F#: The Design of Math Providers

Tuesday, August 10, 2010
3:39 PM

The structure of F# PowerPack Math Providers.

The interface is defined in lapack_base.fs:

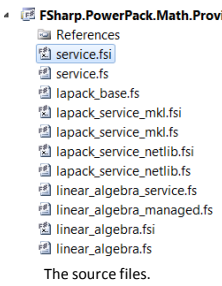
```
type ILapack = interface
    //Matrix-Matrix Multiplication
    abstract dgemm_ : Math.matrix * Math.matrix -> Math.matrix
    //Matrix-Vector Multiplication
    abstract dgemv_ : Math.matrix * Math.vector -> Math.vector
    .. more methods
```

The two actual implementations are done in `lapack_service_mkl.fs` & `lapack_service_netlib.fs` by calling native functions in MKL or Netlib-Lapack math libraries separately.

As the netlib implementation is freely available online. Let's see the design in `lapack_service_netlib.fs`:

```
// part I: the dll imports
module LapackNetlibStubs = begin
  [<System.Runtime.InteropServices.DllImport(@"blas.dll",EntryPoint="dgemm_")>]
  extern void dgemm_(char *transa, char *transb, int *m, int *n, int *k, double
*alpha, double *a, int *lda, double *b, int *ldb, double *beta, double *c, int
*ldc);
  .. More dll imports

// part II: the implementation calling the foreign functions
type LapackNetlibService() = class
 interface ILapack with
///Matrix-Matrix Multiplication
 member this.dgemm_((a:matrix),(b:matrix)) =
  // input copies
  let a = Matrix.copy a
  .. The pattern of each function is
     1) do some variable copying
     2) lock the copied variables into native pointers
     3) call the native function
     4) unlock the pointers
```



The source files.

Side note 1:
The detailed implementation of a function:

```
member this.dgemm_((a:matrix),(b:matrix)) =
  // input copies
  let a = Matrix.copy a
  let b = Matrix.copy b
  // dimensions
  let m = NativeUtilities.matrixDim1 a in
  let k = NativeUtilities.matrixDim2 a in
  NativeUtilities.assertDimensions "dgemm_" ("k","Dim1(b)")
(k,NativeUtilities.matrixDim1 b);
  let n = NativeUtilities.matrixDim2 b in
  // allocate results
  let c = Matrix.zero (m) (n)
  // transpose
  let c = Matrix.transpose c
  // setup actuals
  let mutable arg_transa = 't'
  let mutable arg_transb = 't'
  let mutable arg_m = m
  let mutable arg_n = n
  let mutable arg_k = k
  let mutable arg_alpha = 1.0
  let arg_a = NativeUtilities.pinM a
  let mutable arg_ldk = k
  let arg_b = NativeUtilities.pinM b
  let mutable arg_ldn = n
  let mutable arg_beta = 1.0
  let arg_c = NativeUtilities.pinM c
  let mutable arg_ldm = m
  // call function
  try
    LapackNetlibStubs.dgemm_(&&arg_transa,&&arg_transb,&&arg_m,&&arg_n,
&&arg_k,&&arg_alpha,arg_a.Ptr,&&arg_ldk,arg_b.Ptr,&&arg_ldn,
&&arg_beta,arg_c.Ptr,&&arg_ldm)
  finally
    NativeUtilities.freeM arg_a
    NativeUtilities.freeM arg_b
    NativeUtilities.freeM arg_c
  // INFO
  // fixups
  let c = Matrix.transpose c
  // result tuple
  c
```

```
// part III: the module

module LapackNetlib = begin
                                                                  provider-name     dlls-list                    the implementation
 let NetlibProvider = new Microsoft.FSharp.Math.Experimental.Provider<_>("Netlib",[|"blas.dll";"lapack.dll"|],fun () -> new LapackNetlibService() :> ILapack)
end
```

`linear_algebra_service.fs`

Each linear algebra function calls one of the providers to perform calculations. This file is like the wrapper for the service providers.

```
  let MKLProvider    = LapackMKL.MKLProvider
  let NetlibProvider = LapackNetlib.NetlibProvider
  let LAPACKService = new Service<ILapack>([MKLProvider;NetlibProvider])

  let Service() =
    match LAPACKService.Service() with
    | Some svc -> svc
    | None     -> failwith "LAPACK service either not available, or not started"
```

two exemplar functions:

```
  let SVD a =
    let vs,u,w = Service().dgesvd_ a
    u,vs,w
  /// Given A[n,n] find it's inverse.
  /// This call may fail.
  let inverse a =
    let n,m = matrixDims a
    NativeUtilities.assertDimensions "inverse" ("rows","columns") (n,m)
    let _,_,x  = Service().dgesv_(a,Matrix.identity n)
    x
```

