IT 10 058
Examensarbete 30 hp
Oktober 2010

Designing a Flexible Software Tool for RBF Approximations Applied to PDEs

Danhua Xiang

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Designing a Flexible Software Tool for RBF Approximations Applied to PDEs

Danhua Xiang

This paper aims at addressing how to design a flexible software for RBF-based numerical solution of partial differential equations (PDEs). In the process, object-oriented analysis and design (OOAD) approach combined with feature modeling, is adopted to construct object models of PDE solvers. This project was implemented in Fortran 90, emulating object oriented constructs by encapsulating particular data structures and subroutines in modules to represent classes.

The separation of mathematical domains and numerical domains as well as the introduction of the workflow manager and operations gives a significant flexibility and extendability for the software. For illustration, a solver for Dam seepage problem is constructed with the new design, compared with the one by the pre-existing reference code.

Keywords: PDE, RBF, Feature Modeling, OOAD, dam seepage

Handledare: Elisabeth Larsson / Martin Tillenius
Ämnesgranskare: Michael Thuné
Examinator: Anders Jansson
IT 10 058
Tryckt av: Reprocentralen ITC

Acknowledgments

First, I want to express my gratitude to my supervisor, Elisabeth Larsson, for your generous support, technical as well as non-technical. You have very patiently spent lots of time to answer my questions and draw the general outlines of the project for me. Besides, you are the one who always have new ideas and lead me to learn all kinds of new knowledge. You make me feel really comfortable and free to do this project.

A special acknowledgment goes to assisting advisor, Martin, for helping me implement the most difficult part as co-developer, learning me useful tools and always timely giving me feedback.

Thanks also to my reviewer, Michael, for your valuable suggestions in making our software framework clearer, more reasonable, and my thesis nicer.

Thank you to Amir, for providing and explaining the test problem as well as helping me to evaluate the solution patiently.

Besides, I would like to extend a very special thank you to my parents for their continued support and care, especially economic and spiritual. Thank you to my little sister for taking good care of parents while I am absent, and to my cousin, Junxia Yang, for your more care about me than my own.

Thank you to all my friends, who kept me company during my study abroad.

Finally, thank you to Sweden for giving me a different study and life experience. Even though you have a dark and cold long winter, I really enjoy your mild, beautiful summer and nice people.

Contents

1 Introduction

As we know, there are already some software for scientific computing. But most of them are for specific numerical methods. To introduce some new flexibility or a particular solver in an existing program for new test purposes will require a complete implementation effort and high expense. To have a software library for fast development of test applications would be the trend of scientific programming. In the last decade, a large number of frameworks for flexible construction of Scientific Computing software have been developed, see for example Diffpack [1] [19], Dune [8] [7], Overture [9], Cogito [32], PETSc [35], SAMRAI [18]. However, none of these specifcally support methods based on Radial Basis Functions. The contribution of the present report is a proposal for a design of a framework to support such methods.

The new framework should be much more maintainable and reusable than the existing one. As proved in the previous frameworks, the object-oriented programming is able to approach code re-use and now it becomes more and more popular. Thus, We will use Object-oriented Analysis and Design (OOAD) method to model mathematic notions and use Fortran 90 as implementing language.

Framework. A framework usually provides generic functionality, which is actually a special case of software libraries. The framework code is reusable. In general, users can extend the framework by selective overriding or specialize its functionality, but not modify the framework code [2]. In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes [10].

PDE. A partial differential equation (PDE) is a mathematical relation involving an unknown function of several independent variables and its partial derivatives with respect to those variables [3]. By PDEs, many of the natural laws can be expressed, such as the propagation of sound or heat, fluid flow etc. Usually, it is a hard task to solve a PDE problem accurately, but numerically computing an approximate solution is relatively easy and fast. In this paper, we will consider linear, second-order PDEs, where one of the common representatives is Laplace's equation, which in two dimension is the governing equation of our test problem.

RBF. A radial basis function(RBF) is a function whose value depends on the distance to center points, so that $\phi(x, c) = \phi(||x - c||)$. RBF-based method is a fairly new approach for simulating the numerical solution of PDEs, but it has been concerned by many researchers [16] [24] [25] [22] [23] [34] [12] [31]. A comparison with two standard techniques (a second-order finite difference method and a pseudospectral method) has revealed that RBF-based methods could yield higher accuracy [25].

Feature modeling and OOAD. Feature modeling is one of the most popular domain analysis techniques [26] and is widely used in software product line (SPL) since feature models , which define features and their dependencies, were first introduced by Kang in 1990 [21]. In this project, we will use feature modeling to analyze common features and their variabilities in the problem and numerical method domain.

Object-oriented analysis and design (OOAD) is an effective technique, which is widely used in modern software enterprises. See [17] about all kinds of application of OOAD. OOAD is fundamentally different from traditional structured design approaches, and it is a method that leads us to consider object-oriented decomposition of a complex system. OOAD method requires the design to map objects and entities in the specified problem region of the real

world. This requires the design to be as close as possible to the real world, i.e. to represent entities in the most natural way.

Fortran. Fortran is a portmanteau of Formula Translator and it is a computer programming language. Fortran has been widely used by scientists and engineers for scientific computing and numerical computation since it was developed in 1950s. The reasons why we use Fortran 90 as implementation language are: the old code was written in Fortran 90 and it is easy to write close to mathematics by Fortran with built-in support for mathematical notations, such as complex numbers, high precision and matrices and so on. Although Fortran 90 is not an object-oriented language, it could implement object-oriented design by encapsulating particular data structures and subroutines in modules to represent classes [33], which are sets of objects that represent certain concepts or entities with the same characteristic.

The following outline of this paper: Section 2 briefly describes the test model, the Dam seepage problem. Through feature modeling, section 3 illustrates relative features of the problem domain and numerical method domain and analysis their possible variabilities. Then in Section 4 object models based on the features are derived by the OOAD method and in Section 5 the Dam seepage problem is used to test these models and evaluate flexibilities of our new design by comparing with the previous one. Some implementation discussions and concluding remarks as well as future works are followed in Section 6 and Section 7 respectively .

2 Test problem

2.1 Background

A dam is a barrier that retains water. Dams are large structures used to control and to store the runoff (surface flow) caused by the flood in order to prevent the destruction of infrastructures or flooding land regions. Also the stored water could be utilized as a source of water for irrigating farmland, generating electricity and urban uses. Since the earliest dam was known in Jawa, Jordan, it has been 5000 years. Throughout the history, dams have enormously pushed the development of human civilization. Without dams, modern life would not be the same as we know it.

Based on structure and material used, dams are classified as timber dams, arch-gravity dams, embankment dams or masonry dams. Embankment dams have two types: earth-fill dams and rock-fill dams. Earth-fill dams or simply earth dams constructed of well compacted earth are a sub-type of embankment dams which rely on their weight to hold back the force of water [4].

The main body of earth dam is mainly made up from soil, sand or rocks. A zoned-earth dam, which consists of distinct zones of dissimilar material, typically has a waterproof clay core. The impounded water pushes against the upstream surface of the earth dam. The pressure of water increases linearly with its depth. If the structures of an earth dam are not enough stable, the dam would not stand up the pressure of water and eventually collapse. It would cause disastrous flood and tragic economic loss. So the construction and safety of the earth dams have become a common and important research area for civil engineers.

2.2 Seepage in an earth dam

Since the body of earth dam is not rigid, water seeps in to the pores of the soil. The line of seepage flow is called the phreatic line with almost a parabolic shape, as shown in Figure 1.

If the phreatic line overtops the break level threshold of the earth dam, it would cause vast flooding through eventual failure of the earth dam. In order to decrease the pressure of the seeping water, modern earth dams employ filter and drain to collect and remove the seep water and to lower the level of phreatic line (Figure 2).

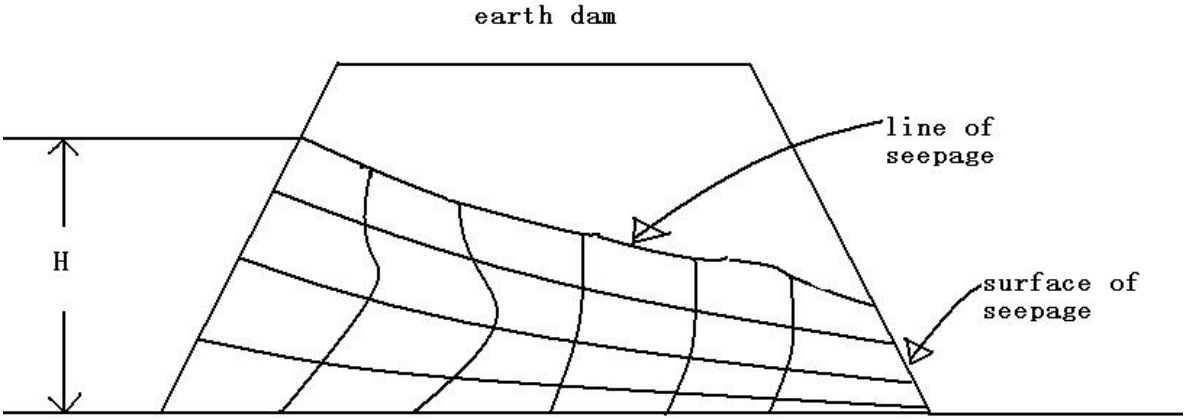

Figure 1: Seepage and potential lines in an earth dam

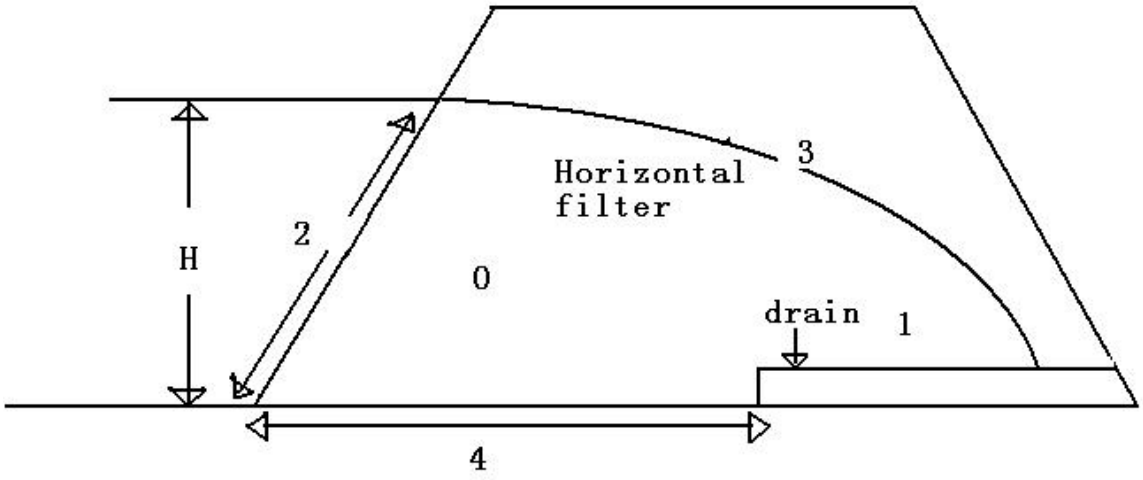

Figure 2: Drain installation as an alternative to control seepage

2.3 Governing partial differential equation

In order to research the seepage problem or decrease the affect of seepage to the dam, a model could be formulated in the form of a partial differential equation (PDE). The governing PDE of seepage in porous media is Laplace's equation as following:

$$\frac{\partial}{\partial x}(k_x \frac{\partial h}{\partial x}) + \frac{\partial}{\partial y}(k_y \frac{\partial h}{\partial y}) = 0 \tag{1}$$

Where h is the water pressure on the dam, (x, y) represents a certain point on the dam, k_x, k_y are coefficients with respect to x, y. When $k_x = k_y$ it transforms to Equation 2:

$$\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} = 0 \tag{2}$$

From Figure 2 above, we can see that it forms a region, which is the domain of this partial differential equation, under the phreatic line and over the surface of the drain. Laplace's equation is a typical elliptic PDE, for which there is no exact analytical solution. I will consider the seepage analysis as representative model in the process of software design. The numbers around the domain represent the five different boundary conditions, expressed as :

0. $\Delta h = 0$ (Interior points)
1. $h = y$
2. $h = H$ (of Dirichlet type)
3. $h = y$ (of Dirichlet type) and $\frac{\partial h}{\partial n} = 0$ (of Neumann type), where n is the normal vector
4. $\frac{\partial h}{\partial y} = 0$

3 Feature and variability modeling

The feature and variability modeling is used commonly in the area of product line design [20] and has been applied to numerical software by Ljungberg [27]. It is a way to capture requirements of the applications through explicitly establishing what kinds of features should be included in an application and the desired variability of these features. In this context, we focus mainly on features related to flexibilities. Generally, the variability of an application is an implicit consequence of some implementation mechanism. It indicates the possibility to modify the applications on the basis of features.

