

# Outline of the RBF-DDM code

Jordi Gutiérrez Hermoso

August 17, 2008

Thank you for your interest in my code. I will first try to give you here a brief explanation of what it is and where I'm going with it. Later I'll present to you the problems I'm facing and the sort of help I would appreciate.

My code is currently hosted in a Mercurial repository at <http://inversethought.com/jordi/hg>. Some fairly complete Doxygen documentation is available at <http://inversethought.com/jordi/rbf-doc>.

## 1 Mathematical Outline

### 1.1 Radial Basis Functions

We are concerned with approximating a solution of the general boundary value problem (BVP)

$$\begin{cases} \mathcal{L}u = f & \text{in } \Omega, \\ \mathcal{B}u = g & \text{on } \partial\Omega \end{cases} \quad (1)$$

with the following ansatz

$$\tilde{u}(x) := \sum_{\xi \in \Xi} \lambda_{\xi} \varphi_{\xi}(x). \quad (2)$$

Here  $\mathcal{L}$  is any differential operator on the domain  $\Omega$ , almost always taken to be linear, and  $\mathcal{B}$  is any other boundary differential operator as appropriate for Dirichlet, Neumann, or Robin boundary conditions, or a mixture of these on different parts of the boundary. The functions  $f$  and  $g$  give values on the interior and the boundary of  $\Omega$ . The ansatz  $\tilde{u}$  is a linear combination of *radial basis functions* (RBFs)  $\varphi_{\xi}$ . A radial basis function, like its name indicates, is any function from  $\mathbb{R}^n$  to  $\mathbb{R}$  such that  $\varphi_{\xi}(x) = \tilde{\varphi}(\|x - \xi\|)$ , i.e., its value only depends on the distance from some point  $\xi$ . This has the obvious advantage that the dimension of  $\Omega$  does not affect the formulation of the ansatz in any significant matter, the upshot being that as a replacement for the finite element method, this method saves us the computation of a mesh, and hence it's known as a *meshless method*. The set  $\Xi$  contains collocation points of  $\Omega$  where the RBFs are centred, and it can further be partitioned into  $\Xi = \Xi_I \uplus \Xi_B$  where  $\Xi_I$  are the collocation points in the interior of  $\Omega$  and  $\Xi_B$  are the boundary points on  $\partial\Omega$ .

The most common RBFs are summarised in table 1

Of course, there are further complications with this method. Let us see what happens. Substituting the ansatz into the BVP (1) and evaluating at each point of  $\Xi$ , we obtain the linear system

$$\begin{bmatrix} \mathcal{L}\varphi_{\xi_I} & \mathcal{B}\varphi_{\xi_I} \\ \mathcal{L}\varphi_{\xi_B} & \mathcal{B}\varphi_{\xi_B} \end{bmatrix} \begin{bmatrix} \lambda_{\xi_I} \\ \lambda_{\xi_B} \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$
$$A\vec{\lambda} = \vec{f}.$$

The structure of the Gram matrix  $A$  in this so-called *unsymmetric collocation method* (which so far I've employed exclusively) is separated into four major blocks, depending if we're using the interior or boundary

Table 1: Some radial basis functions

Piecewise polynomial	$r^n, n$ odd
Thine plate splines	$r^n \log r, n$ even
Multiquadrics	$\sqrt{1 + (\epsilon r)^2}$
Inverse multiquadric	$(1 + (\epsilon r)^2)^{-1/2}$
Inverse quadratic	$(1 + (\epsilon r)^2)^{-1}$
Gaussian	$e^{-\epsilon r^2}$

operators, and if we are evaluating the operator on the interior or boundary RBFs. In general, the Gram matrix is full, unsymmetrical (with this method) and ill-conditioned. It is possible to use a symmetric collocation method with a little more work, but I have opted not to do so, because the benefits it yields are marginal. Compactly-supported RBFs also exist, but again, from reading the available literature, it seems their benefits are also marginal.

A more pressing problem is the size of having to invert a full and ill-conditioned matrix. For  $N$  collocation points, this amounts to inverting a  $N \times N$  matrix<sup>1</sup>. Also, since more collocation points means less distance between the points (so that rows that are near in the matrix become more alike with more collocation points), the condition number also increases.

## 1.2 Domain Decomposition Methods

To alleviate the problem of the size of the matrix and the ill-conditioning (which incidentally only is relevant for certain classes of problems, as 200 collocation points or fewer can be quite sufficient sometimes), domains can be partitioned according to various *domain decomposition methods* (DDMs). For RBF collocation, the most common DDMs so far are the *Schwartz methods*, in their *additive* and *multiplicative* form.

