Regional Research Institute
West Virginia University

Technical Document Series

Object-Oriented Interindustry Systems: Proof of Concept

Péter Járosi and Randall Jackson

RRI TechDoc 2015-03

Date submitted: August 27, 2015
Key words/Codes: Object oriented modeling, Interindustry Systems, Python; C67, C68, C63, R15

Object-Oriented Interindustry Systems: Proof of Concept

Péter Járosi*
Randall Jackson†

August 27, 2015

Abstract

This document provides a proof-of-concept demonstration of an object-oriented approach to modeling an inter-industry system. The example framework uses a small CGE model based on a three-sector social accounting matrix (SAM). The economy is shocked by changing total factor of productivity in the production function, the new equilibrium is determined in classical CGE fashion, and the original SAM is updated to conform to the new equilibrium solution. In this way, the efficiency of the Object-oriented modeling (OOM) approach can be emphasized in the context of the computational simulations of interindustry systems by a simplified CGE example written in Python. Since this example implemented as only a possible application of the OOM, the proof of the concept should be interpreted as a particular but among the most difficult economic modeling cases.

*Innovation and Economic Growth Research Group, University of Pécs. E–mail: jarosip@ktk.pte.hu

†Director, Regional Research Institute, and Professor, Department of Geology and Geography, West Virginia University. E–mail: Randall.Jackson@mail.wvu.edu

1 The Problem Context

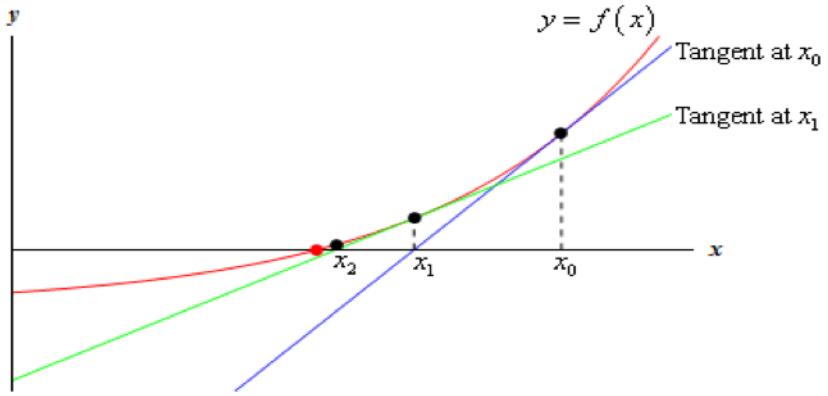
This document presents the first practical example of a response to Jackson's 1994 call for object-oriented modeling in the social sciences, and regional science in particular Jackson (1994). The problem context is presented in two ways. First, by keeping example problem small we are able to present the entire problem domain in two spreadsheet pages. Of course, one of the advantages of the object-oriented approach is that additional sectors could be added easily. The first page is used to calibrate the classical 3×3 CGE, and the second page is where the shocks are entered and solutions are computed and displayed. The object-oriented proof-of-concept is demonstrated using the Python programming language, and comprises two files: CDIO_Economy.py and Example3×3OfCDIO.py.

2 A Brief Description of the Model

There is a SectorClass that defines the attributes and behaviors of the industry sectors. There are three instances: agriculture, manufacturing, services. The industries are characterized by constant scale Cobb-Douglas production functions. Capital may be substituted for labor as a function of factor prices. Labor supply and capital supply are fixed, and there is are labor and capital demand functions. The numéraire is the price of the capital. There is no need to check the capital market equilibrium because of Walras Law.

There is a Cobb-Douglas type household class and a StoneGeary household class derived from the parent Cobb-Douglas household class. The parent-child relationship demonstrates the inheritance and polymorphism features of the object-oriented approach. There are no savings, income taxes, etc., in this simplified model.

The final class is the simplified market equilibrium class, which implements a Newton tangent method based solver to determine the market equilibrium. The figure below is a graphical characterization of the tangent method.



The Excel spreadsheet file and the two Python files are provided as a supplement to this Technical Document in compressed (.zip) format.

