Manual de Usuario

Primera Propuesta – Febrero 2009

por

José Antonio Muñoz Gómez
jose.munoz@cucsur.udg.mx

Resumen

En este manual se describen las clases y los métodos o funciones requeridas para poder trabajar con las funciones de base radial generalizadas para datos no equiespaciados en 1D, 2D y 3D. Con base en dichas funciones radiales, se muestra como construir la matriz de interpolación para posteriormente interpolar los datos. Las clases y funciones mostradas son genéricas en el sentido de que podemos trabajar con distintos tipos de datos, así como con las generalizaciones de las funciones radiales.

Este manual forma parte del proyecto "Métodos de Funciones de Base Radial para la Solución de Ecuaciones en Derivadas Parciales" coordinado por el Dr. Pedro González-Casanova. http://www.dci.dgsca.unam.mx/pderbf

Funciones Radiales Generalizadas

En la siguiente tabla mostramos las funcione de base radial generalizadas con las que podemos trabajar. La generalización de las FBR se basa en la selección del parámetro o factor β. Dependiendo de la función radial este valor puede ser un número para, impar o no se requiere.

Nombre	$\phi(r)$	Parámetros	Polinomio
Multicuádrico	$(-1)^{\lceil \beta/2 \rceil}(r^2+c^2)^{\beta/2}$	$\beta > 0, \beta \notin 2\mathbb{N}$	$\lceil \beta \rceil$
Placa Delgada	$(-1)^{1+\beta/2} r^\beta \log r$	$\beta > 0, \beta \in 2\mathbb{N}$	$1+\beta/2$
Potencias	$(-1)^{\lceil \beta/2 \rceil} r^\beta$	$\beta > 0, \beta \notin 2\mathbb{N}$	$\lceil \beta/2 \rceil$
Gausiana	$e^{-r^2 c}$	$c > 0$	0
Inverso Multicuádrico	$(r^2+c^2)^{-\beta/2}$	$\beta > 0, \beta \notin 2\mathbb{N}$	0

En la implementación de dichas funciones por defecto al momento de seleccionar la función radial a utilizar se establece el valor de β mínimo requerido. A partir del factor β, se determina el grado del polinomio requerido en el problema de interpolación. Es conveniente recordar que el polinomio es necesario para poder tener un problema bien planteado.

Cada función radial está implementada en una clase genérica. Los nombres de cada clase se muestran a continuación:

Función Radial	Clase C++
Multicuádrico	MQ
Placa Delgada	TPS
Potencias	POT
Gausiana	GAU
Inverso Multicuádrico	IMQ

Para utilizar una función radial en C++ requerimos instanciarla a un tipo de dato particular T. Por ejemplo, para utilizar el núcleo multicuádrico requerimos hacer:

```
MQ<T>  rbf;
```

donde

T tipo de dato básico, que puede ser float, double o long double.
rbf es el nombre de la variable, podemos utilizar cualquier nombre en la variable.

Una vez creada la variable, podemos acceder a sus métodos de la clase para poder evaluar la función radial en algún punto en 1D, 2D o 3D, así como poder parametrizar la función radial.

En las siguientes funciones se indica a detalle como podemos evaluar las funciones radiales, así como la modificación de los parámetros generalizados. Posteriormente,

damos algunos ejemplos de como utilizar dichas funciones.

int get_min_degree_pol( )

Descripción: obtiene el mínimo grado del polinomio requerido en la interpolación con funciones radiales. Esto se requiere para que el problema de interpolación esté bien planteado, es decir, para que la matriz de interpolación sea invertible. El grado depende de la función radial utilizada.

Parámetros de Salida
ENTERO: el mínimo grado del polinomio requerido por la FBR.

Parámetros de Salida
ninguno

set_degree_pol(m)

Descripción: establecemos el grado del polinomio a utilizar en el problema de interpolación. Marca error sí m < get_min_degree_pol. No es estrictamente necesario establecer este parámetro, por defecto se establece el grado del polinomio en función del núcleo radial utilizado.