The Schwartz methods rely on partitioning  $\Omega = \Omega_1 \uplus \dots \uplus \Omega_n$  such that each domain overlaps with at least one other domain. Overlap means that they must overlap on the interior; domains that merely touch each other along the boundary are not overlapping. For simplicity, let us take  $n = 2$ . Let  $\Gamma_1$  be the *artificial boundary* of  $\partial\Omega_1$ , which is the boundary of  $\Omega_1$  that lies in the interior of  $\Omega_2$ , and let  $\Gamma_2$  similarly be the artificial boundary for  $\Omega_2$  which lies in the interior of  $\Omega_1$ . The additive Schwartz DDM is iterative, so let  $u_1^n$  and  $u_2^n$  represent the solutions at the  $n$ th step of the BVPs

$$\begin{cases} \mathcal{L}u_1^n = f & \text{in } \Omega_1 \\ \mathcal{B}u_1^n = g & \text{on } \partial\Omega_1 \setminus \Gamma_1 \\ u_1^n = u_2^{n-1} & \text{on } \Gamma_1 \end{cases} \quad \begin{cases} \mathcal{L}u_2^n = f & \text{in } \Omega_2 \\ \mathcal{B}u_2^n = g & \text{on } \partial\Omega_2 \setminus \Gamma_2 \\ u_2^n = u_1^{n-1} & \text{on } \Gamma_2 \end{cases} .$$

Notice that we use the values from the previous iteration for Dirichlet boundary conditions along the artificial boundary conditions. This is what characterises the additive method. The alternative is to use the newest values available, which is known as the *multiplicative Schwartz DDM*. The additive method is analogous to the Jacobi iterative method for solving linear systems, and the multiplicative method is analogous to the Gauss-Seidel method.

The additive method has the advantage that it's slightly easier to implement than the multiplicative method, the latter of which requires some hackery to properly colour or group the subdomains. Further, the

<sup>1</sup>Of course, not literally inverting the matrix. Factorisations like  $LU$  tend to be sufficient.

additive method is amenable to parallelisation. The multiplicative method can also be parallelised, but it's more difficult.

The generalisation to multiple domains is straightforward for the Schwartz DDMs with the modification that wherever domains overlap and the value at that point is desired, we take the average from each domain's ansatz.

### 1.3 Different Kinds of BVPs

The basic scheme above can be expanded in two further directions, which I have already partially done. We can take a time-dependent BVP,

$$\begin{cases} \frac{\partial u}{\partial t} + \mathcal{L}u = f & \text{in } \Omega, \\ \mathcal{B}u = g & \text{on } \partial\Omega \end{cases} \quad (3)$$

or a system of BVPs,

$$\begin{cases} \mathcal{L}_1(u, v) = f_1 & \text{in } \Omega, \\ \mathcal{L}_2(u, v) = f_2 & \text{in } \Omega, \\ \mathcal{B}u = g_1 & \text{on } \partial\Omega \\ \mathcal{B}v = g_2 & \text{on } \partial\Omega \end{cases} \quad (4)$$

or combine these approaches. I'm particular interested in the shallow water equations myself,

$$\begin{aligned} u_t &= -uu_x - vv_y - gh_x & &=:g_1(u, v, h) \\ v_t &= -uv_x - vv_y - gh_y & &=:g_2(u, v, h) \\ h_t &= -uh_x - hu_x - vh_y - hv_y & &=:g_3(u, v, h), \end{aligned}$$

which is a time-dependent coupled quasilinear system. One approach for this system, which doesn't quite work for this system (it's not stable) but at least is illustrative, is to first discretise in time,

$$\begin{aligned} u_{n+1} &= u_n + \Delta t g_1(u_n, v_n, h_n) & &=:f_u(u_n, v_n, h_n, \Delta t) \\ v_{n+1} &= v_n + \Delta t g_2(u_n, v_n, h_n) & &=:f_v(u_n, v_n, h_n, \Delta t) \\ h_{n+1} &= h_n + \Delta t g_3(u_n, v_n, h_n) & &=:f_h(u_n, v_n, h_n, \Delta t), \end{aligned}$$

where the  $g$ 's are the relevant functions of the dependent variables, and the  $f$ 's are merely another abbreviation, and then to timestep by evaluating the functions  $f_u$ ,  $f_v$  and  $f_h$  with the previous iterations of  $u_n$ ,  $v_n$ , and  $h_n$ . The next iterations  $u_{n+1}$ ,  $v_{n+1}$ , and  $h_{n+1}$  are obtained by *interpolating* these computed values obtained from the  $f$ 's. Boundary conditions can be imposed at each timestep too. Observe that this is essentially an Euler time-stepping. Stability can be vastly improved with an RK4 method, at the expense of a slightly higher difficulty of implementation.