A PDE solver is usually huge and accomplished via dividing it into smaller pieces. There are many different ways to divide the PDE solver. A traditional approach is also dividing the solver into parts, where each function represents a part of the algorithm. In this thesis, we will divide it into two parts, i.e. mathematical problem and numerical method. The features of each part will be addressed individually. According to different concerns, users could also be categorized into problem oriented user and method oriented user. Also, these two user categories plus developers lead to different level possibilities to vary the features of this application. We could consider the variability for each feature from the perspective of problem oriented users.

- Problem oriented users are general users who just want to solve some problem using this software tool without caring too much about the numerical method. They could select or modify the features of the application via user interfaces.

Furthermore, method oriented users and developers can extend the framework by overriding or adding new modules. In other words, the framework allows two different levels of extensibities with respect to the roles people play.

- Method oriented users or expert users are usually experts in this area, who want to develop or demonstrate some numerical method. They are allowed to add new method modules to extend this framework.
- Developers can decide to support features arbitrarily by modification of code or adding modules. Usually, these features do not fit within the model and it requires significant effort to add these features.

In the following section, we will exploit the features existed in this framework and their variabilities, as well as possible extensibilities which will be covered with the variabilities.

3.1 Problem features

Scientists are used to model phenomenon by PDEs defined over a domain. PDEs include governing equations representing the problem numerically, and boundary conditions which could be different types, e.g. linear operator or complex conditions like Dirichlet-to-Neumann map (DtN) radiation boundary condition [28]. And the domain is commonly in the form of a geometry. For the time-dependent model, initial conditions also need to be considered, that can be represented by means of equations.

3.1.1 Equation

Equations are mathematical description of the PDE problem, which represent governing equations, initial conditions and boundary conditions.

Subfeatures: An equation can be an *algebraic equation* or a *differential equation* such as

$$u - f = 0, \quad \text{(interpolation)},$$

and Helmholtz equations:

$$\begin{cases} \Delta u - \kappa^2 u = 0 & \text{(Helmholtz equation, PDE)} \\ -u_x - i\beta u = -2i\beta sin(\alpha x) & \text{(Left boundary condition, PDE)} \\ u_x - i\beta u = 0 & \text{(Right boundary condition, PDE)} \\ u(\underline{x}) = 0 & \text{(Top and bottom/Dirichlet boundary condition)} \end{cases}$$

It can be *linear* as in the examples above or *non-linear* as

$$uu_x + u_{xx} = 0.$$

Equations can be *time-dependent* or *time-independent*. Whether the coefficients or the unknown u is with time, we say the equation is time-dependent. If the u is time-dependent, equations could have different order derivatives with respect to time. All equations are considered to be *space-dependent*, but the coefficient functions can be variable or constant. Our intention is not to offer specific equations, rather to allow building any equation. The feature diagram is shown in Fig. 3.

Variability: We do not consider equations containing time derivatives and only support linear equations. To add the non-linear feature, we would need to develop entirely new code as developers. If it is time-dependent is implicitly determined by the coefficient functions and can hence change during the build up of the equation. At the same time, users can easily decide to choose a variable function or a constant for the coefficient at run time.

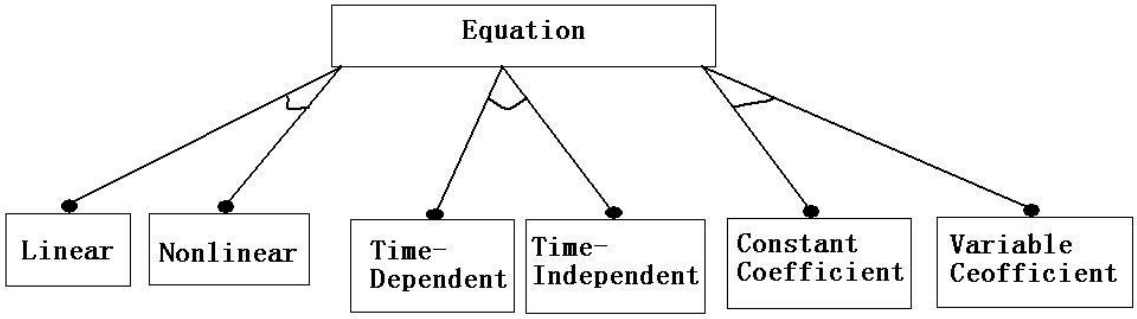

Figure 3: Feature diagram of equation, FMN [11]

3.1.2 Geometry

Subfeatures: The feature geometry represents the domains of problem models. The domains can be infinite or finite. When the problem model has enough boundary conditions, then the infinite domains can be translated into finite.

Thus, the geometries can be regular or irregular multi-dimensional, i.e. the shape of geometry is flexible. Moreover, the shape of geometries or the size of the domain could be static or dynamic.

Variability: For time-independent model, the shape of geometry should be determined during setup of the problem model. However, the geometrical shape can change with time elapsing for time-dependent model, and it is determined once the time is chosen.

By developers, they should consider to support as many types of entities as possible and allow users to freely utilize these entities to construct any reasonable geometry. Otherwise, expert users are welcome to complement the entities if needs.

3.2 Approximation features

The numerical method used in this thesis is RBF interpolation. So in this section, we will discuss features associated with RBF approximation.

3.2.1 Basis function

Basis functions are used for computing approximations in the process of representing the solution of the PDE or interpolation problem.

Subfeatures: A basis function could be a radial basis function or a non-radial basis function. A radial basis function (RBF) only depends on the distance between points, i.e. *r-dependent* (r is the distance between two node points) and is of the form $\phi(r)$, such as

$$\phi(r) = |r|^n, \quad n\ odd \quad \text{(Piecewise Polynomial(Rn))},$$
$$\phi(r) = |r|^n \ln|r|, \quad n\ even \quad \text{(Thin Plate Spline(TPSn))},$$

The RBF may also have a shape parameter ε and is of the form $\phi(r, \varepsilon)$, such as

$$\phi(r, \varepsilon) = \sqrt{1 + (\varepsilon r)^2}, \quad \text{(Multiquadric(MQ))},$$
$$\phi(r, \varepsilon) = e^{-(\varepsilon r)^2}, \quad \text{(Gaussian(GS))},$$

Conversely, a non-radial basis function does not depend on distances, for instance polynomial basis function and RBF-QR [14]. A polynomial basis function is $\underline{x}$*-dependent* ($\underline{x}$ is the placement of points) as

$$1 + \alpha_1 x + \alpha_2 x^2 + \cdots$$

And the RBF-QR basis function has the same form as polynomial basis function, but it is $\phi - basis$ as $\Phi(\underline{x})$.

All the examples above are time-independent but basis functions can also be time-dependent. See Fig. 4 about its features.

Variability: Which basis functions are supported (i.e. the leaves of the feature tree Fig. 4) is determined by developers during implementation or added by expert users later. And the other features e.g radial or non-radial basis function are activated by users once they choose certain basis function for test purposes.

The matrix of interpolation and approximation could be singular when using inadequate shape parameter for some RBFs. It is necessary to introduce the polynomial basis functions in order to eliminate singularity. If the polynomial basis function is needed depends on RBF itself. Thus, we would support both radial and non-radial basis functions in our system.

With respect to implementation, each commonly used RBF would be hard coded. If a new RBF need to be investigated, it still need hand code which has to be done. But the polynomial basis function could be relatively flexible, supporting any order.

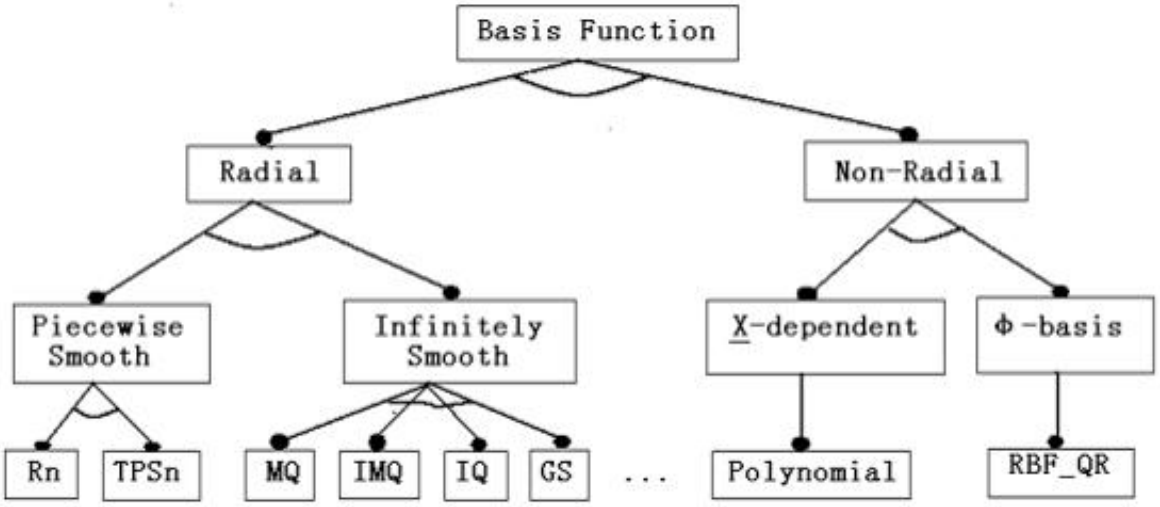

Figure 4: Feature diagram of basis function

3.2.2 Shape parameter

There is a feature called shape parameter, commonly noted as ε , which is involved in most of RBFs $\phi(r, \varepsilon)$, such as MQ, GS, with distance r between node points.

Subfeatures: The data type of shape parameter could be real or complex. We could use constant or variable shape parameter strategy which refers to uses a possibly different value of the shape parameter at each center, i.e. each node point with its own constant shape parameter value.

Moreover, if the shape parameter is variable, it could be determined randomly or by some functions and can be changed before computing each RBF interpolant on all node points. [30] compares four different types of shape parameter strategies (constant, linearly varying, exponentially varying and random shape parameter) and also introduce a new random variable shape parameter where ε_0 scaled with the inverse distance to the nearest neighbor.

Variability: In terms of the flexibility, as developers, we prefer to implement both real and complex data type, and support several strategies for the shape parameter to facilitate users. More strategies are welcome to be added by developers or expert users.

If the data type is real or complex, and which strategy is used, are determined by users while setting the shape parameter. However, the values of the shape parameter could also be assigned automatically in the program according to the different RBFs.

3.2.3 Approximation

An approximation is an inexact representation of the unknown function $u(t, \underline{x})$ that needs to be solved in the problem, but there should be no significant deviation with the real form.

