

# **Embedding and Using Sophisticated Mathematics in Firebird: Fast Fourier Transforms, Data Smoothing and High Order Derivatives**

Andrew Morgan

Analytical Logic cc

Johannesburg, South Africa

October 2005

[Andrew.Morgan@telkomsa.net](mailto:Andrew.Morgan@telkomsa.net)

## **Abstract**

The Firebird database engine ships very few internal functions. This paper describes several User Defined Functions (UDF) for performing both simple and sophisticated mathematics.

Simple mathematics are defined as those with scalar input/output without any internal state or memory requirement. An example would be Cosine.

Sophisticated mathematical functions by contrast are those operating on arrays of data and requiring internal state information and possibly internal storage which must be managed carefully.

The paper shows an implementation of the Fast Fourier Transform (FFT) and their manipulation from within stored procedures. Built on the FFT are the data smoothing and derivative engines which operate on the supplied data in the frequency domain. The procedures usually return their result sets to the end user through result sets of selectable stored procedures.

# Table of Contents

1	Introduction.....	3
2	Fourier Implementation.....	4
2.1	Exponent Class.....	4
2.2	Fourier Class.....	5
2.3	Container Class.....	6
2.4	Main.....	7
2.5	Declaring the Functions.....	9
2.6	Using Fourier.....	10
3	Smooth Implementation.....	12
3.1	smSMOOTH.....	12
3.2	smGET.....	12
3.3	Using Smooth.....	12
	Appendix A FastMath Module.....	14
	Appendix B String Module.....	16
	Appendix C Fourier Module.....	17
	Appendix D Smooth Module.....	18

# Table of Illustrations

# 1 Introduction

All the mathematics presented in this paper are implemented as user defined functions (UDF) and stored procedures. Unlike simple functions (such as `double COS(double)`) which receive and return a double, advanced functions may operate on arrays of data, and may require internal storage and state information.

There is presently no way for a stored procedure to instantiate a C++ class and manipulate it, so how does one achieve this? If the class cannot be instantiated from a stored procedure, where is it instantiated, how many, and how are they managed?

It should be clear that some form of instantiation needs to happen prior to the stored procedure executing. In order to support multiple instances of a class, a managed array would be useful, and so the first part of the solution is at hand.

A container class must exist prior to the execution of the stored procedure. The container obviously then has a higher scope than the stored procedure, and so disposal will require careful consideration.

The problem remaining is how the stored procedures communicate with the container. By carefully designing the container interface the essential methods can be exposed as simple functions!

## 2 Fourier Implementation

### 2.1 Exponent Class

The header below shows the design of the TExponent template class. The class performs the task of the universal complex exponent lookup.

```
//=====
// TExponent.
//=====
template<typename Type>
    class TExponent
{
public:

    // Constructors.
    TExponent(int Stages = 3);

    // Finalisers.
    void Clear();

    // Destructors.
    ~TExponent();

    // Read Interface.
    int Points();
    int Stages();
    TComplex<Type>* M();
    TComplex<Type>* W();

    // Write Interface.
    void Resize(int Stages);

private:

    // variables.
    int points,stages;
    TComplex<Type> *m,*w;
};
//-----
// Pre-defined Types.
//-----
typedef TExponent<int> TIExponent;
typedef TExponent<float> TFEExponent;
typedef TExponent<double> TDEExponent;
//-----
```

