figura 15).

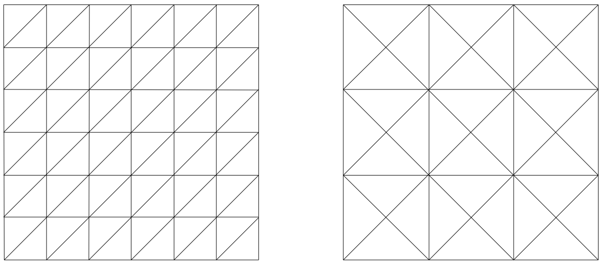

Figura 15: Triangulações clássicas: Coxeter-Freudenthal (esquerda) e J_1 (direita).

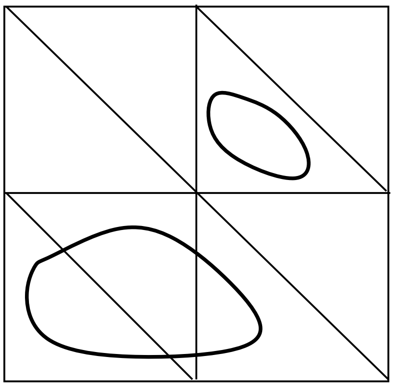

Figura 16: *Aliasing* em enumeração.

4.5 Enumeração adaptativa

A maior desvantagem de enumeração uniforme completa é ter que usar células pequenas para obter uma boa aproximação, embora a maior parte das células não contribuam para a aproximação (Figura 14). O método de enumeração adaptativa abaixo, a exemplo do método de amostragem adaptativa para curvas e superfícies paramétricas, explora recursivamente o domínio de h, gerando uma *quadtree*:

```
explore(r):
    if haszero(r)
        if toosmall(r)
            visit(r)
        else
            r1, r2, r3, r4 ← split(r)
            explore(r1)
            explore(r2)
            explore(r3)
            explore(r4)
```

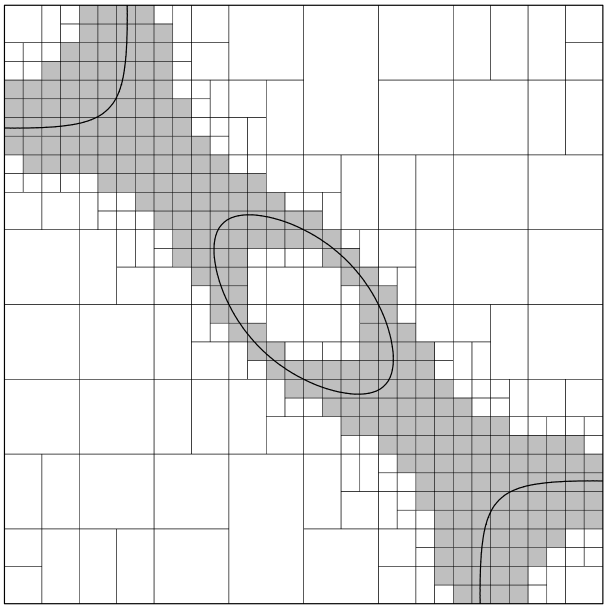

Figura 17: Enumeração adaptativa de uma quártica com aritmética intervalar ($h(x,y) = x^2 + y^2 + xy - (xy)^2/2 - 1/4$, $\Omega = [-2,2] \times [-2,2]$).

O critério de fim de recursão, representado pela função `toosmall`, depende da aplicação. Para síntese de imagens, pode significar "o retângulo r é menor do que um *pixel*". Para outras aplicações, pode significar $|h(r)| \leq \varepsilon$. Uma questão mais difícil é como decidir se r contém zeros de h, representada pela função `haszero`. Queremos evitar amostragem para poder eliminar partes grandes do domínio, e não precisar usar alta resolução. A resposta está em usar aritmética intervalar [14, 30, 33, 34]: dado um retângulo r, usamos aritmética intervalar para calcular uma estimativa conservadora para a imagem $h(r)$, isto é, um intervalo que contenha $h(r)$ (Apêndice A). Se este intervalo não contiver zero, então r não contém zeros de h. A Figura 17 mostra um exemplo de enumeração adaptativa usando aritmética intervalar. Note como partes grandes do domínio são descartadas.

As estimativas dadas pela aritmética intervalar são excessivamente conservadoras, chegando a serem inúteis em cálculos longos, principalmente porque não levam em consideração acoplamento de subexpressões. Uma generalização recente de aritmética intervalar, chamada aritmética afim, considera acoplamentos e, consequentemente, é capaz de produzir estimativas melhores (Figura 18). Enumeração adaptativa usando aritmética afim promete ser um excelente método para aproximação objetos implícitos, principalmente para síntese de imagens [11].

