SAWP (http://www.sawp.com.br/blog/)
Pedro Garcia's Blog

3.1.7 Aproximação de Funções — Interpolação — Splines

Posted on 24 de February de 2011 (http://www.sawp.com.br/blog/?p=1130) by SAWP (http://www.sawp.com.br/blog/?author=2)

1. Interpolação por Splines

O conceito de spline foi originado de uma técnica de desenho arquitetônico em que usa-se um barbante flexível — chamado *spline* — para desenhar uma curva lisa passando por um conjunto de pontos. Nessa técnica, o desenhista coloca papel sobre uma base fixa e prega pequenos pinos ao papel sobre os pontos dados. Uma curva contínua é usada para intercalar estes pontos afixados, podendo ser reta (linear), quadrática ou cúbica.

Esta ideia foi utilizada em análise numérica para ajuste de funções. Neste caso, utilizamos polinômios de pequeno grau para união dos pontos consecutivos da amostragem.

1.1. Splines Lineares

O ajuste mais simples que podemos fazer com dois pontos é uni-los por uma reta. Os splines de primeiro grau para um conjunto de pontos amostrados podem ser definidos como um conjunto de funções lineares, tais que

$$\begin{aligned} f(x) &= f(x_0) + m_0(x - x_0) \text{ , para } x_0 \leq x \leq x_1 \\ f(x) &= f(x_1) + m_1(x - x_1) \text{ , para } x_1 \leq x \leq x_2 \\ &\vdots \\ f(x) &= f(x_{n-1}) + m_{n-1}(x - x_{m-1}) \text{ , para } x_{n-1} \leq x \leq x_n \end{aligned}$$

onde m_i é o coeficiente angular da reta que liga os pontos:

$$m_i = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Essas equações podem ser utilizadas para calcular a função em qualquer ponto entre x_0 e x_n . Notamos que este método é idêntico à interpolação linear.

1.2. Splines Quadráticos

Este método é obviamente idêntico à interpolação por splines lineares. A única diferença está na função de união dos pontos, que deve ser quadrática. Para isso, o objetivo dos splines quadráticos é determinar um polinômio de segundo grau para cada intervalo entre os pontos dados. Esse polinômio é representado pela função

$$f_i(x) = a_i x^2 + b_i x + c_i$$

Para $n+1$ pontos dados $(i = 0, 1, 2, 3, \ldots, n)$, existem n intervalos e, portanto, $3n$ constantes indeterminadas para calcularmos: a_i , b_i e c_i . Portanto, criamos $3n$ equações para calcular estas incógnitas. São elas:

1. Dos valores da função e dos polinômios adjacentes que devem ser iguais aos pontos interiores. Estas equações são:

$$a_{i-1}x_{i-1}^2 + b_{i-1}x_{i-1} + c_{i-1} = f(x_{i-1})$$
$$a_i x_{i-1}^2 + b_i x_{i-1} + c_i = f(x_{i-1})$$

   para $i = 2 \cdots n$. Como apenas os pontos internos amostrados foram usados, as equações acima fornecem cada uma $n-1$ equações para um total de $2n-2$ equações.

2. A primeira e a última função deve passar pelos pontos extremos. Com isso, temos duas equações adicionais:

$$a_i x_0^2 + b_1 x_0 + c_1 = f(x_0)$$
$$a_n x_n^2 + b_n x_n + c_n = f(x_n)$$

3. As primeiras derivadas nos pontos interiores devem sempre ser iguais. Isto é, como a função de ajuste é de segundo grau, sua derivada terá a forma

$$f'(x) = 2ax + b$$

   Portanto, a condição geral será

$$2a_{i-1}x_{i-1} + b_{i-1} = 2a_i x_{i-1} + b_i$$

   para $i = 2, \cdots, n$. Isso fornece outras $n-1$ equações para um total de $3n-1$.

4. A última condição supõe que a segunda derivada seja nula no primeiro ponto. Isto é

$$a_1 = 0$$

1.3. Splines Cúbicos

O objetivo nos splines cúbicos é determinar um polinômio de terceiro grau para cada intervalo entre os pontos amostrados. Ou seja, a aproximação de pontos consecutivos obedece à função

$$f_i(x) = a_i x^3 + b_i x + c_i x + d_i$$

Logo, para $n+1$ pontos dados $(i = 0, 1, 2, \ldots, n)$, existem n intervalos e, consequentemente, $4n$ constantes indeterminadas.

Para dedução dos splines cúbicos, nos baseamos na observação de que, como cada par de pontos amostrados é ligado por um polinômio cúbico, a segunda derivada no interior de cada intervalo é uma reta. A equação acima pode ser derivada duas vezes para verificar esta observação. Com base nisso, as segundas derivadas podem ser representadas por um polinômio interpolador de Lagrange de primeiro grau:

$$f_i"(x) = f_i"(x_{i-1})\frac{x-x_i}{x_{i-1}-x_i} + f_i"(x_i)\frac{x-x_{i-1}}{x_i-x_{i-1}}$$

onde $f_i"(x)$ é o valor da segunda derivada em um ponto qualquer x no $i-esimo$ intervalo. Portanto, essa equação é uma reta ligando a segunda derivada no primeiro ponto $f"(x_{i-1})$ com a segunda derivada no segundo ponto $f"(x_i)$.

