Implementation of a library for modelling in economics: design guidelines

João Pedro Pedroso

Riken Institute, Laboratory for Information Synthesis
Hirosawa 2-1, Wako-shi, Saitama 351-01, Japan
e-mail: jpp@brain.riken.go.jp

***Abstract*—** In this paper we propose some design guidelines for the implementation of a library for computer aided modelling and analysis in economics. We base our approach in object-oriented programming. We also introduce a prototype implementation, where a set of algorithms and data structures appropriated for economic modelling is supplied.

I. Introduction

There are many concrete economic problems for which an analytical solution is not obtainable, due to the size of the problem or to its complexity. In these situations it would be desirable to obtain numerical solutions, and a library providing algorithms and data structures for computationally modelling economic problems, and for the corresponding numerical solution, can be a valuable tool.

Let us first define a *library* as a collection of reusable code, i.e., a set of computer instructions which can be used, without change, as the basis for the construction of new programs. Properties that are usually required for a library are that it is easy to find and understand, that there is a reasonable assurance that its code is correct, and that no changes are required in the library's code for building a new program that uses it [2].

For the design of a library for analysis in economics, we propose that it should provide the programmer with the capability of starting from the simulation of the behaviour of the basic agents of the market, and then pursue by integrating their behaviour, thus constructing abstract aggregate agents. These, in turn, can be further aggregated, until achieving the desired abstraction level required for the model. This way, more and more complex situations can be implemented, by progressively developing the pieces that compose the economic situation we want to solve.

We believe that the most appropriate programming paradigm for the implementation of such a library is the object-oriented programming paradigm. This is because classes in an object oriented language can be very intuitively associated to the types of agents that actually exist in markets, and classes' member functions to the agents behaviour. There is also an intuitive correspondence between the information that is transmitted between the economic agents and the messages that can be passed between objects in a program.

In this paper we also introduce a prototype implementation using an object-oriented approach, where a set of algorithms and data structures appropriated for economic modelling is supplied. The classes provided in this library are closely related to the types of entities that are considered in the economic analysis. The programmer can create objects (i.e., variables) for each of the classes provided, as well as widen the set of classes through inheritance mechanisms.

II. Object oriented programming

Objects and Classes. In object-oriented programming an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.

A class is a specification of a structure, the behaviour, and the inheritance scheme for objects. Classes also specify access permissions, visibility and member lookup resolution.

For example, we could have a Consumer class, which would implement an algorithm to react to a price message by returning the optimal quantity purchased. An object of that class could be AdamSmith, which would be characterised by a particular budget constraint and utility function. Then, if we call AdamSmith.Quantity(Price = 100), this object will return the quantity that this particular consumer would optimally buy at price 100. We could, alternatively, call AdamSmith.Price(Quantity = 100), and it would returned the price that this consumer would be disposed to pay in order to buy a quantity of 100.

It is the responsibility of the object to satisfy the request specified in the message. The actual way of satisfying it does not need to be known by a user of that class—who often does not want to know the details, only requires the results.

Messages and methods. Methods are defined in classes and implement how objects of that class react, when they receive a message. A message encodes a request for a specific action, and is possibly accompanied by additional information (arguments) needed to carry it out.

Inheritance. Inheritance is a relationship between classes where one class is a base class of another; this means that all the (public or protected) elements of the base class are available, without change.

This way, classes can be organised into a hierarchical inheritance structure. A derived class inherits attributes from base classes. For example, we could have a class ConsumerCES which would be a specialisation of the consumer class, defining consumers whose utility function has constant elasticity of substitution (CES). The behaviour of the Consumer class is present, and in addition the utility function of objects of this class is instantiated.

III. Basic algorithms and classes

The first part of the library that we propose consists of a set of numerical analysis tools, which we summarise in this section. As most of the interesting problems arising in economics are nonlinear, and many times nonconvex and nonsmooth, the library should provide algorithms for tackling them. For this reason, at the core of this library we integrate methods for global optimisation and equilibrium computation described in [4] and [5], respectively.