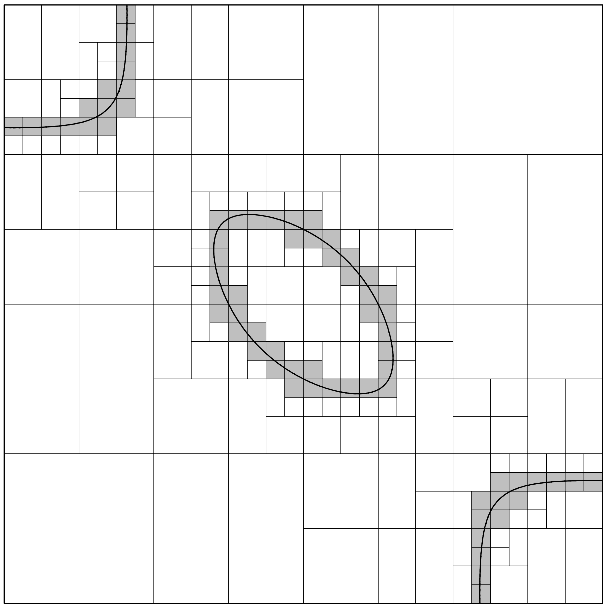

Figura 18: Enumeração adaptativa da quártica com aritmética afim.

4.6 Aproximação por subdivisão

Uma outra maneira de evitar o alto custo de enumeração uniforme de alta resolução é usar baixa resolução para localizar a curva ou superfície obtendo uma aproximação topologicamente correta mas geometricamente grosseira, e depois refinar a geometria usando subdivisão da aproximação grosseira.

Considere o círculo da Figura 14. Para melhorar essa aproximação poligonal, podemos subdividi-la nos pontos médios dos segmentos. Claro que só subdividir não adianta, pois os novos vértices não estão sobre o círculo, mas isso é facilmente corrigido projetando os novos vértices no círculo. Uma extensão trivial deste método obtém ótimas aproximações da esfera a partir dos sólidos platônicos [6]. Estas aproximações da esfera têm a vantagem de serem bem regulares, ao contrário das aproximações geradas pela parametrização clássica da Seção 3.

No caso geral, o refinamento também é feito subdividindo a aproximação grosseira e "projetando" os novos vértices na curva ou superfície. Para isso, podemos usar os novos vértices como estimativa inicial para um método de solução de equações (Apêndice C). Explorando o fato que estamos lidando com objetos implícitos, podemos tomar um corretor de Newton, como no método de Euler-Newton (Seção 4.3). Uma outra possibilidade é usar um corretor pseudo-físico [35]:

$$
\begin{array}{l}
p' \leftarrow p - \delta \,\operatorname{sign}(h(p))\, \nabla h(p) \\
\text{if } h(p)h(p') < 0 \\
\qquad \delta \leftarrow \delta/2
\end{array}
$$

Além disso, basta subdividir segmentos ou polígonos somente enquanto a aproximação não é boa; deste modo obtemos um método de aproximação adaptaviva à curvatura do objeto [35]. Para outros métodos adaptativos, veja [3, 23].

5 Agradecimentos

Este mini-curso é uma extensão de um mini-curso ministrado na UNICAMP no verão de 1994, durante uma visita financiada pela FAPESP. As Figuras 12, 13 e 15 foram gentilmente cedidas por Luiz Velho. As Figuras 17 e 18 foram gentilmente cedidas por Jorge Stolfi. Agradeço o apoio do CNPq, do Projeto VisGraf no IMPA, do TeCGraf da PUC-Rio, e do Computer Systems Group da Universidade de Waterloo.

A Aritmética intervalar

Um dos principais problemas no estudo de algoritmos numéricos é o controle de erros devidos à representação de números reais no conjunto discreto dos números de ponto flutuante de precisão fixa. Esses erros são diferentes dos erros de aproximação de cada algoritmo, que ocorrem por que as soluções desejadas não podem ser calculadas com um número finito de operações elementares.

Aritmética intervalar (IA) é uma técnica para atacar os erros de ponto flutuante de forma automática [27]. Com IA, um número real x é representado por um par de números de ponto flutuante, $[a, b]$, representando um intervalo garantido conter x, isto é, tal que $a \leq x \leq b$. Deste modo, temos não só uma estimativa para o valor de x (por exemplo, dada pelo centro do intervalo), mas também uma medida da qualidade dessa estimativa, dada pela largura do intervalo.