Integrando a última equação acima duas vezes, obtemos uma expressão para $f_i(x)$. Entretanto, essa expressão irá conter duas constantes de integração indeterminadas. Tais constantes podem ser determinadas invocando-se a condição de igualdade das funções. Isto é, $f_i(x) = f(x_{i-1})$ em x_{i-1} e $f_i(x)$ deve ser igual a $f_i(x_i)$ em x_i . A partir desses cálculos, obteremos a seguinte equação cúbica

$$f_i(x) = \frac{f_i"(x_{i-1})}{6(x_i-x_{i-1})}(x_i-x)^3 + \frac{f_i"(x_i)}{6(x_i-x_{i-1})}(x-x_{i-1})^3 + \left[\frac{f(x_{i-1})}{x_i-x_{i-1}} - \frac{f"(x_{i-1})(x_i-x_{i-1})}{6}\right](x_i-x) + \left[\frac{f(x_i)}{x_i-x_{i-1}} - \frac{f"(x_i)(x_i-x_{i-1})}{6}\right](x-x_{i-1})$$

Observe que esta equação possui apenas dois coeficientes indeterminados: as segundas derivadas $f"(x_{i-1})$ e $f"(x_i)$. Estas segundas derivadas podem ser calculadas usando-se a condição de que as primeiras derivadas nos pontos amostrados são contínuas. Isto é:

$$f_i'(x_i) = f_{i+1}'(x_i)$$

Derivando a equação de $f_i(x)$, encontramos uma expressão para a primeira derivada. Se isso for feito para os $i-esimos$ e $(i-1)-esimos$ intervalos e se os dois resultados forem igualados de acordo com a essa função, obtemos a seguinte relação:

$$(x_i-x_{i-1})f"(x_{i-1}) + 2(x_{i+1}-x_{i-1})f"(x_i) + (x_{i+1}-x_i)f"(x_{i+1}) = \frac{6}{x_{i+1}-x_i}[f(x_{i+1})-f(x_i)] + \frac{6}{x_i-x_{i-1}}[f(x_{i-1})-f(x_i)]$$

Se esta última equação for escrita para todos os pontos interiores amostrados, temos $(n-1)$ equações simultâneas com $n+1$ segundas derivadas. Contudo, como esse é um spline cúbico, as segundas derivadas nos extremos são nulas e o problema se reduz a $(n-1)$ equações com $(n-1)$ incógnitas. Além disso, se observamos a relação entre as variáveis, temos apenas a relação entre $(i-1)$, i e $(i+1)$, o que caracteriza um sistema tridiagonal. Com isso, temos uma implementação extremamente simples e rápida de se resolver.

2. Implementação

```
def spline(x, x_j, f_j, n=3):
    """
    Return x evaluated in Spline interpolation function.

    spline(x, x_j, f_j, n=3)

    INPUT:
      * x: float, evalue at this point
      * x_j: LIST, tabular points
      * f_j: LIST, tabular points (must be same size of x_j)
      * n: spline degree (1=linear,2=quadratic,3=cubic(default),4= quartic)

    return:
      * f(x): interpolated and evaluated x value

    Author: Pedro Garcia [sawp@sawp.com.br]
    see: http://www.sawp.com.br

    License: Creative Commons
             http://creativecommons.org/licenses/by-nc-nd/2.5/br/

    Feb 2011
    """

    def cubic_spline(xi, fi):
        """ Fit a curve using a cubic spline"""
        n = len(xi) - 1
        h = []
        g = []
```

```
    d = []
    b = []
    c = []
    # diferences between points
    for i in xrange(n):
        hi = xi[i + 1] - xi[i]
        gi = fi[i + 1] - fi[i]
        h.append(hi)
        g.append(gi)
    # generate 3-diagonal matrix
    for i in xrange(n-1):
        di = 2 * (h[i + 1] + h[i])
        bi = 6 * (g[i + 1] / h[i + 1] - g[i] / h[i])
        ci = h[i + 1]
        d.append(di)
        b.append(bi)
        c.append(ci)
    # solve the tridiagonal system
    g = tridiagonal_solve(d, c, c, b)
    # solution
    pol = [0.0 for i in xrange(n)]
    pol[n-1] = 0.0
    for i in xrange(1, n):
        pol[i] = g[i-1]
    return pol

if type(x_j) != type(f_j) or type(x_j) != type([]):
    print "Error: wrong type parameters"
    return float("NaN")

if len(x_j) != len(f_j):
    print "Error: the tabular points must have same size"
    return float("NaN")

if n == 1:
    # TODO
    pass
elif n == 2:
    # TODO
    pass
elif n == 4:
    # TODO
    pass
else:
    polinomial = cubic_spline(x_j, f_j)

# this is just to assure convergence
m = 100
for i in xrange(m):
    k = 1
    dx = x - x_j[0]
    while dx >= 0:
        k = k + 1
        dx = x - x_j[k]
    k = k - 1
    # encontra um valor para a funcao f(x)
    dx = x_j[k + 1] - x_j[k]
    alpha = polinomial[k + 1] / (6 * dx)
    beta = - polinomial[k] / (6 * dx)
    gamma = f_j[k + 1] / dx - dx * polinomial[k + 1] / 6
    ro = dx * polinomial[k] / 6 - f_j[k] / dx
    f = alpha * (x - x_j[k]) ** 3 + \
        beta * (x - x_j[k + 1]) ** 3 + \
        gamma * (x - x_j[k]) + \
```

```
            ro * (x - x_j[k + 1])
    return f
```

Esta função está disponível em http://www.sawp.com.br/code/interpolation/splines.py (http://www.sawp.com.br/code/interpolation/splines.py) assim como um exemplo de como utilizá-la.

3. Copyright

References

[1] Anthony Ralston and Philip Rabinowitz, *A First Course in Numerical Analysis* (2nd ed.), McGraw-Hill and Dover, (2001).
[2] N.B Franco, *Cálculo Numérico*, Pearson Prentice Hall (2006).