## 2.2 Fourier Class

The header below shows the design of the TFourier template class.

```
//-----
enum EFourier { foAnalysis, foSynthesis };
//=====
// TFourier.
//=====
template<typename Type>
    class TFourier
{
public:

    // Constructors.
    TFourier(int Points = 0, TExponent<Type> *Look = NULL);
    TFourier(int Points, Type *Signal, TExponent<Type> *Look = NULL);
    TFourier(int Points, TComplex<Type> *Signal, TExponent<Type> *Look =
NULL);
    TFourier(TFourier &Right);
    TFourier(TFourier *Right);

    // Destructors.
    ~TFourier();

    // Operators.
    inline TFourier& operator=(TFourier &Right);
    inline TComplex<Type>& operator[](int Point);

    // Initialisers.
    void Init(int Points, TExponent<Type> *Look = NULL);
    void Init(int Points, Type *Signal, TExponent<Type> *Look = NULL);
    void Init(int Points, TComplex<Type> *Signal, TExponent<Type> *Look =
NULL);

    // Finalisers.
    void Clear();

    // Interface.
    int Points();

    // Transforms.
    void DFT(EFourier Direction);
    void FFT(EFourier Direction);

    // Filters.
    bool LPF(int Retain);
    bool LPF(double Retain);
    bool HPF(int Discard);
    bool BPF(int Retain, int Discard);
    bool NPF(int Retain, int Discard);
```

```

        // Manipulation.
        void Mask(int Points, Type *Mask);
        void Diff(Type Interval, int Retain = 0);

private:

        // Variables.
        Type theta;
        int points, stages;
        TComplex<Type> *data;
        TExponent<Type> *lookup;

        // Internal Methods.
        bool Allocate(int Points, TExponent<Type> *Look);
        void FFT(TComplex<Type> *W);
        void DFT(Type Theta);
};
//-----
// Pre-defined Types.
//-----
typedef TFourier<int> TIFourier;
typedef TFourier<float> TFFourier;
typedef TFourier<double> TDFourier;
//-----

```

## 2.3 Container Class

The container class must manage an array of classes. Any of the standard container classes will probably work just as well.

```

//-----
template<class Type>
    class TClassArray
{
public:

        // Constructors.
        TClassArray(int Grow = 8);
        TClassArray(TClassArray &Right);

        // Destructors.
        ~TClassArray();

        // Operators.
        inline Type& operator[](int Index);
        inline TClassArray& operator=(TClassArray &Right);

        // Initialisers.
        void Init(int Grow);

```

```

        // Finalisers.
        void Clear();

        // Read Interface.
        int Size();
        int Used();
        int Grow();
        const Type** Data();

        // Write Interface.
        bool Resize(int value);
        void Grow(int value);
        int Insert(Type *Item);
        void Sort(bool Sort = true);
        bool Delete(int Index);

private:

        // Variables.
        Type **data;
        int size,used,grow,*sort;

        // Internal Methods.
        void QuickSort(int Left,int Right);
};
//-----

```

## 2.4 Main

The task now is to build a DLL with functions that expose the interface of the Fourier class via the class container. There are crucially important things to notice in the code below:

***the fourier container and lookup class are instantiated as the DLL loads since they are global variables.***

The UDF's act as brokers between the stored procedures and the container. Each UDF will be briefly covered.

```

//-----
#include <windows.h>
#pragma hdrstop
//-----
#include "TArray.h"
#include "TFourier.h"
//-----
TClassArray<TDFourier> fourier;
TDExponent lookup;
//-----
#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,unsigned long reason,void* lpReserved)
{
    return(1);
}

```

```
}  
//-----  
extern "C" int __export foNEW(int *Points)  
{  
    return(fourier.Insert(new TDFourier(*Points,&lookup)));  
}
```

This creates a new double precision Fourier object of the given size (\*Points), using the universal lookup (&lookup). The new object is then inserted into the fourier container, and the index of the object in the container is returned. This is vitally important because all subsequent calls to the UDF's must specify the index so that the container knows which object is being manipulated.

```
//-----  
extern "C" int __export foSIZE(int *Index)  
{  
    return(fourier[*Index].Points());  
}
```

This returns the size of the specific (\*Index) object. Note that this is a power-of-2 implementation, and so the size may differ from the number of points used to create it.

```
//-----  
extern "C" int __export foDEL(int *Index)  
{  
    fourier.Delete(*Index);  
    return(0);  
}
```

This function deletes the specific (\*Index) object from the container. Although not necessary, the int return type is retained for future use.

```
//-----  
extern "C" int __export foSET(int *Index,int *Point,double *Re,double *Im)  
{  
    fourier[*Index][*Point].Init(*Re,*Im);  
    return(0);  
}
```

Since internally the fourier object is an array, and I currently do not know how to pass an array from a stored procedure, a specific (\*Point) point of a specific (\*Index) fourier object is set to (\*Re,\*Im). Although not necessary, the int return type is retained for future use.

```
//-----  
extern "C" double __export foRE(int *Index,int *Point)  
{  
    return(fourier[*Index][*Point].Re());  
}
```