Para implementar um algoritmo numérico com IA, basta usar variantes das operações elementares e composição. Por exemplo, para as operações aritméticas elementares temos:

$$
\begin{array}{rcl}
[a,b] + [c,d] & = & [a+c, b+d] \\
[a,b] - [c,d] & = & [a-d, b-c] \\
[a,b] \times [c,d] & = & [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)] \\
[a,b] \,/\, [c,d] & = & [a,b] \times [1/d, 1/c], \quad \text{se } 0 \notin [c,d]
\end{array}
$$

Com um pouco mais de esforço, podemos escrever fórmulas para as funções elementares como raiz quadrada, seno, coseno, logaritmo, exponencial, etc. Desse modo, se sabemos escrever um algoritmo para calcular uma função f de n variáveis $x_1, \ldots, x_n$, então sabemos escrever um algoritmo para calcular uma extensão $\bar{f}$ de f em IA, isto é, uma função tal que

$$x_i \in \bar{x}_i \Rightarrow f(x_1, \ldots, x_n) \in \bar{f}(\bar{x}_1, \ldots, \bar{x}_n).$$

Em outras palavras, $\bar{f}$ dá uma estimativa conservadora para o conjunto dos valores de f. Como vimos, vários algoritmos em computação gráfica podem ser acelerados com este tipo de estimativa robusta.

As estimativas fornecidas por IA tendem a ser muito conservadoras, isto é, os intervalos calculados para uma expressão podem ser muito maiores do que o intervalo verdadeiro. Este problema é ampliado em longos cálculos encadeados, e frequentemente obtemos estimativas inúteis, de tão grandes.

A principal razão para esse conservadorismo excessivo é que as fórmulas de IA assumem que os argumentos são independentes. Se isso não é verdade, então as fórmulas não são as melhores possíveis, porque nem todas as combinações de valores dos argumentos são válidas. Esse problema é chamado o *problema de dependência* em IA. Um exemplo extremo é a expressão $x - x$, que embora identicamente nula, é calculada cegamente em IA como:

$$\begin{aligned} x &= [a \,..\, b] \\ x - x &= [a - b \,..\, b - a] \end{aligned}$$

Assim, um intervalo que deveria ter largura zero tem largura o dobro da largura do argumento. Um exemplo menos extremo é $x(10 - x)$ para x em $[4 \,..\, 6]$:

$$\begin{aligned} \bar{x} &= [4 \,..\, 6] \\ 10 - \bar{x} &= [10 \,..\, 10] - [4 \,..\, 6] = [4 \,..\, 6] \\ \bar{x}(10 - \bar{x}) &= [4 \,..\, 6] \cdot [4 \,..\, 6] = [16 \,..\, 36]. \end{aligned}$$

Note que o resultado $[16 \,..\, 36]$ é 20 vezes maior do que $[24 \,..\, 25]$, o valor real de $x(10 - x)$ em $[4 \,..\, 6]$. Essa discrepância é devida à relação inversa existente entre x e $10 - x$, que não é levada em consideração na fórmula de multiplicação em IA.

B Aritmética afim

Aritmética afim (AA) é um modelo de cálculo numérico proposto para atacar o problema de dependência em IA [7]. AA não só controla automaticamente erros de arredondamento e truncamento como IA, mas também controla possíveis *correlações* entre as quantidades num cálculo. Graças a isso, AA consegue fornecer estimativas muito melhores do que IA, principalmente em longos cálculos encadeados.

Uma quantidade x é representada em AA por uma *forma afim*, que é um polinômio de grau 1:

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n.$$

Os x_i são coeficientes reais e os ε_i são variáveis simbólicas, chamados *símbolos de ruído*, cujos valores são desconhecidos mas assumidos estarem no intervalo $\mathbf{U} = [-1 \,..\, +1]$. Os símbolos de ruído representam fontes independentes de incerteza no valor de x; o coeficiente x_i dá a magnitude da incerteza devida à fonte ε_i.

A principal vantagem de representar quantidades com formas afins em vez de intervalos é que duas ou mais quantidades podem compartilhar alguns símbolos de ruído, significando que elas não são independentes entre si. Como essa dependência é representada, ainda que implicitamente, AA consegue fornecer estimativas melhores, como mostra o exemplo no final desta seção.

É simples usar AA para obter estimativas intervalares. Primeiro, converta todos os argumentos de uma expressão para formas afins. A seguir, calcule a expressão desejada usando estas formas afins. No final, converta o resultado para um intervalo. Vejamos como fazer esses três passos.

