

# Endogenous matching functions: An agent-based computational approach\*

Michael Neugart<sup>†</sup>

September 2003

DRAFT – PLEASE DO NOT CITE WITHOUT  
PERMISSION OF THE AUTHOR

## Abstract

The matching function has become a popular tool in labor economics. It relates job creation – a flow variable – to two stock variables: vacancies and job searchers. In most studies the matching function is exogenous and assumed to fulfill certain properties. This study looks at the properties of an endogenous matching function. For that purpose we program an agent-based-computational labor market model with endogenous job creation and endogenous job search behavior. Our simulations suggest that job creation is increasing in the number of job searchers and vacancies. The endogenous matching technology is subject to decreasing returns to scale. The Beveridge

---

\*The paper is prepared for the Wild@Ace Workshop on Industry and Labor Dynamics to be held on October 3rd and 4th in Turin. I would like to thank Richard Freeman for encouraging discussions that substantially motivated me to do research with agent-based computational models. There is no agent-based modelling without knowledge of an appropriate programming language. While I was on leave from the Social Science Center Berlin (WZB) at Harvard University I had the chance to participate in Lars-Eric Cederman's course 'Introduction to computational modelling for social scientists' to learn RePast. My thanks go also to him. None of the two takes any responsibility for the content of the paper.

<sup>†</sup>Chemnitz University of Technology, Department of Economics, Reichenhainer Strasse 39, 09107 Chemnitz, Germany, e-mail: michael.neugart@wirtschaft.tu-chemnitz.de

curve reveals substitutability of job searchers and vacancies for a small range of inputs but is flat for relatively high numbers of job searchers and vertical for relatively high numbers of vacancies. It occurs that the matching technology changes with labor market policies. This raises concerns about the validity of labor market policy evaluations conducted with flow models of the labor market employing exogenous matching functions.

*Keywords:* Endogenous matching functions, agent-based-computational labor market

## 1 Introduction

The matching function has become a widely used tool in modern labor economics. It plays a central role in flow models of the labor market explaining equilibrium unemployment (see Pissarides (1990) or Mortensen (1982)). The matching function has also undergone extensive testing (see Petrongolo and Pissarides (2001)). The key idea of the matching function is that it summarizes in a neat way the behavior of firms trying to fill vacancies and workers looking for jobs, and relates it to job creation. As a modelling device certain attributes are generally assumed for the matching function: constant returns to scale, and job creation being concave and increasing in both arguments, job searchers and vacancies.

The aim of the paper is to study properties of an endogenous matching function and compare them to assumptions usually imposed on exogenous matching functions in the theoretical and empirical labor economics literature. Rather than to assume a holistic function we are interested in whether properties like concavity, job creation being an increasing function of vacancies and job searchers, or constant returns to scale can be generated by a model that cares about the micro-behavior of firms and workers.

The second topic is the endogeneity of the matching function with respect to labor market policies. Policy experiments conducted on the basis of a labor market model with an exogenous matching function may be misleading if the policies change the properties of the matching function. Such a concern is not farfetched as the properties usually imposed on exogenous matching

functions are justified on agents' micro-behavior. As policies are targeted to change agents' choices, for example by firm subsidies or mobility vouchers to job searchers, it may very well also affect the properties of the matching technology.

We employ an agent-based computational setting. The main advantage of such a setup is that it allows capturing heterogeneity on the side of the firms and workers to a larger extent than analytical models can deal with, without becoming intractable. We depart from the existing literature on endogenous matching functions in a second respect. While agents are usually modelled as optimizers having rational expectation, agents in this study stick to the strategies they are born with. This means that workers always send the same number of applications and firms always post the same number of vacancies over their life-cycle. However, workers and firms have to exit the market if their behavior does not yield positive payoffs. In that case they are replaced by new agents that copy strategies of the surviving workers and firms, respectively. Thus, over time the market will be populated with agents that behave such that they 'survive'.

Agent-based computational research is gaining interest among economics' scholars (see for example Tesfatsion (2001) or Tesfatsion (2002)) and other social sciences disciplines.<sup>1</sup> The appeal of agent-based computational economics (ACE) is its flexibility of modelling. Heterogeneity of agents and interaction of agents can explicitly be taken care of. After the characteristics of the agents and rules that govern their interaction are programmed, the emerging macro-behavior can be studied. One of the roles ACE could play in modern economics has been very well described by Roth and Peranson (1999) who undertake computational experiments to test the clearinghouse with which physicians are matched on the U.S. labor market. Their analogy is that of a civil engineer constructing a suspension bridge. Based on Newtonian physics the engineer will write down a simple model of the new bridge. But as nature is much more complicated to be captured by a small model – think of the different conditions of soil, changing weather and the like – every bridge project undergoes extensive computer simulations before construction is started. This type of analysis maybe less elegant than a simple Newtonian

---

<sup>1</sup>Cederman (2001) surveys, for example, agent-based modelling in the political sciences.

model but it is what makes the bridge stand. And quite importantly, there may be important feedback processes from computer simulations by which modelers learn about more complicated phenomena for building analytical models. The complementary feature of ACE is also evident where agent-based modelers work together with experimental economists. Agent-based models can be used to test learning behavior, as derived from experiments that usually have to stick to shorter horizons, in the longer run (Pingle and Tesfatsion (2001)). There are also attempts to model the behavior of agents, evident from experiments, in an agent-based setting, and then test the predictions from the simulations with experiments again (e.g. Duffy (2001)).