Subfeatures: When you use Radial Basis Functions, there are two formulas for computing approximation depending on each part of the equation which is studied.

$$s(\underline{x},\varepsilon) = \sum_{j=1}^{n} \lambda_j \phi_j(\underline{x},\varepsilon) + \sum_{i=1}^{k} \alpha_i p_i(x) \tag{3}$$

$$s(\underline{x},\varepsilon) = \sum_{j=1}^{n} \lambda_j L_j \phi_j(\underline{x},\varepsilon) + \sum_{i=1}^{k} \alpha_i p_i(x) \tag{4}$$

$$\sum_{j=1}^{n} \lambda_j L_j p_i(x_j) = 0 \quad i = 1 \dots k \quad \text{(constraint)} \tag{5}$$

where $\phi_j(x,\varepsilon)$ is basis function and L_j is an operator,such as Δ; p_i is polynomial basis function. The polynomial part depends on radial basis function ϕ_j itself. And its degree should be defaulted to zero.

Equation 5 represents constraints which define the feasible set of candidate solution of the unknowns. The purpose of introducing constraints is to make the numbers of equations and unknowns to be equal. Whether constraints are needed or not depends on the particular problem.

Which formula should be used or which operator should be applied to the basis function (RBF and polynomial basis) for each type of node points, depends on node-related equations and this could be done pointing to the equations.

For an unknown function or more that need to be solved, different basis funtions and a range of ε values could be applied to get various approximations.

Variability: The approximation depends on RBFs, shape parameter and point sets and could vary according to these three objects. The combination of them is quite variable and flexible, which could form numerous approximations. Developers should consider how to combine the RBFs and shape parameter values or strategies with each point set and provide combination options for users. A flexible idea is to assign a different part of the RBFs and ε values to each type of point set, that arrives to an approximation whose elements have totally no common. To ease the implementation, usually, a general and simple rule may be to only share all the RBFs and ε values.

Thus, the approximation can be determined once the users specify RBFs, shape parameter strategies or values , point sets, and the combination method if there is.

3.3 Methods

Lots of possible methods appeared in the RBF simulation of PDEs, for instance point sets construction, collocation methods, and methods to solve the system, as well as all kinds of evaluation operators and operations.

Features: The structure of points sampled from the domains could be global, stencil or partition node point set. They will lead to different coefficient matrices. It should

be possible to create stencils and partitions from global point sets. That means the representation of points is flexible.

Collocation methods are a battery of methods by collocation to satisfy both PDE and boundary conditions, which can be various based on point types: straightforward RBF-based collocation [22] [23]; symmetric collocation [34] which modifies the basis functions in the interpolant by using the operator in the PDE; direct collocation [12] which collocates both with boundary condition and the PDE at the boundary points.

The method for solving the linear system could be direct solution method (Lu-factorization) or least squares. Least squares is a method for seeking optimum function of data by minimizing the sum of squares of residuals, a residual being difference between an observed value and the value provided by a model. There are linear and nonlinear least squares depending on whether or not the residuals are linear in all unknowns [5]. The least squares method could combine with collocation methods.

When the number of collocation points equal to the number of basis functions ($M = N$), only Lu-factorization needs; when $M > N$, then least squares is used; in other case, the combination method of these two could be adopted, i.e. Lu-factorization for some points and least squares for the other points.

There are still many methods which could be considered to extend the flexibility.

Cauchy method [15] allows numerically stable computations of radial basis function interpolants for all shape parameter values by removing the restriction the ε be a real parameter and allowing ε to be complex. But it is time-consuming.

Instead of the Cauchy method, we introduce RBF-QR which employs a change of basis function to allow computation for arbitrarily small values of the shape parameter. This method is perfectly stable and can be used for large number of node points [14].

Probably, we could use Galerkin method which converts a continuous operator problem to a discrete problem [6] for computing interpolants. Also see [31] about the use of this method.

Moreover, the software tool should support kinds of evaluations of the solution or its derivatives, i.e. the evaluation operator can vary due to needs. And users should be able to do many different operations on the same problem in any order, e.g. composing matrices, computing coefficients and evaluating solutions etc.

Variability: From the user perspective, it is preferable that users are allowed to create different structures of point sets via feature selection.

Whether or not a collocation method is used could also be determined by users or problem models needs to be investigated while constructing the models. And which method is applied to solve the system depends on the relation of collocation points and basis functions. That is, it is activated after the user sets these two parameters.

In aspect of implementation, we would probably only support the first two methods for solving the linear system, and the combination method could be supplemented by developers or expert users in the future for more complex investigation. In the meantime, the Cauchy method and the Galerkin method will not be included either.

For the evaluation operators, developers should consider the supported maximum order of the derivatives with respect to the space and time. And users is totally free to do any evaluation as long as it does not exceed the limits.

The use of operations is much more flexible. The problem oriented users, are just responsible to make sure the involved operations in a correct order. And expert users or developers could add any reasonable operations to extend the software for new requirements at any time.

4 Design

After the analysis of the relevant features in this framework, we come to the design phase, which is extremely important and time-consuming in the whole process of developing the framework. A good design will often lead to an unexpectedly flexible and easy-to-use product. Before presenting our design outcome, we need to talk about some issues occurred during the design and how we overcame them.

4.1 Design issues

1. **Design philosophy.**
   The purpose of software development is to meet user requirements. So it is crucial to design from the user perspective, i.e. considering what would be most natural for the user. As a user, whether expert or non-expert, he would expect that the currently used software is sufficiently efficient and flexible. For example, the user hopes that this software is suitable for any problem and it is able to solve synchronously multiple problems within an acceptable time. According to these requirements, we introduce the ideas of parallelization, separation of problem and method, and the workflow manager etc, which will be explained in detail below.

2. **Separation of problem and method.**
   The old code has very low flexibility. Every time, the users try to solve a new type of problem, they require changing the method part of the code, which is a hard work and probably only possible for experts to do.

   We think a better way is to allow the user to specify a new problem and apply existing methods. In view of this, we got an idea that if it is possible to separate the mathematical problem from the numerical method as Krister Åhlander did in [29]. Then the problems and the methods could be combined arbitrarily, i.e. one problem could be solved by different methods and vice versa. But the idea of separation requires an effort to give a type of specification that allows the method part to automatically determine the details of how to do it.

3. **Consideration about the equations.**
   In the general procedure of solving PDE problems, PDEs and operators are essential features. PDEs are the mathematic representations of the physical problems and operators are necessary for the solution evaluation. These two are similar and should be represented by the same class. The only big difference is with or without right hand side. Should we seperate the left hand side of PDEs from the right hand side? And

use two different classes to cover two sides of the equations? Do we really need the *right_hand_side* class as did in the old code?

Maybe not. Actually, it is easy to make PDEs and operators be in the analogous form. What we need to do is to move all terms to the left side and let the right sides of all PDEs be zero. Then, both of PDEs and operators are without right hand side and could possibly be involved in one class called *expression*. Furthermore, the right hand side vectors could also acquired simultaneously during building the interpolation matrices and the user does not need to bother his head about that.

Then, the consideration comes to the feature selection of the equations. Since to support a non-linear system would be totally different, we would mainly focus on the linear system. Sometimes, problems are time critical and the previous software tool can only support some time-independet problems, that is indeed low flexible and limited. We prefer to have time-dependent equations but no big changes to the current low-level classes. Our suggested idea is to use RBF approximation for the spatial operator, but typically an ODE (ordinary differential equation) time-stepping method for the time-derivative. The time-operations needs to be added by an expert user. Furthermore, we want to support both the constant and variable coefficients, this job is possible and easy to do.

4. **Coupling of PDE and geometry.**
   The two features of the problem, PDE and geometry, should be separate but must have links. when the user specifies the equations and the domain of the investigated problem, the relations of these objects, i.e. which equation is associated with which geometrical object, are not created yet. But the question is how to link them and who to make sure the correct relations?

   Our solution is to define two classes for PDEs and domains separately, and use the same name as a mark for the relevant objects of these two parts. Then another *problem* class is used to check their compatibilities, that is, to see if every equation has an associated geometric object. Renaming might be needed for flexibility when PDEs and domains are coupled.

5. **The workflow manager and operations as well as the data manager.**
   Another deficiency that existed in the old code is that all of the problems are solved in the same workflow scheme, from building matrices to evaluating the solutions. In other words, the loops on operations are always in the same order, unchanged. Sometimes, users probably do not want to run to the end directly and prefer to halt halfway or try some new ways. Clearly, a single loop-order usually could not meet the frequently changing needs.

   The user should be able to set up workflow scheme by himself, i.e. easily changing operations to a needed order. Thus, the idea of workflow manager and operations was born. The workflow manager is in charge of organizing operations added by the user in a proper sequence, and then execute all of the operations successively as the user expects.

   In order to allow for the reuse of data and improve the efficiency, we tend to use a data structure in the workflow manager to keep trace of some data for instance distances.

Since the workflow manager needs to call operations where lots of data will be delivered, if we keep all the data in the workflow manager, it definitely would cause a circular dependency problem. Namely, the workflow manger and the operations will need to use each other. Therefore, we separated this data structure from the workflow manager as a new *data_manager* class to manage all of the input and output data.

6. **Internal and external interfaces.**
   Usually, for each class, there are lots of methods. Some of them need to be invoked in other classes or used by users to create objects or get some information, and the others would only be for internal use in the class. The previous one could be defined as public interfaces and the later are surly private.

   Sometimes, users need to call some public interfaces in the low-level classes but we do not expect users to be in touch with them directly. Besides, it is not necessary for users to know the whole structure of the software and they would prefer to easily use some interfaces in the high-level classes. Thus, as developer, we must think about how to ease the difficulties of interfaces use and to provide nicer interfaces for the user.

   The traditional solution is to define public interfaces in the high-level class for the relevant interfaces of the low-level classes. This high-level class probably could be an aggregate class of these low-level classes. These interfaces in the high-level classes are called external interfaces, which are actually mirror images of the one in the low-level classes, and could be accessed by users conveniently. But, if other classes need to call these interfaces, it is preferable and more efficient to still use the one with the same function in the low-level classes, which are called internal interfaces.

4.2 Object model

According to the OOAD method, in this section, we will identify suitable objects or classes that together will cover the features space and provide desired variability. For each class, we will describe what it does or which feature it associates with, part of implementation in current code or interfaces provided in the new class. The main structure of classes follows as:

- Data structures for one kind of particular objects
- Constructors for allocating memory and producing objects, and destructor for releasing memory.
- Query-functions for sizes etc
- Interfaces to methods which are provided if necessary and will be specifically described in the individual class.

4.2.1 Expression

This class represents mathematical expressions involving an unknown solution function $u(t, \underline{x})$ and its spatial derivatives and known functions $a_j(t, \underline{x})$ and their derivatives. See the representative class *Expression* in Fig 10.

Expression is the generalization of equation and operator, which represents only linear expressions in this context. This expression class does not exist in the current version of the

code. The PDE and the evaluation operator are hand coded in the operator class for each problem. This leads to a large programming effort every time a new problem is investigated, which is undesirable.

Specification: We will move all things to the left side of the equations. Then operators and equations have the same form, without right side. The expression class could use just one type of data structure (type *expression*) to represent them.

The *expression* can represent array of equations and each of its elements can have a different number of terms. Then we could define new data types *equation* and *term* to represent one equation and one term respectively. Equations must be able to be identified by different names. Namely, the *equation* should contain a variable for names, which is the connection to geometric objects. And the data type *term* could be with operators, coefficient variables, sign and flag variables for time-dependent or time-independent etc. This class should also have some relative operations referring to the terms or expressions.

The following operations must be supported:

- A function for constructing expressions through adding terms into the expression instances. This function could have different versions depending on if operator, coefficient, coefficient function, or operator on coefficient function are present. And all the versions will be integrated into a generic interface.

- It is necessary to have a function to compute coefficients of a certain term on particular points. It might be with alternative parameter lists depending on if it is time-dependent.

Probably, some get_methods or query_methods are needed to find out how many equations or terms and if they are time-dependent etc.

4.2.2 Geometry

This *Geometry* class keeps track of geometric objects in the problem, such as the domain, boundary segments, corners, etc. Namely, the feature geometry is implemented clearly in this class. We mainly consider finite domain. For boundary value problems, their domains are usually decomposed into interior of computational domain and boundary segments. Thus, the domain for a particular problem can be described by various geometrical entities, such as, points, point sets, lines or closed curves.

