Cosine interpolation
Sine interpolation
Interpolation of arbitrary series with multiplicative coefficients
Scilab codes

Andrej Liptaj*

Abstract

This text summarizes methods for (pure) cosine and sine interpolations and reminds the reader of the matrix-inversion method valid for any interpolation using series with multiplicative coefficients.

Introductory note: This text was previously published on Scribd[1].

1 Introduction

In my previous text [1] I gave detailed instructions concerning interpolation of data by Fourier series (with choice among different possible cutoffs in case of even number of points). It seemed to me a good idea to complete that text with interpolation based on pure cosine and sine series. Meanwhile I realized that interpolation for any series with multiplicative coefficients is trivial - probably no surprise for an educated reader. Still, for pedagogical reasons, I give a description of the latter in a dedicated section and provide Scilab code for all procedures.

2 Cosine and Sine interpolation

Both are based on a simple trick with subsequent use of the general procedure for Fourier interpolation as described in [1].

2.1 Cosine

We start with a data set

$$S_1 = (x_i, y_i).$$

One can define a mirror-symmetric (with respect to the y-axis) data set

$$S_2 = (-x_i, y_i)$$

and their union

$$S = S_2 \cup S_1$$

and use the general Fourier interpolation method. Because the set S is symmetric, even functions (i.e. cosines) will be automatically filtered out from the Fourier series whereas all sine coefficients will vanish. The only cumbersomeness which remains because of the even number of points is, that, depending on the high-frequency cutoff, one cosine too much could be present. Indeed, if the number of points is N, we are supposed to have one constant term $\frac{a_0}{2}$ and $N-1$ cosine terms. By adding the symmetric data set we get $2N$ points, which can be described by one $\frac{a_0}{2}$ term, N cosine and N sine terms (these ones vanishing) with, however, cutoff freedom. Carrying out the cutoff properly we can use this freedom to remove the high-frequency (N-th) cosine. The general procedure for doing it is described in Section 4 of [1]. I refer the reader to that text for what follows. In case of the high-cosine cutoff one has

$$\begin{aligned} c_{-N} &= \frac{1}{2} i b_n, \\ c_N &= -\frac{1}{2} i b_n, \end{aligned}$$

*Institute of Physics, Bratislava, Slovak Academy of Sciences, andrej.liptaj@savba.sk

I am willing to publish any of my ideas presented through free-publishing services in a journal, if someone (an editor) judges them interesting enough. Journals in the "*Current Contents*" database are strongly preferred.

[1] *https://www.scribd.com/document/289673276/Cosine-and-sine-interpolation-Interpolation-for-arbitrary-series-with-multiplicative-coefficients*

leading to

$$K = -\frac{1}{2} i b_n$$

and

$$p_0 + qK = \frac{1}{2} i b_n.$$

This results to

$$K = -\frac{p_0}{1+q}.$$

With this constant we construct the interpolation polynomial (as described in [1]) and read-out the interpolation coefficients. An alternative way of interpreting this procedure is as follows: we add to the symmetrized data set a new point at $x = 0$, thus keeping the symmetry and preventing any sine function from appearing, and we adjust its vertical position (y-coordinate) so as to make vanish the coefficient in front of the highest-frequency cosine.

The Scilab code for the cosine interpolation follows in Appendix.

2.2 Sine

The sine interpolation is straightforward, cumbersomeness-free. We produce an anti-symmetric data set (symmetry w.r.t. the origin)

$$S_2 = (-x_i, -y_i),$$

and make the union

$$S = S_2 \cup (0,0) \cup S_1,$$

since each sine interpolation needs to go through the origin. Adding this point (the origin) we get an odd number of data points which leads to unambiguous Fourier interpolation with vanishing cosine terms. The Scilab code is in Appendix.

