B-Splines

EP2 de MAC0210

Renato Lui Geh, NUSP: 8536030

1 Introdução

O EP foi feito na linguagem Python. Foi usada a biblioteca pyglet[1], que age como um *wrapper* de OpenGL para Python. Toda a parte de desenho e GUI foi feita com esta biblioteca.

Para rodar o EP, é preciso do Python 3 e que a biblioteca pyglet esteja instalada. Além disso, alguns cálculos foram feitos com NumPy. Foram usadas várias funções elaboradas durante o EP1.

2 Uso

O EP2, ao contrário do EP1, não é interativo. Ao invés disso, o usuário deve executar o EP2 da seguinte forma.

```
$ python3 main.py
---
Usage: main.py data l
  data - dataset file
  l    - lambda constant for L2
```

[1] Disponível em `https://bitbucket.org/pyglet/pyglet/wiki/Home`

Os parâmetros `data` e `l` são obrigatórios. O primeiro é o caminho para o arquivo de dataset. O segundo é a constant λ para regularização L2. O arquivo `data` é um `plaintext` simples com o valor de cada par de valores (t_i, y_i). Abaixo segue um exemplo de cinco pares deste formato.

```
$ cat in.put
---
0.000000000000000000e+00 4.229733993338091835e+01
1.000000000000000000e+00 4.242230348764690717e+01
2.000000000000000000e+00 4.093792562561398540e+01
3.000000000000000000e+00 4.009299524534277737e+01
4.000000000000000000e+00 4.108056259456515846e+01
...
```

Este arquivo é gerado através de um gerador aleatório `generator.py`. Para gerar um arquivo `data`, requerem-se quatro parâmetros.

```
$ python3 generator.py
---
Usage: generator.py filename sdv n m
  filename - file name to save to
  sdv      - gaussian standard deviation to sample with
  n        - number of samples to generate
  m        - max value for samples
```

O primeiro, `filename`, refere-se ao nome do novo dataset file. O argumento `sdv` é o valor do desvio padrão da gaussiana usada; `n` é o número de amostras a serem geradas; e `m` é o valor máximo que cada valor y_i pode alcançar.

A construção dos pares (t_i, y_i) é bem simples, e é descrita no Algoritmo 1. Desta forma garante-se que todos valores estejam no intervalo $[0, m]$. Como são amostrados por uma gaussiana, podemos amostrar diferentes curvas alterando o valor de σ. Quanto maior σ, mais "bruscas" as curvas.

Abaixo segue o algoritmo descrito.

Algoritmo 1. `generator.py`: gerador de dataset

Entrada Desvio padrão σ, número de amostras n e valor máximo m
Saída Conjunto de pares G

Seja G uma lista vazia
$\mu \leftarrow \frac{m}{2}$
para todo inteiro i no intervalo $[0, n)$ **faça**
    Seja μ uma mostra de $\mathcal{N}(\mu, \sigma^2)$
    se $\mu > m$ **então**
        $mu \leftarrow mu - 2\sigma$
    senão se $\mu < 0$ **então**
        $mu \leftarrow mu + 2\sigma$
    Insere par (i, μ) em G
retorna G

Um exemplo de parâmetros para gerar o dataset:

```
$ python3 generator.py in.put 2 200 40
```

Depois de gerado o dataset, podemos desenhar a spline.

```
$ python3 main.py in.put 0.01
```

Este comando desenhará uma spline com um $\lambda = 0.01$ para regularização L2. A Figura 1 é um exemplo de uma spline desenhada a partir dos pares de pontos gerados pelo gerador. Os pontos em vermelho mostram os pontos reais do dataset. A curva em azul mostra os valores da spline

$$S(t) = \sum_{i=0}^{n} a_i \beta(t - i)$$

para cada valor em $[0, n)$, onde β é a spline básica usada e a é o vetor de coeficientes.

Alterando o λ e o número de splines usadas é possível observar como a curva muda. De fato, é possível perceber que a curva sofre "overfitting" quando λ tende a zero. A Figura 2 mostra este fenômeno com $\lambda = 0.0001$ ("próximo" de zero).