`linear_algebra.fs`:
the actual exposed linear algebra interface to the end user

```
  let Lapack         = LinearAlgebraService.LAPACKService // The service/provider object
```

Notice the variable Lapack here. the type of it is **Service<Ilapack>**.

```
  module Locals =
    let HaveService() = Lapack.Available()
  open Locals
```

a typical linear algebra function is implemented as:

```
  let QR a =
    if HaveService() then LinearAlgebraService.QR a
                     else LinearAlgebraManaged.QR a
```

the LinearAlgebraManaged module contains an incomplete list of linear algebra functions written in F#.

Side note 2:
the service<_> module (service.fs)

As said in the source file, this is a general DLL service module. The Service<'a> object contains a set of providers, each of which is a native DLL function provider.

Service<'a> is defined as:

```
type Service<'a>(providers:Provider<'a> seq) =
    let mutable providers = Seq.toArray providers      // possible providers
configuration state
    let mutable state = ServiceEnabledUninitialised   // service state
```

Thus a service has a set of Provider<_>s:

```
/// Generic provider with unmanaged DLL dependencies.
type Provider<'a>(name:string,requiredDLLs:string[],provide:unit -> 'a) =
    // NOTE: The dependencies could be extended to include architecture.
    member this.Name         = name
    member this.RequiredDLLs = requiredDLLs
    member this.Provide()    = provide()
```

Side note 3:

the linear_algebra_managed.fs contains the managed F# implementation of the common linear algebra functions. Should be noted that not all the functions are implemented yet.

```
module LinearAlgebraManaged =

    let NYI () = failwith "Not yet implemented, managed fallback linear
algebra ops coming soon"

    type Permutation = Permutation of int * (int -> int)

    // some are not implemented
    let SVD A = NYI()
    let EigenSpectrum A = NYI()
    let Condition A = NYI()


   // some are implemented
    let QR (A:matrix) =
        let (n,m) = matrixDims A
        let mutable Q = Matrix.identity n                                  //
Keeps track of the orthogonal matrix.
        let R = Matrix.copy A

        // This method will update the orhogonal transformation fast when
given a reflection vector.
        let UpdateQ (Q:matrix) (v:vector) =
            let n = Vector.length v
            let (nQ,mQ) = matrixDims Q
```

From a user's perspective: how to use Math Providers?

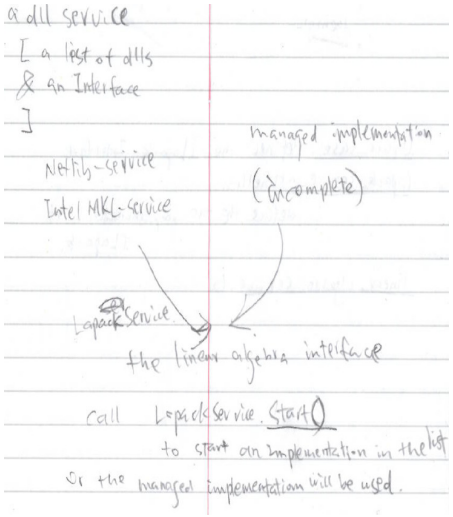
```
let isSucc = Experimental.LinearAlgebra.Lapack.Start()
```

Pasted from <http://fdatamining.blogspot.com/2010/03/matrix-and-linear-algebra-in-f-part-ii.html>

From a user's perspective: how to use Math Providers?

```
let isSucc = Experimental.LinearAlgebra.Lapack.Start()
```

Pasted from <http://fdatamining.blogspot.com/2010/03/matrix-and-linear-algebra-in-f-part-ii.html>




```
        // This method will update the orhogonal transformation fast when
given a reflection vector.
        let UpdateQ (Q:matrix) (v:vector) =
            let n = Vector.length v
            let (nQ,mQ) = matrixDims Q

            // Cache the computation of Q*v.
            let Qv = Vector.init nQ (fun i -> (Q.[i..i,nQ-n..].Row 0) * v)

            // Update the orthogonal transformation.
            for i=0 to nQ-1 do
                for j=nQ-n to nQ-1 do
                    Q.[i,j] <- Q.[i,j] - 2.0 * Qv.[i] * v.[j-nQ+n]
            ()

        // This QR implementation keeps the unreduced part of A in R. It
computes reflectors one at a time
        // and reduces R column by column. In the process it keeps track of
the Q matrix.
        for i=0 to (min n m)-1 do
            let v = HouseholderTransform R i
            UpdateQ Q v
        Q,R
```