Specification: This class should have a *geometry* data type, which at least has two vectors, one for numerical or analytical representations of sub-domains or boundaries and the other one for domain names as indexes. It might need various variables for keeping trace. It can be extended for implementation requirements.

It should support operations as follows:

- An interface to specify the geometry entity for a domain via a friendly name or domain index.

- An interface to produce points according to a given domain name or index.

- Also, it should have methods for computing normals.

4.2.3 Problem

This problem class is an abstract class for all problem models, which refers to class *Problem* in Fig 10. An investigated problem is commonly expressed by using numerical or analytical representations, i.e. governing equations and domains.In other words, it is an aggregate class of expression and geometry class, who keeps track of the relationship between equations and geometric objects for the domain of a currently studied problem. It makes sure that each equation is correctly associated with the right geometric object, i.e. the expression object and the geometry object is compatible.

Specification: The purpose of this class is to check the compatibility between equations and geometric object, interior of computational domain or boundary segment. Thus, we will first define a *problem* data type with two vectors to trace objects of equation and geometry individually. And The only operation needed in this class is an interface for checking the relations of expression and geometry objects via the name comparison.

4.2.4 Basis function

Apparently, the basis function objects cover the feature basis function, which are expressed in the abstract class *Basis_function*. From this abstract class, we could derive other basis_funtion classes with specific features, for example radial basis function, polynomial basis function and RBF-QR.

Specification: Currently, there are two derived classes: class *Radial_basis_function* for radial basis functions and class *polynomial_basis_function* for polynomial basis functions. Also, we want to create a class for RBF-QR method which could have a function to form basis functions and a function to evaluate basis functions.

The base class *Basis_function* probably consists of the common interface, i.e. the default constructor and destructor. In addition to the common interfaces, the *Radial_basis_function* class should be with

- A generic interface for evaluating RBFs or their derivatives. It should allow alternative parameter lists depending on which types of derivatives are to be computed.

The class *Polynomial_basis_function* has two public functions and one interface.

- A function for getting the number of polynomial basis functions applied for probably eliminating singularity.
- Another funtion for getting the powers of the basis polynomials.
- A interface with alternative parameter lists for evaluating a polynomial basis at node points or evaluation points.

4.2.5 Shape parameter

There is a shape parameter class, which covers the feature shape parameter, in the current program (see class *Shape_parameter* of Fig 10). Users can create shape parameter instances of this class with values. And the instance indexes should be used as input parameters of some subroutine or function in the approximation class to define approximations.

Specification: This class supports two types of shape parameter, real and complex. It consists of a data type *epsilon* and lots of functions.

The data type *epsilon* has four one-dimensional vectors for storing data on each component of the different coordinate systems, Cartesian or Polar.

There are three different versions of generators for producing batches of shape parameter values. Two of them refer to coordinates and another is for the small value ε on a circular path.

- An interface to get one shape parameter value according to a given place index or to get all the data on one component of the given coordinate system . Similarly, it must have several versions for different data types and coordinates.
- An interface to get the minimal value of the shape parameter set. The minimum is needed in the process of shape parameter range partition, to decide whether or not Cauchy method is used for computing approximation.

4.2.6 Point sets

Points sampled from the domain and used in RBFs can be represented by the *Point_set* class which has been implemented currently. This class is enough flexible to support any large number of points with multi-dimensionality.

Specification: In this class, there is a data type called *point_set* and some interfaces.The data type includes two fields which are sets of the array with different sizes , another embedded data type for dimensional matrices. One is as index for tracing points of different types, and the other one is as the points storage.

- An interface for getting a part of points from an existing point set. It should have several versions refer to different dimensions.
- An interface for getting the relative place range of certain type of points in the point set.

4.2.7 Approximation

This class (class *Approximation* in Fig 10) describes what kind of approximation is used for the unknown function $u(t, \underline{x})$ in the expression class as mentioned above. It does not actually contain numerical data. That is, it only has to do with how we represent our solution function.

This class covers the approximation feature and its subfeatures. It's a new developed class. In the previous code, approximation computation is implemented in the solver class by calling some functions in the operator class. Currently, we would like to acquire multi-approximations for the same problem with respect to different basis functions and shape parameters.

Specification: A approximation class should have a data type *approximation* with a list of node point type for each approximation element, and variables refer to RBFs, shape parameter objects and polynomial, constraint etc.

Every approximation element, which relates to one of the geometric objects (interior or boundary segments) or equations, allows to connect to a few basis functions and a range of shape parameter values. From the implementation aspect, it means to call and use approximation block by block during composing matrix blocks.

4.2.8 Operations

This class is related to the subfeature operations of the methods (see class *Operation* in Fig 10). There are lots of operations needed in the process of constructing PDE solvers. Thus, an abstract *operation* class will be introduced to aggregate them. We will illustrate each operation in an independent class and then gather into a so-called operation component. This method makes it easy to develop new operations in order to extend this component and efficient to reuse existing operations. The introduction of some basic operations combined into the entire procedure of solving PDEs is covered in the following subsections.

1. **Compose (build RBF matrices)**
   The compose operation is actually a matrix builder, being responsible to create the matrices A and B or their blocks in the equation $A\lambda = f$ or $B\lambda = Lu$, where A is matrices for collocation points and B are matrices for evaluation points, L is an operator. This operation should be able to take input in vector form and generate many blocks. These blocks could be floating (to be kept as individual blocks) or assembled into one matrix. The number of blocks depends on the type number of point sets. And the number of matrices is determined by the numbers of RBFs and epsilons or their collocation.

   By old code, this process of composing the matrix A or B is implemented in the class *solver* by calling the *assign_operator* function of the class *operator*. The reason we separate this process from the *solver* class as an independent module is that it is more flexible to call and to implement parallel computing later. Especially also that we want to be able to assemble several matrices at the same time and then do something with them.

   Specification: Probably, we will define a data type *block* with the point type to store parts of a matrix for each type of node points. Then this class could have another data type with blocks via assembling blocks to form a matrix. Other index or size variables need to be figured out during the implementation process.

   The only function is to compute all blocks of the matrices for different points collocations. Any auxiliary functions could be introduced if needed, such as to merge operators for each type of node points and center points.

2. **Compose (build right hand side vectors)**
   This compose operation is to compute function values f on node points in the equation $A\lambda = f$. The right hand side could be used to compute the coefficients λ_j for the RBF interpolant. However, the right hand side vectors are determined by the node points, terms without unknown function $u(t, \underline{x})$ and operators of these terms in PDEs.

   There is a *right_hand_side* class currently. It supports multiple right hand sides through combination with one other class *function_value* which is hard coded to compute the enumerated function values. However, it always needs to add new test functions for the right hand side to satisfy any new requirement. If it could support several user specified functions at a time to form multiple right hand sides, it will be more flexible.

   Specification: To implement this operation, we just need a function to get node points and terms for the right hand side in the expression object as input parameters, and then produce the output vectors via calling functions in the *function_value* class.

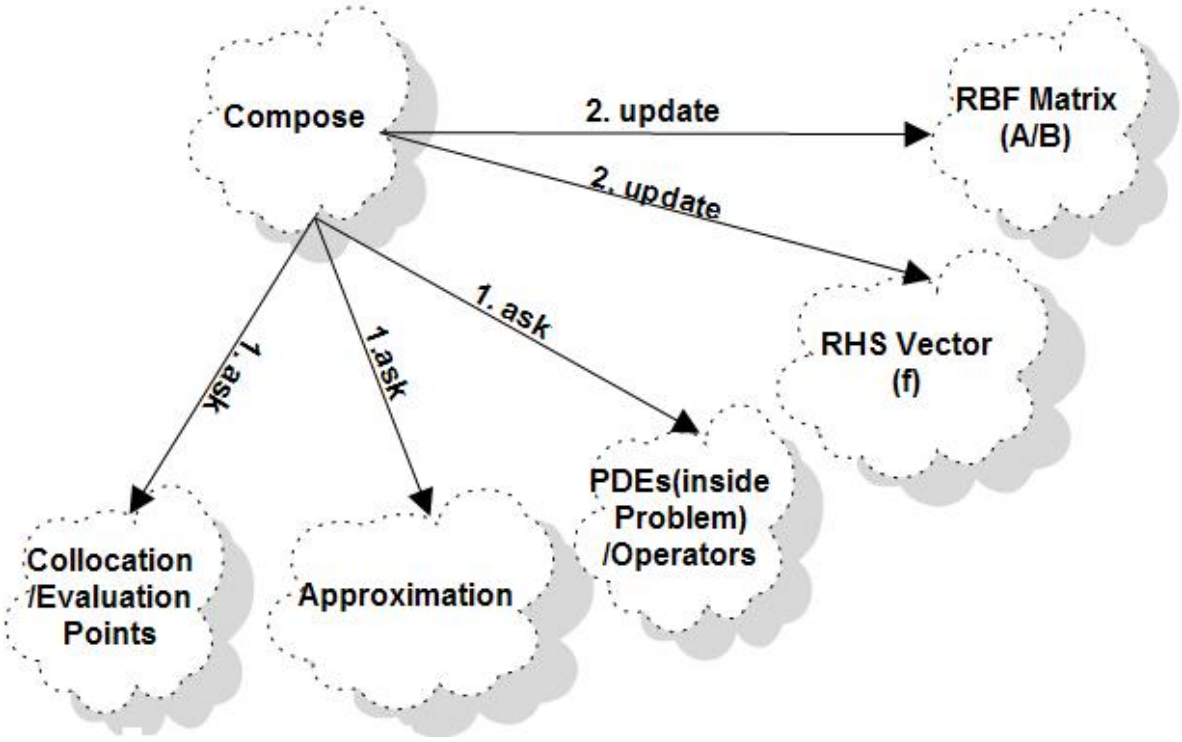

Figure 5: Operation Compose

3. **Solve (compute coefficients λ)**
   The *solve* operation takes matrix A, right hand side vector f as inputs and use them to compute the coefficient λ as output, i.e. $\lambda = A^{-1} * f$. It is related to the feature solving method, Lu-factorization and least squares. see Fig 6.

   For the current code, there is also a function called *solve* in the solution class. But, it actually gets the solution of the PDE problem. Of course, the process of computing coefficients λ is included and just the first step in that function.

   Specification: This operation is also relatively simple and it follows the *compose* operation. After the user adds the operations for building matrix A and vector f, this operation only needs some linear algebra subroutines for instance LU-factorization or least squares, to get coefficient λ with matrix A and vector f as inputs.

4. **Solve (compute the differentiation matrix C)**
   The second functionality of the *solve* operation is to get the differentiation matrix C with these two matrices A and B as inputs , i.e. $C = B * A^{-1}$, where the matrix A^{-1} could be considered as multiple right hand side. See Fig 7 for the operation procedure. The purpose of this operation is to avoid computing the coefficient λ explicitly.

   Specification: To compute C is typically done as a multiple right hand side transposed solve operation. Namely, this function could be implemented by calling the previous solve operation for computing λ.