Since we are limited by the return types, and Firebird does not support a complex return type, the data must be retrieved as primitives. The function returns the real part of a specific (\*Point) point of a specific (\*Index) fourier object.

```
//-----  
extern "C" double __export foIM(int *Index,int *Point)  
{
```



```
return(fourier[*Index][*Point].Im());
}
```

Similarly, this function returns the imaginary part of a specific (\*Point) point of a specific (\*Index) fourier object.

```
//-----
extern "C" int __export foANALYSIS(int *Index)
{
fourier[*Index].FFT(foAnalysis);
return(0);
}
```

This function performs a Fourier analysis of a specific (\*Index) object's internal data. Although not necessary, the int return type is retained for future use.

```
//-----
extern "C" int __export foSYNTHESIS(int *Index)
{
fourier[*Index].FFT(foSynthesis);
return(0);
}
//-----
```

This function performs a Fourier synthesis (inverse analysis) of a specific (\*Index) object's internal data. Although not necessary, the int return type is retained for future use.

## 2.5 Declaring the Functions

```
DECLARE    EXTERNAL FUNCTION FONEW INTEGER
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foNEW' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FOSIZE INTEGER
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foSIZE' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FODEL INTEGER
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foDEL' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FOSET INTEGER,INTEGER,DOUBLE PRECISION,DOUBLE
PRECISION
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foSET' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FORE INTEGER,INTEGER
RETURNS    DOUBLE PRECISION BY VALUE ENTRY_POINT 'foRE' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FOIM INTEGER,INTEGER
RETURNS    DOUBLE PRECISION BY VALUE ENTRY_POINT 'foIM' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FOANALYSIS INTEGER
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foANALYSIS' MODULE_NAME 'Fourier';

DECLARE    EXTERNAL FUNCTION FOSYNTHESIS INTEGER
RETURNS    INTEGER BY VALUE ENTRY_POINT 'foSYNTHESIS' MODULE_NAME 'Fourier';
```

## 2.6 Using Fourier

So far, so good. The next step is to create a stored procedure with which to test the module.

```

ALTER      PROCEDURE PCostest(Points INTEGER)
RETURNS    (Re DOUBLE PRECISION,Im DOUBLE PRECISION) AS
DECLARE    VARIABLE i INTEGER;
DECLARE    VARIABLE j INTEGER;
DECLARE    VARIABLE k INTEGER;
DECLARE    VARIABLE theta DOUBLE PRECISION;
BEGIN
--          Sampling angle.
theta      = PI(2.0)/Points;

--          Create Fourier object. Note that "i" is the container index!
i          = foNew(Points);

--          Populate Fourier object. Note index "i" and point "j"!
j          = 0;
WHILE      (j < Points) DO
    BEGIN
        Re      = COS(j*theta);
        Im      = 0;
        k       = foSet(i,j,Re,Im);
        j       = j + 1;
    END

--          Analysis of "i"th object.
k          = foAnalysis(i);

--          Get actual size of the "i"th object.
Points     = foSize(i);

--          Return data from the "i"th object and "j"th point.
j          = 0;
WHILE      (j < Points) DO
    BEGIN
        Re      = foRe(i,j);
        Im      = foIm(i,j);
        j       = j + 1;
    SUSPEND;
    END

--          Very important! Cleanup "i"th object when done.
k          = foDel(i);
END#

```

This procedure will return the fourier spectrum of a single period cosine sampled by the specified number of points. Note that all indices are zero-based, and that the data for this procedure was not packed (the imaginary portion was zero). To pack the data, halve the number of points to *foNew*, and use both the real and imaginary slots. The theoretical

## 2 Fourier Implementation

spectrum for this procedure for a power-of-2 number of points should be all zero except  $F[1] = F[\text{Points}-1] = \text{Points}/2$ ;

Do not be deceived by the apparent length of this procedure; it will perform an unpacked 132K-point FFT in 750ms (2.2G Mobile AMD-64)!

Note that these procedures use functions supplied by the ***FastMath*** module.