3 The Essence of the Object Oriented Modeling Approach

By the example described above we illustrate the object-oriented modeling approach. This modeling framework came from computer science and software programming languages, but its value extends beyond popular programming techniques and data abstraction to other problem domains. Moreover, it has the potential to become one of the most effective scientific methods for conceptualizing problem domains in the field of economics, as a new way of problem abstraction and reduction.

Because we can characterize virtually all of the salient elements of an economy as object, we can develop powerful abstractions of economic entities. In our CGE example, the households and the industrial sectors are the objects. In computer programming languages an object is a collection of attributes (data) and behaviors (or *methods*, typically implemented as functions), so the almost perfect analogy becomes readily apparent. For an economic object, we can collect not only data about the economic entities but also behaviors, methods, etc. —whatever we can implement as a function inside the class which represents the group of the economic objects. For example the operation of the industry can be described by its production

function, input cost shares, factor demand function, etc., and similarly the behavior of the household can be defined by the commodity demand function derived from a utility function.

A class in object oriented modeling is a construct that defines the structure of a set of objects all of which share common characteristics. It identifies what attributes and behaviors objects of the class will have, and it can additionally define default values for attributes and default functionality for behaviors. The class, sometimes referred to as an *object class*, characterizes the entities, but it exists as a construct and does not itself have an *identity*. An object is an instance of a class, and it does have identity. There are numerous real world examples. There is a class of real world objects that we know as *bicycle*, for example, all of which have two wheels, a color, a seat, etc. A bicycle is a type of thing, a construct without identity, whereas the bicycle in your garage is a bicycle object, one that you can identify and that has specific values for its attributes.

In our model there are two household classes (i.e., two types of households), one of which is derived from the other —the former is a subclass, and one class for the industrial sectors. In the concrete model the three sectors are manifested as the instances of the same class called "SectorClass". They inherit the structure of the SectorClass, and are assigned their attribute values during model calibration when they were *instantiated* by the constructor "__init__" method. Later the values of the attributes can be changed as a result of the classes behaviors. It is also possible to provide access to selected objects attribute values with set behaviors (e.g., bicycle.set_color(red)). Our program code has been divided into two parts. The first block contains the code for the definition of the sectors class and the two types of the households. The second part is the specified example of the model.

The object-oriented programming literature identifies the primary characteristics of this modeling approach: encapsulation, identity, classification, aggregation, inheritance and polymorphism. Now we illustrate these principles in general and also in accordance with our specific economic problem.

3.1 Encapsulation

The combining of data and functions which operate on that data is the substance of the object-oriented modeling. We can imagine it as a capsule and in this capsule we have all of its attributes and functions, encapsulation is the essence of OOM. Encapsulation is essential to polymorphism, defined below, in that behaviors with the same name in two different classes can invoke different actions. In our model, both of the classes, the "SectorClass" and the "HouseholdClass" have the "calcCommodityDemand" function with exactly the same name, but each invokes a different behavior. Encapsulation eliminates name conflict in the namespace of our model; because the names belong to different classes, they are of different types. We use the "dot" notation to implement the encapsulation principle.

3.2 Identity

A class is an invariant description of object structure, which means the class is nothing but a construct from the viewpoint of the existing model unless and until we create its instances calling the constructor method with different parameter lists. Attributes of objects can change in their lifetimes. For example, we can give a total factor productivity shock to the system increasing the value of the "tfp" parameter of the sector instances. The identity of the object remains the same, only the value of its attributes has been changed.

3.3 Classification

Conceptualizing the class hierarchy for the problem domain is the crucial moment of the abstraction process, and commences before writing any program code. Without proper class definitions the model development can quickly become a confused, chaotic, and possibly hopeless attempt. Sometimes it is easy to find the common attributes and behaviors in the elements of the model, for example see the class of the sectors in our example. In many other cases it can be extremely difficult and can result a scientific break-through. There are some graphical tools and methods that can help this thinking process, and these are recommended and highly useful. Many computer drawing and diagramming tools support object-oriented modeling.

3.4 Aggregation

Aggregation is a rarely cited essence of the object-oriented programming but in the economic modeling it often becomes a pivotal step of the abstraction process. Although composite objects are frequently used in economic simulations, how the attributes and functions of the lower level entities can be aggregated is not a always a simple and straightforward question. The simplified market equilibrium class may not be the most elegant answer for this question in this context, but it is satisfactory for the proof-of-concept. We are developing more sophisticated aggregations for the market types, solver classes, etc. to supporting easily understandable modeling structures for the open source collaboration we are currently initiating.