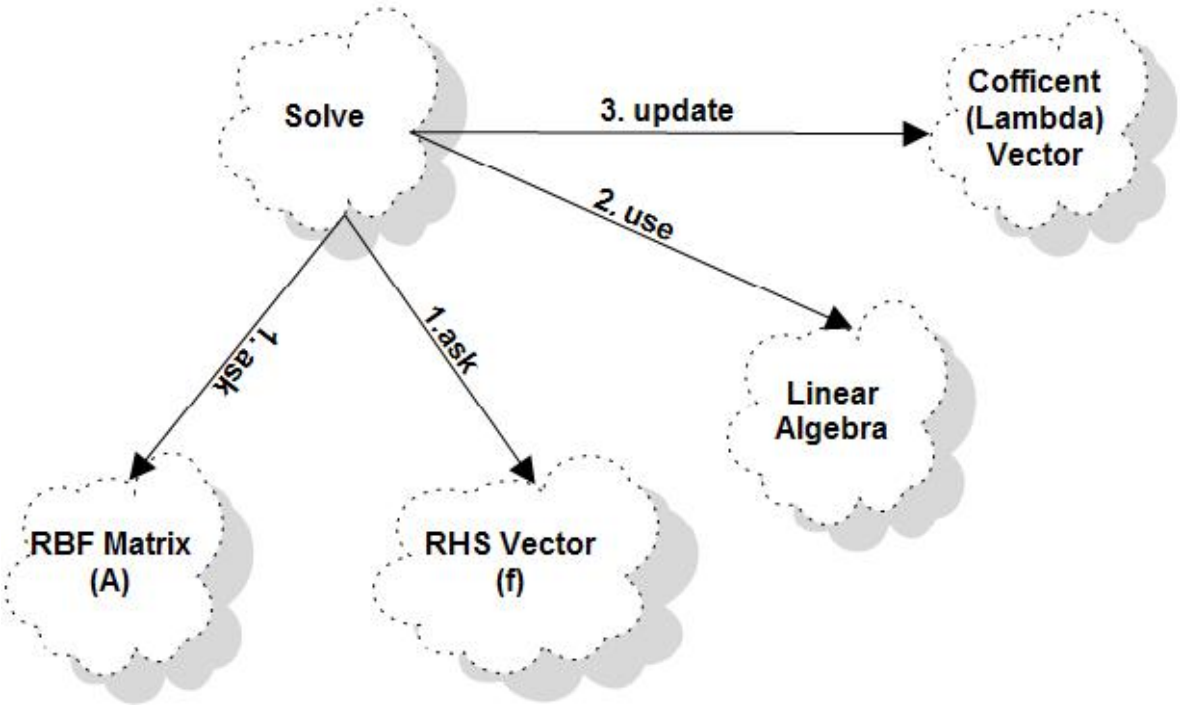

Figure 6: Operation Solve for computing λ

5. **Evaluate (evaluate operators)**
   The evaluate operation is primely used to get solution of the problem and evaluate its other properties on evaluation points, such as various derivatives of the RBF-based simulation solution or residuals. see Fig 8.

   The current software just supports to evaluate the fundamental solution of PDEs, not other properties. It is not enough and lacks of flexibilities for research purpose. This is done through the use of *solve* function in the solution class as mentioned in the computing coefficient λ operation.

   Specification: The main task of this operation is actually matrix-vector multiplication. For different cases, the matrix-vector multiplication could be $B * \lambda$ or $C * u$ (here, u is equal to f above) for evaluating operators finally.

4.2.9 Workflow manager

This class (*Workflow_Manager* of Fig 10) is used to set up workflow schemes by the user and then it works as the scheme by means of combining all the things. It means that users can do operations in any reasonable order they would like, not just as regular flow process. Researchers are allowed to map items of a certain problem directly into code through the use of this class. They should provide all inputs including scheme and specify output at the end of the pipe-line. Then the workflow manager will process the operations one by one and produce output as expected. The idea of workflow manager mainly derives from the observation that

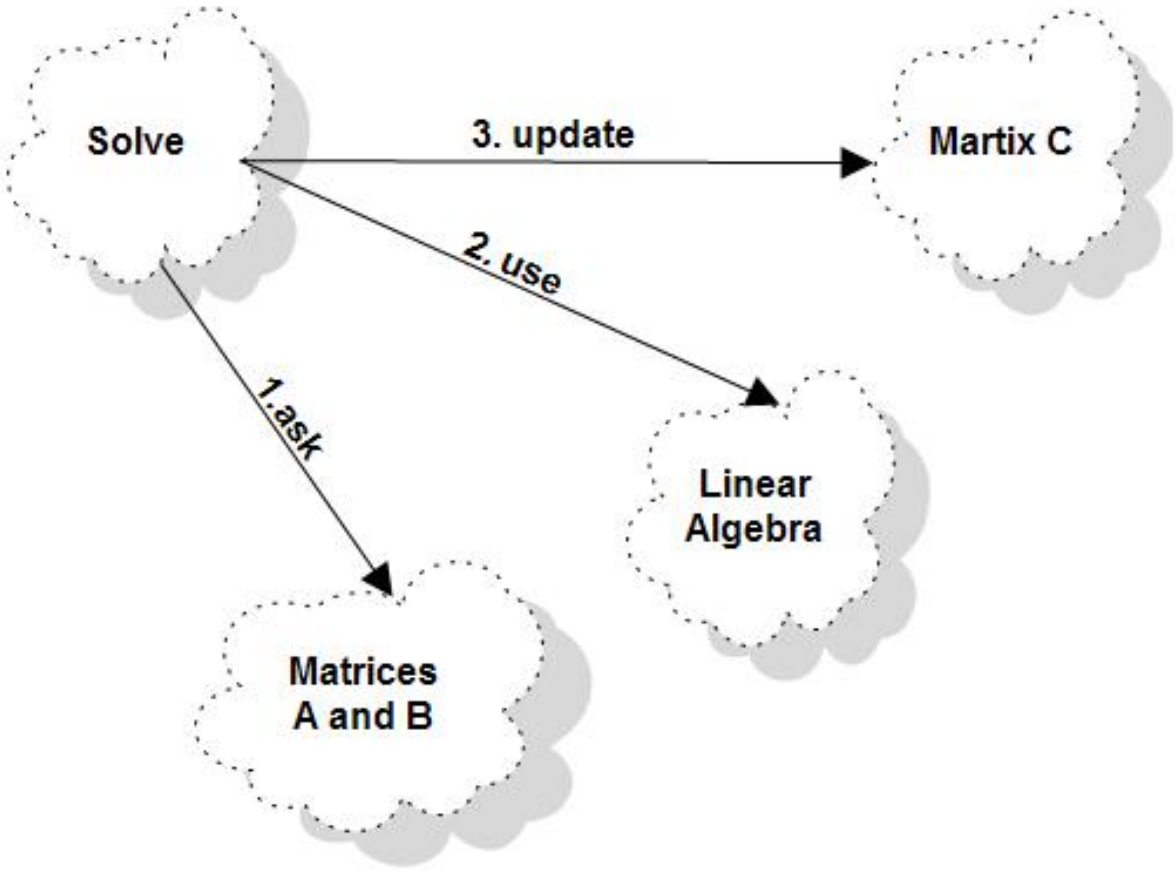

Figure 7: Operation Solve for computing C

the change of loop order can significantly improve memory utilization.The main process of this class see Fig 9

In the previous software tool, the operations, coefficient computation and solution evaluation are integrated. And the user could only evaluate the solution. There is no option to arrange the order of operations and to evaluate other properties of the solution. This class is a brand new idea and it's very nice and flexible for users.

Specification: Since the high flexibility, this class could be quite complicated. This class has two main tasks, to establish workflow shcemes and to execute all the operations according to the scheme.

The supported external interfaces are:

- A subroutine to add operations into the *data manager*.
- A manager to take care of all the operations, which includes getting other input for each operation, executing each operation and extracting outputs.

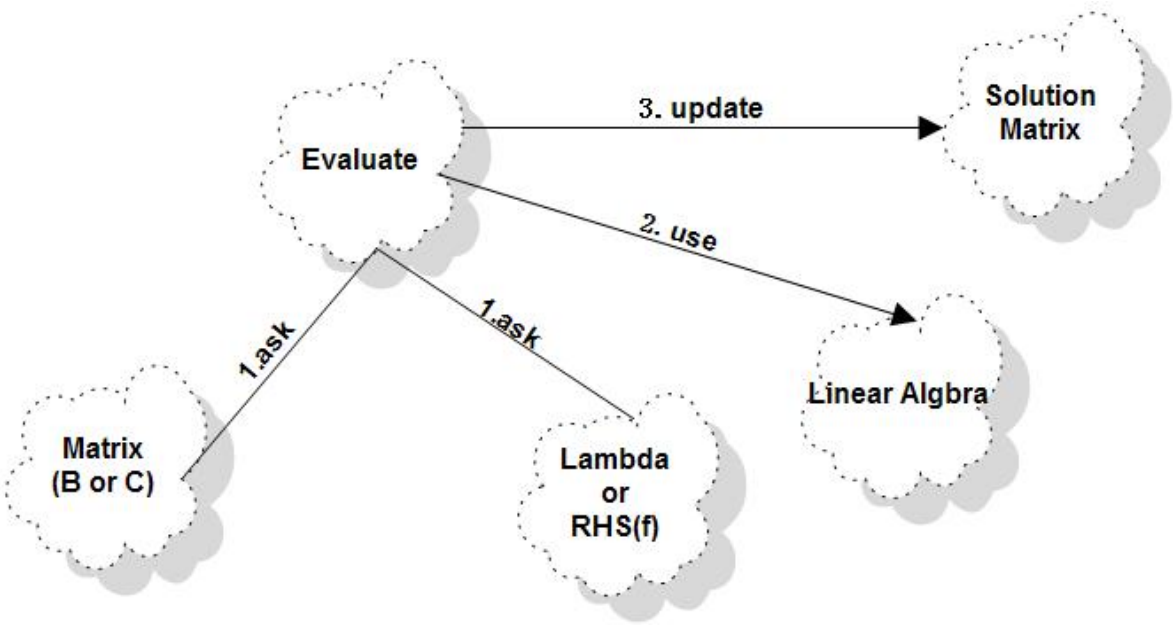

Figure 8: Operation Evaluate

4.2.10 Data manager

As indicated in the design issues section, the separation of data is to eliminate circular dependency between the *Operation* class and the workflow manager. This *Data_Manager* class is used for storing sequent operations, problem and approximation objects, and all the inputs from users and outputs from the execution results of operations.

Specification: This class does not indeed need any methods and in fact it is a data structure to comprise all the data involved in the process of simulation to PDEs.

After the analysis above, we get a preliminary structure of object models about the framework, shown in Fig 10.

4.3 Parallelization

Since a PDE solver usually involves large scale of computation, parallel computing is extremely necessary in order to advance the performance efficiency. Additionally, most computers now are parallel computers with multi-core processors, that provides the potential and feasibility for parallel programming.

There are two parallel programming models available and supported in Fortran: the shared name space model (OpenMP) and the local memory or message passing model (MPI). In this work, we prefer to add OpenMP directives for high level parallelization of independent operations. Probably, two-level or multi-level parallelization schemes will be introduced depending on the algorithm and code implementation as well as the number of cores available.

