When using this online resource, please cite the following *Journal of Management Accounting Research* article:

Anand, V., R. Balakrishnan, and E. Labro. XXXX. A Framework for Conducting Numerical Experiments on Cost System Design. *Journal of Management Accounting Research* 31 (1): XXX-XXX. https://doi.org/10.2308/jmar-52057

Appendix: Computer Code

A-1 Introduction

This appendix describes the computer code published online alongside the paper. We intend this code as a template that researchers can modify for use in numerical experiments involving cost systems. The code is adapted from two prior numerical experiments conducted by the authors, Balakrishnan, Hansen, and Labro (2011) and Anand, Balakrishnan, and Labro (2017), and currently implements simplified versions of those experiments. Other researchers should be able to adapt this code to their own research.

The code generates a random sample of firms, each of which is modeled as a mapping between resources (inputs) and products (outputs); see section 2. For each firm, the code generates a set of cost systems, each with a different design, and each possessing limited information about the firm's production technology; see section 3. Each firm produces a subset of its available products and observes total resource consumption. It then iterates over its cost systems, using each to estimate the costs of its products. In the portion of the code that is loosely based on Balakrishnan, Hansen, and Labro (2011), the reported costs from each cost system are compared to the true (full-information) costs and an error metric is reported. In the portion of the code that is loosely based on Anand, Balakrishnan, and Labro (2017), each firm revises its production decision after observing its reported costs, producing (dropping) all products that appear (un)profitable. After implementing the revised decision, each cost system reports new estimates for the products' costs. This process repeats until the firm finds a fixed-point decision, a cycle of decisions, or a death spiral.

This appendix contains two major sections. Section *A-2 Descriptions of Classes*, describes the main classes (data structures) defined in the code. For each class, we provide details about the encapsulated data and methods. In section *A-3 Modifying and Running the Code*, we detail the steps

that a researcher should follow to modify this code for use in a numerical experiment involving cost systems and then describe the process for running the code.

A-1.1 Downloading the Code

The code, organized as a Visual Studio Solution, can be downloaded from the first author's website, http://vicanand.weebly.com/abl_jmar_code.html.[1] We request that anybody who downloads the code and amends it for their research acknowledges their use of this code and references this paper in an acknowledgement section.

A-1.2 Programming Language and Overview of Object-Oriented Programming

The code is written in C#, currently ranked as the fifth most popular programming language worldwide (*TIOBE Index* 2017).[2] C# is an object-oriented language, and we designed the code accordingly. Object-oriented programming is a programming paradigm in which the organization of data drives the design of the program. Data is organized into classes that *encapsulate*, or contain, the data. Classes are used to represent or model structures of interest.[3] Classes are blueprints for objects; objects are instances of classes. Classes store data using variables that are referred to as *fields*. Fields are often designated as private and therefore not visible to other classes. Thus, objects can interact without knowing the details of each other's implementation. To create, or instantiate, an object, one must call the class's *constructor*. A constructor is a special function whose sole purpose is to create an object by assigning values to some of the object's fields. Classes can expose their data to other classes through *properties*, which are mechanisms to provide restricted access to a class's fields. Thus, fields are private while properties are public.[4] *Methods* define the actions that a class can perform. Methods, which are implemented as functions and can therefore accept input arguments, can change the internal state of an

[1] Visitors of this website must enter a name and email address to download the code. We request this information so that we can notify users of changes to the code, if any.

[2] Until recently, Microsoft supported C# only on the Windows® operating system. However, Microsoft now provides native support for C# and its Visual Studio integrated development environment (IDE) on Macintosh computers. See https://blogs.msdn.microsoft.com/visualstudio/2016/11/16/visual-studio-for-mac/.

[3] For example, in our code, we define a *Firm* class and a *CostSys* class.

[4] While it is possible to designate fields as public, many consider that poor programming practice. We therefore mark all fields as private and use properties to expose an object's data to other objects.

object by changing values of the class's fields, or use the class's fields to compute a value of interest. Thus, methods use the class's data to perform an action without necessarily exposing that data to other classes.[5] We use the terms *field*, *property*, *constructor*, and *method* throughout this appendix, and recommend that readers familiarize themselves with object-oriented programming concepts before attempting to use our code.[6]

A-1.3 Open Source License

The code contains an open source license, the Microsoft Public License (Ms-PL).[7] The terms of this license imply three things. First, anybody may freely reproduce and distribute the original code, or a derivative version. Second, the code lacks any warranty or guarantee. Finally, the license file (license.txt) found in the code must accompany any future distribution of the original or modified code.

A-1.4 Future Changes to the Code

While we have thoroughly tested and debugged the code, we cannot guarantee that the code is error-free; bugs may exist. If we discover a bug, we will modify the code and update our code repository at the download location, http://vicanand.weebly.com/abl_jmar_code.html. We will then notify those who have previously downloaded our code of the changes.

If you believe there is an error or bug in our code, please notify one of the authors. We are also open to suggestions for new features, particularly those developed as part of a published or accepted paper. However, since we intend this code as a template, we will not add new features that are specific to individual research projects.

---

[5] Events are another important construct in object-oriented programming. Events enable a class or object to notify other classes or objects when something of interest occurs. Our code does not currently use events.

[6] For more information on object-oriented programming, see https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/object-oriented-programming.

[7] https://opensource.org/licenses/MS-PL

A-2 Descriptions of Classes

The major classes (objects) are *Firm* and *CostSys*. The *Firm* class encapsulates the production technology of an organization as well as details of the organization's input and output markets. The *CostSys* class models the cost system of an organization. The *InputParameters* class handles input; it validates and stores user-supplied parameters read from an input file. The *Output* class handles output; it writes user-defined output to CSV files. The *Main* method is contained within a static class[8] called *Program*; this method defines the control flow for the experiment. Finally, helper classes provide support for random numbers and matrix algebra. In the following subsections, we provide details of the fields, properties, constructor, and methods of each class in the code. We denote vector quantities using vector notation (e.g. $\overrightarrow{RCC}$) and matrices and scalars in italics (e.g. g). We denote elements of vectors using subscripts, e.g., element i of vector $\overrightarrow{RCC}$ as RCC_i. We use lower case for field names, and upper case for property names.

A-2.1 Firm Class

We model a firm as a mapping between its resources (inputs) and its products (outputs) using a Leontief technology. Currently, the code does not implement uncertainty in the production technology. We assume the firm can sell either some fixed amount of each product, or zero units.[9] We also assume the firm purchases its resources from perfectly competitive input markets at known, fixed prices.

A-2.1.1 Fields and Properties

Table 1 lists all fields of the *Firm* class, their data types, and their corresponding public properties.[10] Fields are private and cannot be accessed by other objects. Properties are public, and all

[8] A static class cannot be instantiated. It is typically used as a container for methods (functions) that do not require fields to store state between method calls.

[9] See Anand, Balakrishnan, and Labro (2017) for extensive discussion of this design choice. Future research can modify this design choice to model settings where the firm either has market power or operates in perfectly competitive markets for its outputs.

[10] Many classes in this code rely on an open source numerical library called *Meta Numerics* (http://www.meta-numerics.net), which provides data types and methods for linear algebra and statistics. We discuss this library in section A-3.2.

properties return copies of the fields. This distinction will be useful when modeling what can or cannot be observed under limited information circumstances. In Table 1, we refer to the following parameters from the input file :[11]

- RCP is the (user-supplied) number of input resources.
- CO is the (user-supplied) number of products (cost objects) that the firm can produce. The program enforces the constraint: $RCP \geq CO$.

Table 1: Fields and properties of the *Firm* class.

Field	Data Type	Public Property	Description
res_cons_pat	RectangularMatrix (Dim: *RCP x CO*)	RES_CONS_PAT	Element ij of the matrix is the number of units of resource i (row) consumed by product j (column).
$res\_cons\_pat\_prct$	RectangularMatrix (Dim: *RCP x CO*)	$RES\_CONS\_PAT\_PRCT$	Percentage of each resource consumed by each product. Each row will sum to 1.0.
$pearsoncorr$	SymmetrixMatrix (Dim: *RCP x RCP*)	$PEARSONCORR$	Correlation matrix showing correlations of resource consumptions.
g, also referred to as $DISP2$	Double precision	G	Percentage of total resources accounted for by the top *DISP1* resources, where *DISP1* is supplied by the user in the input file. Note: in the input file, the user must specify bounds for parameter *DISP2*. The program then chooses g from a uniform distribution with those bounds.
d, also referred to as DNS	Double precision	D	The density (sparsity) of the *RES_CONS_PAT* matrix. Note: in the input file, the user must specify bounds for parameter *DNS*. The program then chooses d from a uniform distribution with those bounds.

[11] The researcher can specify levels for many of the parameters in the input file. We describe the input file in sections A-2.3 and A-3.4.