3 Interpolation of arbitrary series with multiplicative coefficients

Focusing on the "genuine" trigonometric interpolation[2] I overlooked the basic fact, that any expression of the form

$$F_N(x) = \sum_{n=1}^{N} a_n f_n(x)$$

can be interpolated through data points using linear algebra. This holds of course on some general conditions, assuming the functions f_n are not behaving bad (e.g. not all off them vanishing at some x_i with nonzero y_i, not being linear dependent, etc.). If one has N experimental points and N functions then the interpolation is usually unique and therefore the method described in this section is equivalent to any other interpolation method. It is of course as "genuine" as any special recipe which is based on the properties of the functions f_n. Actually, in some sense it is even more general: if we apply it for example on the Fourier-type series, it automatically allows for an arbitrary cutoff (high-frequency sine or cosine, symmetric cutoff, see [1]) or even for cutting terms inside the series. The only true question when comparing to other approaches is the computing time (computational complexity).

In this general approach we can forget about details of the functions f_n, the only information needed are their values at x_i, let me note them $f_{i,n}$. These numbers naturally build a matrix, which, when multiplied by the coefficient vector a_i, needs to give the y_i values,

$$\sum_n f_{i,n} a_n = y_i,$$

or written with full explicitness

$$\begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_{N-1}(x_1) & f_N(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_{N-1}(x_2) & f_N(x_2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ f_1(x_{N-1}) & f_2(x_{N-1}) & \dots & f_{N-1}(x_{N-1}) & f_N(x_{N-1}) \\ f_1(x_N) & f_2(x_N) & \dots & f_{N-1}(x_N) & f_N(x_N) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{N-1} \\ a_N \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix}.$$

[2] Finding several text with this topic on internet, I was thinking only in the given direction, forgetting other possibilities. It is maybe pity that the "matrix inversion" approach, as a natural way of finding the interpolation, is not mentioned on (most) Web resources related to the trigonometric interpolation.

One then needs to invert the matrix $(f)_{i,n}$ and compute the coefficients a_n

$$a_i = \sum_{n=1}^{N} \left(f^{-1}\right)_{i,n} y_n.$$

The summary is

- Choose a sequence of your basic functions f_n with all specificity you are asking for. For example, if you wish, in case of the Fourier-series and even number of data points, to cutoff the high-frequency cosine, your sequence should end with a high-frequency sine term: $f_1 = \frac{1}{2}$, $f_2 = \cos(x)$, $f_3 = \sin(x)$, $f_4 = \cos(2x)$, $f_5 = \sin(2x)$, ... , $f_{N-2} = \cos\left(\frac{N-2}{2}x\right)$, $f_{N-1} = \sin\left(\frac{N-2}{2}x\right)$, $f_N = \sin\left(\frac{N-1}{2}x\right)$.
- Build the matrix $(f)_{i,n}$ by evaluating each basic function at each x_i.
- Invert the matrix $(f)_{i,n}$, getting $\left(f^{-1}\right)_{i,n}$.
- Multiply the inverted matrix by the vector of values y_i, getting thus a vector with the a_n coefficients as elements.

The Scilab code is in Appendix.

4 Closing remarks

- For what concerns the interpolation based on trigonometric functions, the "symmetrization" procedure can lead to two separate group of points (S_1 and S_2) lying far from each other. This is, of course, only a cosmetic feature without any impact on the interpolation procedure. To concentrate the data (in any case, not only when "symmetrizing") one is completely free to move their x-coordinates by any (entire) multiple of 2π, and put them into an interval of 2π length.
- Numerical calculations I have performed seem to suggest that the Fourier series interpolation gets very quickly numerically unstable with increasing number of data points. It seems to be an intrinsic issue, independent on the method. The problem starts at such small number of points as 10. I see two possible go-arounds:
  - Use an arbitrary-precision computation software. Doing this, I was able to interpolate approximately 100 data points (on a standard computer). I failed at 300 points: the required numerical precision is so high that the computational time becomes very long.
  - Use an approximate regularization procedure. For example, in the last piece of code in Appendix (matrix inversion) the Scilab software contains an implementation of the pseudo-inverse of a matrix with, when desired, user given precision (the commented line "iM = pinv(M)").