Parámetros de entrada
m ENTERO especifica el grado del polinomio a utilizar.

Parámetros de salida
ninguno

Es conveniente aclarar que el grado del polinomio es a lo mas m-1. Por ejemplo, si establecemos m=1, el grado del polinomio requerido es 0 lo cual es un polinomio con una constante. Si establecemos m=1, se requiere de un polinomio lineal.

m ← get_degree_pol( )

Descripción: obtiene el grado del polinomio a utilizar en la función radial. Este valor puede ser cambiado mediante la función set_degree_pol.

Parámetros de Salida
m ENTERO especifica el grado del polinomio a utilizar

Parámetros de Salida
ninguno

beta ← get_beta( )

Descripción: obtiene el factor factor β utilizado en las funciones radiales generalizadas. La función radial Gausiana GAU no utiliza este factor, es requerido en las funciones radiales: MQ, IMQ, TPS y POT.

Parámetros de Entrada

niguno

Parámetros de Salida

beta ENTERO positivo que especifica el factor o exponente utilizado en las funciones radiales generalizadas.

Para todas las FBR se regresa el valor de beta previamente asignado en la función set_beta o bien, el valor que tiene por defecto la función radial.

set_beta(beta)

Descripción: establece el factor β requerido en las funciones radiales generalizadas MQ, IMQ, TPS y POT.

Parámetros de Entrada

m ENTERO positivo que el factor en las funciones radiales generalizadas.

Parámetros de Salida

ninguno

Dependiendo de la función radial el valor de beta puede ser un número par o impar.

RBF	m
MQ	1, 3, 5, 7, …
IMQ	1, 3, 5, 7, …
GAU	no requiere
TPS	2, 4, 6, 8, …
POT	1, 3, 5, 7, …

En el constructor de cada función radial se tiene por defecto un valor establecido, es por ello que no es estrictamente necesario establecer dicho valor. El valor por defecto en cada función es:

RBF	beta
MQ	1
IMQ	1,
GAU	no requiere
TPS	2
POT	1

Esta función es la que nos permite tener funciones radiales generalizadas.

En la función radial Gausiana GAU no se requiere dicho factor. Con el núcleo GAU se puede utilizar esta función y no marca ningún error ni tiene efecto sobre la función radial. Esta función se deja eb GAU para tener una entrada común con el resto de las FBR.

T ← eval(x, xj, c)

Descripción: evalua la función radial $\phi(r)$, donde $r = \sqrt{(x - x_j)^2}$ para datos en una dimensión. La implementación depende de la función radial utilizada. El parámetro *c* solo tiene efecto en las funciones MQ, IMQ y GAU. En las funciones TPS y POT no se requiere dicho parámetro, sin embargo se deja este parámetro para tener una entrada común a todos las evaluaciones de las FBR.

Parámetros de Entrada

x	T	primera componente
xj	T	segunda componente

Parámetros de Salida

T	Regresa la evalución de la función radial. El tipo de dato depende de como instanciamos la función radial. Por lo general lo instanciamos del tipo DOUBLE.

T ← eval(x, y, xj, yj c)

Descripción: evalua la función radial $\phi(r)$, donde $r = \sqrt{(x - x_j)^2 + (y - y_j)^2}$ para datos en dos dimensiones. La implementación depende de la función radial utilizada. El parámetro *c* solo tiene efecto en las funciones MQ, IMQ y GAU. En las funciones TPS y POT no se requiere dicho parámetro, sin embargo se deja este parámetro para tener una entrada común a todos las evaluaciones de las FBR.

Parámetros de Entrada

x	T	primera componente
y	T	segunda componente
xj	T	primera componente
yj	T	segunda componente

Parámetros de Salida

T	Regresa la evalución de la función radial. El tipo de dato depende de como instanciamos la función radial. Por lo general lo instanciamos de tipo DOUBLE.