Field	Data Type	Public Property	Description
$\overrightarrow{mxq}$	ColumnVector (Dim: *CO* x 1)	$\overrightarrow{MXQ}$	A vector of maximum production quantities of the products (cost objects).
$\overrightarrow{rcu}$	RowVector (Dim: 1 x *RCP*)	$\overrightarrow{RCU}$	Vector of unit resource costs (price the firm pays per unit of each resource).
$\overrightarrow{sp}$	RowVector (Dim: 1 x *CO*)	$\overrightarrow{SP}$	Vector of selling prices of products (cost objects). This parameter corresponds to $\overrightarrow{P_t}$ in the paper.
$\overrightarrow{initial\_rcc}$	RowVector (Dim: 1 x *RCP*)	$\overrightarrow{Initial\_RCC}$	Randomly generated vector of resource costs ($\overrightarrow{RCC}$) used to create a Firm object. The sum of the elements of this vector will be TR, the total resources parameter supplied by the user in the input file.
$\overrightarrow{initial\_rank}$	Integer array (Length: RCP)	$\overrightarrow{Initial\_RANK}$	The ranking of resources, by size, in the initial $\overrightarrow{RCC}$ vector.
$costSystems$	List of *CostSys*	$costSystems$	A list of *CostSys* objects (cost systems) based on this firm object.
$profitt0$	Double	$PROFITT0$	The profit generated by producing the profit-maximizing product mix.

A-2.1.2 Constructor

The *Firm* constructor performs the following steps to instantiate a *Firm* object.

1. Choose random values for the *g* and *d* fields.
2. Randomly generate a vector of true product margins ($\overrightarrow{MAR}$).
    a. Using $\overrightarrow{MAR}$, compute the profit-maximizing production decision vector $\overrightarrow{DECT0}$.
3. Randomly generate the vector of maximum production quantities of the products ($\overrightarrow{MXQ}$).
    a. Using $\overrightarrow{MXQ}$and $\overrightarrow{DECT0}$, compute the vector of profit-maximizing production quantities of the products, $\overrightarrow{QT}$.
4. Randomly generate the resource consumption pattern matrix (RES_CONS_PAT).
5. Randomly generate the vector of total resource costs ($\overrightarrow{RCC}$) and from that, compute vector of unit resource costs ($\overrightarrow{RCU}$), and true (benchmark) product costs $\overrightarrow{PC^B}$.
6. Using the above, compute vector of selling prices ($\overrightarrow{SP}$) and profit from producing the profit-maximizing product mix ($PROFITT0$).

7. Compute rank vector $\overrightarrow{RANK}$ and matrix of percentage resource consumption ($RES\_CONS\_PAT\_PRCT$).
8. Compute matrix of correlations in resource consumption ($PEARSONCORR$).
9. Log the data from this *Firm* using the *Output* class.

We will discuss each of these steps in turn. Note that we make frequent use of the Hadamard, or Schur, product of two vectors. This operator, denoted by ∘, performs element-wise multiplication of two vectors. Thus, $Z = X \circ Y \Leftrightarrow \forall i, Z_i = X_i \cdot Y_i$. Note that when the product mix decision is exogenous, as in Balakrishnan, Hansen, and Labro (2011), steps 2, 3, and 6 can be omitted. In such an experiment, the firm would produce all available products and the quantity of interest would be a vector of true product costs, $\overrightarrow{PC^B}$.

A-2.1.2.1 Choose random values for the g and d fields

For each firm, the program chooses a value for *g* (dispersion in resource costs) from a uniform distribution with bounds $DISP2\_MIN$ and $DISP2\_MAX$ specified in the input file.[12] For each firm, the program chooses a value for *d* (resource sharing, implemented as density (sparsity) of the resource consumption pattern matrix RES_CONS_PAT) from a uniform distribution with bounds DNS_MIN and DNS_MAX specified in the input file.[13]

A-2.1.2.2 Randomly generate a vector of true product margins (MAR) and a decision vector (DECT0)

$\overrightarrow{MAR}$ is a vector of dimension $[1 \times CO]$ of product margins. We define a product's margin as the ratio of its selling price to its cost. Thus, profitable products have margins greater than 1.0, breakeven products have a margin of exactly 1.0, and loss-making products have a margin between 0.0 and 1.0. This definition allows for easy computation of selling prices from costs and vice versa. We distinguish between the true (full-information) margin, which can only be known inside the *Firm* class, and the margin reported by a cost system, which is based on limited information.

[12] We discuss our random number generator in section A-2.6.1.

[13] We discuss the input file in sections A-2.3 and A-3.4.

The program chooses the true margin of each of the firm's products from $U[MARLB, MARUB]$ where $MARLB$ and $MARUB$ are the upper and lower bounds on margins specified in the input file. The choice of *MARLB* in particular allows for consideration of products whose "true" costs exceed their prices.

We define the benchmark (profit-maximizing) decision vector $\overrightarrow{DECT0}$ as follows. $DECT0_i = 0$ if $MAR_i < 1.0$, and 1 otherwise. Thus, $DECT0_i = 1$ if product i is profitable (or breakeven) and should be produced in the profit-maximizing product mix, and 0 otherwise.

If the product mix decision is exogenous, as in Balakrishnan, Hansen, and Labro (2011), then this step can be omitted, and the variables $\overrightarrow{MAR}$ and $\overrightarrow{DECT0}$ deleted. Alternatively, the margins can be set to a very high arbitrary positive value, and the decision vector to a vector of all 1's.

A-2.1.2.3 Randomly generate the vector of maximum production quantities of the products ($\overrightarrow{MXQ}$), vector of production quantities ($\overrightarrow{QT}$)

$\overrightarrow{MXQ}$ is a $[CO \times 1]$ vector of maximum production quantities. For each product, the program chooses its maximum production quantity from $U[10,40]$. The bounds of the distribution were chosen to generate unit resource prices and selling prices within ranges consistent with those used in Balakrishnan, Hansen, and Labro (2011). Create the vector of production quantities as $\overrightarrow{QT} = \overrightarrow{DECT0} \circ \overrightarrow{MXQ}$. Note that $\forall j$: $0 \leq QT_j \leq MXQ_j$.

If the product mix decision is exogenous, as in Balakrishnan, Hansen, and Labro (2011), then this step can be omitted, and the variables $\overrightarrow{MXQ}$ and $\overrightarrow{QT}$ deleted. Alternatively, arbitrary positive values can be chosen for the maximum quantities, and because $\overrightarrow{DECT0}$ will be a vector of all 1's, $\overrightarrow{QT}$ set to $\overrightarrow{MXQ}$.

A-2.1.2.4 Randomly generate the resource consumption pattern matrix (RES_CONS_PAT)

Generating the resource consumption pattern matrix is complicated because the matrix must satisfy two constraints. First, the matrix must have a density (or sparsity) as specified by the *d* parameter. Second, there are many correlation parameters in the input file that must be satisfied. We

therefore use the heuristic developed by Balakrishnan, Hansen, and Labro (2011), and describe that heuristic here.

Generate the matrix RES_CONS_PAT, of dimension $[RCP \times CO]$, as follows. Remember that RCP and CO refer to the (user-supplied) number of resources available to the firm, and the number of cost objects (products) that the firm can produce, respectively.

a) Generate a random, standard normally-distributed vector of baseline resource $\vec{X}$ of dimension $1 \times CO$, used by all cost objects.
b) Generate $(DISP1 - 1)$ further random standard normally distributed vectors $\overrightarrow{Y_2}$ through $\overrightarrow{Y_{DISP1}}$. Define $\overrightarrow{Z_\iota} = COR1 \times \vec{X} + \sqrt{1 - COR1^2} \cdot \overrightarrow{Y_\iota}$, with $COR1$ drawn from a uniform distribution $U[COR1LB, COR2UB]$, where the bounds are specified in the input file. By multivariate normality, $\vec{X}$ and $\overrightarrow{Z_\iota}$ have a correlation of $COR1$.
c) Generate $(RCP - DISP1)$ further random standard normally distributed vectors $\overrightarrow{Y_{DISP1+1}}$ through $\overrightarrow{Y_{RCP}}$. Define $\overrightarrow{Z_\iota} = COR2 \times \vec{X} + \sqrt{1 - COR2^2} \cdot \overrightarrow{Y_\iota}$ with $COR2$ drawn from $U[COR2LB, COR2UB]$, where the bounds are specified in the input file. By multivariate normality, $\vec{X}$ and $\overrightarrow{Z_\iota}$ have a correlation of $COR2$.
d) For each resource $\vec{X}$ and $\overrightarrow{Z_\iota}$, take the absolute value of all elements. Multiply all elements (in all rows $\overrightarrow{Z_\iota}$ and $\vec{X}$) by 10. Replace all elements with the nearest integer that is greater than or equal to the number (ceiling function).
e) For each resource $\overrightarrow{Z_\iota}$, loop over the elements. For each element, generate a random number $rnd \sim U[0.0, 1.0]$. If $rnd < d$, keep the element as is; otherwise, set the element to zero.
f) Place the vector $\vec{X}$ in the first row and all the vectors $\overrightarrow{Z_\iota}$ in the remaining rows of the matrix in the order as simulated. Check that $\forall i : \exists a_{ij} \neq 0$ and $\forall j : \exists a_{ij} \neq 0$ and that the first row does not have any zeros $(\forall j : a_{1j} \neq 0)$.[14] If not, throw away RES_CONS_PAT and start again at step (a). This ensures that each resource is at least used by 1 cost object, and each cost object uses at least 1 resource.

A-2.1.2.5 Randomly generate the vector of total resource costs ($\overrightarrow{RCC}$), compute vector of unit resource costs ($\overrightarrow{RCU}$), and compute vector of true product costs ($\overrightarrow{PC^B}$)

Given the resource consumption pattern matrix RES_CONS_PAT, true, error-free product costs can be computed as $\overrightarrow{PC^B} = \overrightarrow{RCU} \times RES\_CONS\_PAT$, where $\overrightarrow{RCU}$ is a $[1 \times RCP]$ vector of unit resource costs. We compute $\overrightarrow{RCU}$ by randomly generating $\overrightarrow{RCC}$, a $[1 \times RCP]$ vector of total resource costs, and

[14] This step is redundant if $\vec{X}$ is correctly simulated as a vector without zeros, but can be used to check that the program executes correctly.

then dividing each element of $\overrightarrow{RCC}$ by the total units of that resource consumed. This procedure is nontrivial because the input file specifies constraints on the concentration of resources in $\overrightarrow{RCC}$. In the following paragraphs, we elaborate on each of the items just described.