In labor economics agent-based computational modelling has been attributed great potential (see Freeman (1998)), especially in studying market outcomes in a world where labor market institutions set by policy makers govern the behavior of agents, and outcomes may lead policy makers to reconsider institutional choices and thereby alter labor demand and supply.

Our aim in this paper is much more humble. Institutions are exogenous. We are only interested in the labor market outcomes when firms and workers make their individual choices in a stable institutional setting. Nevertheless, we think that endogenous matching functions are an interesting research issue that so far was only dealt with on the basis of analytical model. The following section surveys this literature. As those approaches are extensions of the urn ball model which is generally taken to illustrate how frictions arise on labor markets, we briefly sketch it first. Section 3 describes our model. Section 4 reports on the simulation results. The last section concludes and gives some implications of our findings for the suitability of exogenous matching functions in the labor economics literature.

## 2 A review of the literature

The urn ball model<sup>2</sup> serves as an intuitive approach for a micro-foundation of the matching function based on coordination failures. Those coordination

---

<sup>2</sup>Those models were analyzed by Butters (1977) and Hall (1979) among the first. See Petrongolo and Pissarides (2001) for a brief summary of the problem of coordination failure.

failures arise because workers simultaneously apply for jobs not knowing where the other workers send their applications. As a result some vacancies may get more than one application while others do not receive a single application and have to stay unproductive. In the urn ball model vacancies are considered as urns and workers as ball that have to be placed into the urns for a job to be created. Assume that there were  $U$  workers looking for a job and  $V$  vacancies. All workers simultaneously send one application. With probability  $1/V$  a single vacancy receives the application of a worker. And with  $1 - 1/V$  this vacancy receives no application. As there are  $U$  workers looking for a job the probability that a vacancy gets no application from any worker is  $(1 - 1/V)^U$  and the probability that it gets at least one application writes  $1 - (1 - 1/V)^U$ . Hence, the matching function in this example becomes  $M = V(1 - (1 - 1/V)^U)$ . For large  $V$ ,  $(1 - 1/V)^U$  can be approximated with  $e^{-U/V}$  so that  $M \approx V(1 - e^{-U/V})$ . It can easily be seen that this matching function has constant returns to scale. Also,  $M$  is increasing in its arguments,<sup>3</sup> and concavity is given.<sup>4</sup> While all the properties usually imposed on matching functions are fulfilled,  $M = V(1 - e^{-U/V})$  does not fit the data very well as an increase in unemployment relative to vacancies empirically comes with higher duration of unemployment than the function implies.

More recent approaches studying endogenous matching function are rooted in the urn ball model but try to develop a richer labor market context.

One such extension is analyzed by Albrecht et al. (2003) who allow for multiple applications of job seekers. The idea is that with multiple applications a new coordination failure as compared to the standard urn-ball model will arise. In the latter, the coordination failure leads to a situation where some vacancies do not get applications so that they stay unproductive. With multiple applications very likely every vacancy will get at least one application but still a coordination failure may be given. It arises from the competition among firms for single candidates. Once a firm has chosen an applicant to make him an offer other firms may have done so, too. A certain firm may

---

<sup>3</sup> $\partial M/\partial U = e^{-U/V} > 0$  and for the case that there are sufficiently many job searchers relative to vacancies - which is usually true in real labor markets -  $\partial M/\partial V = 1 - (1 + U/V)e^{-U/V} > 0$ .

<sup>4</sup> $\partial^2 M/\partial^2 U = (-1/V)e^{-U/V} < 0$  and  $\partial^2 M/\partial^2 V = (-U^2/V^3)e^{-U/V} < 0$ .

not get its candidate because it was hired away by a competing firm. As a result more applications per worker may not increase the matching efficiency of labor markets. The authors also show that for high but finite numbers of market participants the matching functions exhibits constant returns to scale.

