3
Theoretical background

In the previous chapter, we introduced the smart well technology concepts and identified the problem of value its flexibility in the presence of uncertainty. We described how the existing methodologies seek to address the problem, pointing out the importance of considering available information to reduce uncertainty and consequently make better decisions.

In this chapter, we present a theoretical background in order to provide a formal basis for the approach proposed in this thesis. We begin by describing dynamic programming (DP), which can be used to provide an exact valuation in the context of uncertainty and flexibility, incorporating future information. Then explain approximated dynamic programming that can handle this problem with sufficient speed and computational efficiency to make it practical, being an alternative that offers a powerful set of strategies for complex problems that require an extremely large number of evaluations and cannot be addressed by dynamic programming.

Expensive-to-evaluate functions are often encountered in industrial optimization problems, where the function value may be the output of complex computer simulations, such as reservoir simulations. For this reason, we also describe in this chapter existing theories and ideas to optimize expensive functions with a reduced number of evaluations, including optimizations based on proxies.

3.1. Dynamic programming

Optimization problems are ubiquitous in many knowledge areas, and in some cases the decision need to be make under uncertainty conditions and later, as uncertainty are resolved over a project's lifetime, this need to be remake. We can find example in sequential decision problems that can be easy to model but very difficult to solve (Powell, 2011). According to Barreto (2008), the main characteristic of this kind of problems is the decisions has a cumulative effect, i.e., the consequence of one specific action can be extended for an undefined time interval. So in sequential decision problems at the time of making the decisions we have to consider the information available. The DP is a standard solution method developed for sequential decision problems, based on Bellman's Principle of Optimality, that affirms: “An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision” (Bellman, 1957).

DP is recognized as a solution model to solve sequential decision problems (Barreto, 2008). A “*stage*” is defined as a step of a sequential process. In the end this sequential process there are alternatives called “*decisions*” and inside each *stage* there is the process condition called “*state*”. Each *stage* needs a *decision* to be taken that may or may not change the process *state.* This implies a *state* transition, between the present and the future. Figure 3.1 shows schematically the relationship between *stages, states* and *decisions*.

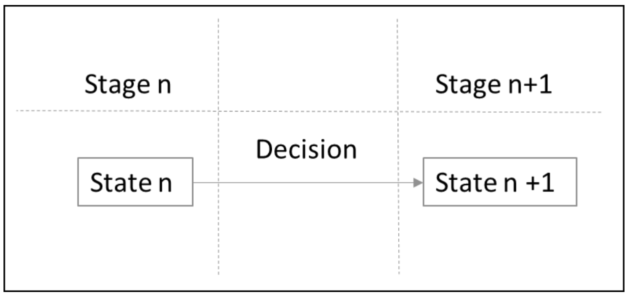


Figure 3.1: Decision process in stages. Schematic relationship between stages, states and decisions

This methodology is an optimization approach that transforms a complex problem into a sequence of simpler problems (Dasgupta, 2006); its essential characteristic is the multistage nature of the optimization procedure. DP provides a general framework for analyzing many problem types, where a variety of optimization techniques can be employed to solve particular aspects of a more general formulation.

Using DP, each problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until all problems solved. So, according to Bertsekas (2007), we can take as an example one problem *P*, that can be divided in 2 subproblems, P_1 and P_2, considering that there is dependencies between the subproblems: if to solve subproblem P_2 we need the answer of the subproblem P_1, then there is a conceptual edge from P_1 to P_2. In this case, P_1 is thought of as a smaller subproblem than P_2 — and it will always be smaller.