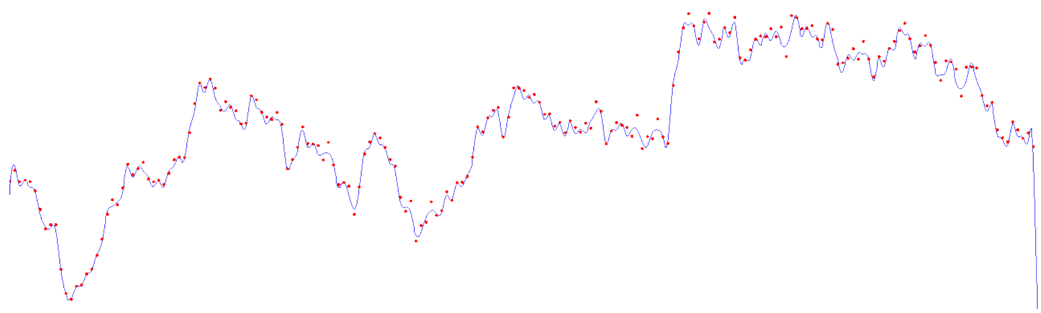

Figura 1: Spline desenhada com $\lambda = 0.01$ e 200 amostras.

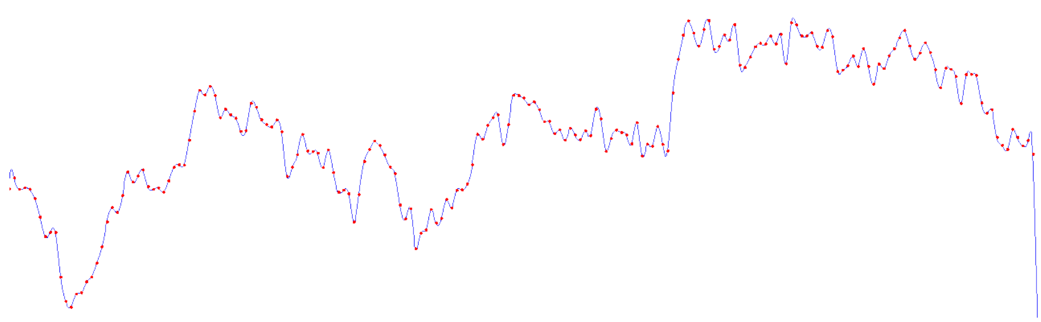

Figura 2: Spline desenhada com $\lambda = 0.0001$ e 200 amostras.

Quando λ é muito grande (no caso observei empíricamente que isso ocorre quando $\lambda \geq 0.1$), a curva perde o sentido, aumentando o erro demais, como mostra a Figura 2.

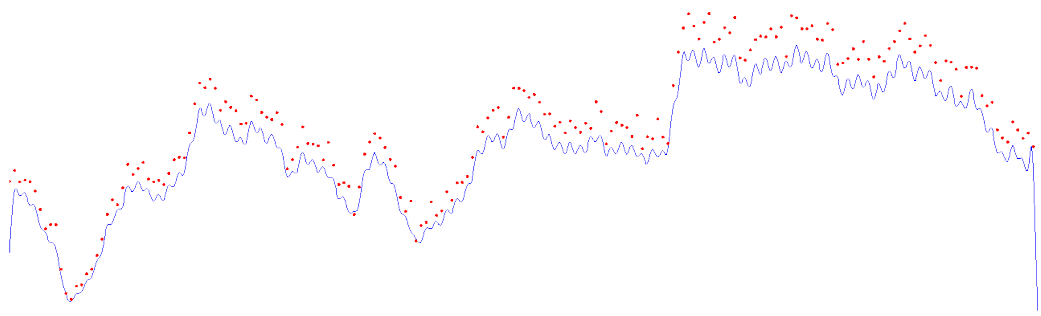

Figura 3: Spline desenhada com $\lambda = 0.1$ e 200 amostras.

Com poucas amostras podemos observar melhor a suavidade da curva com um λ maior.

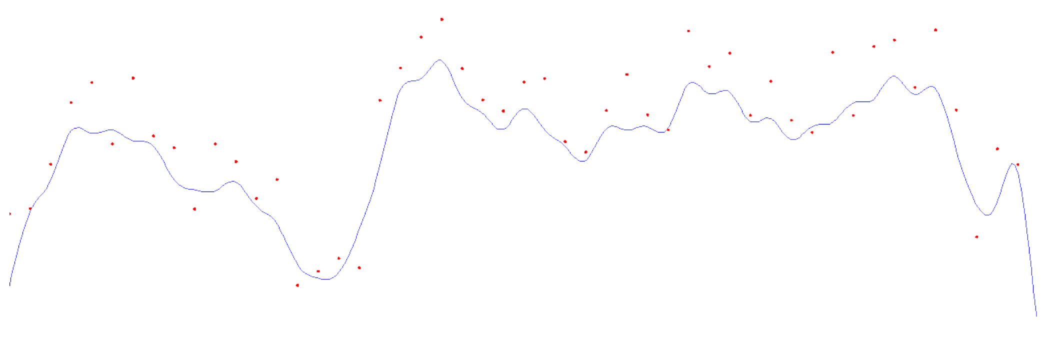

Figura 4: Spline desenhada com $\lambda = 0.1$ e 50 amostras.