A-2.1.2.5.1 Compute vectors of unit resource consumption, $\overrightarrow{TRU}$ and $\overrightarrow{MAXRU}$

Let $\overrightarrow{MAXRU}$ be a $[RCP \times 1]$ vector of maximum unit resource consumption. This can be computed as: $\overrightarrow{MAXRU} = RES\_CONS\_PAT \times \overrightarrow{MXQ}$, where RES_CONS_PAT is the $[RCP \times CO]$ resource consumption pattern matrix and $\overrightarrow{MXQ}$ is the $[CO \times 1]$ vector of maximum production quantities of the products. Element i of $\overrightarrow{MAXRU}$ is then the number of units of resource i needed to produce $\overrightarrow{MXQ}$.

Let $\overrightarrow{TRU}$ be a $[RCP \times 1]$ vector of unit resource consumption needed to produce product mix $\overrightarrow{QT}$. This can be computed as: $\overrightarrow{TRU} = RES\_CONS\_PAT \times \overrightarrow{QT}$, where RES_CONS_PAT is the $[RCP \times CO]$ resource consumption pattern matrix and $\overrightarrow{QT}$ is the $[CO \times 1]$ vector of production quantities of the products. Element i of $\overrightarrow{TRU}$ is then the number of units of resource i needed to produce $\overrightarrow{QT}$. Note that $\overrightarrow{TRU} \leq \overrightarrow{MAXRU}$ since $\overrightarrow{QT} \leq \overrightarrow{MXQ}$.

When decisions are endogenous, as in Anand, Balakrishnan, and Labro (2017), the product mix $\overrightarrow{QT}$ is endogenous. Thus, the total units of each resource needed, $\overrightarrow{TRU}$, will be a function of $\overrightarrow{QT}$. However, when decisions are exogenous, as in Balakrishnan, Hansen, and Labro (2011), the firm is assumed to always produce all of its available products. Thus, the total units of each resource needed, $\overrightarrow{TRU}$, will always equal the maximum, $\overrightarrow{MAXRU}$. In the following, we ensure that the resource concentration constraint is met when the firm produces maximum quantities of all products, and compute (infer) unit resource costs $\overrightarrow{RCU}$. We then re-compute resource consumption using the actual product mix $\overrightarrow{QT}$.

A-2.1.2.5.2 Compute initial vector of dollar resource consumption, $\overrightarrow{RCC}$, given resource concentration constraint

$\overrightarrow{RCC}$ is a $[1 \times RCP]$ vector of resource consumption, in dollars (i.e. element i of the vector is the total dollar value of resource i needed to produce the current product mix). We generate this vector randomly by distributing total resources (TR, a parameter supplied in the program's input file) across the RCP elements of the vector (thus, the sum of all elements of the vector will equal TR). We ensure that the top $DISP1$ resources account for $DISP2$ percent of TR ($DISP1$ and $DISP2$ are specified in the program's input file). This statement implies 3 constraints. The statement itself can be written as $\sum_{i=1}^{DISP1} RCC_i = DISP2 \cdot TR$, the first constraint. The first constraint implies a constraint on the remaining $(RCP - DISP1)$ resources. Specifically, the values of the remaining resources must sum to $(1 - DISP2) \cdot TR$, and therefore $\sum_{i=DISP1+1}^{RCP} RCC_i = (1 - DISP2) \cdot TR$. Finally, the top $DISP1$ resources must be the top resources – the dollar value of each of these resources must exceed that of the remaining resources. In the following paragraphs, we describe our method for generating a vector of resource costs, $\overrightarrow{RCC}$, that satisfies these constraints.

We begin by noting that, for some values of $DISP1$ and $DISP2$, no vector of resource costs exists. To see this, assume that the elements of $\overrightarrow{RCC}$ are sorted in descending order, with the largest resource at index 1. If resource concentration is as low as possible (i.e. every element of $\overrightarrow{RCC}$ equals TR/RCP), then the sum of the first $DISP1$ resources will be $DISP1 \cdot \frac{TR}{RCP}$. Since, by the first constraint, this sum must equal $DISP2 \cdot TR$, we can write $DISP1 \cdot \frac{TR}{RCP} = DISP2 \cdot TR$, which simplifies to $\frac{DISP1}{RCP} = DISP2$. If resource concentration is higher than its minimum, $DISP2 \geq \frac{DISP1}{RCP}$. Valid vectors $\overrightarrow{RCC}$ exist only when this inequality is satisfied. In the following, we refer to the top $DISP1$ resources as the *big* resources, and the remaining $(RCP - DISP1)$ resources as the *small* resources.

Generating *Big* Resource Values

There is a minimum allowable value for the *big* resources, r_{min}. We derive r_{min} by noting that the sum of the *small* resources must equal $(1 - DISP2) \cdot TR$. Thus, if the minimum value of *big* resource is r_{min}, the largest allowable value of a *small* resource is also r_{min}. Since the sum of the *small* resources must equal $(1 - DISP2) \cdot TR$, then:

$$r_{min} = \frac{(1 - DISP2) \times TR}{RCP - DISP1}$$

The minimum allowable value for *big* resources implies a maximum allowable value for *big* resources: if one of the *big* resources is too large, then the remaining *big* resources will have to be small (since the sum of the big resources must equal $DISP2 \cdot TR$), and one might violate the r_{min} constraint. Thus, when generating the first *big* resource, we must leave enough available resource dollars so that all the other *big* resources can be greater than r_{min}. Denoting the maximum value of the first resource as r_1^{max}:

$$(DISP2 \cdot TR) - r_1^{max} = (DISP1 - 1) \cdot r_{min}$$

The left-hand side is the remaining dollar value of the *big* resources. The right-hand side is the number of remaining *big* resources times their minimum value, r_{min}. Rearranging:

$$(DISP2 \cdot TR) - (DISP1 - 1) \cdot r_{min} = r_1^{max}$$

When determining the maximum for the second *big* resource, we need to consider the amount assigned to the first resource. Denote the value of the first resource as r_1, where $r_1 \leq r_1^{max}$. Using the same logic as above:

$$r_2^{max} = (DISP2 \cdot TR - r_1) - (DISP1 - 2) \cdot r_{min}$$

Generalizing this, it can be shown that for any *big* resource i,

$$r_i^{max} = \left(DISP2 \cdot TR - \sum_{k=1}^{i-1} r_k \right) - (DISP1 - i) \cdot r_{min}$$

We now state our heuristic for generating random values of big resources, given the user-supplied values of TR, $DISP1$, and $DISP2$, and assuming that $DISP2 \geq \frac{DISP1}{RCP}$.

1. Calculate r_{min} as $\left(r_{min} = \frac{(1-DISP2)\cdot TR}{RCP-DISP1}\right)$.
2. Calculate r_1^{max} as $(r_1^{max} = (DISP2 \cdot TR) - (DISP1 - 1) \cdot r_{min})$.
3. Adjust r_{min} upwards by $(r_1^{max} - r_{min}) \times 0.025$.[15]
4. For $i = 1 \ldots (DISP1 - 1)$
    a. Compute r_i^{max}
    b. Generate $r_i = U[r_{min}, r_i^{max}]$
5. Set r_{DISP1} equal to $DISP2 \cdot TR$ minus the sum of the previous *big* resources. This ensures that the sum of all *big* resources is exactly $(DISP2 \cdot TR)$.
6. Find the biggest resource among the *big* resources and move it to the head of the list of *big* resources.

Generating *Small* Resource Values

The procedure for generating the *small* resource values is relatively straightforward. It generates a list of numbers drawn from $U[0.05, 0.95]$ and then normalizes the list so its sum is 1.0. It then multiplies each element of the list by the total value of *small* resources, $(1 - DISP2) \cdot TR$. It then ensures that all *small* resource values are less than the smallest *big* resource value.

1. Generate $RCP - DISP1$ random numbers from $U[0.05, 0.95]$.
2. Normalize the list (divide each element by the sum of the list) so that the sum of the list is 1.0.
3. Multiply every element of the list by $(1 - DISP2) \cdot TR$. Now the sum of this list contains the proper fraction of total resources.
4. The following step ensures that no *small* resource is larger a *big* resource.
5. While (maximum *small* resource) > (minimum *big* resource) do:
    a. Sort the list of small pool resources in descending order.
    b. For $i = 1 \ldots n$, where n is the number of *small* resources
        i. $\text{Overage} = MAX(r_i - [\min big \text{ resource}], 0.0)$
        ii. $r_i = r_i - \text{Overage}$
        iii. $r_{n-i+1} = r_{n-i+1} + \text{Overage}$
6. Shuffle the list of small pool resources

[15] We found that if we do not make this adjustment to r_{min}, then there is too little variance in the values of the *small* resources. We arrived at the 2.5% adjustment factor through simulation and trial-and-error.