Powell (2011) explains Bellman's equation supposing to solve a deterministic shortest path problem where S_t is the index of the node in the network where we have to make a decision. If we are in state $S_t = i$ (i.e., we are at node *i* in our network) and take action $a_t = j$ (i.e., we wish to traverse the link from *i* to *j*), the transition function will tell us that we are going to land in some state as $S_{t+1} = S^M(S_t, a_t)$ (in this case, node *j*). What if we had a function $V_{t+1}(S_{t+1})$ that told us the value of being in state S_{t+1} (giving us the value of the path from node *j* to the destination)? We could evaluate each possible action at and simply choose the action at that has the largest one-period contribution, $C_t(S_t, a_t)$, plus the value of landing in state $S_{t+1} = S^M(S_t, a_t)$, which we represent using $V_{t+1}(S_{t+1})$. Since this value represents the money we receive one time period in the future, we might discount this by a factor γ. In other words, we have to solve

$$a_t^*(S_t) = \arg\max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma V_{t+1}(S_{t+1}) \right) \tag{3.1}$$

where "arg max" means that we want to choose the action at that maximizes the expression in parentheses. We also note that S_{t+1} is a function of S_t and a_t, meaning that we could write it as $S_{t+1}(St, a_t)$. Both forms are fine. It is common to write S_{t+1} alone, but the dependence on S_t and at needs to be understood.

The value of being in state *St* is the value of using the optimal decision $a_t^*(S_t)$. The optimality equation for deterministic problems is

$$\begin{aligned} V_t(S_t) &= \max_{a_t \in \mathcal{A}_t} \left(C_t(S_t, a_t) + \gamma V_{t+1}(S_{t+1}(S_t, a_t)) \right) \\ &= C_t(S_t, a_t^*(S_t)) + \gamma V_{t+1}(S_{t+1}(S_t, a_t^*(S_t))). \end{aligned} \tag{3.2}$$

When we are solving stochastic problems, we have to model the fact that new information becomes available after we make the decision a_t. The result can be uncertainty both in the contribution earned and in the determination of the next state we visit, S_{t+1}.

For the uncertainty problems, we can work out the probability that S_{t+1} will take on a particular value. These probabilities depend on S_t and a_t, so we write the probability distribution as $\mathbb{P}(S_{t+1} | S_t, a_t)$ = probability of S_{t+1} given S_t and a_t. We can then modify the deterministic optimality equation (in eq. 3.2) by simply adding an expectation, giving us

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \Big(C_t(S_t, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(S_{t+1} = s' | S_t, a_t) V_{t+1}(s') \Big). \tag{3.3}$$

We refer to this as the standard form of Bellman's equations, since this is the version that is used by virtually every textbook on stochastic, dynamic programming. An equivalent form that is more natural for approximate dynamic programming is

$$V_t(S_t) = \max_{a_t \in \mathcal{A}_t} \Big(C_t(S_t, a_t) + \gamma \mathbb{E}\{ V_{t+1}(S_{t+1}(S_t, a_t, W_{t+1})) | S_t \} \Big), \tag{3.4}$$

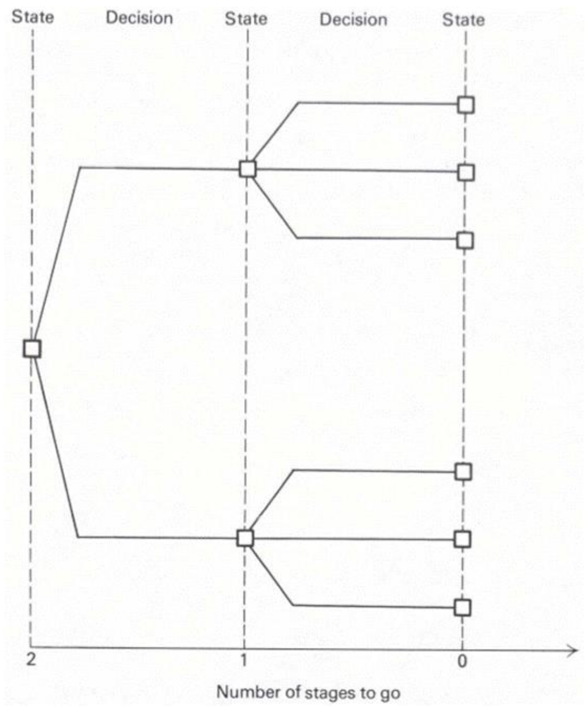
where we simply use an expectation instead of summing over probabilities. We refer to this equation as the expectation form of Bellman's equation.