Below it is a simple example to show how to use OpenMP to implement parallel computing in the Fortran. The parallel region starts with the **!$OMP PARALLEL** and ends at **!$OMP END PARALLEL**. This region is to be executed by multiple threads in parallel. We also

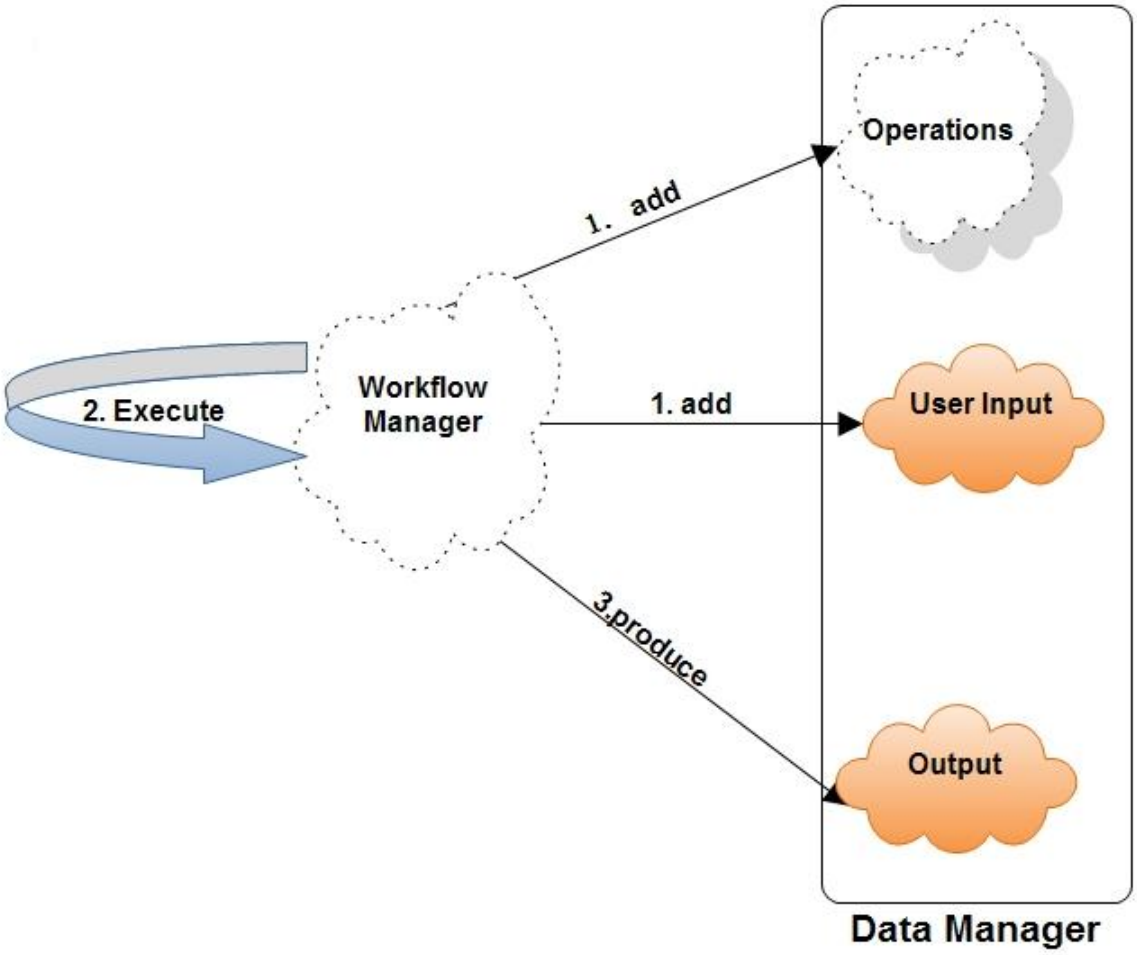

Figure 9: Workflow manager

use nested parallelism for the loops of RBFs and shape parameters to arrive to the two-level parallelization scheme. It is still possible to parallel the loop of operations synchronously.

```
subroutine execute(DM)
  .
  .

c Enable nested parallelism
  call OMP_SET_NESTED(.true.)

c Start parallel region
!$OMP PARALLEL
!$OMP DO
  do i=1,nphi      ! Loop for RBFs
   ......get RBF(phi)........
```

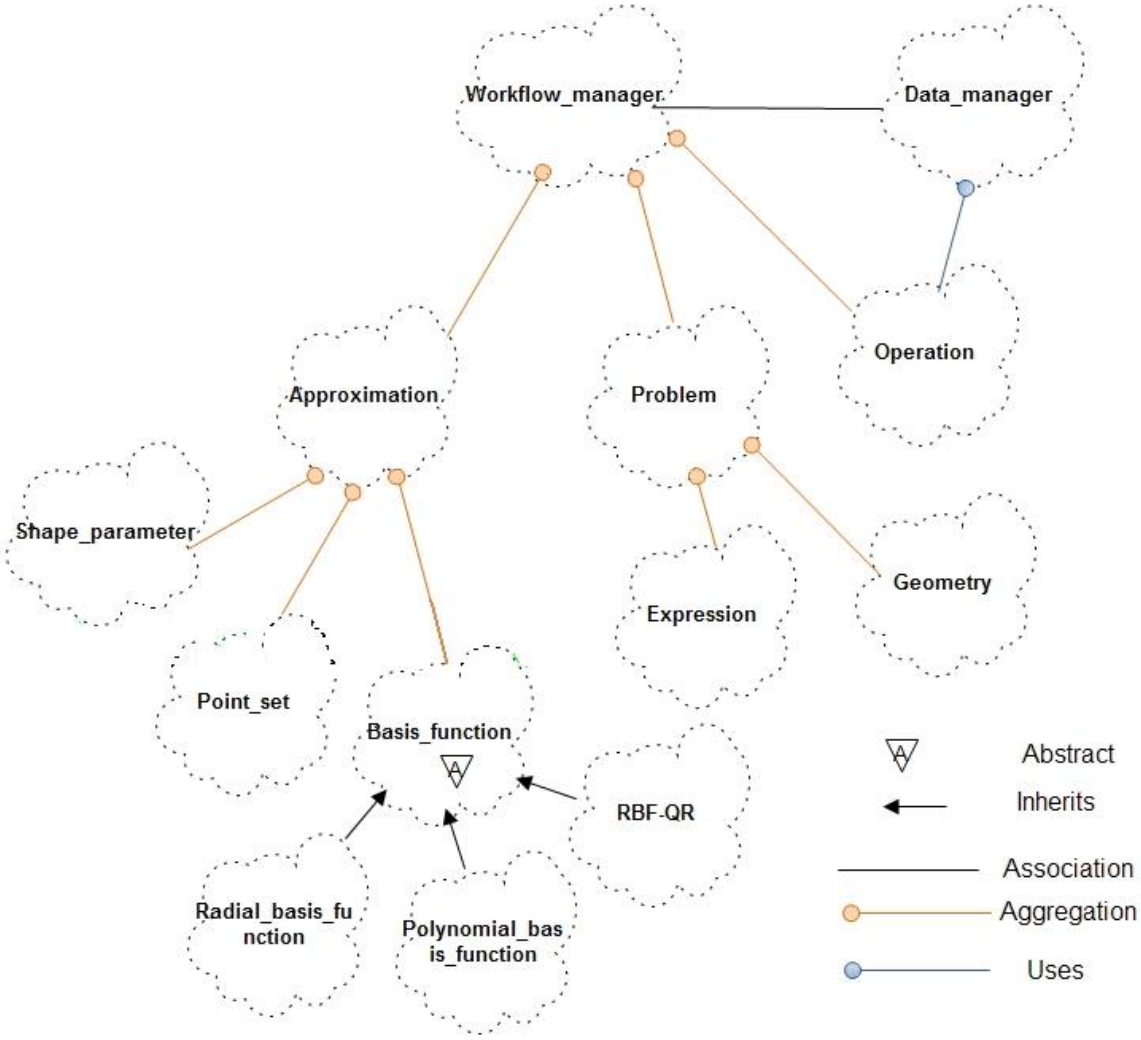

Figure 10: The overall object model

```
!$OMP PARALLEL
!$OMP DO
  do j=1,nep       ! Loop for epsilons
          .....get epsilon(ep)......


       do n=1,nop  ! Loop for operations
      .......get operation(op)......
 call execute_operation(op, phi, ep)
       end do
```

```
   end do
!$OMP END PARALLEL

  end do
!$OMP END PARALLEL

end subroutine execute
```

5 Evaluation of design

In this section, we will discuss how to construct a PDE solver in this framework using Dam seepage problem described in section 2 as a motivating example. Comparison to the previous version will be accompanied with the explanation. Through the evaluation to our design, users could feel the ease to solve a PDE problem and the flexibility to extend this framework.

5.1 New problem

As illustrated in the object model section, the problem class is the aggregate class of equations and domains. so, equations and domains need to be defined before we create the problem object.

5.1.1 New equations

We call the *new_express* function of *expression* class to create an object *(*pde) for PDEs and then add five equations representing the five conditions of Dam seepage problem (see the conditions 0-4 in section 2.3) into *pde*. See the following program 1.

```
----------------------------
     Program 1
----------------------------
type(expression) :: pde
character(len=80) :: fun
real(kind=rfp), parameter :: co_r = 1.0_rfp
!
! Create an expression object with 5 equations
pde = new_express(6)       ! Specify 6 for extension later
!
! Set a user friendly name and add two terms for equation 0
call set_equation(pde, 1, 'interior', 2)
call add_term(pde, 1, 'L', co_r)
fun = 'zero'
call add_term(pde, 1, fun)

! Set a user-friendly name and add two terms for equation 1
call set_equation(pde, 2, 'b1', 2)
call add_term(pde, 2, '0', co_r)
```

```
fun = 'y'
call add_term(pde, 2, fun)

! Set a user-friendly name and add two terms for equation 2
call set_equation(pde, 3, 'b2', 2)
call add_term(pde, 3, '0', co_r)
fun = 'Height30'
call add_term(pde, 3, fun)

! Set a user-friendly name and add three terms for equation 3
call set_equation(pde, 4, 'b3', 3)
fun = 'Normalx'
call add_term(pde, 4, 'X', co_r, COEFF_F=fun)
fun = 'Normaly'
call add_term(pde, 4, 'Y', co_r, COEFF_F=fun)
fun = 'zero'
call add_term(pde, 4, fun)

! Set a user-friendly name and add two terms for equation 4
call set_equation(pde, 5, 'b4', 2)
call add_term(pde, 5, 'Y', co_r)
fun = 'zero'
call add_term(pde, 5, fun)
```

5.1.2 New geometry object

In this test example, we use sets of points sampled in the domain and along its sides to represent geometric objects(domain and its boundaries). It's easy to create a geometry object from a outside file with categorized points via calling the *geom_new* interface as below:

```
type(geometry) :: geom
!
geom = geom_new('x.dat')
```

It also possible to create a geometry object for the domain through adding geometric entities manually by users. To simplify the code, we will create an echelon domain for the example case instead, see program 2.

```
----------------------------
     Program 2
----------------------------
type(geomPoint) :: p1, p2, p3, p4
type(geomEntity) :: l1, l2, l3, l4
type(geomDomain) :: d
type(geometry) :: geom
!
real(kind=rfp), dimension(1:2), parameter :: point1 = (/  0.0_rfp,  0.0_rfp /)
```

```
real(kind=rfp), dimension(1:2), parameter :: point2 = (/  5.0_rfp,  30.0_rfp /)
real(kind=rfp), dimension(1:2), parameter :: point3 = (/  25.0_rfp,  30.0_rfp /)
real(kind=rfp), dimension(1:2), parameter :: point4 = (/  30.0_rfp,  0.0_rfp /)

! Create square domain
p1 = geomPoint_new(point1)
p2 = geomPoint_new(point2)
p3 = geomPoint_new(point3)
p4 = geomPoint_new(point4)

l1 = geomLine_new(p3, p4, '[', ')')  ! boundary 1
l2 = geomLine_new(p1, p2, '[', ')')  ! boundary 2
l3 = geomLine_new(p2, p3, '[', ')')  ! boundary 3
l4 = geomLine_new(p4, p1, '[', ')')  ! boundary 4
d = geomDomain_new([l1, l2, l3, l4]) ! interior

geom = geom_new(5)         ! Specify 5 for extension later
call geom_setDomain(geom, 'b1', l1)
call geom_setDomain(geom, 'b2', l2)
call geom_setDomain(geom, 'b3', l3)
call geom_setDomain(geom, 'b4', l4)
call geom_setDomain(geom, 'interior', d)
```

Program 2: *First, we define four points (p1,p2,p3,p4) at the corner of the domain and then use them as endpoints to create lines (l1,l2,l3,l4) for the border of the domain. And the domain (d) is currently specified by a list of lines by creating a simple closed oriented curve. The square bracket '[]' means the endpoints are included when creating lines and not for the parentheses '()', for each point on the border of a domain can only belong to a single point or line. This is to solve the problem that if two adjacent edges have different boundary conditions, it is unspecified what happens in the common point. At last, we add the boundaries and the domain with specified names into the geometry object geom.*

5.1.3 New problem object

Once equations and their related domain (pde and geom above) are created, it is pretty simple to produce problem object to check the compatibility between the expression object and the geometry object. Namely, to find out if these two objects are correctly associated by the means of checking names of their each parts. This check operation could be done in the problem constructor.

```
type(problem) :: problem

! Check compatibility and then create a problem object
problem = new_problem(pde, geom)
```

5.2 New approximation object

Here, we show how to define an approximation for the dam seepage problem through describing separately how to create other objects of the relative classes. The following *Program 3* explains how the process is done.