A-2.1.2.5.3 Computing the vector of unit resource costs ($\overrightarrow{RCU}$)

At this point, we have a $[1 \times RCP]$ vector $\overrightarrow{RCC}$ of total resource costs. Divide each element of this vector by the corresponding element of $\overrightarrow{MAXRU}$, the vector maximum unit resource consumption computed in section A-2.1.2.5.1. The resulting vector $\overrightarrow{RCU}$ contains unit resource prices. Note that $\overrightarrow{RCU}$ is computed using the maximum production quantities of all products.

A-2.1.2.5.4 Re-compute vector of total resource costs $\overrightarrow{RCC}$

The initial vector $\overrightarrow{RCC}$ is based on maximum production quantities of all products. It spreads the total resource costs TR across the individual resources, consistent with the user-supplied resource dispersion parameters $DISP1$ and $DISP2$. For experiments in which the product mix decision is exogenous or unneeded, such as Balakrishnan, Hansen, and Labro (2011), this step is unnecessary. However, when the product mix decision is endogenous, as in Anand, Balakrishnan, and Labro (2017), the vector $\overrightarrow{RCC}$ is a function of the product mix decision. Thus, for the benchmark (profit-maximizing) product mix $\overrightarrow{QT}$, re-compute $\overrightarrow{RCC}$ as $\overrightarrow{RCC} = \overrightarrow{RCU} \circ \overrightarrow{TRU}$, where $\overrightarrow{TRU}$ is the vector of unit resource consumption needed to produce product mix $\overrightarrow{QT}$ computed in section A-2.1.2.5.1.

A-2.1.2.5.5 Compute vector of true product costs ($\overrightarrow{PC^B}$)

Compute the vector of true product costs as $\overrightarrow{PC^B} = \overrightarrow{RCU} \times RES\_CONS\_PAT$. Recall that $\overrightarrow{RCU}$ is the vector of unit prices of the resources, and that element a_{ij} of the resource consumption pattern matrix RES_CONS_PAT is the number of units of resource i needed to produce one unit of product j. $\overrightarrow{PC^B}$ is then a linear combination of the rows of RES_CONS_PAT.

A-2.1.2.6 Compute vector of selling prices ($\overrightarrow{SP}$) and profit from producing the profit-maximizing product mix ($PROFITT0$)

Using the above, we compute the vector of sales prices $\overrightarrow{SP}$ through the Hadamard product (element-wise multiplication) of the margins ($\overrightarrow{MAR}$) and total cost ($\overrightarrow{PC^B}$) vectors, $\overrightarrow{SP} = \overrightarrow{MAR} \circ \overrightarrow{PC^B}$. We then compute the revenue and profit of the benchmark (profit-maximizing) product mix. Revenue

$TRV = \overrightarrow{SP} \times \overrightarrow{QT}$. Total costs $TCT0 = \overrightarrow{RCU} \times RES\_CONS\_PAT \times \overrightarrow{QT}$. Profit $PROFITT0 = TRV - TCT0$.

If the product mix decision is exogenous, as in Balakrishnan, Hansen, and Labro (2011), then this step and the variables $\overrightarrow{SP}$ and $PROFITT0$ can be omitted.

A-2.1.2.7 Compute rank vector ($\overrightarrow{RANK}$) and matrix of percentage resource consumption (RES_CONS_PAT_PRCT)

The vector $\overrightarrow{RANK}$ is computed by sorting the vector of total resource consumption in dollars ($\overrightarrow{RCC}$) in descending order, and then replacing each element with its index from the unsorted vector. Thus, $\overrightarrow{RANK}$ contains a list of resource indices, sorted by the dollar value of the resource. The vector $\overrightarrow{RANK}$ is useful when forming cost pools and choosing cost drivers.

To compute the matrix of resource consumption patterns expressed in percentages, $RES\_CONS\_PAT\_PRCT$, we perform the following steps. Iterate over the rows of RES_CONS_PAT. For each row i, perform element-wise multiplication with the vector of production quantities $\overrightarrow{QT}$. Letting $\vec{r_i}$ denote the i^{th} row of RES_CONS_PAT, we are constructing a vector $\vec{v}$ such that $\vec{v} = \vec{r_i} \circ \overrightarrow{QT}$. Thus, $\vec{v}$ is a vector of the total number of units of resource i needed for each product. Divide each element of $\vec{v}$ by the i^{th} element of the vector of total resource usage $\overrightarrow{TRU}$ $\left(\vec{w} = \frac{\vec{v}}{TRU_i}\right)$. Each element of $\vec{w}$ is the percentage of resource i needed to produce product j. Set the i^{th} row of the matrix $RES\_CONS\_PAT\_PRCT$ to $\vec{w}$.

A-2.1.2.8 Compute correlations in resource consumption (PEARSONCORR)

$PEARSONCORR$ is a symmetric matrix of dimension $[RCP \times RCP]$. The value of element $a_{ij} = a_{ji}$ is the correlation between rows i and j of the $RES\_CONS\_PAT\_PRCT$ matrix. We compute these correlations using numerical library Meta Numerics.

A-2.1.2.9 Log the data from this **Firm** *using the* **Output** *class*

See section A-2.4 for descriptions of the output files.

A-2.1.3 Methods

A-2.1.3.1 Calculate Resource Consumption (CalcResConsumption)

This method takes a single argument, a column vector of production quantities, $\vec{q}$, and returns a vector ($\overrightarrow{TRU}$) containing the total consumption (number of units) of each resource, given production quantities $\vec{q}$.

$$CalcResConsumption(\vec{q}) = RES\_CONS\_PAT \times \vec{q}$$

Thus, the method multiplies the resource consumption pattern matrix by its argument and outputs a column vector of the number of units of each resource needed to produce $\vec{q}$. This method can be called without the caller having knowledge of RES_CONS_PAT, thus hiding the production technology within the *Firm* object. It represents a situation in which a firm produces a product mix and observes the resulting consumption of resources.

A-2.1.3.2 Calculate Total Costs (CalcTotCosts)

This method takes a single argument, a column vector of production quantities, $\vec{q}$, and returns the total cost of producing $\vec{q}$, a scalar.

$$CalcTotCosts(\vec{q}) = \overrightarrow{RCU} \times RES\_CONS\_PAT \times \vec{q}$$

A-2.1.3.3 Calculate True Product Costs (CalcTrueProductCosts)

This method returns a row vector containing the true (full-information) costs of the firm's products.

$$CalcTrueProductCosts(\,) = \overrightarrow{RCU} \times RES\_CONS\_PAT$$

A-2.1.3.4 Calculate the Optimal Production Decision (CalcOptimalDecision)

This method computes the profit-maximizing decision ($\overrightarrow{DECT0}$), given true costs and selling prices. Each element i of the return vector $\overrightarrow{DECT0}$ is 1 if the product is profitable, and 0 otherwise. The method first calls *CalcTrueProductCosts()* and stores the result as a vector $\overrightarrow{PC^B}$. For each product i, $\overrightarrow{DECT0}$ is 1 if $SP_i \geq PC_i^B$, and 0 otherwise.

A-2.1.3.5 Calculate the Optimal Production Quantities (CalcOptimalVol)

This method returns the Hadamard product (element-wise multiplication) of the vectors $\overrightarrow{MXQ}$ (maximum production quantities) and $\overrightarrow{DECT0}$ (optimal production decision). The method calls *CalcOptimalDecision()* and stores the result as $\overrightarrow{DECT0}$. It then returns $\overrightarrow{MXQ} \circ \overrightarrow{DECT0}$.

A-2.2 CostSys Class

We model a cost system as an entity that possesses a subset of the information encapsulated in a firm. As we use a Leontief technology that we represent as a matrix, we operationalize limited information as knowledge of some of the rows of the RES_CONS_PAT matrix. Because each row represents the consumption of a single resource by all the firm's products, we assume the cost system knows the consumption of some, but not all, of the resources by the firm's products. This is analogous to direct and indirect costs; firms have precise information about the consumption of the former by their products, but not of the latter.

We further assume that, for any production decision, the accounting system records the total value (in units and dollars) of the consumption of resources by the firm (through the *CalcResConsumption()* method above). The cost system assigns resources to cost pools and, by choosing a cost driver for each pool, allocates costs from each pool to its products to estimate its products' costs.

A-2.2.1 Fields and Properties

Table 2: Fields and properties of the CostSys class.

Field	Data Type	Public Property	Description
Firm	Firm	---	Pointer to the firm object upon which this cost system is based.
B	Array of (list of integer)	B_as_String	B[i] is the set of resource indexes that are aggregated into activity cost pool i. For example, consider a 2-pool system with 6 resources. A sample B array is [[1, 4], [3, 5, 2, 6]], indicating that resources 1 and 4 are aggregated into the first activity cost pool, and the remaining resources are aggregated into the second pool. The property returns a string representation of the array.
D	Array of (list of integer)	D_as_String	D[i] is the set of resource indexes that constitute drivers for activity cost pool i. For example, consider a 2-pool system with 6

Field	Data Type	Public Property	Description
			resources. A sample D array is [[1], [3]], indicating that resource 1 is the driver for pool 1, and resource 3 is the driver for pool 2. The property returns a string representation of the array.
a	Integer	*A*	Number of activity cost pools. Note: The input file contains a parameter ACP. The value of *a* is chosen from the input parameter ACP.
p	Integer	*P*	Method for grouping resources into activity cost pools. Note: The input file contains a parameter PACP. The value of *p* is chosen from the input parameter PACP.
r	Integer	*R*	Driver selection method. Note: The input file contains a parameter PDR. The value of *r* is chosen from the input parameter PDR.