Dado um intervalo $\bar{x} = [a \,..\, b]$ representando uma quantidade x, uma forma afim equivalente é $\hat{x} = x_0 + x_k\varepsilon_k$, onde

$$x_0 = \frac{b + a}{2} \quad \text{and} \quad x_k = \frac{b - a}{2}\,.$$

Um novo símbolo ε_k deve ser usado para cada variável independente na expressão. Reciprocamente, o valor de uma quantidade x repreentada por uma forma afim $\hat{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$ é garantido estar no intervalo

$$[\hat{x}] = [x_0 - \xi \,..\, x_0 + \xi], \text{ onde } \xi = \|\hat{x}\| := \sum_{i=1}^{n} |x_i|\,.$$

Note que $[\hat{x}]$ é o menor intervalo que contém todos os valores possíveis de $\hat{x}$, assumindo que cada ε_i varia independentemente no intervalo $\mathbf{U} = [-1 \,..\, +1]$.

Para calcular uma expressão em AA, basta usar uma versão AA de cada operação elementar $z = f(x, y)$. As operações afins são as mais fáceis porque podem ser calculadas exatamente. Dadas formas afins

$$\begin{aligned} \hat{x} &= x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n \\ \hat{y} &= y_0 + y_1\varepsilon_1 + \cdots + y_n\varepsilon_n \end{aligned}$$

e $\alpha \in \mathbf{R}$, temos

$$\begin{aligned} \hat{x} \pm \hat{y} &= (x_0 \pm y_0) + (x_1 \pm y_1)\varepsilon_1 + \cdots + (x_n \pm y_n)\varepsilon_n \\ \alpha\hat{x} &= (\alpha x_0) + (\alpha x_1)\varepsilon_1 + \cdots + (\alpha x_n)\varepsilon_n \\ \hat{x} \pm \alpha &= (x_0 \pm \alpha) + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n. \end{aligned}$$

Note que $\hat{x} - \hat{x}$ é identicamente nula em AA.

Para operações não afins, usamos a melhor* aproximação afim, e adicionamos um termos $z_k\varepsilon_k$ para

*no sentido de Chebyshev de minimizar o maior erro.

representar o erro dessa aproximação:

$$\hat{z} = z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n + z_k\varepsilon_k.$$

Novamente, ε_k é um símbolo novo, e z_k é uma estimativa para o erro de aproximação. Desse modo, podemos encontrar fórmulas para todas as operações e funções elementares. Por exemplo, a multiplicação é dada por:

$$\begin{aligned} z_0 &= x_0 y_0 \\ z_i &= x_0 y_i + y_0 x_i \qquad (i = 1..n) \\ z_k &= ||\hat{x}||\,||\hat{y}||. \end{aligned}$$

Para ver o efeito de AA no problema de dependência, considere novamente o exemplo: calcular $z = x(10 - x)$ para x em $[4 \,..\, 6]$:

$$\begin{aligned} \hat{x} &= 5 + 1\varepsilon_1 \\ 10 - \hat{x} &= 5 - 1\varepsilon_1 \\ \hat{z} = \hat{x}(10 - \hat{x}) &= 25 + 0\varepsilon_1 - 1\varepsilon_2 \\ [\hat{z}] &= [25 - 1 \,..\, 25 + 1] = [24 \,..\, 26]. \end{aligned}$$

Note que a correlação entre x e $10 - x$ cancelou ε_1 no produto. Note também que o intervalo associado a $\hat{z}$ é muito mais perto do valor real $[24 \,..\, 25]$, e muito melhor do que a estimativa de IA, $[16 \,..\, 36]$.

Embora AA seja mais difícil de implementar e mais cara de executar do que IA, muitas vezes as estimativas melhores fornecidas por AA tornam os algoritmos globalmente mais rápidos.

C Equações não lineares

É frequentemente necessário resolver equações $f(x) = 0$, onde f é uma função $\mathbf{R} \to \mathbf{R}$. Um número real $\hat{x}$ tal que $f(\hat{x}) = 0$ é chamado uma *raiz* da equação $f(x) = 0$, ou um *zero* de f. Existem fórmulas simples para achar zeros de funções lineares ou quadráticas; existem até fórmulas para equações de grau 3 e 4, mas elas não são simples e não são muito utilizadas na prática. (Por exemplo, para resolver uma equação cúbica geral, é necessário usar números complexos.) Para funções gerais, não necessariamente polinomiais, não existem fórmulas, somente métodos para calcular soluções aproximadas. Uma solução aproximada pode ser um ponto $\bar{x}$ tal que a equação $f(x) = 0$ é satisfeita aproximadamente, isto é, tal que $|f(\bar{x})| < \varepsilon$; ou então um ponto $\bar{x}$ que está perto de uma raiz de f, isto é, tal que $|\bar{x} - \hat{x}| < \delta$.