3.5 Inheritance

Inheritance can be interpreted through the relationship between the subclasses and superclasses. In the CGE example the household with the Cobb-Douglas utility function (CD) is defined as the parent class (called as superclass earlier) and the Stone-Geary type household (SG) was derived from it as a child object (subclass). As the children inherit all of the of properties from their parents, but redefine (overwrite) some and potentially add others.

3.6 Polymorphism

After declaring the subclass, some of its attributes and functions can be overwritten and others can be left unchanged. In the case of the StoneGeary household, the demand for commodities is different from the case of the CobbDouglas household, which explains why the function "calcCommodity-Demand" had to be overwritten according to the different utility function. The "market equilibrium"object is indifferent to which type of households are used, because the *interfaces* of the commodity demand functions are the same, and the calculated demand can be obtained for the solver algorithm without any changes in the code of the latter class. An object's interface provides other objects with access to its properties. As examlples, the *set*

and *get* functions are commonly used by external objects to set an object?s attributes and to get the values of object attributes.

Of course, the example here is not the only one possible or correct way to implement these two types of household. It is not even the simplest way, which would be a trivial implementation where only the SG household has been defined as a class, and the CD case can be used by the code of SG with the zero values of the subsistence levels of consumption without using any subclass definition. However, that example would not be suitable to demonstrate inheritance and polymorphism. But there are a lot of possible extensions of the model where a similar trivial approach would be impractical, for example suppose what if the production function should be changed from Cobb-Douglas to CES. Since the Cobb-Douglas is the special case of the CES, more accurately it is the limit of the CES function when the elasticity of substitution approaches one in the limit, this is not a trivial case and the modification of the model surely would be easier by the principle of polymorphism than programming some clever code into only one class definition. Such clever coding solutions also tend to obfuscate the conceptual underpinnings of these modeling frameworks rather than clarify and make them more transparent, which defeats the purpose of object oriented modeling in the first place. A future technical documents will demonstrate the example of the replacement of the production function.

4 Conclusion

Defining new classes in a programming language means that beyond using the built-in types of the language the developer creates new user defined types. In this way the new elements of the language or a new language altogether can be invented according to the terminology of the given problem domain as it was illustrated above by the CGE example. So we do not need to think about integers, floats, vectors, matrices, etc. anymore but we can operate with the terms of sectors, households, demand, supply, etc. when we write the code of the model. Not only the name and the attributes of the new entity can be defined by this approach but its behaviors, functions etc. can be bound together into one entity. Our longer term goal is to generalize and extend the modeling framework and the Python language for many spe-

cial purposes: including interindustry systems, environment problems, etc., energy technology transitions, water quantity and water quality issues, and more.

Supporting Algorithm(s)/Code

The contents of the Python files are listed below, but we recommend that if you plan to use this code, you extract the files from the zip file that accompanies this document to ensure proper formatting.