The Figure 3.2 (by Bertsekas, 2007) depicts dynamic programming as a decision tree, comparing deterministic dynamic programing and dynamic programing under uncertainty, where squares represent states where decisions have to be made and circles represent uncertain events whose outcomes are not under the control of the decision maker. These diagrams can be quite useful in analyzing decisions under uncertainty if the number of possible states is not too large. The decision tree provides a pictorial representation of the sequence of decisions, outcomes, and resulting states, in the order in which the decisions must be made and the outcomes become known to the decision maker. Unlike deterministic DP wherein the optimal decisions at each stage can be specified at the outset, in dynamic programming under uncertainty, the optimal decision at each stage can be selected only after we know the outcome of the uncertain event at the previous stage. At the outset, all that can be specified is a set of decisions that would be made contingent on the outcome of a sequence of uncertain events. In DP under uncertainty, since the stage and resulting stage may both be uncertain at each stage, we cannot simply optimize the sum of the stage-return functions. Rather, we must optimize the expected return over the stages of the problem, taking into account the sequence in which decisions can be made and the outcomes of uncertain events become known to the decision maker. In this situation, backward induction can be applied to determine the optimal strategy, but forward induction cannot.

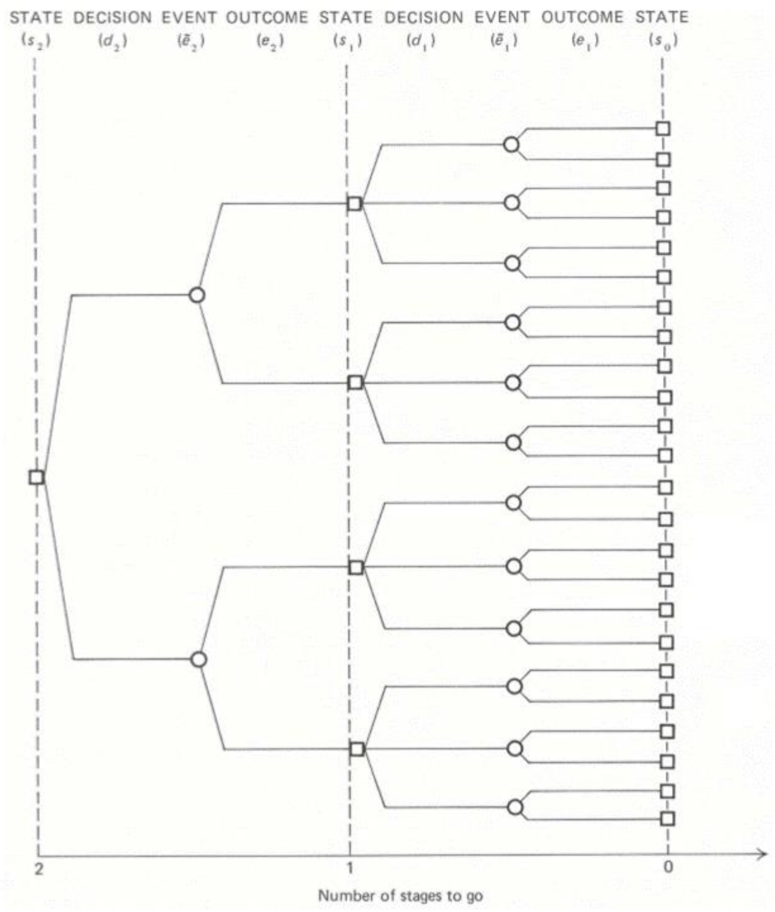
The difficulty with forward induction is that it is impossible to assign values to states at the next stage that are independent of the uncertain evolution of the process from that future state on. With backward induction, on the other hand, no such difficulties arise since the states with zero stages to go are evaluated first, and then the states with one stage to go are evaluated by computing the expected value of any decision and choosing optimally.