We next discuss the methods for grouping resources into cost pools and for driver selection.

A-2.2.2 Constructor

Values for the properties A, P, and R are passed as arguments to the constructor. The user specifies valid values of these parameters in the input file. The program allows the user to specify multiple values for each of these parameters, and the main program loops over each value, thereby creating a separate cost system for every combination of these parameters. Using the supplied values of A, P, and R, the constructor creates the *B* and *D* arrays.

If $A = 1$ (a single activity cost pool), then:

- The array *B* contains a single list. That list contains the index of every resource, sorted in descending order by resource size in dollars.
- The array *D* contains a single list. If the driver selection method (R) is bigpool, then the program chooses the first element of *B* (largest resource) as the driver. If the driver selection method is indexed driver, then the first NUM resources are chosen as the composite driver. NUM is a user-supplied parameter from the input file.

If A is greater than 1 (multiple activity cost pools), then the program considers the aggregation method for grouping resources into activity pools, as modeled by parameter (P). This parameter can take on the following values. In the following, MISCPOOLSIZE is a user-supplied parameter that indicates

the desired size of the miscellaneous resource pool. MISCPOOLSIZE is a number in the range [0.0, 1.0]

that indicates the target fraction of total resources that should be assigned to the miscellaneous pool.

0. If $P = 0$, then the program seeds the first $(A - 1)$ pools with the largest $(A - 1)$ resources. All remaining resources assigned to the last pool, which is designated the miscellaneous pool. In other words, each large resource gets its own pool and serves as a driver (i.e. it's a direct cost). The remaining resources are aggregated into a miscellaneous pool whose driver is the largest miscellaneous resource.
1. If $P = 1$, then the program seeds the first $(A - 1)$ pools with the largest $(A - 1)$ resources. The program then finds the highest correlation for the remaining resources, and assigns that high-correlation resource to the relevant pool. It then checks to see if the ratio of unassigned resources to total resources is greater than MISCPOOLSIZE. If yes, find the next resource with the highest correlation to an assigned resource, assign that resource to a pool, and recheck the ratio. When the ratio of unassigned resources to total resources is less than MISCPOOLSIZE, then pool the remaining resources into the miscellaneous pool. The program ensures that at least one nonzero resource is assigned to the miscellaneous pool.
2. If $P = 2$, then the program seeds each of the $(A - 1)$ cost pools with the largest $(A - 1)$ resources. It then allocates the remaining resources to the $(A - 1)$ pools at random until the ratio of unassigned resources to total resources is less than MISCPOOLSIZE. The remaining resources are then assigned to the miscellaneous pool. The program ensures that at least one nonzero resource is assigned to the miscellaneous pool.
3. If $P = 3$, then the program seeds the first pool with the largest resource. It then iterates over the other pools. For each remaining pool, select a seed resource that is the largest of the remaining, unassigned resources, and assign it to the pool. Form a correlation vector (a list), which is the correlation of each resource in the remaining unassigned resources with the seed resource. If the highest correlation is greater than CC, a user-supplied correlation cutoff, and there are enough remaining resources to fill the remaining pools and satisfy the constraint on the miscellaneous pool size, then assign resource with the highest correlation to the current pool. Once there are just as many resources remaining as there are pools, assign one resource to each remaining pool.

Drivers are chosen as follows:

0. If $R = 0$, then the bigpool driver selection method is in use. For each list in *B*, sort it in descending order by resource size. Then choose the first (largest) element as the driver for that pool. Thus, $D[i] = B[i][0]$.
1. If $R = 1$, then the indexed driver selection method is in use. For each list in *B*, sort it in descending order by resource size. Then choose the first (largest) NUM elements as the drivers for that pool, where NUM is a user-supplied parameter. Those NUM resources will serve as an indexed (composite) driver for the pool.

A-2.2.3 Methods

The *CostSys* object has two methods. The first, *CalcReportedCosts()*, serves as the heart of any numerical experiment based on this code. *CalcReportedCosts()* takes as its input a product mix, M, and, using the information in the *B* and *D* array, computes the estimated (reported) costs of *every* product available to the firm, based on the resource consumption pattern that results from producing M. This method can be used in models without a decision-making objective, such as Balakrishnan, Hansen, and Labro (2011). The second method, *EquilibriumCheck()*, iterates *CalcReportedCosts()*. After each iteration, it updates M based on the reported costs, adding profitable products to M and removing unprofitable ones. It continues this process until a terminal outcome is found. This process is related to that described in Anand, Balakrishnan, and Labro (2017). This second method is required only to iterate toward an informationally consistent cost system / decision pair.

A-2.2.3.1 Calculate Reported Costs (CalcReportedCosts)

CalcReportedCosts() takes two arguments, an *InputParameters* object (see section A-2.3) and a product mix decision $\overrightarrow{DECF0}$, specified by binary row vector of length CO in which a value of 1 (0) in element i indicates that product i will (not) be produced. It returns a row vector of length CO containing the estimated, or reported, cost of each product available to the firm. The method works as follows:

1. Create a column vector $\overrightarrow{q0}$ as $\overrightarrow{q0} = \overrightarrow{MXQ} \circ \overrightarrow{DECF0}$. This vector contains the production quantities of each product. $q0_i$ is 0 if $DECF0_i$ is 0, and MXQ_i if $DECF0_i$ is 1.
2. Calculate the vector of total unit resource consumption, $\overrightarrow{TRU\_F}$, required to produce $\overrightarrow{q0}$. Thus, $\overrightarrow{TRU\_F} = CalcResConsumption(\overrightarrow{q0})$.
3. Calculate the vector of total resource consumption in dollars, $\overrightarrow{RCC\_F}$, as $\overrightarrow{RCC\_F} = \overrightarrow{RCU} \circ \overrightarrow{TRU\_F}$.
4. Compute the dollar value of resources in each activity cost pool and store in array AC.
    a. The value of each element of AC is obtained by iterating over the B array. Each list in B is a list of resource indices assigned to a pool. Iterate over that list, replacing each element with the corresponding value from $\overrightarrow{RCC\_F}$. Then sum that list to get the dollar value in that activity cost pool.
    b. Since some decisions specify the production of only a small subset of the available products, it is possible that some activity cost pools will have $0 assigned to them. Delete each such pool, and create an array AC' that contains only pools with nonzero

amounts. Remove the corresponding lists from array B to create array B', and from array D to create array D'.

5. Choose the largest resource(s) for each activity cost pool for which the resource usage is not zero. Choose a single resource if the driver selection method (r) is bigpool, or NUM resources if indexed drivers are used.
6. Compute rates and product costs as follows:
    a. Bigpool driver selection method
        i. For each pool, rate is determined by dividing the amount in the pool by the total units of the driver for that pool.
        ii. For each product, calculate the amount to be allocated from each pool by multiplying the rate for that pool (calculated in the previous step) by the number of units of the driver consumed by that product. Then sum the allocations from each pool to obtain that product's cost.
    b. Indexed driver selection method
        i. For each pool, divide the amount in the pool by NUM. This will provide NUM subpools. For each, compute a rate as in 6.a.i above.
        ii. For each product, calculate the amount to be allocated from each subpool by multiplying the rate for that subpool (calculated in the previous step) by the number of units of the driver consumed by that product. Then sum the allocations from each subpool to obtain that product's cost.

A-2.2.3.2 Iterate Until a Terminal Outcome is Found (EquilibriumCheck)

This method takes two arguments, an *InputParameters* object (see section A-2.3) and a product mix decision $\overrightarrow{DECF0}$, specified by binary row vector of length CO in which a value of 1 (0) in element i indicates that product i will (not) be produced. It returns a 2-tuple containing a "stop code" and an ending product mix. The stop codes are "Equilibrium", "Cycle" or "Zero Mix".

Starting from the initial decision $\overrightarrow{DECF0}$, this method iterates the *CalcReportedCosts()* method, which returns a vector of reported costs $\overrightarrow{PC^R}$. Upon observing $\overrightarrow{PC^R}$, it compares these reported costs to the vector of selling prices $\overrightarrow{SP}$ and updates the decision $\overrightarrow{DECF0}$ to a new decision $\overrightarrow{DECF1}$. It drops products from $\overrightarrow{DECF0}$ that appear unprofitable, given $\overrightarrow{PC^R}$, and adds products that appear profitable but were not produced under $\overrightarrow{DECF0}$. If $\overrightarrow{DECF0} = \overrightarrow{DECF1}$, then the decision is a fixed point, and iteration stops. The stop code reports "Equilibrium". Other possible outcomes are a set of decisions that form a "Cycle", e.g. $\overrightarrow{DECF0} \rightarrow \overrightarrow{DECF1} \rightarrow \overrightarrow{DECF2} \rightarrow \overrightarrow{DECF3} \rightarrow \overrightarrow{DECF2}$. In this example, $\overrightarrow{DECF2}$ and $\overrightarrow{DECF3}$ form a cycle and will repeat infinitely unless detected. Another possible outcome is the "Zero

Mix", in which case the firm goes into a death spiral that ends in zero production. Anand, Balakrishnan, and Labro (2017) found that a terminal outcome is typically detected within 6 iterations, even when the decision space is very large, so computer time is not a concern with this method.

A-2.3 InputParameters Class

The *InputParameters* class processes the input file, which contains user-supplied parameters for the numerical experiment. The *InputParameters* constructor validates all user-supplied parameters and stores these in an *InputParameters* object for use during the experiment. Users should add or remove parameters as needed for their own numerical experiments. We describe the process for modifying this class in section A-3.