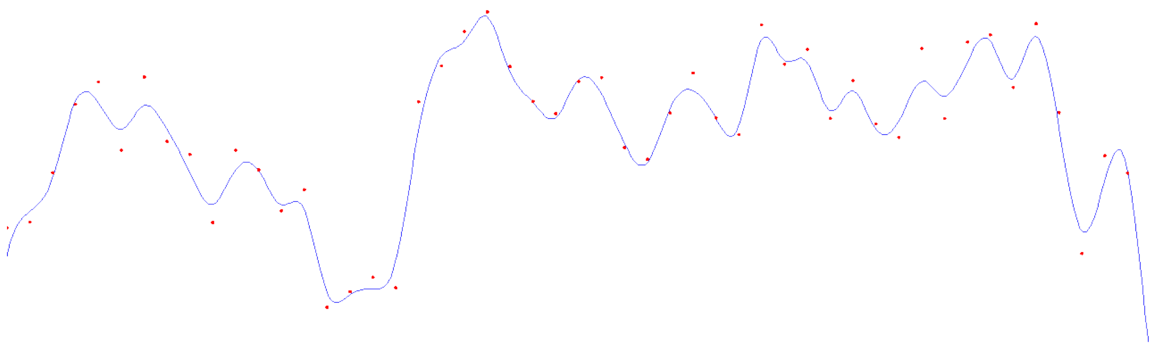

Figura 5: Spline desenhada com $\lambda = 0.01$ e 50 amostras.

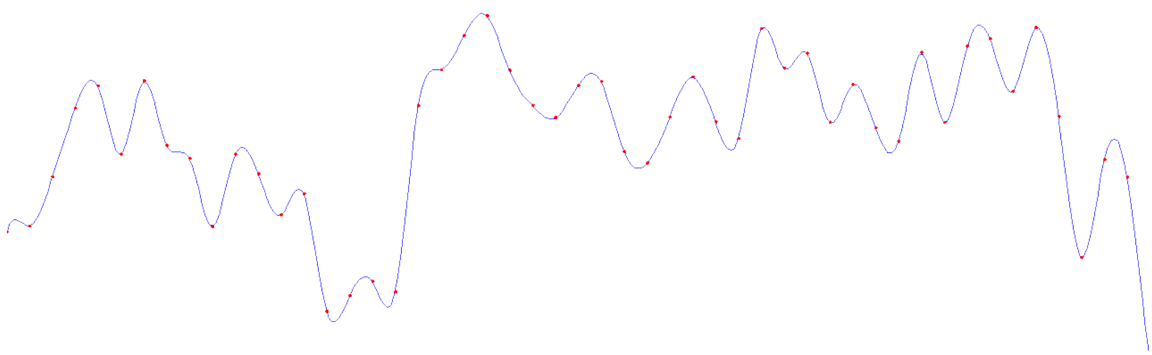

Figura 6: Spline desenhada com $\lambda = 0.0001$ e 50 amostras.

3 Estrutura do código

Em `main.py`, lêem-se os parâmetros dados pela `command line` e em seguida cria-se uma instância da classe `Dataset`, que agrupa os pares (t_i, y_i) para serem lidos pela spline. Após a criação do `Dataset`, é gerada a janela e a tela de desenho.

O arquivo `frame.py` define a janela e cuida de desenhar os elementos no canvas do OpenGL. A classe `Frame`, que representa a janela e o canvas, cria a classe `Spline` e passa o `Dataset` a ela.

Para construir a spline, computa-se primeira a matriz M definida por

$$M_{i,j} = 2\int_0^n \beta(t-i)\beta(t-j)\mathrm{d}t,$$

onde n define um bound para t_i tal que $0 \leq t_i \leq n$. Esta matriz independe do conjunto de dados, e portanto é pré-computada.

Em seguida, usa-se o método `Spline.fit(data, lambda)` para achar um vetor de coeficientes $a \in \mathbb{R}^n$ que modelem o conjunto de dados `data` com constante de regularização L2 $\lambda =$ `lambda`. Para isso, constrói-se a matriz B onde cada elemento toma a forma

$$B_{i,j} = \beta(t_i - j).$$

A matriz M é esparsa, já que quando $|t-k| > 2$, $\beta(t-k) = 0$. Portanto, podemos evitar computar os valores de cada elemento de M, que demoraria tempo $\mathcal{O}(n^2)$, e ao invés disso computar apenas a região da diagonal até uma distância dois para cada lado. Deste jeito, é possível computar M em tempo $\mathcal{O}(n)$.