The backward induction process starts by computing the optimal-value function at stage 0. This amounts to determining the value of ending in each possible stage with 0 stages to go. This determination may involve an optimization problem or the value of the assets held at the horizon. Next, we compute the optimal-value function at the previous stage. To do this, we first compute the expected value of each uncertain event, weighting the stage return plus the value of the resulting state for each outcome by the probability of that outcome. Then, for each state at the previous stage, we select the decision that has the maximum expected value. Once the optimal-value function for stage 1 has been determined, we continue in a similar manner to determine the optimal-value functions at prior stages by backward induction.

The application of DP is limited by the curse of dimensionality (Bellman, 1957). This approach requires many iterations, accounting for all possible solutions for the optimization problem. The number of stages to be considered grows as we increase the number of input variables, also requiring for memory space to save all the answers in each stage. Therefore, DP is only numerically treatable if the problem has small sets of actions, small state space and easily computable expectations.



(a) Decision tree for deterministic dynamic programming



(b) Decision tree for dynamic programming under uncertainty

Figure 3.2: Decision tree for deterministic and uncertainty problems (Bertsekas, 2007).

3.2.
Approximate Dynamic Programming

In order to avoid the course of dimensionality, the approximate dynamic programming (ADP) approach tries to combine the best elements of classic dynamic programming with heuristics to adequately approximate functions that represent the future stages. The main difference between DP and ADP is the optimality criteria. The ADP emphasizes the search for a reasonably high value, not necessarily the optimum value, since it ensures proximity, which allows finding good solutions within an acceptable simulation range.

According to Powell (2011), the foundation of ADP is based on an algorithmic strategy that steps forward through time, and the author describes this approach through an example as follows. Suppose that we want to solve a problem using classical dynamic programming, we would have to find the value function $V_t(S_t)$ using

$$\begin{aligned} V_t(S_t) &= \max_{a_t \in \mathcal{A}_t} \left(C(S_t,\, a_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}\right) \\ &= \max_{a_t \in \mathcal{A}_t} \left(C(S_t,\, a_t) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}(s'|S_t,\, a_t) V_{t+1}(s')\right) \end{aligned} \tag{3.5}$$

for each value of S_t. The maximization problem is written as one of choosing the best $a_t \in A_t$. A_t depends on the state variable S_t and the dependence is expressed by indexing the action set by time t.

But we find some issues in solving the previous problem using DP, because it requires that to loop over all possible states (in fact we usually have three nested loops, two of which require that we enumerate all states while the third enumerates all actions). With approximate dynamic programming we step forward in time. In order to simulate the process forward in time, we need to solve two problems. The first is that we need a procedure to randomly generate a sample of what might happen (in terms of the various sources of random information). The second is that we need a procedure to make decisions. We start with the problem of making decisions first, and then turn to the problem of simulating random information.

When we used exact dynamic programming, we stepped backward in time, exactly computing the value function that will be used to produce optimal decisions. When we step forward in time, we have not computed the value function, so we have to turn to an approximation in order to make decisions. Let $\overline{V}_t(S_t)$ be an approximation of the value function. This is easiest to understand if we assume that we have an estimate $\overline{V}_t(S_t)$for each state S_t, but since it is an approximation, we may use any functional form we wish.

ADP proceeds by estimating the approximation $\overline{V}_t(S_t)$ iteratively. We assume we are given an initial approximation $\overline{V}_t^0$ for all t (we often use $\overline{V}^0{}_t = 0$ when we have no other choice). Let $\overline{V}^{n-1}{}_t$ be the value function approximation after $n-1$ iterations, and consider what happens during the nth iteration. We are going to start at time $t = 0$ in state S_0. We determine a_0 by solving

$$\begin{aligned} a_0 &= \arg\max_{a\in\mathcal{A}_0} \left(C(S_0,\, a) + \gamma\mathbb{E}\{\overline{V}_1(S_1)|S_0\}\right) \\ &= \arg\max_{a\in\mathcal{A}_0} \left(C(S_0,\, a) + \gamma \sum_{s'\in\mathcal{S}} \mathbb{P}_0(s'|S_0,\, a)\overline{V}_1(s')\right) \end{aligned} \tag{3.6}$$

where a_0 is the value of a that maximizes the right-hand side of eq. 3.6, and as $\mathbb{P}$ $(s'|S_0,a)$ is the one-step transition matrix (which we temporarily assume is given). Assuming this can be accomplished (or at least approximated), we can use eq. 3.6 to determine a_0.