CDIO_Economy.py

```
#Initialize a sector with the Leontief coefficients and the
#parameters of the Cobb-Douglas production function
    def __init__(self, indexOfThisSector, expOnInterGoods,
        expOnFactors, givenFactorSupply):
        #The ordinality of this sector in the input-output table
        self.ordSect = indexOfThisSector
        #The cardinality of the sectors in the input-output table
        self.cardSect = len(expOnInterGoods)
        #The cardinality of the factors in the production function
        self.cardFact = len(expOnFactors)
        #We suppose the consistency of the SAM
        self.totalOutlays = sum(expOnInterGoods) + sum(expOnFactors)
        self.nominalOutput = self.totalOutlays
        self.priceIndex = 1.00
        self.realOutput = self.nominalOutput / self.priceIndex
        #Calibrate the column vector of the coefficients in the
        #input-output table
        self.coefficients = [expOnInterGoods[i] / self.totalOutlays \
        for i in range(self.cardSect)]
        #Calibrate the parameters (Partial elasticities and tfp)
        #of the Cobb-Douglas production function
        self.elasticities = [expOnFactors[i] / sum(expOnFactors) \
        for i in range(self.cardFact)]
        #The initial value of the tfp (total factor of productivity)
        denominator = 1.00
        for i in range(self.cardFact):
            denominator *= givenFactorSupply[i]**self.elasticities[i]
        self.tfp = self.realOutput / denominator
```

```
        self.factorPrice = [expOnFactors[i] / givenFactorSupply[i] \
        for i in range(self.cardFact)]
        #In the case of the calibration:
        self.initialVaPerUnit = 1.00 - sum(self.coefficients)
        self.currentVaPerUnit = self.initialVaPerUnit
        self.factorSupply = givenFactorSupply
        #self.factorDemand = givenFactorSupply
        self.factorDemand = [self.elasticities[i] * \
        self.currentVaPerUnit * self.realOutput/self.factorPrice[i] \
        for i in range(self.cardFact)]
        #edf meaning: excess demand of factors
        self.edf = [self.factorDemand[i] - self.factorSupply[i] \
        for i in range(self.cardFact)]
        #Calculate the initial intermediate demand of commodities
        self.commodityDemand = [self.coefficients[i]*self.realOutput \
        for i in range(self.cardSect)]

    def calcVaPerUnit(self, pricesOfFactors):
        #Calculate the new Value Added per Unit depending from
        #the prices of factors
        self.factorPrice = pricesOfFactors
        numerator = 1.00
        denominator = self.tfp
        for i in range(self.cardFact):
            numerator *= pricesOfFactors[i] ** self.elasticities[i]
            denominator *= self.elasticities[i] ** self.elasticities[i]
        self.currentVaPerUnit = numerator / denominator
        return self.currentVaPerUnit

    def calcFactorDemand(self, pricesOfFactors, givenRealOutput):
        #In the Cobb-Douglas production fuction there are
        #substitutions between the factors
        self.calcVaPerUnit(pricesOfFactors)
        self.realOutput = givenRealOutput
        self.factorDemand = [self.elasticities[i] * \
        self.currentVaPerUnit * self.realOutput \
        / pricesOfFactors[i] for i in range(self.cardFact)]
        return self.factorDemand

    def calcExcessDemandOfFactors(self, pricesOfFactors,
        givenRealOutput):
        self.calcFactorDemand(pricesOfFactors, givenRealOutput)
        self.edf = [self.factorDemand[i] - self.factorSupply[i] \
        for i in range(self.cardFact)]
        return self.edf
```

```
    def calcCommodityDemand(self, givenRealOutput):
        #In the Leontief framework production the demand
        #of commodities are independent
        self.realOutput = givenRealOutput
        self.commodityDemand = [self.coefficients[i] * \
        self.realOutput for i in range(self.cardSect)]
        return self.commodityDemand

class HholdCobbDouglas:
#The parent object of the households

    def __init__(self, expOnFinalGoods):
        #Initialize the Cobb-Douglas utility function.
        self.disposableIncome = sum(expOnFinalGoods)
        #The cardinality of the different goods by
        #the sectors in the consumption
        self.cardSect = len(expOnFinalGoods)
        self.setBetaParams(expOnFinalGoods)
        self.commodPrice = [1.00 for i in range(self.cardSect)]
        #Calculate the initial final demand of commodities
        self.commodityDemand = [self.betaParams[i] * \
        self.disposableIncome / self.commodPrice[i] \
        for i in range(self.cardSect)]

    def setBetaParams(self, resExpenditures):
        #Calibrate the beta parameters of the utility function
        #by the residual income and expenditures
        resIncome = sum(resExpenditures)
        self.betaParams = [resExpenditures[i] / resIncome \
        for i in range(self.cardSect)]

    def calcCommodityDemand(self, pricesOfGoods,
        givenDisposableIncome):
        #Calculate the final demand of hoseholds
        self.disposableIncome = givenDisposableIncome
        self.commodityDemand =[self.betaParams[i] * \
        self.disposableIncome / pricesOfGoods[i] \
        for i in range(self.cardSect)]
        return self.commodityDemand


class HholdStoneGeary(HholdCobbDouglas):
#The child object of the households
```

```
    def __init__(self, expOnFinalGoods, subsistenceCons):
        #Initialize the Stone-Geary utility function.
        HholdCobbDouglas.__init__(self, expOnFinalGoods)
        self.subCons = subsistenceCons
        self.resExps = [expOnFinalGoods[i] - subsistenceCons[i] \
        for i in range(self.cardSect)]
        self.setBetaParams(self.resExps)

    def calcCommodityDemand(self, pricesOfGoods,
        givenDisposableIncome):
        #Calculate the final demand of hoseholds
        self.disposableIncome = givenDisposableIncome
        self.residualIncome = givenDisposableIncome - \
        sum([pricesOfGoods[i]* self.subCons[i] \
        for i in range(self.cardSect)])
        self.commodityDemand =[self.subCons[i] + self.betaParams[i] \
        * self.residualIncome / pricesOfGoods[i] \
        for i in range(self.cardSect)]
        return self.commodityDemand
```