## 2 C++ implementation

Let us now talk about the actual code in more detail.

### 2.1 Overview

I am particularly interested in making my code *general-purpose*. Ideally, you have a simple BVP in mind, and you just define the parameters of the BVP, a few details of which RBFs you want to use to solve the BVP, and my code should do the rest. For simple BVPs, this is largely already done. The additive Schwartz DDM is also largely implemented, although it currently needs a few extensions. I am not sure I will get around to implementing more DDMs, since I don't seem to have a pressing need for them.

My code is supposed to be highly modular, and I try to make it as readable as possible. For linear algebra and other basic numerical methods, I chose to depend on the GNU Scientific Library (GSL). My code is not as optimised as it could be. For instance, there is no template metaprogramming to avoid temporaries within certain arithmetic expressions. I do, however, overload as many operators as seem useful, and I try to provide sensible class interfaces in my header files.

Besides the GSL, I also employ a few Boost functions. Right now, I only use `boost::shared_ptr` and `boost::hash` functions and classes. In the future, I plan to further depend on Qt classes for GUI functionality. I also liberally employ as many C++ features as seem reasonable, particularly, I happily use all the C++ standard library classes I can.

My code's modularity is supposed to make future optimisations possible. Efficiency is a high priority.<sup>2</sup> For example, should I ever decide that the GSL linear algebra functions are not optimised enough for me, I should be able to only change the implementation of my linear algebra C++ classes without affecting the rest of the code. Similar optimisations should be similarly localised elsewhere.

Lastly, my code should be a flexible building block for others. Should you have a more complex BVP than what my code already provides, you should be able to use my code to take care of the more tedious and routine aspects of how you plan to solve that BVP using RBFs. It will have to be free software as long as I keep linking to the GSL, so eventually I will give write access to the Mercurial repository to more people.

## 2.2 General Structure of my Code

While the gritty details of my code can be found in the Doxygen documentation or in the code itself, I do want to discuss in broad strokes how it's structured.

**Linear algebra classes** These are contained in the `linalg` namespace. I have some classes like `linalg::matrix` and `linalg::vector` which are simply wrapper classes for GSL linear algebra functions.

**Error classes** In the `error_handling` namespace. I use exceptions throughout for error handling. I thus also provide exception classes for GSL error codes, plus a few of my own exception classes that aren't associated to GSL errors.

**BVP definition classes** In order to define a BVP, we need to define a domain, differential operators (two of them), and two real-valued functions for the RHS of the BVP. Those classes are in the `bvp` namespace.

**Radial basis functions** Of course every radial basis function has its own class, with a very obvious class hierarchy.

**Interpolator class** Most of the work in my code is done by the ansatz (2). Within my code, the ansatz is referred to as an `interpolator`, and it's templated by an RBF. The `interpolator` class is also in the `bvp` namespace.

**Domain decomposition classes** So far, I have only implemented the additive Schwartz DDM, but I have provided a general class hierarchy for other DDMs should I or someone else ever decide to implement more. Another family of classes in the `bvp` namespace.

## 2.3 Problems

The interface is very important for me. That being said, I think I see the following problems with my code's interface, and I would appreciate your comments on the matter.

1. I think I have too many namespaces. There's five of them, and I don't have classes within different namespaces with the same name. The way I've seen other projects, all the project goes under one namespace. Should I do the same?

---

<sup>2</sup>But getting the code working first, albeit slightly inefficiently, is a higher priority.

2. I'm not sure I'm happy with the current state of `bvp::interpolator`. The problem is that I'm trying to cram both interpolation and BVP-solving abilities into that class. However, the ansatz is exactly the same; the only difference is how it receives and returns interpolation data. I am thinking that it might make more sense to split the class into a purely interpolating class and a purely BVP-solving class with their common functionality in a parent class. Or am I worrying about nothing?
3. Is it easy to understand what my code is for and how to use it? I provide right now some examples in the `main-*.cpp` files (found in the Mercurial repository) which I will eventually move to an `example/` directory
4. Anything further that you notice about the current condition of my code that you would like to comment on?

Again, I thank you for your time and for the interest you have shown in discussing these issues with me. I eagerly await your input.