T ← eval(x, y, z, xj, yj , zj, c)

Descripción: evalua la función radial $\phi(r)$, donde $r = \sqrt{(x - x_j)^2 + (y - y_j)^2 + (z - z_j)^2}$ para datos en tres dimensión. La implementación depende de la función radial utilizada. El parámetro *c* solo tiene efecto en las funciones MQ, IMQ y GAU. En las funciones TPS y POT no se requiere dicho parámetro, sin embargo se deja este parámetro para tener una entrada común a todos las evaluaciones de las FBR.

Parámetros de Entrada

x	T	primera componente
y	T	segunda componente
z	T	tercera componente
xj	T	primera componente
yj	T	segunda componente
zj	T	tercera componente

Parámetros de Salida

T	Regresa la evalución de la función radial. El tipo de dato depende de como instanciamos la función radial. Por lo general lo instanciamos de tipo DOUBLE.

Construcción de la Matriz de Gramm

Una vez seleccionada la función radial a utilizar, podemos construir el sistema lineal de ecuaciones para el problema de interpolación con funciones radiales. La estructura general de la matriz requerida es:

$$\begin{bmatrix} A & P \\ P^T & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} f \\ \bar{0} \end{bmatrix} \qquad (1)$$

esta matriz la podemos construir de manera automática mediante las funciones:

```
fillGramm(rbf, c, A, b, x , f);                datos en 1D
fillGramm(rbf, c, A, b, x , y, f);             datos en 2D
```

Parámetros de Entrada

- rbf — función radial seleccionada
- c — parámetro positivo libre de tipo T requerido en las funciones radiales MQ, IMQ, GAU. En las funciones TPS y POT no se requiere este parámetro, sin embargo se deja este parámetro para tener una entrada común a todos las evaluaciones de las FBR. Para TPS y POT podemos darle cualquier valor al parámetro libre *c* y no afecta en nada.
- x — vector de datos de tipo vector<T> con los nodos a utilizar en la FBR que sirven para construir la matriz del interpolante
- y — vector de datos de tipo vector<T> con los nodos a utilizar en la FBR que sirven para construir la matriz del interpolante
- f — vector de datos de tipo vector<T> con los valores de los datos a interpolar

Parámetros de Salida

- A — matriz de tipo Matrix<T> en la cual se almacena el sistema lineal de ecuaciones; ver ecuación de arriba.
- b — vector de datos de tipo vector<T> en donde se almacena f y si se requiere un vector de ceros adicional; ver ecuación de arriba

Es importante mencionar que en estas funciones no se especifica el polinomio P requerido en la construcción de la matriz. La inclusión del polinomio depende de la función radial seleccionada. Mediante la FBR podemos controlar el grado del polinomio requerido.

Internamente estas funciones utilizan la clase Polinomio, la cual construye de manera automática el polinomio requerido mediante la defición del grado del polinomio mediante la FBR seleccionada.

Las funciones mostradas son genéricas, en el sentido de que podemos seleccionar cualquiera de las funciones radiales generalizadas: MQ, IMQ, GAU, TPS y POT.

Solución del sistema Ax = b

Una vez construido el sistema lineal de ecuaciones (1), el siguiente paso es resolver dicho sistema de ecuaciones. Esto lo realizamos mediante el operador "/" sobrecargado en la clase Matriz<T>. Simplemente, realizamos lo siguiente:

```
lambda = A/b;
```

donde

lambda vector de salida de tipo vector<T> en donde se almacena la solución del sistema lineal de ecuaciones

A,b matriz y vector de entrada determinados en la sección anterior

El esquema numérico empleado es Gauss con pivoteo parcial cuya complejidad computacional es O(N^3) en tiempo. Como trabajo futuro se tiene contemplado la creación de una clase Solver en la cual se puedan tener distintas maneras de resolver el sistema lineal de ecuaciones.