Appendix

Scilab code for cosine interpolation

```
// INTERPOLATING POLYNOMIAL
function [f] = interpolation_poly(x,y)
    nData = length(x)

    atom(1) = poly(1,"x","coeff")

    for i=2:nData
        atom(i) = atom(i-1)*poly(x(i-1),"x")
    end

    poly_interpol = poly(0,"x","coeff")
    for i=1:nData
        const = ( y(i) - horner(poly_interpol,x(i)) )/horner(atom(i),x(i))
        poly_interpol = poly_interpol + const*atom(i)
    end
```

```
    f = poly_interpol
endfunction

// COMPUTING COEFFICIENTS
function [a,N] = trigo_cfs(X,Y)
    dim=1;

    Z = flipdim(X,dim)
    Z = -1.0.*Z
    x = cat(dim,Z,X)

    W = flipdim(Y,dim)
    y = cat(dim,W,Y)

    nDat = length(x)

    N = nDat/2

    for i=1:nDat
        z(i)= exp( %i*x(i))
        Y(i) = (z(i))^N*y(i)
    end

    z_polynomial = interpolation_poly(z,Y)

    // *** Cut high-frequency COSINE ****
    p0 = coeff(z_polynomial,0)
    q = 1

    for i=1:nDat
        q = q*(-z(i))
    end

    K = -p0/(1+q)
    polyToAdd = K*poly(z,"x")
    z_polynomial = z_polynomial + polyToAdd
    // *** END ***

    cfs = coeff(z_polynomial)

    for i=-N:N
        if i<0 then
            continue
        end
        coef = cfs(i+N+1)
        a(i+1)=2*real(coef)
    end
endfunction

// COMPUTING TRIGONOMETRIC INTERPOLATION
function [f] = interpolation_trigo(x,a,N)
    f = a(1)/2
    for i=1:N
        f = f + a(i+1)*cos(i*x)
    end
endfunction

// START OF THE PROGRAM FLOW
```

```
dataSet = read("twoColumnDataFile.dat",-1,2)
nDat = length(dataSet)/2

x = dataSet(:,1)
y = dataSet(:,2)

[a_cfs,N] = trigo_cfs(x,y)
disp(N)
disp(a_cfs(N+1)) // should be zero

xmin = min(x)
xmax = max(x)
dx = (xmax-xmin)/10
xmin = xmin-dx
xmax = xmax+dx

x_ax = [xmin:0.01:xmax]
y_ax = interpolation_trigo(x_ax,a_cfs,N)

ymin = min(y)
ymin = min(ymin,0)
ymax = max(y)
ymax = max(ymax,0)
dy = (ymax-ymin)/10;
ymin = ymin-dy
ymax = ymax+dy

plot2d(x_ax,y_ax,rect=[xmin,ymin,xmax,ymax])
plot(x,y,'*')
```