The study of an endogenous matching function by Burdett et al. (2001) differs from the urn ball model as firms post wages and are heterogenous in their size measured by the number of vacancies they post. In one version of the model the number of vacancies each firm offers differs exogenously, while in the other it is made endogenous. The process of exchange is such that each firm simultaneously posts a wage. Workers know all vacancies and posted wages. They simultaneously choose a vacancy for application. If there is more than one application, firms randomly choose one applicant who gets the job at the posted wage. The authors find a matching function with decreasing returns to scale that converges to constant returns to scale as the number of market participants increases. They also find an outward shift of the Beveridge curve as more firms offer fewer jobs which leads them to claim that empirically observable shifts in the Beveridge could be explained by shifts in the job distribution over firms.

Smith and Zenou (2003) extent heterogeneity to the labor supply side. In addition to jobs having different skill requirements, workers have different skills. Thereby, the authors explicitly allow for skill mismatch. With respect to the properties of the matching function they find with a finite and an infinite number of agents that the matching function is concave and that matches are increasing in both arguments. But only for infinite inputs the matching functions has constant returns to scale.

There is homogenous labor supply and demand in the model by Cao and Shi (2000) who, however, emphasize the role of wages in their matching model. An endogenous matching functions arises from a labor market in which firms post wages to attract workers. A relatively high wage is not only considered as a cost factor. Firms also take into account that it will lead to more applications and therefore make it more likely that the vacancy gets filled. Workers value relatively high wages. But on the other hand workers also anticipate that jobs offering relatively high wages may be crowded with

applications making an offer less likely. The trade-offs for the workers and firms drive their wage posting and application strategies. Cao and Shi (2000) are interested in the effect of the coordination failure on welfare. They find that welfare loss is highest with equal numbers of traders on both sides of the market, no matter how large the market is.

The key difference in Julien et al. (2000) from other papers on endogenous matching functions is their auction model for wage determination. Whereas in wage posting models it is assumed that firms offer a binding wage and then are approached by applicants among whom they choose, the process is reversed with auctions. Here, workers announce a reservation wage for which they are willing to work and then are approached by firms. If more than one firm contacts a worker, the worker can bid up his wage. This gives the authors even under the assumption of homogenous firms and workers wage dispersion. As in the other models frictions enter because of coordination problems, capacity constraints, and in a version where vacancy creation is endogenous, externalities associated with new vacancies entering the market. The properties of the endogenous matching function are constant returns to scale and matches that are increasing in vacancies and job searchers.

When deriving matching functions from the micro-behavior of agents, it is usually assumed that a single worker does not know what other workers do, or that firms make offers not knowing what their competitors' choices are. This assumption enters matching models in the form of random behavior of agents. The contribution by Lagos (2000) is to show that even when the assumption of 'nobody knows what the others do' is dropped, frictions – firms and workers do not come together even though both sides are willing to trade – nevertheless occur. To get the intuition of this result it may be helpful to briefly sketch the model. Lagos (2000) considers a grid where cabs can locate and passengers stochastically want to move from one location to another but can only do so by taking a cab. It is shown that cabs may optimally locate on the grid such that some passengers do not get served. The properties of the endogenous matching function are constant returns to scale and a right angle shape of the iso-matching curve (or Beveridge curve). Non-substitutability between cabs and passengers is not an empirical characteristic of matching functions but may vanish with the introduction of not-administered prices

in the market. The upshot of the argument is that when frictions are the result of optimal choices of agents, policy analysis on the basis of exogenous matching functions may be misleading as agents' new choices may also affect the matching technology of the market.

As Lagos (2000) we are also interested in the endogeneity of the matching function with respect to labor market policies. An issue that so far has rarely attracted attention. Furthermore, our work relates to the existing literature as we also analyze the properties of endogenous matching function. Contrary to the existing studies on endogenous matching functions, we model an agent-based computational labor market. This allows for more structure on both sides of the market. For example, vacancy creation and job search is endogenous in our model. We also depart from other studies by not assuming agents with rational expectations. Our agents simply stick to their strategies with which they were born. However, those that do not come up with positive payoffs have to leave the market. They are replaced by newly born agents that adopt a strategy of the surviving firms and workers, respectively.

### 3 The model

There are  $m > 0$  workers and  $n > 0$  firms. Firms can create vacancies at a cost  $costVac$ . Those costs can be interpreted as costs for advertising a vacancy but also as some fixed capital cost like e.g. buying a computer that equips the job. The initial number of vacancies  $numVac_i$  that each firm  $i$  posts is randomly drawn from the interval  $[0, numFirms]$  where  $numFirms$  is the number of firms in the market.<sup>5</sup>

Workers are born with different reservation wages  $r_j$  reflecting different tastes for leisure. Each worker is initially assigned an application strategy  $numAppli_j$  out of the interval  $[0, numWorkers]$  with  $numWorkers$  being the number of workers in the labor market.