Example3x3OfCDIO.py

```
#A 3 sectors example of the Input-output economy
#with Cobb-Douglas production function
#version 0.3 beta, Aug, 2015.
from CDIO_Economy import SectorClass, \
HholdCobbDouglas, HholdStoneGeary
import numpy as np

class SimplifiedEquilibriumOfMarkets:

    def __init__(self, indexOfLabor, listOfSectors, listOfHholds):
        #The index of the Labor in the list of the Factors
        self.iol = indexOfLabor
        #The cardinality of the sectors in the input-output table
        self.cardSects = len(listOfSectors)
        self.sectors = listOfSectors
        #The cardinality of the households
        self.cardHhols = len(listOfHholds)
        self.hholds = listOfHholds
        #The accuracy and the step options of the solver
        self.solvOptTolFun = 0.000001
        self.solvOptTolX = 0.0000001
        self.coeffMatrix = np.matrix.transpose(np.matrix
```

```
        ([self.sectors[i].coefficients
        for i in range(self.cardSects)]))
        self.leontiefInverse = np.linalg.inv(np.identity
        (self.cardSects) - self.coeffMatrix)
        #For debugging only
        #print(self.leontiefInverse)

    def calcTheNewRealOutput(self):
        totalIncome = sum([np.dot(self.sectors[j].factorPrice,
        self.sectors[j].factorSupply)
        for j in range(self.cardSects)])
        rowVectorOfVaPerUnit = np.matrix([self.sectors[j].
        calcVaPerUnit(self.sectors[j].factorPrice)
        for j in range(self.cardSects)])
        rowVectorOfCommPrices = np.dot(rowVectorOfVaPerUnit,
        self.leontiefInverse)
        pricesOfGoods = np.array(rowVectorOfCommPrices). \
        flatten().tolist()
        colVectorOfFinalDemand = np.matrix.transpose(np.matrix
        ([self.hholds[0].calcCommodityDemand
        (pricesOfGoods, totalIncome)]))
        colVectorOfRealOutput = np.dot(self.leontiefInverse,
        colVectorOfFinalDemand)
        realOutput = np.array(colVectorOfRealOutput). \
        flatten().tolist()
        for j in range(self.cardSects):
            self.sectors[j].calcExcessDemandOfFactors(
            self.sectors[j].factorPrice, realOutput[j])

    def stepTowardEquilibrium(self, indexOfSector):
        i = indexOfSector
        if abs(self.sectors[i].edf[self.iol]) > self.solvOptTolFun:
            lastFactorPrice = self.sectors[i].factorPrice[self.iol]
            tempFactorPrice = self.sectors[i].factorPrice[self.iol] \
            + self.solvOptTolX
            lastExcessDemandOfFactors = self.sectors[i].edf[self.iol]
            self.sectors[i].factorPrice[self.iol] = tempFactorPrice

            self.calcTheNewRealOutput()
            tempExcessDemandOfFactors = self.sectors[i].edf[self.iol]

            nextFactorPrice = lastFactorPrice - self.solvOptTolX * \
            lastExcessDemandOfFactors \
            / (tempExcessDemandOfFactors-lastExcessDemandOfFactors)
```

```
                #Error handling
                if nextFactorPrice <= 0.00:
                    nextFactorPrice = lastFactorPrice / 2.00
                #Set the new FactorPrice
                self.sectors[i].factorPrice[self.iol] = nextFactorPrice

                self.calcTheNewRealOutput()
                nextExcessDemandOfFactors = self.sectors[i].edf[self.iol]