A. Maximisation class

The most important of the classes implemented to support the development of the economics library is undoubtfully the Maximisation class. This class provides a very high level interface to the numerical optimisers.

The constructor of this class takes as an argument an Functor object, which provides the objective function, and a Constraints object, where the constraints of the optimisation problem are specified. This class provides the methods Solution and Objective, for accessing the solution vector and the corresponding evaluation, respectively.

The actual solver used depends on the objective function and constraints passed as arguments for the construction of a Maximisation object. For example, if they are linear, a simplex-like optimiser could be called; on the most general case, both objective and constraints are nonlinear and nonconvex, and an appropriate solver for this case would be binded.

The algorithms that underly the methods provided by this class are crucial for the whole development of the library, as they should provide a way of solving, with a good degree of confidence, any optimisation problem whose solution is not hopeless.

B. SimultaneousEquilibria class

This class implements a fixed point iteration method for the solution of simultaneous optimisation problems, and is described in detail in [5].

For the construction of one SimultaneousEquilibria object the programmer must supply a structure where the moves of all the players are stored, together with a Maximisation object for each of the players. Each of these Maximisation objects is parameterised on the moves of the other players; i.e., it will maximise an objective which depends the actions that were taken by the other market agents.

The Solve member function of this class performs the iterative solution of the system, by successively carrying out the optimisation for each of the players and updating the solution data structure, where the moves of the players are stored.

C. AsynchronousEquilibria class

Asynchronous equilibria are characterised by a player (the leader) which optimises its actions taking into account the optimal reaction of the other player (the follower).

Objects of the AsynchronousEquilibria class perform an optimisation such that for each evaluation of the objective, it is carried out another, nested optimisation. By other words, every time the objective function of the first stage problem (the leader) is called, there is a solution of the optimisation problem of the second stage (that of the follower).

Notice that objects of the AsynchronousEquilibria class can themselves be used as the input for the construction of a SimultaneousEquilibria object. We can therefore have a sequence of nested simultaneous and asynchronous problems, and construct a multiple stage problem. The only limits concern computational time, which can rapidly become rather restrictive.

IV. Classes for market agents

A. Consumers

The Consumer is the base class for simulating the behaviour of a consumer. It provides several methods that correspond to actions that consumers are supposed to take in an given economic environment, and should provide the basis for simulation of what is generally found in common for all consumer instances.

What the base class does is to implement a maximisation problem (hence it has a Maximisation object), which finds the quantities that are optimally bought by the consumer, given its utility function, its budget, and the price vectors stated by the market. Alternatively, it can find the price that the consumer would be disposed to pay for buying a given quantity. This class also provides a method to determine the indirect utility (the utility at the optimal quantity).

As an example, consider the case where one wants to simulate the behaviour of a consumer whose utility function is $U(x,y) = [\alpha\ y^{\rho} + (1-\alpha)x^{\rho}]^{\frac{1}{\rho}}$, where x is the quantity of the good under analysis, and y is the quantity of other goods that the consumer buys with its budget. This is a constant elasticity of substitution (CES) function, concerning the substitution between this good and other goods. We firstly implement a functor class for this utility function, the CesFunctor. Then, we create an object of this class, and use it as an argument for constructing a consumer object:

CesFunctor CesUtility	*The utility function object.*
CesUtility.Alpha(0.2)	*Set α=0.2*
CesUtility.Rho(0.8)	*and ρ=0.8*
Consumer CesConsumer(CesUtility)	*Create a consumer with this utility function,*
CesConsumer.Budget(100)	*and set its budget to 100.*
CesConsumer.Quantity(10)	*This returns the optimal quantity bought at a relative price of 10.*
CesConsumer.Price(1)	*This returns the relative price at which the quantity bought is 1 unit.*

The optimal quantity that is purchased by a consumer of this type at a given price, Quantity() is determined by the maximisation of its utility subject to the budget constraint. The Price() member returns the inverse of this function.