5.2.1 New RBF object

For this dam seepage problem, we choose the infinitely smooth 'gauss'(GS) RBF to simulate its solution. See part a) of *Program 3*.

5.2.2 New shape parameter object

Users can define the range of shape parameter value in which they would like to test or shape parameter is well defined. Additionally, they would also need to indicate the number of shape parameter value and choose means of producing these values. Now we use real data type for shape parameter with a range of 3.5 to 4.2 and use logarithmic distribution along an interval method to produce 200 shape parameter values. See part b) of *Program 3*.

5.2.3 New point sets

We get the collocation point sets from an outside file and then produce center points according to the collocation points. By $nak = 0, dupl = .false.$, we do not use the generalized Not-a-Knot(NaK) boundary condition [13]. This condition could be imposed with a positive integer for nak and a TRUE for $dupl$. Here, nak represents the percentage of centers which are moved out of the domain. See part c) of *Program 3*.

5.2.4 Specify other information

As we have discussed in the feature section, the approximation encompasses two subfeatures: polynomial and constraint. For this example, we do not introduce polynomial and constraint to the coefficient matrices by setting the degree of polynomial to zero and assigning False to variable *constr*. See part d) of *Program 3*.

5.2.5 New approximation object

After we create all kinds of approximation-related objects or define needed information, it is of obvious simple to produce an approximation object by just invoking the constructor as done in part e) of *Program 3*.

```
----------------------------------
         Program 3
----------------------------------
!
! Declare variables first
!
logical :: dupl, logl, constr
character(len=80) :: fname
```

```
character(len=80), dimension(:), allocatable ::  phi
integer :: pdeg, nak
real(kind=rfp), dimension(1:2) :: rgx
!
type(point_set) :: x, c
type(epsilon) :: ep
type(approximation) :: approx
!
! a. Create RBFs
!
  allocate(phi(1:1))
  phi(1) = 'gauss'
!
! b. Create the set of epsilon values
!
  logl = .true.
  rgx(1) = 3.5_rfp; rgx(2) = 4.2_rfp;
  ep = new_eps(logl,rgx,200)
!
! c.Create the collocation point set, and the center points.
!  RBF boundary conditions. None are used.
  nak = 0
  dupl = .false.
  fname = 'x.dat'
  x = new_pts(fname)
  c = new_center_pts(x,.true.,nak,dupl)
!
! d. Specify the degree of the polynomial basis (if there is one)
!  And if need constraint(None).
  pdeg = -1
  constr = .false.
!
! e.Create approximation
  approx = new_approx(phi, ep, c, pdeg, constr)
!
```

5.3 New workflow manager

The third and also the last step is to build a workflow scheme for the dam seepage application. First, we call the constructor *new_WFManager* of *workflow_manager* class to create a manager and then set the entire procedure by adding operations into this manager. *Program 4* gives two different workflow schemes: one is for the commonly used standard elliptic case and the other shows the process for partition of unity case.

The only difference between standard and partition case is the third step. For the standard elliptic case, it needs to compute coefficient λ first; but in the partition of unity case, we do not work with λ explicitly and get the solution by means of matrices transformation. Figure

11 and Figure 12 separately show the main part of the workflow schemes for these two cases, from building the interpolation matrix to the evaluation of the solutions.

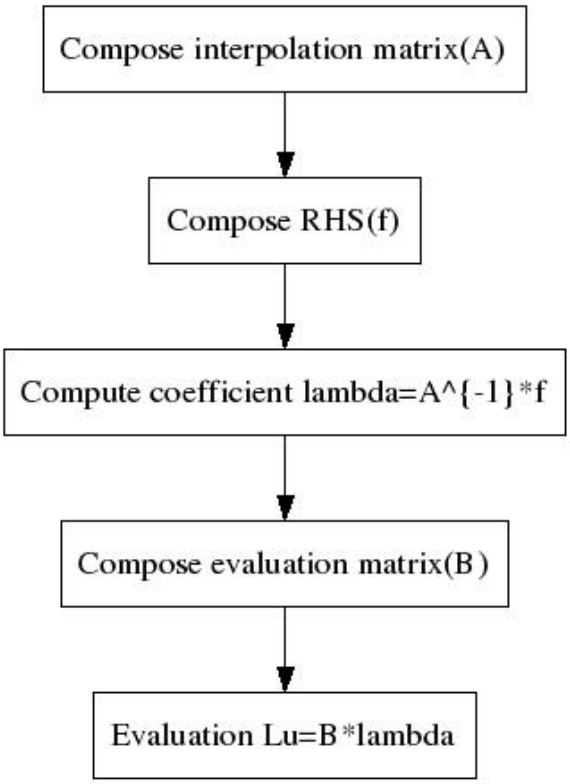

Figure 11: Workflow for standard case

```
--------------------------------------
          Program 4
--------------------------------------
type(wfmanager) :: wfm       ! Workflow manager
type(expression) :: oper        ! Evalutation operator
type(point_set) :: x, xe       ! Collocation and evaluation point sets
!
! Create a workflow manager and add operations
!
  write(*,*) 'Give the number of steps needed for the work flow: '
  read(*,*) n
  wfm = new_WFManager(prob, approx, n)
!
! example: work flow for standard elliptic case
!
  call add_operation(wfm, 'compose', 'matrixA', x)   ! compose matrix A
  call add_operation(wfm, 'compose', 'rhs')       ! compose f
```

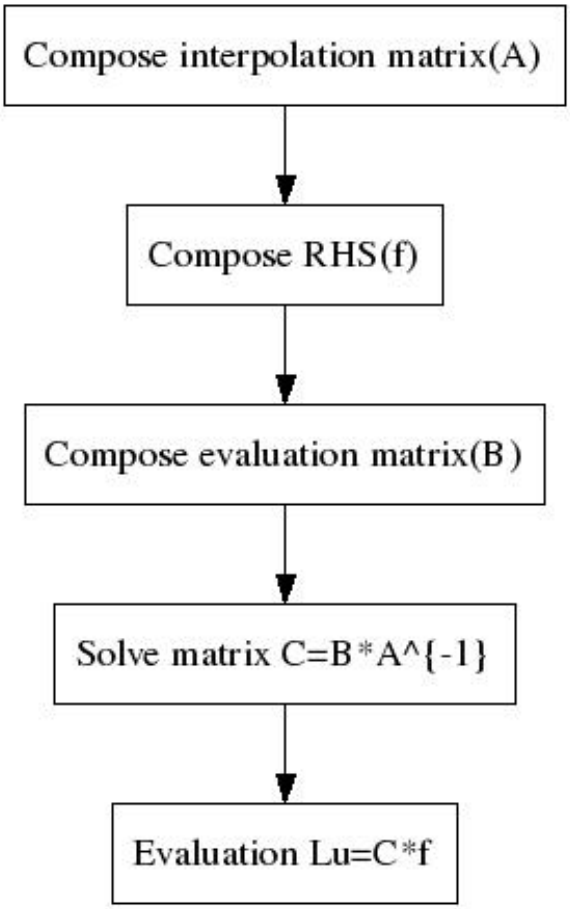

Figure 12: Workflow for partition case

```
  call add_operation(wfm, 'solve', 'lam')         ! compute lambda = f/A
  call add_operation(wfm, 'compose', 'matrixB', xe, oper)   ! compose matrix B
  call add_operation(wfm, 'evaluate', 'standard')  ! get Lu = B*lambda
  call add_operation(wfm, 'solve', 'residual')         ! get residual = Lu - f(xe)
  call add_operation(wfm, 'output', 'lambda')      ! output lambda
  call add_operation(wfm, 'output', 'solution')      ! output Lu
  call add_operation(wfm, 'output', 'error')        ! output error
!
! example: work flow for partition of unity case
!
  call add_operation(wfm, 'compose', 'matrixA', x)    ! compose matrix A
  call add_operation(wfm, 'compose', 'rhs')        ! compose u = f
  call add_operation(wfm, 'compose', 'matrixB', xe, oper)   ! compose matrix B
  call add_operation(wfm, 'solve', 'c')      !compute C = B*A(-1)
  call add_operation(wfm, 'evaluate', 'partition')    ! get Lu = C*u
  call add_operation(wfm, 'solve', 'residual')  ! get residual = Lu - f(xe), xe = xs
```

```
  call add_operation(wfm, 'output', 'solution')     ! output Lu
  call add_operation(wfm, 'output', 'error')        ! output error
!
```

5.4 Comparisons

Comparison to how to solve this motivating example in the previous code will be addressed in this section. Through the followed explanation, we could feel what's the flexibility about our new design and the limitation of this currently existing application.

This old software tool supports to solve seven sorts of problems: three of them are for Helmholtz equation with different dimensionality or domain; another three, actually are three collocation methods mentioned in [25] which are implemented as different problem models; the last one is the standard case corresponding to the three collocation methods, for PDEs with a disk unit domain.

Apparently, every time extending this software to solve a new problem, for instance dam seepage problem, users need put great effort into understanding the code and into modifying *class_operator* and *class_rhs*, that is certainly the job of developers. It is totally time-consuming and not reusable for non-expert users. *Program 5* and *Program 6* describe how to extend *class_operator* and *class_rhs* class respectively in order to construct a new PDE solver for the dam seepage problem.

```
----------------------------------------
          Program 5
----------------------------------------
function new_rhs(funcs,op,x) result(rhs)
 .
 .
 .
select case (trim(oper))
 .
 .
 case ('dam')
    if (id == 1) then     ! condition 1: h = y
       nprime = 'Y'
    else if(id == 2) then
       nprime = '0'       ! Dirichlet condition
    else
       nprime = 'L'
    end if

    if (reel) then     ! For real data type
       call evalf(funcs(k),nprime,xp(rg(1):rg(2),:), fp_r(fpos+1:fpos+xsz,k))
    else            ! For complex data type
       call evalf(funcs(k),nprime,xp(rg(1):rg(2),:), fp_c(fpos+1:fpos+xsz,k))
    end if
    fpos = fpos + xsz
```

```
  .
  .
  .
 end function new_rhs
```