O algoritmo abaixo computa M em tempo linear. Para computar o valor da integral abaixo, usa-se a lei de Simpson, que simplifiquei para o intervalo $[0, n]$:

$$\int_0^n f(x)\mathrm{d}x \approx f(0) + f(n) + \sum_{i=1}^{n} 2^{(i \mod 2)+1} f(i)$$

Algoritmo 2. `Computa` M

Entrada Bound n tal que $0 \leq t_i \leq n$ é válido para todo $t_i \in D$, D dataset
Saída Matriz M

1: Seja M uma matriz nula de dimensão $n \times n$
2: **para todo** inteiro i no intervalo $[0, n)$ **faça**
3:     **para todo** inteiro d no intervalo $[-4, 4]$ **faça**
4:         $j \leftarrow i + d$
5:         **se** $j \geq 0$ e $j < n$ **então**
6:             $M_{i,j} \leftarrow 2 \int_0^n \beta(t-i)\beta(t-j)\mathrm{d}t$
7: **retorna** M

A matriz B é um tanto diferente, e o mesmo raciocínio não pode ser aplicado neste caso. Já que B depende de cada valor t_i do conjunto de dados, não basta apenas computar a diagonal, já que o conjunto pode não estar ordenado. Portanto, acabei deixando $\mathcal{O}(nm)$. Como assume-se que n é relativamente pequeno, podemos considerar isso praticamente linear.

Agora que temos B e M, podemos computar o vetor coeficiente a partir da derivação do gradiente.

$$\nabla f(a) = (B^\intercal B + \lambda M)a - B^\intercal y = 0 \Rightarrow (B^\intercal B + \lambda M)a = B^\intercal y$$

Algoritmo 3. `Computa` a

Entrada Matrizes M e B, vetor y do dataset e constante λ
Saída Vetor de coeficientes a

1: $A \leftarrow B^\intercal B + \lambda M$
2: $b \leftarrow B^\intercal y$
3: Soluciona sistema linear $Ax = b$ em função de x
4: $a \leftarrow x$
5: **retorna** a

Para achar a solução do sistema linear, usou-se `numpy.linalg.solve`.

Agora que temos a, podemos desenhar a spline. Para isso, decidi percorrer por "todo" (tomando um certo ϵ suficientemente pequeno para a iteração) o domínio da spline S, e para valor de t no domínio, achar o valor de

$$S(t) = \sum_{i=0}^{n} a_i \beta(t-i). \qquad (3.1)$$

No entanto, esta tarefa é $\mathcal{O}(nm)$, e será computada para cada instante de renderização. Portanto precisei otimizar esta parte do código. Otimizar esta soma foi fácil. É fácil ver que $\beta(t-i) = 0$ fora do intervalo $[-4, 4]$. Neste caso, podemos substituir a Equação 3.1 pela forma

$$S(t) = \sum_{i=t-2}^{t+2} a_i \beta(t-i).$$

O algoritmo fica portanto:

Algoritmo 4. `Computa` $S(t)$

Entrada $t \in \mathbb{R}$
Saída $S(t)$

$z \leftarrow 0$
para todo inteiro i no intervalo $[-4, 4]$ **faça**
  $k \leftarrow \lfloor t \rfloor + i$
  se $k \geq 0$ e $k < n$ **então**
    $z \leftarrow z + a_k \beta(t-k)$
retorna z

A função β escolhida foi:

$$\beta(t) = \begin{cases} \frac{(2-t)^3}{4} & \text{, se } 1 \leq t < 2; \\ \frac{3t^3}{4} - \frac{3t^2}{2} + 1 & \text{, se } 0 \leq t < 1; \\ \frac{-3t^3}{4} - \frac{3t^2}{2} + 1 & \text{, se } -1 \leq t < 0; \\ \frac{(t+2)^3}{4} & \text{, caso contrário.} \end{cases}$$

Por causa desta escolha, β toma valores não zero apenas no domínio $[-4, 4]$, e tem imagem em apenas $[0, 1]$. Isto trouxe problemas na hora de desenhar a spline, já que consideramos cada unidade atômica um pixel. Para solucionar este problema, aplicamos uma escala durante a computação de $S(t)$.

Desenhar cada valor de $S(t)$ consistia em desenhar o ponto $(s_x \cdot t, s_y \cdot S(t))$. Esta escala foi escolhida a partir dos valores do tamanho da janela, e de n e m. Seja w e h as dimensões da janela.

$$
\begin{aligned}
s_x &= w/n \\
s_y &= 0.6 \frac{h}{\max\{y^* \in y\}}
\end{aligned}
$$

Desta forma, toda a spline fica visível na tela, desde o primeiro ponto até o último. Além disso, pela escala s_y, a spline fica mais ou menos no meio da tela.