## 3 Smooth Implementation

The Smooth Module manages an array of smooth objects which manipulate internal Fourier objects. The interface is quite similar to Fourier providing the ability to create, set, get, destroy and manipulate smooth objects. Of particular interest are *smSMOOTH* and *smGET*.

### 3.1 smSMOOTH

```
extern "C" int __export smSMOOTH(int *Index,int *Retain,int *Discard,
                                int *Depth,double *Interval)
```

This function performs the actual smoothing of the data:

Index is the object within the container to smooth

Retain is the number of terms to retain for a low pass filter (excluding DC)

Discard is the number of terms to zero for a high-pass filter (excluding DC)

Depth is the number of derivatives to compute

Interval is the sampling interval (time) of the data (inverse of sampling frequency)

Using Retain and Discard together yields a band pass filter.

After running smSMOOTH, smGET is used to retrieve the results.

### 3.2 smGET

```
extern "C" double __export smGET(int *Index,int *Derivative,int *Point)
```

This function retrieves that data and smoothing and differentiating:

Index is the object within the container to smooth

Derivative is the derivative requires (0 = smoothed data, 1=1<sup>st</sup> derivative etc)

Point is the point within the array

### 3.3 Using Smooth

The next step is to create a stored procedure to manipulate a smooth object:

```
ALTER      PROCEDURE PcheckUDF(Process VARCHAR(64),serID INTEGER,
                                p1 DOUBLE PRECISION,p2 DOUBLE PRECISION,p3 DOUBLE PRECISION)
RETURNS    (x          DOUBLE PRECISION,
            y0         DOUBLE PRECISION,
            y1         DOUBLE PRECISION,
            y2         DOUBLE PRECISION,
            y3         DOUBLE PRECISION,
            y4         DOUBLE PRECISION) AS
DECLARE    VARIABLE i INTEGER;
DECLARE    VARIABLE code INTEGER;
DECLARE    VARIABLE retain INTEGER;
DECLARE    VARIABLE object INTEGER;
DECLARE    VARIABLE points INTEGER;
```

```

DECLARE VARIABLE interval DOUBLE PRECISION;
BEGIN
--      Check that the series exists!
SELECT COUNT(*)
FROM Tinput
WHERE serID = :serID
INTO :points;
IF (points > 1)
THEN BEGIN
--      Return 4 derivatives without filtering (all frequencies).
retain = points/2 - 1;

IF (UPPER(Process) = 'DIFF')
THEN BEGIN
interval = PI(2.0)/points;
object = smNEW(points);
i = 0;

--      Set data to differentiate.
FOR SELECT inpY
FROM Tinput
WHERE serID = :serID
ORDER BY inpX
INTO :y0 DO
BEGIN
code = smSET(object,i,y0);
i = i + 1;
END

code = smSMOOTH(object,retain,0,4,interval);
i = 0;

--      Since x is not in smooth, use cursor.
FOR SELECT inpX
FROM Tinput
WHERE serID = :serID
ORDER BY inpX
INTO :x DO
BEGIN
y0 = smGet(object,0,i);
y1 = smGet(object,1,i);
y2 = smGet(object,2,i);
y3 = smGet(object,3,i);
y4 = smGet(object,4,i);
i = i + 1;
SUSPEND;
END

code = smDEL(object);
END