                #Debugging the values of the most important variables
                #during the iteration process
                #print("sector ", i, self.sectors[i].factorPrice)
                #print(self.sectors[i].factorSupply, self.sectors[i].
                #factorDemand)
                #print(lastExcessDemandOfFactors,
                #nextExcessDemandOfFactors)
                #print("")

#Initialize the sector instances (Index in the IO table,
    #[Expenditures on Intermdiate Goods], \
    #[Expenditures on Factors],
    #[Factor Supplies])
agricult = SectorClass(0, [256466.975822654, 408396.165980749, \
    311743.90414051], \
    [532904.12448497, 431069.657967317], \
    [300000.00, 431069.657967317])

manufact = SectorClass(1, [532853.858444792, 355800.894077304, \
    608714.436575474], \
    [1024107.22949434, 1421295.81744353], \
    [200000.00, 1421295.81744353])

services = SectorClass(2, [378117.307470861, 785721.936621465, \
    540607.901795493], \
    [1654889.68703088, 1112063.10268159], \
    [300000.00, 1112063.10268159])

threeSectors = [agricult, manufact, services]
#Initialize the household instance ([Expenditures on Final Goods])
cdHhold = HholdCobbDouglas([773142.686657892, 2392853.23935591, \
    3010333.69308882])
sgHhold = HholdStoneGeary([773142.686657892, 2392853.23935591, \
    3010333.69308882], \
    [773142.686657892/2.00, 2392853.23935591/3.00, \
    3010333.69308882/4.00])
```

```
#oneHhold = [cdHhold]
oneHhold = [sgHhold]

#Initialize the instance of the SimplifiedEquilibriumOfMarkets
#lmdEqu means labor market driven equilibrium
lmdEqu = SimplifiedEquilibriumOfMarkets(0, threeSectors, oneHhold)
print("Before the shock: (see the CalibrationOfThe3x3Cge sheet!)")
print("Sectoral total factor of productivity: ")
print([lmdEqu.sectors[j].tfp for j in range(lmdEqu.cardSects)])
print("Real Output: ")
print([lmdEqu.sectors[j].realOutput for j in range(lmdEqu.cardSects)])

#Give tfp shocks into all of the sectors
#Increase the the total factor of productivity
#(agricult +4%, manufact +5%, services +6%)
for i in range(lmdEqu.cardSects):
    lmdEqu.sectors[i].tfp = lmdEqu.sectors[i].tfp * (1.04
    + float(i)/100)
    lmdEqu.sectors[i].calcExcessDemandOfFactors(lmdEqu.sectors[i].
    factorPrice, lmdEqu.sectors[i].realOutput)
print
#Conroll the maximum number of iterations
solvOptMaxIter = 0
while (max([abs(lmdEqu.sectors[j].edf[0]) for j in \
range(lmdEqu.cardSects)]) > lmdEqu.solvOptTolFun) \
and (solvOptMaxIter < 1000):
    for i in range(lmdEqu.cardSects):
        lmdEqu.sectors[i].calcExcessDemandOfFactors(
        lmdEqu.sectors[i].factorPrice, lmdEqu.sectors[i].realOutput)
        lmdEqu.stepTowardEquilibrium(i)
        #for debugging only:
        #print(lmdEqu.sectors[i].factorPrice)
        #print(lmdEqu.sectors[i].edf)
    solvOptMaxIter += 1
print("After the shock: (see the SolutionOfTheClassic3x3Cge sheet!)")
print("Number of iterations:", solvOptMaxIter)
print("Sectoral total factor of productivity: ")
print([lmdEqu.sectors[j].tfp for j in range(lmdEqu.cardSects)])
print("Real Output: ")
print([lmdEqu.sectors[j].realOutput \
for j in range(lmdEqu.cardSects)])
```

References

Jackson, R. (1994). Object-oriented modeling in regional science: an advocacy view. *Papers in Regional Science*, 73(4):347–367.