Biseção

Suponha que conhecemos um intervalo $[a, b]$ no qual $f(a)$ e $f(b)$ tem sinais opostos. Se f é contínua em $[a, b]$, então f tem pelo menos uma raiz em $[a, b]$, pelo teorema do valor intermediário.

O método da biseção calcula uma sequência de intervalos encaixados, todos guarantidos conter uma raiz de f por terem valores de sinais opostos nos extremos. Para refinar um intervalo $[a, b]$, tomamos o seu ponto médio $c = (a+b)/2$ e examinamos o sinal de $f(c)$. Se $f(c)$ não tiver o mesmo sinal de $f(a)$, então o próximo intervalo é $[a, c]$; caso contrário, o próximo intervalo é $[c, b]$. Esse processo é repetido até que o intervalo seja suficientemente pequeno, quando então tomamos o seu ponto médio como valor da raiz.

```
bisect(a, b, fa, fb):
    c ← (a + b)/2
    if |a − b| < δ return c
    fc ← f(c)
    if fa fc < 0
        return bisect(a, c, fa, fc)
    else
        return bisect(c, b, fc, fb)
```

Este algoritmo obtém um dígito binário da raiz a cada iteração pois o tamanho dos intervalos é dividido por 2 a cada passo. Portanto, são necessários $\log_2 |a - b|/\delta$ passos para atingir uma tolerância δ. Note que podemos garantir que a aproximação calculada por esse método está a menos de δ de uma raiz de f, mesmo sem conhecer essa raiz. Embora seja lento na prática, uma vez conhecido um intervalo no qual f muda de sinal, o método da biseção sempre converge para uma raiz de f.

Regula falsi

O método da biseção divide cegamente o intervalo corrente em duas partes iguais, não levando em consideração os valores de f nos extremos. Por exemplo, se $|f(a)|$ é grande e $|f(b)|$ é pequeno, então provavelmente a raiz de f está mais perto de b do que de a. Assim, em vez de usar o ponto médio para refinar o intervalo $[a, b]$, podemos calcular c por interpolação linear:

$$\frac{c - a}{0 - f(a)} = \frac{b - a}{f(b) - f(a)}$$

ou seja

$$c = a - f(a)\frac{b - a}{f(b) - f(a)}.$$

Apesar do argumento acima, o método da *regula falsi* não é necessariamente mais rápido do que o método da biseção, podendo ser até mais lento. Tipicamente, método da *regula falsi* não consegue achar um intervalo pequeno contendo uma raiz, embora consiga achar pontos nos quais f é pequena.

Newton

O método de Newton usa a derivada de f num ponto perto de uma raiz para obter um ponto ainda mais perto, calculando assim uma sequência (x_n) de aproximações para uma raiz, onde

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

O ponto x_{n+1} é o ponto aonde a reta tangente a f no ponto x_n encontra o eixo x. Em outras palavras, x_{n+1} é a raiz da aproximação linear a f no ponto x_n. Se f fosse linear, o ponto x_{n+1} seria a raiz de f.

O método de Newton é muito mais rápido do que os métodos anteriores, pois dobra o número de dígitos corretos a cada iteração, mas precisa que a função seja bem comportada (e.g., tenha raízes simples) e precisa de uma boa estimativa inicial; caso contrário, ele pode divergir.

Secante

Uma desvantagem do método de Newton é precisar da derivada da função. O método da secante evita esse cálculo, usando a reta secante aos dois últimos pontos, em vez da reta tangente no último ponto:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

O método da secante tem uma convergência mais lenta do que o método de Newton, mas ainda mais rápida do que o método de biseção. Ambos os métodos são *métodos de ponto fixo*. Nessa classe de métodos, a equação $f(x) = 0$ é substituída pela equação $g(x) = x$, tal que se $g(\hat{x}) = \hat{x}$ então $f(\hat{x}) = 0$. A busca por uma solução da equação de ponto fixo $g(x) = x$ é feita calculando uma sequência (x_n), onde $x_{n+1} = g(x_n)$.

É claro que a função g tem que ser construída com cuidado para garantir convergência. o lema da contração de Banach é util aqui: basta que g seja diferenciável com derivada estritamente menor do 1.*

*Exercçio: verifique isso para a função g do método de Newton.