A-2.3.1 Fields and Properties

Each field in the *InputParameters* class has a corresponding property with the same name. All fields are private with lower-case variable names, while their corresponding properties are public with upper-case names. For brevity, Table 3 only lists the public properties.

In C#, class designers can exercise fine control over data access through properties, which consist of *get* accessors and *set* accessors. *Get* accessors return the value of a field (or some transformation of this value), and set accessors set the value of a field, given an input value. We chose to make all *get* accessors of the *InputParameters* class public, and hence read-only. We also chose to make the *set* accessors private, and therefore only accessible from within the *InputParameters* constructor. Thus, fields of this class can only be set once, as assigned in the constructor, and other code cannot accidentally change the value of a field in this class.

Table 3: Fields and Properties of InputParameters class.

Public Properties	Data Type	Description	Valid Values and Constraints
TR	Double	Total value of resources. This is only used to compute the initial $\overrightarrow{RCC}$ vector.	$TR > 0$
CO	Integer	Number of products (cost objects)	$CO > 0$
RCP	Integer	Number of resources	$RCP \geq CO > 0$

Public Properties	Data Type	Description	Valid Values and Constraints
NUM_FIRMS	Integer	The number of firms that will be created. This is the basis of the sample size.	$NUM\_FIRMS > 0$
DISP1	Integer	One of two parameters used to specify dispersion in resource costs. The top DISP1 resources will account for DISP2 percent of the total resources in the initial $\overrightarrow{RCC}$ vector created for each firm. For example, 10 resources (DISP1) might account for 75% (DISP2) of total cost.	$0 < DISP1 \leq RCP$
DISP2_MIN *DISP2_MAX (g)*	Double	See description as for DISP1. Values of DISP2 are chosen from a uniform distribution U[DISP2_MIN, DISP2_MAX]. This parameter is sometimes referred to as *g* in the code.	$DISP2\_MIN \geq \frac{DISP1}{RCP}$ $DISP2\_MAX \geq DISP2\_MIN$ $1 > DISP2\_MAX$
DNS_MIN *DNS_MAX (d)*	Double	Density of the resource consumption pattern matrix (RES_CONS_PAT). A density of 0.8 means that approximately 20% of the elements of the matrix are zero. Values of DNS are chosen from a uniform distribution U[DNS_MIN, DNS_MAX]. This parameter is sometimes referred to as *d* in the code.	$0 < DNS\_MIN$ $DNS\_MIN \leq DNS\_MAX$ $DNS\_MAX < 1$
ACP (a)	List of integers	The number of activity cost pools in the cost systems. If multiple values are specified, the program will loop over them. This parameter is sometimes referred to as a in the code.	$RCP \geq ACP > 0$
PACP (p)	List of integers	A flag indicating how resources will be pooled into activity cost pools in the cost system. This parameter is sometimes referred to as *p* in the code. If multiple values are specified, the program will loop over them. *p* takes a value of 0, 1, 2 or 3 as explained in section 2.2.2. This parameter is sometimes referred to as p in the code.	$PACP \in [0,1,2,3]$
PDR (r)	List of integers	A flag indicating driver selection in the costing system. PDR==0 indicates that the driver is bigpool. PDR==1 indicates that the driver is indexed with NUM resources included in the index.	$PDR \in [0,1]$

Public Properties	Data Type	Description	Valid Values and Constraints
		This parameter is sometimes referred to as r in the code.	
NUM	Integer	The number of resources to be included in indexed drivers.	$NUM > 0$ and $NUM \times ACP \leq RCP$
MISCPOOLSIZE	Double	The fraction of resources that should be in the last activity cost pool.	Between 0 and 1, exclusive
There are RCP resources whose total dollar value will be TR. Assume these are sorted in descending order by dollar value. For the next 4 rows, define the large resources as the first $DISP1$ resources, and the small resources as the remaining $(RCP - DISP1)$ resources.			
COR1LB	Double	Lower bound on COR1, the correlation between the large resources.	In $[-1.0, +1.0]$
COR1UB	Double	Upper bound on COR1, the correlation between the large resources.	In $[-1.0, +1.0]$
COR2LB	Double	Lower bound on COR2, the correlation between the small resources.	In $[-1.0, +1.0]$
COR2UB	Double	Upper bound on COR2, the correlation between the small resources.	In $[-1.0, +1.0]$
CC	Double	Correlation cutoff for correlation-based methods of assigning resources to pools.	In $[0.0, 1.0]$
MARLB	Double	Product margins are determined randomly, and MARLB is the lower bound. Margin is defined as price per unit divided by total cost per unit, so a product that earns zero profit will have a margin of 1.0.	$MARLB > 0$
MARUB	Double	Product margins are determined randomly, and MARUB is the upper bound. Margin is defined as price per unit divided by total cost per unit, so a product that earns zero profit will have a margin of 1.0.	$MARUB \geq MARLB > 0$
STARTMIX	Integer	A value of zero means that the starting mix for the consistency loop is the benchmark mix. A value of 1 means that the starting mix can be adjusted using the EXCLUDE parameter.	0 or 1
EXCLUDE	Double	This is used to adjust the starting mix. This is ignored if STARTMIX == 0. A value of x means that x% of the products from the benchmark mix will be excluded from the starting mix of the *EquilibriumCheck()* method of the *CostSys* class.	In $[0.0, 1.0]$

Public Properties	Data Type	Description	Valid Values and Constraints
USESEED	Boolean	If true, the random number generator will be initialized with the seed provided by the SEED parameter of the input file. If false, the random number generator will be initialized using the system clock.	True or False (case-insensitive)
SEED	Unsigned integer	The seed value used to initialize the random number generator. This parameter is only used when USESEED == 1.	
HYSTERESIS	Double	If a product is not in the current product mix, then in order to add it to the mix, its reported margin, defined as (sales price minus reported cost) divided by reported cost, must exceed this threshold. If a product is in the current product mix, then to drop it from the mix, its margin must be less than this threshold. Setting this parameter to zero means no hysteresis; products that appear (do not appear) profitable are included (excluded) from the mix.	$HYSTERESIS \geq 0$

A-2.3.2 Constructor

The constructor reads the user-supplied input file.[16] It ignores comment lines that begin with two forward slashes (//), and for each data line, attempts to assign the user-supplied value to the appropriate field of the *InputParameters* class. The constructor checks whether all fields have been assigned, and generates an exception otherwise.

The constructor enforces some, but not all, of the constraints specified in Table 3. Other constraints are enforced through the *EnforceConstraints()* method (see section A-2.3.3). The constructor relies on properties' set accessors to enforce single-parameter constraints.

Finally, the constructor sets the seed for the random number generator (see section A-2.6.1). Users have the option of supplying a seed, thereby facilitating replication in their experiments, or

[16] Section A-3 provides for instructions for modifying the input file and running the code.

allowing the program to set a seed based on the system time. We have found it useful to fix the seed, to avoid repetitious calculations when error-checking the code.

A-2.3.3 Methods

Because some constraints rely on the values of multiple parameters, and the constructor cannot guarantee the order in which parameter values are read from the input file, the constructor only enforces constrains that rely on a single parameter (e.g. number of resources, RCP, must be positive).

The *InputParameters* class contains a single method, *EnforceConstraints()*. Users should update this method as needed to supply other constraints on input parameter values, e.g. $RCP \geq CO$. This method throws an exception if a constraint violation is detected.

A-2.4 Output Class

The *Output* class provides static methods to write output from numerical experiments. The output is formatted as comma-separated value (CSV) files, which are platform independent. Users should modify this class to suit the data needs of their experiments.

The class stores the names of the output files as strings. It also provides temporary, in-memory, storage for output.[17] Thus, when other code logs output data, that data is stored in memory. At the end of the experiment, all data is written to the output files. We made this choice to reduce execution times. However, it is easy to modify this class to write data to files as early as possible. Users should consider this if they modify the code, find that it crashes, and want to analyze the intermediate output.

A-2.4.1 Fields

All fields are private and can only be modified through the class's methods.

Table 4: Fields of Output class.

Field	Data Type	Description
file_Firm_SUM	String	Name of file containing summary information about firms
file_Firm_RESCON	String	Name of file containing $\overrightarrow{RCC}$ (resource consumption) and $\overrightarrow{RCU}$ (unit resource price)

[17] We use C#'s StringBuilder class, which provides support for mutable strings.

Field	Data Type	Description
file_Firm_PRODUCT	String	Name of file containing information about each firm's products (resource concentration (g), resource sharing (d), product margins ($\overrightarrow{MAR}$), maximum production quantities ($\overrightarrow{MXQ}$), selling prices ($\overrightarrow{SP}$), profit-maximizing decision ($\overrightarrow{DECT0}$))
file_CostSys_SUM	String	Name of file containing summary information about cost systems (assignment of resources to cost pools (B), driver choices (D))
file_CostSys_ERROR	String	Name of file containing information about error in reported costs
file_CostSys_LOOP	String	Name of file containing the results of executing the *EquilibriumCheck()* method of the *CostSys* class.
sb_Firm_SUM	StringBuilder	Temporary, in-memory storage for output file Firm_SUM
sb_Firm_RESCON	StringBuilder	Temporary, in-memory storage for output file Firm_RESCON
sb_Firm_PRODUCT	StringBuilder	Temporary, in-memory storage for output file Firm_PRODUCT
sb_CostSys_SUM	StringBuilder	Temporary, in-memory storage for output file CostSys_SUM
sb_CostSys_ERROR	StringBuilder	Temporary, in-memory storage for output file CostSys_ERROR
sb_CostSys_LOOP	StringBuilder	Temporary, in-memory storage for output file CostSys_LOOP

A-2.4.2 Methods

A-2.4.2.1 Creating Output Files (CreateOutputFiles)

This method should be called only once during a simulation, before any calls to the logging methods. It creates each output file and writes a header to each file. Users should modify the headers, and hence the structure of the output files, in this method.