```

## Appendix A FastMath Module

```
// Special Constant Functions.
extern "C" double __export fmPI(double *factor)
extern "C" double __export fmEULER(double *factor)

// Rounding Functions.
extern "C" double __export fmROUNDM(double *x,double *next)
extern "C" double __export fmROUNDZ(double *x,double *next)
extern "C" double __export fmROUNDUP(double *x,double *next)
extern "C" double __export fmROUNDN(double *x,double *next)
extern "C" double __export fmROUND(double *x,double *next)

// Basic Functions.
extern "C" double __export fmABS(double *x)
extern "C" double __export fmSIGN(double *x)
extern "C" double __export fmSQR(double *x)
extern "C" double __export fmSQRT(double *x)
extern "C" double __export fmGREATER(double *x,double *y)
extern "C" double __export fmLESSER(double *x,double *y)
extern "C" int __export fmREVERSE(int *x,int *size)

// Angle Functions.
extern "C" double __export fmNORMALISE(double *x)
extern "C" double __export fmDEGTORAD(double *x)
extern "C" double __export fmRADTODEG(double *x)
extern "C" double __export fmDEGTOHMS(double *x)
extern "C" double __export fmHMSTODEG(double *x)

// Factorial Functions.
extern "C" double __export fmFACT(long *n)
extern "C" double __export fmPFACT(long *n,long *m)

// Exponential Functions.
extern "C" double __export fmPOWER2(double *x)
extern "C" double __export fmPOWER10(double *x)
extern "C" double __export fmPOWERE(double *x)
extern "C" double __export fmPOWERA(double *x,double *a)

// Logarithmic Functions.
extern "C" double __export fmLOG2(double *x)
extern "C" double __export fmLOG10(double *x)
extern "C" double __export fmLOGE(double *x)
extern "C" double __export fmLOGA(double *x,double *a)

// Trigonometric Functions.
extern "C" double __export fmSIN(double *x)
extern "C" double __export fmCOS(double *x)
extern "C" double __export fmTAN(double *x)
extern "C" double __export fmCSC(double *x)
```

```
extern "C" double __export fmSEC(double *x)
extern "C" double __export fmCOT(double *x)

// Inverse Trigonometric Functions.
extern "C" double __export fmASIN(double *x)
extern "C" double __export fmACOS(double *x)
extern "C" double __export fmATAN(double *y,double *x)
extern "C" double __export fmACSC(double *x)
extern "C" double __export fmASEC(double *x)
extern "C" double __export fmACOT(double *y,double *x)

// Hyperbolic Functions.
extern "C" double __export fmSINH(double *x)
extern "C" double __export fmCOSH(double *x)
extern "C" double __export fmTANH(double *x)
extern "C" double __export fmCSCH(double *x)
extern "C" double __export fmSECH(double *x)
extern "C" double __export fmCOTH(double *x)

// Inverse Hyperbolic Functions.
extern "C" double __export fmASINH(double *x)
extern "C" double __export fmACOSH(double *x)
extern "C" double __export fmATANH(double *x)
extern "C" double __export fmACSCH(double *x)
extern "C" double __export fmASECH(double *x)
extern "C" double __export fmACOTH(double *x)

// Step Functions.
extern "C" double __export fmBELOW(double *x,double *a)
extern "C" double __export fmABOVE(double *x,double *a)
extern "C" double __export fmSTEP(double *x,double *a,double *b)
```

## Appendix B String Module

```
extern "C" int __export stLEN(char *Source)
extern "C" int __export stPOS(char *Target,char *Source)
extern "C" char* __export stSUB(char *Source,int *Start,int *Length)
extern "C" char* __export stDEL(char *Source,int *Start,int *Length)
extern "C" char* __export stINS(char *Source,char *Target,int *Start)
extern "C" char* __export stDUP(char *Source,int *Count)
extern "C" char* __export stTRIM(char *Source)
```



## Appendix C Fourier Module

```
extern "C" int __export foNEW(int *Points)
extern "C" int __export foSIZE(int *Index)
extern "C" int __export foDEL(int *Index)
extern "C" int __export foSET(int *Index,int *Point,double *Re,double *Im)
extern "C" double __export foRE(int *Index,int *Point)
extern "C" double __export foIM(int *Index,int *Point)
extern "C" int __export foANALYSIS(int *Index)
extern "C" int __export foSYNTHESIS(int *Index)
extern "C" int __export foANALYSIS2(int *Index)
extern "C" int __export foSYNTHESIS2(int *Index)
```

## Appendix D Smooth Module

```
extern "C" int __export smNEW(int *Points)
extern "C" int __export smDEL(int *Index)
extern "C" int __export smSIZE(int *Index)
extern "C" int __export smSET(int *Index,int *Point,double *value)
extern "C" double __export smGET(int *Index,int *Derivative,int *Point)
extern "C" int __export smSMOOTH(int *Index,int *Retain,int *Discard,
    int *Depth,double *Interval)
```

## Bibliography

- 1 Andrew Morgan Database Design with Identifying Surrogate Key (2000)