Aritmética intervalar

O maior problema dos métodos de refinamento de intervalo, como o método da biseção, é encontrar um intervalo inicial no qual a função f muda de sinal, e portanto contém pelo menos uma raiz de f. Aritmética intervalar é a técnica natural para calcular esse tipo de garantia. Uma variante simples do método da biseção usando aritmética intervalar encontra *todos* os zeros de f no intervalo inicial:

```
I_bisect(a,b):
    if 0 ∈ f̄(a,b)
        c ← (a+b)/2
        if |a−b| < δ return c
        I_bisect(a,c)
        I_bisect(c,b)
```

Se a execução for encerrada assim que um zero for encontrado, então esse algoritmo encontra o menor zero de f no intervalo dado, sendo portanto adequado para *ray-casting*. Esse mesmo algoritmo pode ser implementado com aritmética afim, sem modificações, e com provável ganho. De qualquer modo, esse algoritmo acha todos os zeros de f em ordem.

Amostragem adaptativa

Uma amostragem adaptativa da função f permite localizar *todos* os intervalos nos quais f muda de sinal. Além disso, como em cada intervalo f é aproximadamente linear, um método como biseção ou *regula falsi* vai convergir rapidamente para um zero de f.

D Quadratura de funções

O problema de quadratura de funções é aproximar o valor da integral definida

$$I = \int_a^b f(x)\,dx,$$

para uma função $f : [a,b] \to \mathbf{R}$.

Uma família de métodos numéricos para esse problema é obtida fazendo aproximações polinomiais da função f no intervalo $[a,b]$. Para uma aproximação de grau 0, temos a *regra do retângulo*:

$$I \approx (b-a)f(a).$$

Para uma outra aproximação de grau 0, temos a *regra do ponto médio*:

$$I \approx (b-a)f(\frac{a+b}{2}).$$

Para uma aproximação de grau 1, temos a *regra do trapézio*:

$$I \approx \frac{b-a}{2}(f(a)+f(b)).$$

Para uma aproximação de grau 2, temos a *regra de Simpson*:

$$I \approx \frac{b-a}{6}(f(a)+4f(\frac{a+b}{2})+f(b)).$$

Um método para obter essas e outras regras é tomar uma amostra $a = x_0 < x_1 < \ldots < x_n = b$ e considerar aproximações do tipo

$$I \approx A_0 f(x_0) + A_1 f(x_1) + \ldots + A_n f(x_n)$$

que sejam exatas para polinômios que coincidam com f na amostra. Tomando amostras uniformes, obtemos as *fórmulas de Newton-Cotes*, que são exatas para todos os polinômios de grau até n, e que incluem as regras acima como casos particulares [31]. Por outro lado, escolhendo a amostra com cuidado, podemos obter regras de integração com n pontos amostrais que são exatas para todos os polinômios de grau até $2n-1$. Essa técnica é chamada *quadratura gaussiana* [8, 31].

Como não há razão para a função f ser bem aproximada por polinômios de grau baixo, raramente as regras elementares acima dão uma boa estimativa para a integral I. Na prática, escolhe-se uma regra, toma-se uma aproximação polinomial por partes sobre uma amostragem do intervalo $[a,b]$, aplica-se a regra em cada parte polinomial, e soma-se o resultado. Frequentemente, tomamos uma amostra uniforme de tamanho δ. Como os intervalos têm extremos em comum, podemos fazer simplificações, obtendo *regras compostas*. Assim, a regra do retângulo composta é:

$$I \approx \delta(f(x_0) + f(x_1) + \ldots + f(x_n)).$$

A regra do trapézio composta é:

$$I \approx \frac{\delta}{2}(f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)).$$

A regra de Simpson composta é:

$$I \approx \frac{\delta}{6}(f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+\ldots+f(x_n)).$$

Por outro lado, vimos na Seção 2.6 como uma amostragem adaptativa simplifica esse problema.

E Equações diferenciais ordinárias

Uma equação diferencial ordinária (de primeira ordem) é dada por

$$\dot{u} = f(u,t),$$

onde u é uma função da variável independente t com valores em $\mathbf{R}^d$, e $\dot{u}$ é a derivada de u em relação a t. Um problema contendo uma equação diferencial não está completamente especificado pela equação: *condições de contorno* são necessárias. As condições de contorno mais simples e mais frequentes são dadas por valores iniciais. Nestes *problemas de valor inicial*, também chamados de *problemas de Cauchy*, é dado o valor $u(t_0)$ de u em um ponto t_0 no domínio. O que se deseja como solução é o futuro, isto é, $u(t)$ para $t \geq t_0$. Assim, o problema de Cauchy de primeira ordem é:

$$\dot{u} = f(u,t), \qquad u(t_0) = u_0.$$

O teorema de Picard garante que este problema possui uma única solução quando f é suficientemente diferenciável.