B. Demand

The demand classes calculate the aggregate quantity purchased by a set of consumers, which constitute the whole market for a particular good.

One version of these classes is built based on a discrete set of consumers; the aggregate demand for given values of the price (and other parameters) is found by determining the weighted sum of the values of individual demands (i.e., each of the consumer types has a given weight, which corresponds to the number of consumers of that type in the market).

Another one is built based on a particular type of consumers, which supply a member function for being self parameterised. This parameterisation permits differentiating preferences or budget of consumers; the aggregate demand is determined by numerically integrating the individual demands for given regions of the preference parameter.

Finally, a combination of these two classes leads to a composite, flexible demand class.

Demand Part1(CesConsumer, 10)	*In a part of the market, there are 10 consumers like CesConsumer.*
Functor SetBudget = CesConsumer.SetBudget()	*Function for parameterising consumers based on their budget.*
Demand Part2(CesConsumer, SetBudget, 10, 500)	*In another part of the market, there are identical consumers, but whose budget ranges from 10 to 500.*
Demand TotalDemand	*This is the whole demand of the market:*
TotalDemand.Add(Part1)	*Include the first part of the market*
TotalDemand.Add(Part2)	*and the second one.*
TotalDemand.Quantity(10)	*This returns the demand of the whole market for a price of 10.*

The degree of abstraction that the Demand class provides is quite considerable. Its objects can be called like a normal function of the price; for each function call, it calculates the individual demand for all the consumers, by summing and/or integrating their optimal quantities. The individual demands are determined through the optimisation of the utility functions.

C. Firms

The simulation of firms has many similarities with the simulation of consumers: firms, regarded from a competitive behaviour point of view, receive a vector of prices (and possibly more market parameters) as input and decide their production plan by maximising their profit. The Firm class provides methods that correspond to these actions. Objects of this class are created by supplying a production function, that of the firm that is being simulated.

What the base Firm class does is to implement a maximisation problem, which calculates its optimal production plan given its production function, the cost of the inputs, and the prices of the outputs. It also provides a method to determine the (optimised) costs for a given value of the output.

As an example, consider the case where one wants to simulate the behaviour of a firm producing some good, which is characterised by a quality factor (reliability), which is increased by increasing redundancy in production units. This is an usual situation in telecommunication and energy producers. With a number n of units, the firm has a total cost of $n\,U$ (where U is the cost of one unit), and can produce quantity $x = K \cdot j$ at quality $s = \sum_{j=0}^{n} C_j^n \cdot \sigma^{n-j} \cdot (1-\sigma)^j$, where σ is the reliability of each production unit, and K is its capacity. (C_j^n are the combinations of n machines j at a time). For example, having 2 production units (and therefore a cost of 2 U), a firm can either produce a quantity 2 K with reliability σ^2 or a quantity K with reliability $\sigma\ (1-\sigma)$.

The cost minimisation for such firms is quite simple; is consists simply of determining the minimum number of units that satisfy both the demand and the reliability constraint. After creating the class for simulating this, we could use it as follows:

RedundancyProdFunctor Prod	*Create the production functor.*
Prod.Capacity(100)	*Set capacity to 100*
Prod.Reliability(0.99)	*and unit reliability to 99%.*
Firm Factory(Prod)	*Create a firm with this production function.*
Factory.SetQuality(0.999)	*Set the reliability required for the final product to 99.9%.*
Factory.Quantity(10)	*This returns the optimal quantity produced if the market price of the good is 10.*
Factory.Cost(250)	*This is the cost for optimally producing a quantity of 250.*

The Monopoly class is a derivation of Firm, where the price of the product is not taken as given. The Monopoly objects have hence strategic variables concerning the production plan, as well as the price of the output. Its objective is the profit, and its constraints are determined by the production function.

Oligopolies in this library are simulated by the SimultaneousEquilibria or an AsynchronousEquilibria classes, to which we supply the Firms that take part in the game, and the demand function. Nash equilibria are computed by iterating through the optimal quantities that each firm produces, taking into account the other firms quantity, until reaching a stable situation. Stackelberg equilibria are computed by solving the followers problem inside the leaders objective function (what determines that optimal response of the follower for any action of the leader).