We are now going to step forward in time from S_0 to S_1. This starts by implementing the decision a_0, which we are going to use our approximate value function. Given our decision, we next need to know the information that arrived between $t = 0$ and $t = 1$. At $t = 0$, this information is unknown, and therefore random. Our strategy will be to simply pick a sample realization of the information at random, a process that is often referred to as Monte Carlo simulation. Monte Carlo simulation refers to the popular practice of generating random information, using some sort of artificial process to pick a random observation from a population.

Using our random sample of new information, we can now compute the next state we visit, given by S_1. We do this by assuming that we have been given a transition function, which we represent using

$$S_{t+1} = S^M(S_t,\ a_t,\ W_{t+1}) \tag{3.7}$$

where W_{t+1} is a set of random variables representing the information that arrived between t and t+1. Keeping in mind that we assume that we have V_t for all t, we simply repeat the process of making a decision by solving

$$a_1 = \arg\max_{a \in \mathcal{A}_1} \left(C(S_1,\ a) + \gamma \sum_{s' \in \mathcal{S}} \mathbb{P}_1(s'|S_1,\ a)\bar{V}_2(s') \right) \tag{3.8}$$

Once we determine a_1, we, again, sample the new information, compute S_2, and repeat the process. Whenever we make a decision based on a value function approximation (as we do in eq. 3.8), we refer to this as a greedy strategy. This term, which is widely used in DP, can be somewhat misleading since the value function approximation is trying to produce decisions that balance rewards now with rewards in the future. Fundamental to ADP is the idea of following a sample path. A sample path refers to a particular sequence of exogenous information.

The ADP algorithm depends on having access to a sequence of sample realizations of our random variable. How this is done depends on the setting. There are three ways that we can obtain random samples:

1. Real world. Random realizations may come from real physical processes. For example, estimating average demand using sequences of actual demands. We may also be trying to estimate prices, costs, travel times, or other system parameters from real observations;
2. Computer simulations. The random realization may be a calculation from a computer simulation of a complex process. The simulation may be of a physical system such as the fluid flow in the porous in an oil reservoir. Some simulation models can require extensive calculations (a single sample realization could take hours or days on a computer);
3. Sampling from a known distribution. This is the easiest way to sample a random variable. We can use existing tools available in most software languages or spreadsheet packages to generate samples from

standard probability distributions. These tools can be used to generate many thousands of random observations extremely quickly.

In this thesis, we will consider geological realizations as uncertainty scenarios, using numerical reservoir simulations to evaluate them. It is known that the use of reservoir simulations as part of the evaluation function can lead to expensive function evaluations. For that reason, the next section addresses the optimization of expensive to evaluate functions.

3.3. Expensive-to-evaluate functions

Expensive-to-evaluate functions are often encountered in industrial optimization problems, where the function value may be the output of complex computer simulations, or the result of costly measurements on prototypes (Villemonteix *et al.*, 2008). This is the case of reservoir development optimization, where the optimal decisions cannot necessarily be made by intuitive judgement. In such cases automated optimization is an option, however numerical reservoir models are most often computationally expensive making direct optimization prohibitive in terms of CPU requirements (Guyaguler & Horne, 2001). The literature, we can find many global optimization methods commonly applied to solve reservoir decision problems. However, is known that global optimization of continuous black-box functions that are costly (computationally expensive, CPU-intensive) to evaluate is a challenging problem.