Interpolación de Datos

Una vez obtenido el vector solución *lambda* del sistema lineal de ecuaciones, podemos interpolar los datos. Esto se realiza mediante las funciones:

```
fnew  ← interpola(rbf, c, lambda, x , x_new);            datos en 1D
fnnew ←interpola(rbf, c, lambda, x , y, x_new, y_new);   datos en 2D
```

Parámetros de Entrada

rbf función radial seleccionada

c parámetro positivo libre de tipo T requerido en la construcción de la matriz de Gramm; ver pp 7. En las funciones TPS y POT no se requiere este parámetro, sin embargo se deja este parámetro para tener una entrada común. Para TPS y POT podemos darle cualquier valor al parámetro libre *c* y no afecta en nada.

x vector de datos de tipo vector<T> con los que se construyo el sistema lineal de ecuaciones (1).

y vector de datos de tipo vector<T> con los que se construyo el sistema lineal de ecuaciones (1).

x_new vector de datos de tipo vector<T> correspondiente a los sitios en donde queremos interpolar los datos

y_new vector de datos de tipo vector<T> correspondiente a los sitios en donde queremos interpolar los datos

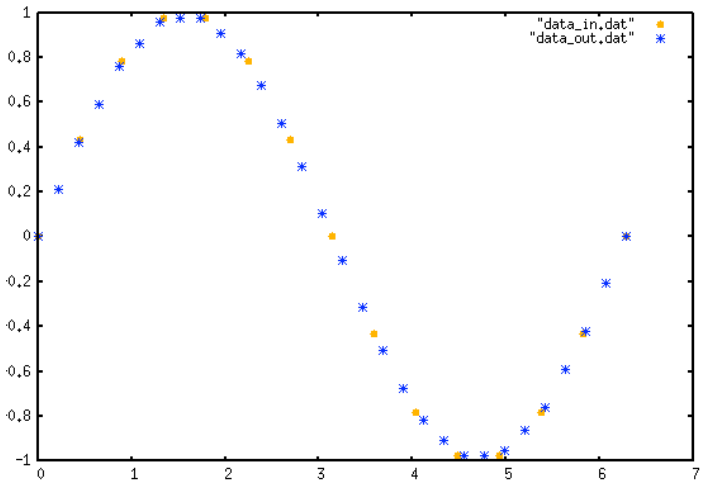
Parámetros de Salida

fnew vector de datos de tipo vector<T> en donde se tienen los valores interpolados

La complejidad computacional en la interpolación es O(Nnew*N) en tiempo, lo cual es prohibitivo cuando queremos interpolar grandes volúmenes de información. El empleo de técnicas cómo multipólos rápidos, hace viable la interpolación de grandes volúmenes de datos.

Ejemplo en 1D

Como primer ejemplo consideremos que tenemos los datos equiespaciados $x_1, x_2, \ldots x_n$ en el intervalo [0, pi]. Sobre dichos nodos tenemos los valores de la función f(x) = sin(x). Lo que queremos es determinar los valores de la f sobre el conjunto de nodos $x_1, \ldots x_{2n}$. Para ello utilizaremos la función radial multicuádrica MQ. En el siguiente fragmento de código se ilustra como resolver el problema planteado. La salida del programa es:



El programa completo se encuentra en la carpeta: examples/1d
Dentro de dicha carpeta, tecleamos ./compila para compilar el programa.

```
//-------------------------------------------------------------
int main(void)
{
 MQ<double>    rbf;  // RBF as kernel we can use : MQ, IMQ, GAU, TPS,
                     // POT
 double           c;
 Matrix<double>   A;
 vector<double>   x,f,b,lambda;
 vector<double>   x_new,f_new;
 int   N;

//defino el numero de nodos
     N = 15;

//Defino el parametro c para mq, imq y gau
     c = 1;

//Opcional: defino el grado del polinomio
     rbf.set_degree_pol(3);

//Creo el vector de datos
     make_data(N,x,f);

//Creo la matriz de Gramm
    fillGramm(rbf,c, A, b, x , f);

//Resuelvo via Gauss con pivoteo parcial
```

```
     lambda = A/b;

//creo el grid en dondo quiero interpolar
     make_data(2*N, x_new, f_new);

//interpolo los datos
     f_new = interpola(rbf,c,lambda,x,x_new);

//salvo los datos interpolados a archivo ASCII
     save_gnu_data("data_in.dat",x,f);
     save_gnu_data("data_out.dat",x_new,f_new);


#ifdef WITH_GNUPLOT
    int pausa;
    GNUplot plotter;
    plotter("reset");
     plotter("set pointsize 2");
    plotter("plot \"data_in.dat\" with point 7, \"data_out.dat\" with point
3");
    std::cout << "\n\n >---> Press any key and then <enter> to finish " ;
     std::cin >> pausa;
#endif

 return 0;
}
```

Si deseamos utilizar otra función de base radial, simplemente modificamos la línea de código:

```
MQ<double>    rbf;
```

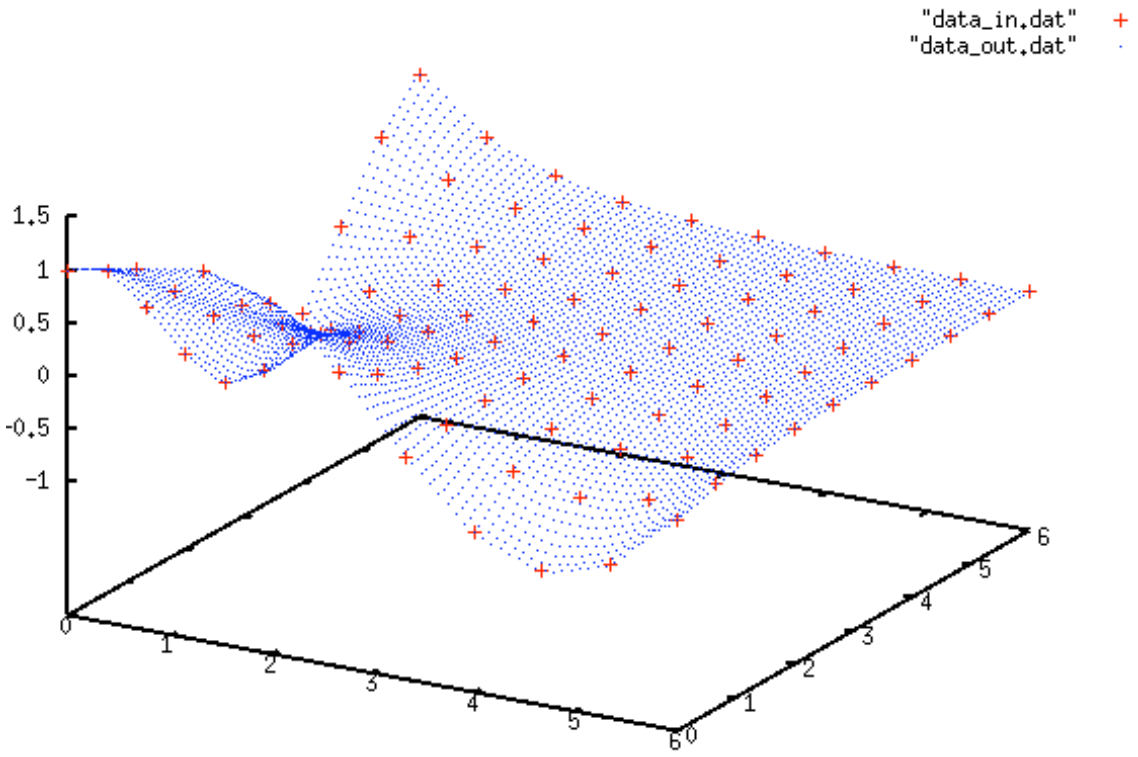
Por ejemplo, si quisieramos utilizar el kernel de placa delgada, tendriamos que escribir:

```
TPS<double>    rbf;
```

y dejamos el resto del programa igual.