Scilab code fore sine interpolation

```
// INTERPOLATING POLYNOMIAL
function [f] = interpolation_poly(x,y)
    nData = length(x)

    atom(1) = poly(1,"x","coeff")

    for i=2:nData
        atom(i) = atom(i-1)*poly(x(i-1),"x")
    end

    poly_interpol = poly(0,"x","coeff")
    for i=1:nData
        const = ( y(i) - horner(poly_interpol,x(i)) )/horner(atom(i),x(i))
        poly_interpol = poly_interpol + const*atom(i)
    end

    f = poly_interpol
endfunction

// COMPUTING COEFFICIENTS
function [b,N] = trigo_cfs(X,Y)
    dim=1;

    Z = flipdim(X,dim)
    Z = -1.0.*Z
```

```
    x = cat(dim,Z,0,X)

    W = flipdim(Y,dim)
    W = -1.0.*W
    y = cat(dim,W,0,Y)

    nDat = length(x)

    N = (nDat-1)/2

    for i=1:nDat
        z(i)= exp( %i*x(i))
        Y(i) = (z(i))^N*y(i)
    end

    z_polynomial = interpolation_poly(z,Y)

    cfs = coeff(z_polynomial)

    for i=-N:N
        if i<0 then
            continue
        end
        coef = cfs(i+N+1)
        b(i+1)=-2*imag(coef)
    end
endfunction

// COMPUTING TRIGONOMETRIC INTERPOLATION
function [f] = interpolation_trigo(x,b,N)
    f = 0
    for i=1:N
        f = f + b(i+1)*sin(i*x)
    end
endfunction

// START OF THE PROGRAM FLOW

dataSet = read("twoColumnDataFile.dat",-1,2)
nDat = length(dataSet)/2

x = dataSet(:,1)
y = dataSet(:,2)

[b_cfs,N] = trigo_cfs(x,y)
//disp(N)
//disp(b_cfs)

xmin = min(x)
xmax = max(x)
dx = (xmax-xmin)/10
xmin = xmin-dx
xmax = xmax+dx

x_ax = [xmin:0.01:xmax]
y_ax = interpolation_trigo(x_ax,b_cfs,N)

ymin = min(y)
ymin = min(ymin,0)
```

```
ymax = max(y)
ymax = max(ymax,0)
dy = (ymax-ymin)/10;
ymin = ymin-dy
ymax = ymax+dy

plot2d(x_ax,y_ax,rect=[xmin,ymin,xmax,ymax])
plot(x,y,'*')
```

Scilab code for interpolation of arbitrary series with multiplicative coefficients

The following code, as an example code, preforms the Fourier series interpolation with basic functions being cosines and sines. In case of even number of data points, it goes for high-frequency sine cutoff. For different interpolation type, one needs to implement its own version of the function "*basicFns*".

The code contains a commented line with pseudo-inversion of the matrix ("iM = pinv(M)"). One can use it in case of numerical instabilities.

```
function [f] = basicFns(x,Z)
    n = floor(Z/2)

    if Z==1 then
        f = 1.0
    elseif modulo(Z,2)==0 then
        f = cos(n*x)
    else
        f = sin(n*x)
    end
endfunction

function [M] = buildMatrix(x,y)
    L = length(x)
    for i=1:L
        for j=1:L
            M(i,j)=basicFns(x(i),j)
        end
    end
endfunction

function [cfs]=getCfs(x,y)
    M = buildMatrix(x,y)
    iM = inv(M)
    //iM = pinv(M)
    cfs = iM*y
endfunction

function [f]=genInterpolation(x,cfs)
    L = length(cfs)
    f = 0
    for i=1:L
        f = f + cfs(i)*basicFns(x,i)
    end
endfunction


// START OF THE PROGRAM FLOW

dataSet = read("twoColumnDataFile.dat",-1,2)
nDat = length(dataSet)/2
```

```
x = dataSet(:,1)
y = dataSet(:,2)
cfs = getCfs(x,y)

xmin = min(x)
xmax = max(x)
dx = (xmax-xmin)/10
xmin = xmin-dx
xmax = xmax+dx

x_ax = [xmin:0.01:xmax]
y_ax = genInterpolation(x_ax, cfs)

ymin = min(y)
ymin = min(ymin,0)
ymax = max(y)
ymax = max(ymax,0)
dy = (ymax-ymin)/10;
ymin = ymin-dy
ymax = ymax+dy

plot2d(x_ax,y_ax, rect=[xmin,ymin,xmax,ymax])
plot(x,y,'*')
```

References

[1] A. Liptaj, "Short notice on (exact) trigonometric interpolation",
https://www.scribd.com/doc/270904435
http://vixra.org/abs/1704.0048