A-2.4.2.2 Logging Information About a Firm (LogFirm)

This method stores information about a single firm, and should be called after a firm is created through its constructor. The method logs the information in 3 files, Firm_SUM, Firm_PRODUCT, and Firm_RESCON. Descriptions of each file follow:

A-2.4.2.2.1 Firm_SUM.csv

This file contains summary information about firms, with one row per firm.

Table 5: Data stored in the file Firm_SUM.csv.

Field	Description
FirmID	A unique identifier for this firm.

Field	Description
G	The randomly selected value of $DISP2$ (the top $DISP1$ resources will account for $DISP2$ percent of the total resources in the initial $\overrightarrow{RCC}$ vector created for each firm).
D	The randomly selected value of the density of the resource consumption pattern matrix.
numRCP	Number of resources
numCO	Number of products (cost objects)
Benchmark revenue	The revenue the firm realizes if it produces its benchmark (profit-maximizing) product mix.
BenchmarkTotCost	The total cost to the firm if it produces its benchmark (profit-maximizing) product mix.
BenchmarkProfit	The profit the firm realizes if it produces its benchmark (profit-maximizing) product mix. This is the difference between the last two values.
NumProdInBenchmarkMix	The number of products in the benchmark (profit-maximizing) product mix. This number will be between 0 and numCO.

A-2.4.2.2.2 Firm_PRODUCT

This file contains information about each firm's products (resource concentration (g), resource sharing (d), product margins ($\overrightarrow{MAR}$), maximum production quantities ($\overrightarrow{MXQ}$), selling prices ($\overrightarrow{SP}$), profit-maximizing decision ($\overrightarrow{DECT0}$)). There is one row per firm.

Table 6: Data stored in the file Firm_PRODUCT.csv.

Field	Description
FirmID	A unique identifier for this firm.
G	The randomly selected value of $DISP2$ (the top $DISP1$ resources will account for $DISP2$ percent of the total resources in the initial $\overrightarrow{RCC}$ vector created for each firm).
D	The randomly selected value of the density of the resource consumption pattern matrix.
MAR_j	The true profitability (margin) of cost object j. Margin is defined as a product's selling price divided by its true cost. Margin is 1.0 for a product that breaks even.
MXQ_j	The capacity (maximum possible production quantity) of cost object j.
SP_j	The selling price of cost object j.
DECT0_j	1 if cost object j is included in the profit-maximizing (benchmark) product mix, 0 if not included.
Rank_by_val_j	Products are ranked by value (total profit) they contribute in the benchmark mix. This field is the rank of product j, by value.
Rank_by_mar_j	Products are ranked by their margins. This field is the rank of product j, by margin.

A-2.4.2.2.3 Firm_RESCON

This file contains the $\overrightarrow{RCC}$ (resource consumption) and $\overrightarrow{RCU}$ (unit resource price) vectors for each firm. There is one row per firm.

Table 7: Data stored in the file Firm_RESCON.

Field	Description
FirmID	A unique identifier for this firm.
G	The randomly selected value of $DISP2$ (the top $DISP1$ resources will account for $DISP2$ percent of the total resources in the initial $\overrightarrow{RCC}$ vector created for each firm).
D	The randomly selected value of the density of the resource consumption pattern matrix.
RCC_i	When creating a firm object, the simulation generates a vector of the total amount of each resource consumed when producing the benchmark (profit-maximizing) product mix. RCC_i is the amount of resource i that is consumed.
RCU_i	The unit price of resource i. This is randomly generated when each firm object is created.

A-2.4.2.2.3 Logging Information About a Cost System (LogCostSys)

This file contains summary information about cost systems (assignment of resources to cost pools (B), driver choices (D)). The file contains one row per firm / cost system pair. Users should call this method after each call to the *CostSys* constructor.

Table 8: Data stored in the file CostSys_SUM.

Field	Description
FirmID	A unique identifier for this firm.
CostSysID	An identifier for this cost system. Note that cost system ID's reset with every firm. Thus, the unique identifier for a cost system is the tuple (FirmID, CostSysID).
PACP	The method of assignment of resources to cost pools. See Table 3 for details.
ACP	The number of activity cost pools in this cost system.
PDR	The driver type. See Table 3 for details.
B	A list of lists. There is one sublist per cost pool. Each sublist contains the indices of the resources that are assigned to that cost pool.
D	A list of lists. There is one sublist per cost pool. Each sublist contains available drivers for each cost pool.

A-2.4.2.2.4 Logging Information About Error in Reported Costs (LogCostSysError)

Each cost system produces a single product mix. Upon producing that mix, observing total resource consumption, assigning resources to cost pools (as per the field B, above), choosing drivers (as per the field D, above), the cost system computes reported product costs. This file compares the

reported costs to the true costs. This file shows the results of the *CalcReportedCosts()* method of the *CostSys* class. Thus, *LogCostSysError()* should be called after each invocation of the *CalcReportedCosts()* method.

Table 9: Data stored in the file CostSys_ERROR.

Field	Description
FirmID	A unique identifier for this firm.
CostSysID	An identifier for this cost system. Note that cost system ID's reset with every firm. Thus, the unique identifier for a cost system is the tuple (FirmID, CostSysID).
PACP	The method of assignment of resources to cost pools. See Table 3 for details.
ACP	The number of activity cost pools in this cost system.
PDR	The driver type. See Table 3 for details.
startDecision_j	1 if product j was included in the starting product mix.
PC_B_j	The true (full-information, benchmark) cost of producing product j.
PC_R_j	The reported cost of producing product j.
MPE	For a given product mix, we compute the reported costs of all products. We then compute the mean percent error by comparing reported costs to true costs. We then average this over all product mixes for a given firm/cost system pair.

A-2.4.2.5 Logging Information about Iterating a Decision (LogCostSysLoop)

This file contains the results of the *EquilibriumCheck()* method of the *CostSys* class. That method takes as its argument a starting product mix. Each cost system produces that starting mix. Upon producing that mix, observing total resource consumption, assigning resources to cost pools (as per the field B, above), choosing drivers (as per the field D, above), the cost system computes reported product costs. Upon observing the reported costs, the firm updates its product mix decision. It adds to the product mix all products that appear profitable, and drops all products that appear unprofitable. If the updated decision is the same as the original decision, an equilibrium (fixed point) has been found. Otherwise, the firm continues to iterate in this manner. This file shows the results of this iteration process, specifically the starting mix, the terminal product mix (if one exists), and a description of the outcome (e.g. fixed point, cycle, etc.). *LogCostSysLoop()* should be called after each invocation of the *EquilibriumCheck()* method.

Table 10: Data stored in the file CostSys_LOOP.

Field	Description
FirmID	A unique identifier for this firm.
CostSysID	An identifier for this cost system. Note that cost system ID's reset with every firm. Thus, the unique identifier for a cost system is the tuple (FirmID, CostSysID).
PACP	The method of assignment of resources to cost pools. See Tale 3 for details.
ACP	The number of activity cost pools in this cost system.
PDR	The driver type. See Table 3 for details.
startDecision_j	1 if product j was included in the starting product mix.
endingDecision_j	1 if product j was included in the final product mix.
Outcome	Equilibrium, Cycle, or Zero-Mix. An equilibrium is a fixed point decision. A cycle is a set of decisions that the firm will iterate over infinitely. The zero mix is a death spiral – after iterating, the firm decides to produce nothing.

A-2.4.2.6 Writing Output Files to Disk (WriteOutput)

This method simply writes the contents of each StringBuilder (temporary, in-memory storage of output data) to disk. This method should be called only once, at the end of the simulation.

A-2.5 Main Method in Program Class

The *Main()* method controls program flow, and implements the experimental design.[18] Users should adapt this method to their experiments. The following outline details the steps in the current *Main()* method. Mandatory items are noted with an asterisk (*).

1. (*) Read input file and create *InputParameters* object.
2. (*) Make a copy of the input file.
3. (*) Create output files using *Output.CreateOutputFiles()* method.
4. For j = 1 to NUM_FIRMS
    a. (*) Create firm j.
    b. (*) Log firm j.
    c. For each value of ACP (A) in the input file:
        i. For each value of resource aggregation method (P) in the input file:
            1. For each value of the driver selection method (R) in the input file:
                a. (*) Create a cost system using A, P, and R.
                b. (*) Log cost system.
                c. Generate starting decision DECF0.

[18] Every C# program must contain a single Main method specifying where program execution is to begin (*Main() and Command-Line Arguments (C# Programming Guide)* 2017). We chose to implement the experimental design within the Main method, but future users could easily move the logic to other methods.

d. Get reported costs from this cost system by calling *CalcReportedCosts()*.
e. Compute error metric between reported costs and true costs.
f. Log error using *Output.LogCostSysError()*.
g. Iterate from DECF0 using *EquilibriumCheck()*.
h. Log error using *Output.LogCostSysLoop()*.