D. Supply

The Supply class provides the same type of aggregation over the Firm class that the Demand provides over Consumer. Hence, it determines the aggregate quantity produced by a set of firms for a given price. All the firms are considered as price takers. It is used mainly for determining perfect competition equilibria, where supply (by price takers) equals demand.

E. Welfare

Market welfare is determined by a Welfare class, which is created based on a Demand and a Supply objects. It computes the total welfare for given values of the price (possibly associated to some other factor, such as a quality level) by numerically integrating the demand function (thus obtaining the consumers surplus), and subtracting the aggregate cost.

F. Regulators

The Regulator class takes as input a set of Firms and a Demand object. It performs the optimisation of the market welfare (a Welfare object), determined through the demand function and the costs of the firms. Different types of regulation are possible, by stating what are the variables that a Regulator object can set (e.g. price, quantity, quality), and the kind of reaction of the other agents in the market.

G. Markets

Markets are simulated by placing together the several objects that constitute it, and assigning an order to the game that they play.

A typical construction is to start by creating the set of consumers, and create their aggregate demand function. The same thing is done to the supply side, by creating a set of firms and the aggregate cost. Based on the demand and supply objects, we can create a welfare, or an equilibrium computation object. The market welfare maximisation, or any type of game between the firms, could be determined through these objects.

In our example, consider the situation where a regulator sets a reliability level to which the firms must conform, and then there is a Nash-Cournot game between the firms for determining their optimal capacity:

```
RegOnQuality Regulator
Firm Leader(Prod)                    Create the leader and the follower
Firm Follower(Prod)                  objects, with a given production function.
Leader.SetQuality( Regulator.Quality() )   Set the imposed
Follower.SetQuality( Regulator.Quality() )   quality.
AsynchronousEquilibria Stackelberg   Create the object for
                                     calculating the equilibrium.
Stackelberg.SetLeader(Leader)        Set the leader and
Stackelberg.SetFollower(Follower)    the follower firms,
Stackelberg.SetDemand(TotalDemand)   and the demand.
Stackelberg.LeaderQuantity()   This returns the leader's quantity
                                     at the equilibrium,
Stackelberg.FollowerQuantity()       this the follower's one,
Stackelberg.Price()                  and this the equilibrium price.
```

V. Conclusion

The library for economic modelling that we describe on this paper was conceived based on the principle of giving the programmer a set of tools for the simulation of economic situations, starting from the simulation of the behaviour of its basic agents.

The simulation process pursues by integrating behaviour of these basic agents, constructing abstract aggregate agents, which. in turn, can be further aggregated, until achieving the desired level required for the simulation. This way, more and more complex situations are implemented, by progressively joining the pieces of the market whose solution we want to know.

The cost of going into such a low level in the simulation process, in terms of computational burden, is rather high. We believe, nevertheless, that this cost is completely justified by the understanding of the market mechanisms that becomes possible; for example, we do not need to restrict the studied models to cases where an analytical solution exists.

References

[1] G. Booch. *Object-Oriented Design With Applications.* Benjamin Cummings, second edition, 1994.

[2] M. D. Carroll and M. A. Ellis. *Designing and Coding Reusable C++.* Addison-Wesley, 1995.

[3] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[4] J. P. Pedroso. Niche search: an evolutionary algorithm for global optimisation. In H.-M. Voigt *et al.*, editors, *Parallel Problem Solving from Nature IV*, volume 1141 of *Lecture Notes in Computer Science*, Berlin, Germany, 1996. Springer.

[5] J. P. Pedroso. Numerical solution of Nash and Stackelberg equilibria: an evolutionary approach. In *Proceedings of the First Asia Conference on Simulated Evolution and Learning*, Korea, 1996.

[6] J. P. Pedroso. *Universal Service: Issues on Modelling and Computation.* PhD thesis, Université Catholique de Louvain, 1996.