As we described in the previous section, the ADP approach tries to find a reasonable solution for optimization problems, not necessarily the optimum solution, and for that reason, we can choose to use approximation methods to optimize instead of traditional methods. Thus, as these simulation-based models can take hours or days to evaluate, optimization of such models, which necessarily requires hundreds of objective function evaluations, can be considerably costly in both time and CPU requirement, especially when constraints are included. It is not the aim of this thesis to develop novel optimization algorithms, but to find the optimum flow control for smart wells we want to use an optimization method that successfully addresses expensive-to-evaluate functions.

We agree with Rashid *et al.* (2013) that while the means to mitigate the high computational burden expected in the optimization process have been previously explored through the use of proxy models (Regis & Shoemaker 2007b), less attention has been paid to the treatment of expensive nonlinear constraints (Regis & Shoemaker 2005). These are simulation-dependent nonlinear constraints that can only be obtained as a consequence of the outcome of the simulation model or one of its sub-systems. Thus, these constraints are also non-trivial and expensive to evaluate, and must therefore be suitably modelled alongside the objective function.

Several approaches based on response surface techniques, most of which utilize every computed function value, have been developed over the years (Holmstrom *et al.*, 2008). According to Villemonteix *et al.* (2008), to optimize an expensive-to-evaluate function, a common approach is to use a cheap approximation of this function, which can lead to significant savings over traditional methods. Sampling methods and approximation methods have been intensively studies in recent years. Pan *et al.* (2014) affirm that one of the most effective approximation methods is the Radial Basis Functions (RBF) interpolation that has been gaining popularity for model approximation because of its simplicity and accurate results for interpolation problems. Holmstrom *et al.* (2008), Rashid *et al.* (2013) and Pan *et al.* (2014) agree that RBF is becoming a better choice for constructing approximated models or finding the global optima of computationally expensive functions by using a limited number of sampling points. Rashid *et al.* (2013) present a method to treat expensive black-box simulation-based nonlinear constrained optimization problems, including integer variables, using adaptive radial-basis function (RBF) approximations, as follows.

An expensive function can be approximated using a radial basis function model and a dataset, and according to Rashid *et al.* (2013) we can conceptually define the RBF model as an input layer of nodes (equal to the dimensionality of the problem), a hidden layer of nodes (equal to the number of samples available) and a single output node (see Figure 3.3). The approximating function is a linear combination of nonlinear processing units, the radial basis functions. Thus, if M samples of the real function are obtained over the domain of interest (with jth data point $Dj = [X\ F\ (X)]$, $X \in R^N$ and $F(X) \in R$), then an RBF model is defined by

$$V(X) = \sum_{j=1}^{M} C_j \Phi(r_j, p_j) \tag{3.9}$$

where Cj is the coefficient of the jth radial basis function ϕ (r_j, p_j), with radius r_j equal to$\|X - X_j\|$ and tuning parameter p_j. Note that r_j defines the Euclidean distance of a given point X from the center X_j of jth basis function, while p_j is a measure of spread or influence of that particular RBF. Note that as the RBF model is designed to have as many centers as known data points, the size of the hidden layer is known a priori. However, this need not be a strict requirement if prototypical samples are extracted for use from the dataset using clustering methods.