Só raramente é possível resolver equações diferenciais exatamente, com fórmulas fechadas; em geral, só é possível obter soluções numéricas. Para isso, precisamos discretizar as equações diferenciais.

Soluções discretas são obtidas discretizando o tempo t em uma sequência $t_0, t_1, t_2, \ldots$, e aproximando a solução contínua $u(t)$ em cada instante t_n, obtendo assim uma sequência $u_0, u_1, u_2, \ldots$, tal que $u_n \approx u(t_n)$.

Os vários métodos numéricos diferem no modo como u_n é calculado. Em geral, u_n é calculado em função dos valores anteriores $u_{n-1}, \ldots, u_0$. Se k valores anteriores $u_{n-1}, \ldots, u_{n-k}$ são usados neste cálculo, então dizemos que o método é um *método de k passos*. Discutiremos somente métodos de um passo.

A técnica fundamental de discretização é transformar equações diferenciais em *equações de diferença*. Conceitualmente, esta transformação é feita substituindo dt por $t_{n+1}-t_n$ e du por $u_{n+1}-u_n$ na equação diferencial, Uma implementação literal desta substituição dá um método de 1 passo, chamado o *método de Euler*:

$$\begin{aligned} u_{n+1} &= u_n + h_n f(u_n, t_n) \\ t_{n+1} &= t_n + h_n. \end{aligned}$$

O passo h_n é arbitrário, e frequentemente tomado independente de n.

O método de Euler é muito simples, mas não é muito preciso nem estável. Outros métodos de discretização usam a expansão de Taylor de u. Enquanto o método de Euler usa uma extrapolação afim no intervalo $[t_n, t_{n+1}]$, estes métodos usam extrapolações polinomiais de grau superior. Além disso, estes métodos evitam o cálculo das derivadas de f, usando, também, aproximações por diferenças. Por exemplo, o método mais simples depois do método de Euler é o *método do ponto médio*:

$$\begin{aligned}
k_1 &= h_n f(u_n, t_n) \\
k_2 &= h_n f(u_n + k_1/2, t_n + h_n/2) \\
u_{n+1} &= u_n + k_2 \\
t_{n+1} &= t_n + h_n.
\end{aligned}$$

Este método é um *método de Runge–Kutta* de segunda ordem. Um outro método de Runge–Kutta de segunda ordem é:

$$\begin{aligned}
k_1 &= h_n f(u_n, t_n) \\
k_2 &= h_n f(u_n + k_1, t_n + h_n) \\
u_{n+1} &= u_n + +k_1/2 + k_2/2 \\
t_{n+1} &= t_n + h_n.
\end{aligned}$$

De longe, o método de Runge–Kutta mais popular é o de quarta ordem:

$$\begin{aligned}
k_1 &= h_n f(u_n, t_n) \\
k_2 &= h_n f(u_n + k_1/2, t_n + h_n/2) \\
k_3 &= h_n f(u_n + k_2/2, t_n + h_n/2) \\
k_4 &= h_n f(u_n + k_3, t_n + h_n) \\
u_{n+1} &= u_n + k_1/6 + k_2/3 + k_3/3 + k_4/6 \\
t_{n+1} &= t_n + h_n.
\end{aligned}$$

Na prática, o método de Runge–Kutta de quarta ordem apresentado acima é combinado com um de quinta ordem para fazer um controle adaptativo do passo h_n. Este método combinado, chamado *método de Runge–Kutta–Fehlberg*, é bastante robusto e merecidamente popular. Entretanto, a função f é avaliada seis vezes neste método, contra somente uma no método de Euler. Em geral, quanto menos vezes f for avaliada, mais rápido o método (mas também menos preciso).

Quando a função f só depende de t, e não de u, o problema de Cauchy

$$\dot{u} = f(u, t), \qquad u(t_0) = u_0$$

é na verdade um problema de integração definida, cuja solução é

$$u(t) = u_0 + \int_{t_0}^{t} f(\tau)\, d\tau.$$

Neste caso, os métodos numéricos apresentados acima para solução do problema de Cauchy correspondem às regras de quadratura apresentadas no Apêndice D. De fato, o método de Euler corresponde à regra do retângulo, o método do ponto médio corresponde à regra do ponto médio,* o outro método de Runge–Kutta de segunda ordem corresponde à regra do trapézio, e o método de Runge–Kutta de quarta ordem corresponde à regra de Simpson.

Referências

[1] E. L. Allgower, K. Georg. *Numerical Continuation Methods: An Introduction*. Springer-Verlag, 1990.

[2] D. H. Ballard. Strip trees, a hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, 1981.