5. (*) Write output files using *Output.WriteOutput()* method.

The program makes a copy of the input file (step 2), adding a timestamp in the filename, to keep a record of which parameters were used during which runs.

A-2.6 Helper Classes

We supply three helper classes that contain useful utilities. *GenRandNumbers* implements a simulation-grade random number generator and contains methods for generating uniform and normal random numbers. *ExtendVectorClass* provides some helpful methods that extend the vector classes of the Meta Numerics library, such as element-wise multiplication of two vectors (Hadamard product). *ListUtil* provides some methods for working with lists, such as shuffling and printing lists with comma delimiters for the output files.

A-2.6.1 GenRandNumbers Class

This class implements the Mersenne Twister random number generator. The class publicly exposes the seed, as well as the following public methods:

- SetSeed(): Sets the seed value of the random number generator. This method has no effect if called more than once.
- GetSeed(): Returns the current seed.
- Next(int maxValue): Returns an integer in the range $[0, maxValue - 1]$
- GenStdNormalVec(int len): Returns a vector of length len with each element drawn from a standard normal distribution.
- GenUniformDbl(double lb, double ub): Generates a random double-precision floating point number in $[lb, ub]$.
- GenUniformDbl(): Generates a random double-precision floating point number in $[0.0, 1.0]$.
- GenUniformInt(int lb, in ub): Generates a random integer in $[lb, ub]$.

A-2.6.2 ExtendVectorClass Class

We rely on the open-source numerical library Meta Numerics (http://www.meta-numerics.net/) for linear algebra objects such as vectors and matrices, as well as for statistics. We extended Meta Numerics's vector class with the following methods:

- ewMultiply. This extension method for Meta Numerics's RowVector and ColumnVector classes performs element-wise multiplication (Hadamard product).
- Map. This extension method for Meta Numerics's RowVector and ColumnVector classes performs an element-wise transformation using a user-supplied function. For example, if one wishes to divide every element in the vector by 10, one need merely map a function that divides a scalar by 10. If V is a vector, one could type `V.Map(x => x / 10.0)`. That line of code says to map the lambda expression `x => x / 10.0` to each element of V and return the result.
- ToCSVString. This extension method for Meta Numerics's RowVector and ColumnVector classes returns a string representation of a vector, e.g. "[1, 7, -9]".
- TrueForAll. This extension method for Meta Numerics's RowVector and ColumnVector classes checks whether a condition, supplied as an argument, is true for every element of the vector.
- CopyRowInto. This extension method for Meta Numerics's Matrix and Vector classes copies the vector into a given row of the matrix.

A-2.6.3 ListUtil Class

This class provides static extension methods for working with lists.

- Normalize: Converts the numbers in a list to proportions of the sum of the list.
- MultiplyBy: Multiplies each element of the list by an argument.
- Shuffle: Shuffles the list using the Fisher-Yates shuffle.
- PrintList: Returns a string representation of the list.
- ToCSVString: Converts the list to a comma-separated string.

A-3 Modifying and Running the Code

This section contains instructions for modifying the existing code for a different numerical experiment. It also contains instructions for running the code.

A-3.1 Recommended prerequisites

Our code is currently built on version 4.6.1 of Microsoft's .NET framework. We recommend you use that version or later since we rely on many of the recently added features of C#. We strongly recommend that you use Microsoft Visual Studio for editing and compiling this code. The Community Edition of Visual Studio is free on Windows and on Mac, and provides ample functionality for code

editing and debugging. Finally, we recommend that you use Visual Studio Online (VSO), which is free, for source control.

A-3.2 Library Dependencies

Our code relies on Meta Numerics, an open source numerical library, as well as the System.ValueTuple library provided by Microsoft. We use Meta Numerics for linear algebra objects and operations, and for statistics. We use System.ValueTuple for returning multiple values, in the form of a tuple, from functions. The capability to return multiple values from a function is a recent addition to C#.

We use the NuGet package manager for both libraries. Therefore, if you download the Visual Studio Solution that accompanies this paper and compile it in Visual Studio, the packages will automatically download and integrate into your code.

A-3.3 Updating the Experimental Design in Program.cs

The most important step in adapting this code to your work is to update the experimental design. We currently use a nested design. We generate a sample of NUM_FIRMS firms, and in each simulated firm we nest multiple cost systems. We iterate over each firm/cost system pair using nested FOR loops. You may wish to modify this design, as well as the dependent and independent variables. Make the necessary changes to the *Firm* and *CostSys* classes, then modify the *Main()* method inside Program.cs. Please adhere to the guidelines in section A-2.5 and retain the mandatory steps of reading the input file, creating the output files, and writing the output files.

A-3.4 Modifying the Input File

To add or remove parameters from the input file, make the following modifications to the file InputParameters.cs:

- Inside the region *Fields and properties*.
    - We recommend that fields should be private, with a public property.
    - Use the get accessor on the public property to retrieve the value.
    - Use the set accessor, with protected accessibility, on the public property to set the field's value in the constructor and to enforce valid values for the field.
- Inside the InputParameters constructor.

    - In the *switch* statement, add a case for each new parameter.
    - Update the dictionary *dictMembers*. The program ensures that every field of InputParameters is assigned a value. If a parameter is missing from the input file, the program will detect it and abort.
- Optionally, specify other constraints on field values in the private method EnforceConstraints() of InputParameters.
    - Since fields can be read in any order, constraints that involve multiple parameters can only be verified once all parameters have been read. That is done in EnforceConstraints().
- If you wish to allow multiple values of a parameter so that your simulation can loop over those values:
    - Make sure the data type for your parameter implements IList (e.g. array, list, etc.).
    - Update the line of code in the InputParameters constructor that enforces only a single value per parameter. Currently, that is line 559, which reads:
      ```
      if ((paramName != "ACP") && (paramName != "PACP") &&
      (paramName != "PDR")) {
      ```

A-3.5 Modifying the Output Files

Decide whether to retain the existing output files and whether to add additional files. To add a new output file, do the following. The procedure to remove an output file is similar.

1. Open the file Output.cs in Visual Studio.
2. Create a string variable containing the filename in the region *Output file names*.
3. Create a StringBuilder variable for in-memory storage of your output in the region *StringBuilders for in-memory storage of output*.
4. Update the method *Output.CreateOutputFiles()*. Create a header for your file and write it to your output file.
5. Create a method for logging your data. The existing logging methods of the Output class (section 0) provide templates. Basically, your method should take some data as arguments and write a comma-separated list of data items to your StringBuilder.
6. Update the method *Output.WriteOutput()* so that it writes your StringBuilder to your file.

A-3.6 Updating the Input File

Inside the Visual Studio Solution is a file called *input.txt*. When you compile the solution, Visual Studio will copy this file to the same directory as your executable. When you run the executable, it will look for this file in its directory. Modify the values in the file to suit your experiment. Ensure that the values in the file are consistent with the constraints in section A-2.3.1, or with the constraints you defined.

We provide 4 input files as samples. Two create products (cost objects) with high margins (HM) and the other two create products with low margins (LM). We also vary the variation in margins at two levels, high variance (HV) and low variance (LV). Thus, the output files span 4 product markets: HMHV, HMLV, LMHV, and LMLV. To use a sample input file, copy its text into the file input.txt.

A-3.7 Compiling the Code

Compile the code using the Build command inside Visual Studio. You may want to change from Debug to Release mode if speed is a concern.

When you compile the code, Visual Studio will create an executable file, currently named CostSystemSim.exe. It will place this file in the bin\Debug or bin\Release subdirectory of your Visual Studio project's folder. It will also copy several DLL (dynamic link library) files, such as the ones for Meta Numerics and System.ValueTuple. Finally, it will copy the input file and the license file.

A-3.8 Running the Code

Run the code by double-clicking on the executable. Alternatively, you can compile and run in a single step within Visual Studio. In the Debug menu, click on "Start Debugging" or "Start without Debugging". A console window will appear and show you the progress of your code. The program will write its output files to the same folder as the executable.

A-3.8.1 Computational Requirements

We note that computational requirements are minimal in the code's current form. Running the simulation with 200 firms, 6 cost systems per firm, requires about 2 seconds on a Core i7-6700K. Total memory usage was 30 MB. In Anand, Balakrishnan, and Labro (2017), we iterate over every possible product mix and determine its terminal outcome. When we did so with 5 products ($2^5 = 32$ possible product mixes), running time was less than 5 seconds. However, when we did so with 20 products ($2^{20} = 1{,}048{,}576$) possible product mixes times 200 firms times 6 cost systems per firm, we used 8 hours of CPU time. However, we note that that was an extreme case. Anand, Balakrishnan, and Labro

(2017) reports on an efficient heuristic procedure to find a solution in this case. We expect that computational requirements will be extremely small for most cost system experiments.

A-4 References

Anand, V., R. Balakrishnan, and E. Labro. 2017. Obtaining Informationally Consistent Decisions When Computing Costs with Limited Information. *Production and Operations Management* 26 (2): 211-230.

Balakrishnan, R., S. Hansen, and E. Labro. 2011. Evaluating Heuristics Used When Designing Product Costing Systems. *Management Science* 57 (3): 520-541.

Main() and Command-Line Arguments (C# Programming Guide). 2017. Microsoft Corporation, May 27 2017 [cited June 13 2017]. Available from https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/main-and-command-args/index.

TIOBE Index. 2017. TIOBE Software BV 2017 [cited June 1 2017]. Available from https://www.tiobe.com/tiobe-index/.