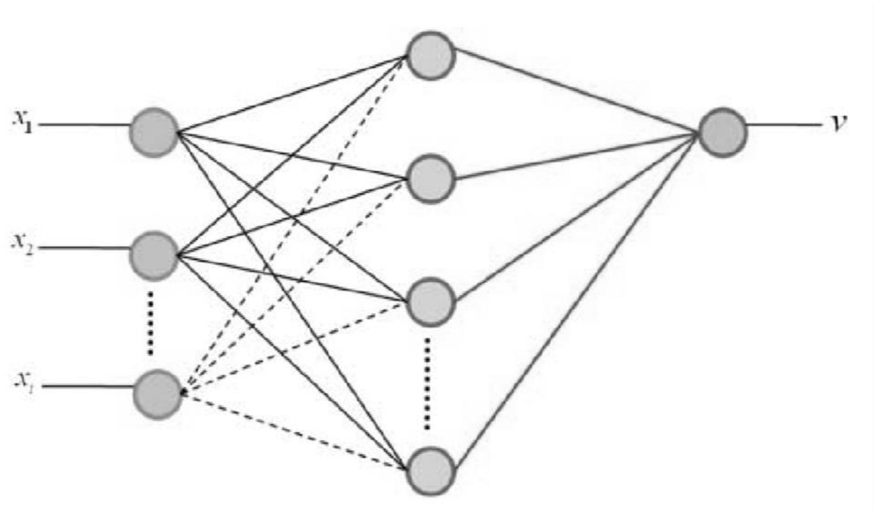


Figure 3.3: An adaptive multiquadric radial basis function method for expensive black-box (Rashid *et al.*, 2013)

If all centers adopt the same value for the tuning parameter p_j, the number of unknowns is M+1and if they are allowed to vary, the number of unknowns is 2M (with M tuning parameters and M coefficients). For the former case, a simple linear relationship between the number of parameters and hidden nodes results. In addition, the coefficients C_j are chosen such that the approximating function has the same value as the modelled function at the known sample points and are obtained from the solution of the following linear system if the tuning parameter is pre-defined by

$$\boldsymbol{C} = \boldsymbol{A}^{-1}\boldsymbol{F} \tag{3.10}$$

where *C* is the column of coefficients ($\in \mathrm{R}^{M\times 1}$), *F* is the column of known target function values ($\in \mathrm{R}^{M\times 1}$) and $\mathrm{A} \in \mathrm{R}^{M\times M}$ is a symmetric square matrix of radial basis functions, in which ϕ_{ij} (the element in the *i*th row and *j*th column) is the radial basis function evaluated with centre X_j from centre X_i.

The radial basis function ϕ (or, more generally, the node function) that is often used, owing to its cited robustness, is the multiquadric (MQ) RBF (Hardy 1971, Alotto *et al.* 1996, Rippa 1999 apud Rashid 2013), given by

$$\text{Multiquadric } \Phi(r,p) = \sqrt{(r^2 + p^2)} \tag{3.11}$$

The MQ-RBF exhibited effectiveness in a range of problems, and particularly in the adaptive optimization procedure, as showed by Rashid *et al.* (2013). We follow describing an adaptive multiquadric radial basis function method for expensive black-box function, proposed by Rashid *et al.* (2013). The Figure 3.4 shows a high-level adaptive procedure proposed by them, for expensive constrained optimization and it is the same as that developed in earlier works for expensive function optimization (Regis & Shoemaker 2005). According Rashid *et al.* (2013), the differences result primarily from the procedures implemented in Steps 0-4. For example, the adaptive procedure can be initialized (Step 0) by evaluation of an initial sample set based on uniform, quasi-random or random sampling, or the use of Latin Hypercubes or Design of Experiments (Regis & Shoemaker, 2005). As these samples are independent, they can be evaluated in parallel, if the provision exists, giving rise to a tabular set comprising the sample point Z, function value *F* and each of the nonlinear constraints *G* (*Z*) in the problem. Note that pre-existing model data could also be retrieved for use.