[3] J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, 1988.

[4] R. E. Chandler. A tracking algorithm for implicitly defined curves. *IEEE Computer Graphics and Applications*, 8(2):83–89, 1988.

[5] R. E. Chandler. A recursive technique for rendering parametric curves. *Computers and Graphics*, 14(3/4):477–479, 1990.

[6] A. H. J. Christensen. A note on geodesic polyhedra: Triangulation and contouring of spheres. *Computers and Graphics*, 3:163–165, 1978.

[7] J. L. D. Comba, J. Stolfi. Affine arithmetic and its applications to computer graphics. *Anais do VI SIBGRAPI*, pp. 9–18, 1993.

[8] S. D. Conte, Carl de Boor. *Elementary Numerical Analysis: An algorithmic approach*. McGraw-Hill, 1980.

[9] M. Crampin, R. Guifo, G. Read. Linear approximation of curves with bounded curvature and a data reduction algorithm. *Computer Aided Design*, 17(6):257–261, 1985.

*Supresa!

[10] L. H. de Figueiredo. Adaptive sampling of parametric curves. In A. Paeth, editor, *Graphics Gems V*, pp. 173–178. Academic Press, 1995.

[11] L. H. de Figueiredo, J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. In *Implicit Surfaces '95*, pp. 161–170, April 1995.

[12] D. P. Dobkin, S. V. F. Levy, W. P. Thurston, A. R. Wilks. Contour tracing by piecewise linear approximations. *ACM Transactions on Graphics*, 9(4):389–423, 1990.

[13] D. H. Douglas, T. K. Peuker. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[14] T. Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):131–138, July 1992.

[15] A. Finkelstein, D. H. Salesin. Multiresolution curves. In *Proceedings of SIGGRAPH '94*, pp. 261–268. ACM Press, July 1994.

[16] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1990.

[17] J. Gomes, G. Tavares. *Métodos Simpliciais em Computação Gráfica*. XVII Colóquio Brasileiro de Matemática, IMPA, 1995.

[18] J. Gomes, L. Velho. *Computação Gráfica: Imagem*. IMPA/SBM, 1995.

[19] B. Guenter, R. Parent. Computing the arc length of parametric curves. *IEEE Computer Graphics and Applications*, 10(3):72–78, 1990.

[20] O. Gunther. *Efficient Structures for Geometric Data Management*, volume 337 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[21] O. Gunther, S. Dominguez. Hierarchical schemes for curve representation. *IEEE Computer Graphics and Applications*, 13(3):55–63, 1993.

[22] O. Gunther, E. Wong. The arc tree: an approximation scheme to represent arbitrarily curved shapes. *Computer Vision, Graphics, and Image Processing*, 51(3):313–337, 1990.

[23] M. Hall, J. Warren. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–42, 1990.

[24] J. Hershberger, J. Snoeyink. Speeding up the Douglas-Peucker line simplification algorithm. In *Proc. 5th Intl. Symp. Spatial Data Handling*, pp. 134–143, 1992.

[25] M. Kosters. Curvature-dependent parametrization of curves and surfaces. *Computer Aided Design*, 23(8):569–578, 1991.

[26] T. Lindgren, J. Sanchez, J. Hall. Curve tesselation criteria through sampling. In D. Kirk, editor, *Graphics Gems III*, pp. 262–265. Academic Press, 1992.

[27] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[28] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

[29] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.

[30] J. M. Snyder. Interval analysis for computer graphics. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):121–130, July 1992.

[31] A. H. Stroud. *Numerical quadrature and solution of ordinary differential equations*, volume 10 of *Applied Mathematical Sciences*. Springer-Verlag, 1974.

[32] K. G. Suffern. Quadtree algorithms for contouring functions of two variables. *The Computer Journal*, 33:402–407, 1990.

[33] K. G. Suffern, E. D. Fackerell. Interval methods in computer graphics. *Computers and Graphics*, 15:331–340, 1991.

[34] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14:14–23, 1994.

[35] L. Velho. Adaptive polygonization made simple. *Anais do VIII SIBGRAPI*, pp. 111–118, 1995.

[36] L. Velho, L. H. de Figueiredo. Adaptive polygonal approximation of parametric surfaces. In *Fourth SIAM Conference on Geometric Design*, November 1995.

[37] B. Von Herzen, A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pp. 103–110, July 1987.

[38] C. A. Wüthrich. An analysis of digitalizations algorithms for non-linear curves. In B. Falcidieno, I. Herman, C. Pienovi, editors, *Computer Graphics and Mathematics*, pp. 285–297. Springer, 1992.