Ejemplos en 2D

En este ejemplo utilizaremos la función radial de placa delgada TPS para interpolar datos en dos dimensiones. El programa se encuentra en examples/2d/eje_2d.cpp y la salida de dicho programa se muestra en la figura de abajo. Los puntos en rojo son los datos con los que construimos el interpolante, los nodos en azul corresponden a los datos interpolados.



El fragmento principal del código es:

```
//-------------------------------------------------------
int main(void)
{
 TPS<double>    rbf; // RBF as kernel we can use : MQ, IMQ, GAU, TPS,
                     // POT
 double           c;
 Matrix<double>   A;
 vector<double>   x,y,f,b,lambda;
 vector<double>   x_new,y_new,f_new;
 int   N;

//defino el numero de nodos
     N = 10;

//Defino el parametro c para mq, imq, gau
     c = 0.1;

//Creo el vector de datos
```

```
    make_data(N,x,y,f);

//Creo la matriz de Gramm
    fillGramm(rbf,c, A, b, x , y, f);

//Resuelvo via Gauss con pivoteo parcial
    lambda = A/b;

//creo el grid en dondo quiero interpolar
    make_data(7*N, x_new,y_new, f_new);

//interpolo los datos
    f_new = interpola(rbf,c,lambda,x,y,x_new,y_new);

//salvo los datos interpolados a archivo ASCII
    save_gnu_data("data_in.dat",x,y,f);
    save_gnu_data("data_out.dat",x_new,y_new,f_new);

    cout<<"Nnew = "<<x_new.size()<<endl;

#ifdef WITH_GNUPLOT
    int pausa;
    GNUplot plotter;
    plotter("set pointsize 1.5");
    plotter("splot \"data_in.dat\" with point 1 , \"data_out.dat\" with dot 3
");
    std::cout << "\n\n >---> Press any key and then <enter> to finish " ;
    std::cin >> pausa;
#endif

 return 0;
}
```

Si deseamos utilizar otra función de base radial, simplemente modificamos la línea de código:

```
TPS<double>    rbf;
```

Por ejemplo, si quisieramos utilizar el kernel de la función inversa multicuádrico, tendriamos que escribir:

```
IMQ<double>    rbf;
```

y dejamos el resto del programa igual. Claro es que tendriamos que seleccionar apropiadamente el valor del parámetro *c* de la función radial IMQ para poder un buen resultado.

Como segundo ejemplo, haremos uso de templates para interpolar un mismo conjunto de datos utilizando todas las funciones de base radial; examples/2d/eje_2d_all.cpp

Lo más importante en el presente ejemplo es mostrar que con base en la programación genérica implementada, podemos fácilmente utilizar distintas FBR. La función principal es:

```
int main(void)
```

```
{
 TPS<double> tps;
 POT<double> pot;
 GAU<double> gau;
 IMQ<double> imq;
 MQ<double>  mq;

test_rbf(tps);
test_rbf(pot);
test_rbf(imq);
test_rbf(mq);
test_rbf(gau);

     return 0;
}
```