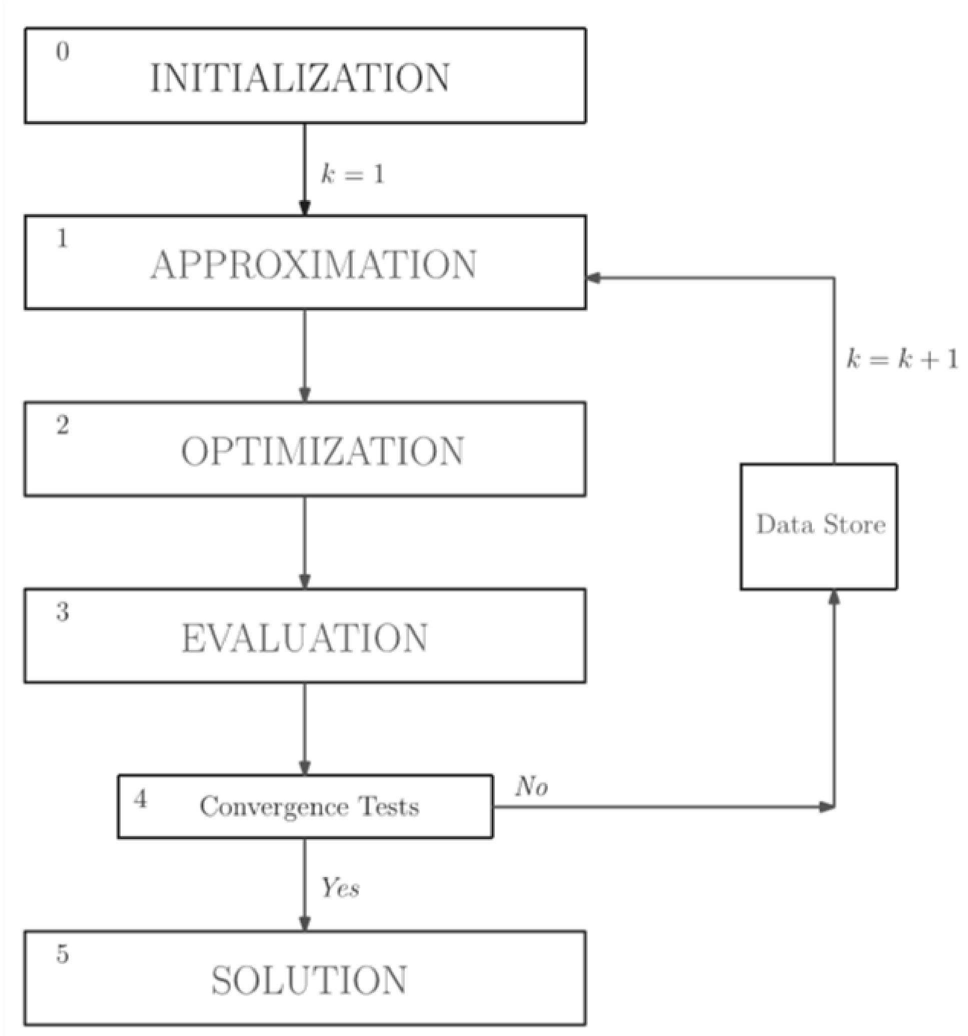


Figure 3-4: High-level methodology by Rashid *et al.* (2013)

In the methodology proposed by Rashid *et al.* (2013), at the approximation stage (*Step 1*), a multiquadric radial basis function (MQ-RBF) proxy model is created for the objective function and for each expensive nonlinear constraint in the problem using the available model data. These proxy models are used in the optimization step (*Step 2*) in place of the actual simulation-based model, with the use of an appropriate solver. In their work, a MINLP (Mixed Integer Nonlinear Programming) solver is employed to manage the optimization problem considered. The optimal solution and some additional samples are evaluated (in parallel) using the real simulation model (*Step 3*). The convergence tests are made in *Step 4* and all three conditions specified must be met. These are norms on the change in the best objective function value and the associated solution in search space over consecutive iterations, alongside the differences between the proxy and the actual model at the current iterate. Stopping conditions based on a no-progress count a real so implemented (Rashid *et al.,* 2009a). If the convergence or stopping conditions are not met, the data base is updated and the procedure is repeated from *Step 1* using all the available model data. Otherwise, the best known solution is returned as the answer to the optimization procedure.

Rashid *et al.* (2013) demonstrate the promising of this methodology and the adaptive RBF scheme is shown to be an effective tool for solving black-box simulation. The dynamic strategy proposed in this thesis is optimizer ambivalent, i.e., can make use of any optimizer, but we choose to use the proxy proposed by Rashid *et al.* (2013).