Program 5: *Modify the* new_rhs *function of* class_rhs *through adding a case option for dam seepage model in order to obtain right_hand_side vector f of equation $A\lambda = f$.*

```
----------------------------------------
          Program 6
----------------------------------------
function new_operator(oper,x,c,xb,pdeg,rtype) result(op)
 .
 .
 .
select case (trim(oper))
 .
 .
 case ('dam')
   call build_dam()
 .
 .
Contains
 .
 .
!
!----------- Build matrices for Dam Seepage model ----------------
!
subroutine build_dam()
    integer :: k,curr,nn
    character(len=80) :: fname
    type(point_set) :: norm
    real(kind=rfp), dimension(:,:), pointer :: np
!
    co_r = 1.0_rfp
!
! get the normal vector from file
!
    fname = 'nhat.dat'
    norm = new_pts(fname)
    np => get_pts_ptr(norm)
    nn = ubound(np,1)
!
    allocate(op%bl_A(1:nxt))
    allocate(op%bl_B(1:1))
    ! All columns in each block
```

```
mc(1) = psize+1; mc(2) = nc + psize
rc(1) = 1; rc(2) = nc

do k=1,nxt
  ! extract information from xtype to get r and rr
  call get\_pts\_id(k,x,curr,rr)
  mr = rr + psize
  if (curr==3*BOUNDARY\_POINT) then
    op%bl_A(k) = new_block(2,mr,mc,rr,rc) ! 2 terms for Neumannn type
  else
    op%bl_A(k) = new_block(1,mr,mc,rr,rc) ! 1 term for other points
  end if
  !
  ! Compute the interpolate matrix A
  !
  if (curr==INTERIOR_POINT) then            ! For left side of equation 0
    op%bl_A(k)%terms(1) = new_term('L',1,co_r)
  else if (curr==BOUNDARY_POINT) then       ! For left side of equation 1
    op%bl_A(k)%terms(1) = new_term('0',1,co_r)
  else if (curr==2*BOUNDARY_POINT) then     ! For left side of equation 2
    op%bl_A(k)%terms(1) = new_term('0',1,co_r)
  else if (curr==3*BOUNDARY_POINT) then     ! For left side of equation 3
  ! dh/dn = n1*(dh/dx) + n2*(dh/dy)
    dim(1:2) = (/1,2/)
    prime(1:2) = (/1,1/)
    op%bl_A(k)%terms(1) = new_term(dim(1:1),prime(1:1),nn,np(:,1))
    op%bl_A(k)%terms(2) = new_term(dim(2:2),prime(2:2),nn,np(:,2))
  else if (curr==4*BOUNDARY_POINT) then     ! For left side of equation 4
  ! dh/dy
    dim(1) = 2
    prime(1) = 1
    op%bl_A(k)%terms(1) = new_term(dim(1:1),prime(1:1),1,co_r)
  end if
  !
  ! Compute the corresponding block in the polynomial part
  !
  if (psize > 0) then ! We know that the op is complex
    call fill_poly_block_r('n',op%pbase_a%r%r2(rr(1):rr(2),:), &
                          xp(rr(1):rr(2),:),op%bl_A(k)%terms)
  end if
end do

mr(1) = 1; mr(2) = nxb
mc(1) = psize+1; mc(2) = nc + psize
rr(1) = 1; rr(2) = nxb
rc(1) = 1; rc(2) = nc
```

```
    op%bl_B(1) = new_block(1,mr,mc,rr,rc)
    op%bl_B(1)%terms(1) = new_term('0',1,co_r)

    !
    ! Compute the constraint block and the polyblock in evaluation matrix B
    !
    if (psize > 0) then
      call fill_poly_block_r('y',op%pconstr%r%r2, cp,constr)
      call fill_poly_block_r('n',op%pbase_b%r%r2,xbp,op%bl_B(1)%terms)
    end if

   end subroutine build_dam
 ! ------------------------------------------------------------
   .
   .

end function new_operator
```

Program 6: *Modify the* new_operator *function of* class_operator *through adding a case option for dam seepage model in order to obtain interpolate matrix A of equation $A\lambda = f$ and evaluation matrix B.*

The extension for solving the dam seepage problem seems easier in the old code than the one in our new design, but this is a false image. We can see that a few subroutines or functions are invoked by *new_rhs* and *new_operator* function, that means, as we have already discussed at the beginning, we must be exactly clear about the roles of these subroutines or functions which are likely to be private. However, according to our suggested design, the user just needs to learn how to use public interfaces which are extremely easy to know. The following four aspects succeed in extending the flexibility:

1. Extend pre-existing objects. We could add new equations and geometric objects into existing PDEs and geometries to arrive to new expression and geometry objects. For example, add an initial condition for the problem and add a decomposed subdomain.

   ```
   ! Set a friendly name and add two terms for equation 6
   call set_equation(pde, 6, 'init', 2)
   call add_term(pde, 6, '0', co_r, COEFF_F= f(x,t))
   fun = 'zero'
   call add_term(pde, 6, fun)


   ! Set a friendly name and add two terms for equation 6
   type(domain) :: d2

   d2 = new_domain(3)
   .
   .
   ```

```
.
call geom_setDomain(geom, 'subdomain', d2)
```

2. Allow to change object features. As expressed in 1), we add a time-dependent equation this time, that makes the time-independent equation systems or PDEs into time-dependent.

3. Reuse objects. Here, we do not mean to extend the objects as 1), but to use existed objects for new problem. Now, for instance, there are one geometry object G1 and two expression objects PDE1, PDE2 which are equations for different problem but with the same domain. So we could combine G1 with PDE1 and PDE2 separately to cover two problem models.

4. Introduction of the workflow manager. Users have the initiative to set their desired workflow through adding orderly operations into workflow manager. In addition, it also allows the introduction of new operations for instance time-stepping method.

6 Implementation

We use Fortran 90 as implementation language which is competitive in scientific computation. Now, most of the new classes are not finished yet and they could be implemented diversely according the interfaces we suggested. In order to improve the computational efficiency, we read and write data using a column-oriented method that is in line with the memory management mechanism of Fortran 90. And to avoid data re-computing, e.g. distances between node points and center points, we introduce a data structure *Data_manager* to manage all of the data. In terms of implementation, we separate this Data_manager from *workflow_manager* in case the *workflow_manager* and the *class_operation* call each other and cause a cyclic dependency.

7 Conclusion

First, in this paper, we have presented object models of our framework by application of object-oriented analysis and design(OOAD) combined with feature modeling. Then we use a dam seepage problem to discuss how to construct a RBF based PDE solver according to these models. Moreover, a comparison was made between our new design and the existing reference code to show how flexible and reusable the proposed design is.

The separation of mathematic domains and numerical domains and the introduction of operations provide greater possibility for users with different concerns to extend this framework. Problem oriented users could vary the features of objects by using public interfaces and changing their relevant parameter values; however, method oriented users are allowed to add new method classes or operations into their corresponding component to extend the framework.

In the future, we could pursue higher degree of computational efficiency via parallelization computing. Also, we can consider how to more reasonably separate classes into multi-level and maximumly avoid users to access low level classes. Furthermore, we could aim to extend the code to allow for equations with derivatives regarding to time. Or to investigate the RBF method for nonlinear partial differential equations, which needs a totally new solver. But some

of the basic classes or components are reusable. We could extend the existing framework to construct new PDE solvers through combination of old classes and new developed classes, for instance to extend the *operation* class. To construct a reusable and flexible framework is always a permanent aim.

References

[1] http://www.nobjects.com/Diffpack.

[2] http://en.wikipedia.org/wiki/Software_framework.

[3] http://en.wikipedia.org/wiki/Partial_differential_equation.

[4] http://en.wikipedia.org/wiki/Dam.

[5] http://en.wikipedia.org/wiki/Least_squares.

[6] http://en.wikipedia.org/wiki/Galerkin_method.

[7] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. II. Implementation and tests in DUNE*, Computing, 82 (2008), pp. 121–138.

[8] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. I. Abstract framework*, Computing, 82 (2008), pp. 103–119.

[9] D. L. Brown, W. D. Henshaw, and D. J. Quinlan, *Overture: An object-oriented framework for solving partial differential equations on overlapping grids*, in Object oriented methods for interoperable scientific and engineering computing: proceedings of the 1998 SIAM workshop, Society for Industrial and Applied Mathematics, May 1999, pp. 245–265.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Pattern*, Wiley, Chichester, 1996.

[11] K. Czarnecki and U. W. Eisenecker, *Generative Programming: methods, tools and applications*, Addison-Wesley, 2000.

[12] A. I. Fedoseyev, M. J. Friedman, and E. J. Kansa, *Improved multiquadric method for elliptic partial differential equations via PDE collocation on the boundary*, Comput. Math. Appl., 43 (2002), pp. 439–455. Radial basis functions and partial differential equations.

[13] B. Fornberg, T. A. Driscoll, G. Wright, and R. Charles, *Observations on the behavior of radial basis function approximations near boundaries*, Comput. Math. Appl., 43 (2002), pp. 473–490. Radial basis functions and partial differential equations.

[14] B. Fornberg, E. Larsson, and N. Flyer, *Stable computations with gaussian radial basis functions in 2-d*, Tech. Rep. 2009-020, Dept. of Information Technology, Uppsala Univ.,Uppsala, Sweden, 2009.

[15] B. Fornberg and G. Wright, *Stable computation of multiquadric interpolants for all values of the shape parameter*, Comput. Math. Appl., 48 (2004), pp. 853–867.

[16] P. González-Casanova, J. A. Muñoz-Gómez, and G. Rodríguez-Gómez, *Node adaptive domain decomposition method by radial basis functions*, Numer. Methods Partial Differential Equations, 25 (2009), pp. 1482–1501.

[17] B. Grady, M. Robert, E. Michael, Y. Bobbi, C. Jim, and H. Kelli, *Object-oriented analysis and design with applications*, Addison-Wesley Professional, 3 ed., 2007.

[18] R. D. Hornung and S. R. Kohn, *Managing application complexity in the samrai object-oriented framework*, Concurrency and computation: practice and experience, 14 (2002), pp. 347–368.

[19] H.P.Langtangen, *Computational partial differential equations cnumerical methods and diffpack programming*, Springer, (1999). an enlarged 2nd edition is to be published in 2002.

[20] J. I., G. M., and J. P., *Software reuse: architecture, process and organization for business success*, ACM Press, 1997.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, *Feature-oriented domain analysis FODA feasibility study*, tech. rep., Carnegie Mellon University, November 1990.

[22] E. J. Kansa, *Multiquadrics—a scattered data approximation scheme with applications to computational fluid-dynamics. I. surface approximations and partial derivative estimates*, Comput. Math. Appl., 19 (1990), pp. 127–145.

[23] E. J. Kansa, *Multiquadrics—a scattered data approximation scheme with applications to computational fluid-dynamics. II. solutions to parabolic, hyperbolic and elliptic partial differential equations*, Comput. Math. Appl., 19 (1990), pp. 147–161.

[24] A. Karageorghis, C. S. Chen, and Y.-S. Smyrlis, *Matrix decomposition RBF algorithm for solving 3D elliptic problems*, Eng. Anal. Bound. Elem., 33 (2009), pp. 1368–1373.

[25] E. Larsson and B. Fornberg, *A numerical study of some radial basis function based solution methods for elliptic PDEs*, Comput. Math. Appl., 46 (2003), pp. 891–902.

[26] K. Lee, K. C. Kang, and J. Lee, *Concepts and guidelines of feature modeling for product line software engineering*, in Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7, Springer Berlin / Heidelberg, 2002, pp. 62–77.

[27] M. Ljungberg, K. Otto, and M. Thune, *Design and usability of a pde solver framework for curvilinear coordinates*, Advances in Engineering Software, 37 (2006), pp. 814–825.

[28] A. A. Oberai, M. Malhotra, and P. M. Pinsky, *On the implementation of the Dirichlet-to-Neumann radiation condition for iterative solution of the Helmholtz equation*, Appl. Numer. Math., 27 (1998), pp. 443–464. Absorbing boundary conditions.

[29] K. Åhlander, *An object-oriented framework for PDE solvers*, PhD thesis, Uppsala University, 1999.

[30] S. A. Sarra and D. Sturgill, *A random variable shape parameter strategy for radial basis function approximation methods*, Eng. Anal. Bound. Elem., 33 (2009), pp. 1239–1245.

[31] E. J. Sellountos, A. Sequeira, and D. Polyzos, *Elastic transient analysis with MLPG(LBIE) method and local RBFs*, CMES Comput. Model. Eng. Sci., 41 (2009), pp. 215–241.

[32] M. Thune, E. Mossberg, P. Olsson, J. Rantakokko, K. Åhlander, and K. Otto, *Object-oriented construction of parallel pde solvers*, Modern Software Tools for Scientific Computing, (1997), pp. 203–226.

[33] D. VK, N. CD, and S. BK, *Expressing object-oriented concepts in fortran90*, In: ACM Fortran Forum, (1997), p. 13.

[34] Z. M. Wu, *Hermite-Birkhoff interpolation of scattered data by radial basis functions*, Approx. Theory Appl., 8 (1992), pp. 1–10.

[35] R. Wyrzykowski, T. Olas, and N. Sczygiol, *Object-oriented approach to finite element modeling on cluster*, Springer-Verlag Lecture Notes in Computer Science, 1947/2001 (2001), pp. 250–257.