Observe que tenemos cinco funciones radiales distintas, las cuales son argumentos de entrada hacía una misma función. La función test_rbf es la misma para todas las FBR seleccionadas, no se requirió programar 5 implementaciones distintas; ésta es una de las ventajas del enfóque genérico diseñado. El código de dicha función es:

```
template<typename RBF>
void test_rbf(RBF rbf)
{
 // rbf:  RBF as kernel we can use : MQ, IMQ, GAU, TPS, POT
 double          c;
 Matrix<double>  A;
 vector<double>  x,y,f,b,lambda;
 vector<double>  x_new,y_new,f_new;
 int  N;

//Muestro el nombre de la funcion radial
     cout<<rbf.name()<<endl;

//defino el numero de nodos
     N = 30;
//Defino el parametro c para mq
     c = 1;

//Creo el vector de datos
     make_data(N,x,y,f);

//Creo la matriz de Gramm
    fillGramm(rbf,c, A, b, x , y, f);

//Resuelvo via Gauss con pivoteo parcial
     lambda = A/b;

//creo el grid en dondo quiero interpolar
     make_data(2*N, x_new,y_new, f_new);

//interpolo los datos
     f_new = interpola(rbf,c,lambda,x,y,x_new,y_new);
     cout<<"Nnew = "<<x_new.size()<<endl;

//Obtengo el maximo error absoluto entre el interpolado y el valor real
     double tmp,emax=0.0;
```

```
    for(int i=0; i<x_new.size();i++)
    {
      tmp = fabs(f_new[i]-myf(x_new[i],y_new[i]));
      emax = tmp>emax? tmp : emax;
    }

    cout<<"e_max = "<<emax<<endl;
    cout<<"-------------------------------------------------"<<endl;
}
```

Observe que la función test_rbf es prácticamente idéntica a la mostrada en el primer ejemplo.

Se utilizaron los parámetros por defecto para la selección del polinomio, así como el factor beta presente en las funciones radiales generalizadas. El presente ejemplo utiliza un gris equiespaciado 30 x 30 para construir el interpolante, sobre un gris cartesiano de 60 x 60 se obtiene el error máximo. Los resultados obtenidos se muestran en la siguiente tabla.

Función Radial	m	M	error
POT	1	1	0.01286044
TPS	2	3	0.00720266
IMQ	0	0	0.00157498
MQ	1	1	0.00060327
GAU	0	0	0.00005513

En la primera columna se indican las funciones radiales utilizadas. Sobre la segunda columna se despliega el grado del polinomio utilizado, recuerde que el polinomio es a lo más de grado m-1. El número de elementos de la base del polinomio se muestran en la tercera columna. El error cometido se indica en la última columna. Los valores de m y M son determinados de manera automática en las FBR. Para las funciones IMQ, MQ y GAU se utilizo un mismo parámtro $c = 1$.

Como tercer ejemplo para datos en 2D, realizaremos un experimento numérico fijando la función radial TPS y variaremos el factor β. Esto es semejante al p-refinamiento en elemento finito.

En la siguiente tabla se muestran los resultados obtenidos (ver eje_2d_tps.cpp).

β	m	M	error
2	2	3	7.202656e-03
4	3	6	2.302889e-03
6	4	10	4.018997e-04
8	5	15	7.584174e-05
10	6	21	1.962668e-05
12	7	28	1.734104e-05
14	8	36	1.219242e-03

El grado del polinomio m se ajusta de manera automática dentro de la función radial, recuerde que el polinomio es a lo más de grado m-1 El número de elementos de la base del polinomio M se ajusta de manera automática dentro de la clase Polinomio.

Observe en la tabla que conforme incrementamos el valor de β el error disminuye. Esto es consistente con la teoría. Para valores de β>12 se tienen oscilaciones numéricas en el error, considero que esto se debe al mal condicionamiento de la matriz. Utilizando la función radial POT se obtuvo un resultado similar, en este caso para β>11 se observo el deterioro.

El presente ejemplo es importante debido a que fácilmente podemos hacer refinamiento sobre β sin tener que cambiar ninguna línea de código.

Ejemplo en 3D
(pendiente)

Trabajo Futuro

Generalizar con templates la clase Polinomio
Revisar y discutir con Daniel la presente propuesta
Ver si queremos utilizar Blitz++ para los vectores y matrices, el performance es comparable con Fortran
Validar el código
Hacer más ejemplos
Definir como vamos a manejar los errores
Crear la documentación con Doxygen o algo por el estilo
Crear una clase para la construcción de la matrizde Gramm
Crear una clase para la interpolación
Revisar y modificar el presente manual de usuario
Mostrar el diagrama UML de las clases
Decirle a Luis Miguel que estamos utilizando algunas secciones de su código
……..