We assume that the market is sufficiently large so that there is no strategic interaction among workers about where to send applications. A worker  $i$  sends applications, given the behavior of all other workers. Technically work-

---

<sup>5</sup>Figure 1 gives the pseudocode for our model.

ers randomly send applications. Applications are sent to firms that have at least one vacancy. No worker applies twice at the same firm. The assumption of random applications captures the idea of coordination failures on the labor supply side of the market that are the reason for frictional unemployment in the model.

Firms may have received no application or at least one. For the case that no application arrived, the vacancy cannot be filled with a worker and the job does not become productive. If there is more than one applicant for a vacancy at a firm, the firm makes a binding offer to the worker with the lowest reservation wage. Reservation wages are known to the firms as they ask applicants when posting a vacancy what wage they would like to get. Such requests can often be found in newspaper or online ads. The order at which firms are allowed to make a job offer is random, approximating simultaneous and uncoordinated choices of firms. The worker who gets a job offer accepts it and is paid his reservation wage. Hence, all the ‘bargaining power’ is with the firms. And as a consequence, there is wage dispersion within firms. Workers in the same firm may earn different wages.

The payoff function of the firm  $i$  writes:

$$payOff_i = \begin{cases} y * numJobs_i - wageSum_i - costVac * numVac_i & \text{if at least one vacancy is filled,} \\ -costVac * numVac & \text{otherwise} \end{cases}$$

A filled job creates output  $y$  from which the firm has to deduct the wage costs and the costs for creating vacancies. If no vacancy was filled the payoff is equal to the vacancy posting costs. A worker who finds a jobs receives his individual reservation wage. In both cases, employed and unemployed he has to carry the application costs

$$payOff_j = \begin{cases} wage_j - costAppli * numAppli_j & \text{if employed,} \\ -costAppli * numAppli_j & \text{otherwise} \end{cases}$$

The market selection mechanism is such that workers and firms that do not have positive payoffs are eliminated from the market. Those who do not survive are replaced by new agents. The new agents are assigned strategies of

the surviving firms and workers, respectively. A new born worker is randomly assigned an application strategy of a surviving worker. The same mechanism applies to a new born firm, respectively. Thus, the new born agents profit from the ‘knowledge of the market’ that carries on which strategies are helpful for not being kicked out.

After every period all jobs are dissolved. A new application and hiring process starts. Note, however, that all (surviving) agents stick to their strategies over the life-cycle.

```

Create  $n$  firms each posting  $numVac_i$  vacancies
Create  $m$  workers with reservation wages  $r_j$  sending applications  $numAppli_j$ 
for  $k$  periods
  Applying
  for each worker  $j$ 
    selects all firms  $i$  with  $numVac_i > 0$ 
    applies randomly at firms  $i$ 
  end for each worker
  Hiring
  for each vacancy of  $m$  firms
    randomly draw vacancy to be filled
    if workers applied and at least one is still available
      firm selects worker  $j$  with lowest reservation wage  $r_j$ 
    else
      vacancy is not filled
    end each vacancy of  $m$  firms
  Market selection
  for each firm  $i$ 
    if  $payoff$  of firm  $i$  smaller or equal  $0$ 
      firm  $i$  exits
      new firm is born with strategy  $numVac$  randomly adopted from
      surviving firm
    end each firm
  for each worker  $j$ 
    if  $payoff$  of worker  $j$  smaller or equal  $0$ 
      worker  $j$  exits
      new worker is born with strategy  $numAppli$  randomly adopted
      from surviving worker
    end each worker
  All jobs are dissolved
end  $k$  periods

```

Figure 1: Pseudocode

## 4 Simulations

For the simulations we normalize per period labor productivity  $y$  to one. Thus, workers’ reservation wages are assigned from the interval  $[0, 1]$ . Our baseline simulation refers to the case where the applications costs are  $costAppli =$

Table 1: Parameters

Parameter	Value
Labor productivity	$y = 1$
Application cost for workers	$appliCost = 0.1$
Costs for posting a vacancy	$vacCost = 0.4$
Number of Workers	$m = 10, 11, \dots, 50$
Number of Firms	$n = 10, 11, \dots, 50$

0.1 and the costs for opening up a vacancy  $costVac = 0.4$ . We run simulations for different combinations of firms and workers. Starting with 10 workers and 10 firms the numbers are increased by one. That makes 1,681 cases. Each case is replicated 10 times so that we arrive at 16,810 runs. Finally, each run consists of 20 periods from which we report the market outcome of the last period in all runs.

Figure 2 plots one such run for a labor market with 30 workers and firms for 40 periods. The left hand scale refers to the average of vacancies posted by firms and the average of applications sent by workers. Both variables start off at relatively high values, driven by the random assignment of strategies, but drop quickly to one as the firms and workers with negative payoffs are selected out. As there are relatively many applications written and vacancies posted, employment is also high (right hand scale). But as workers send fewer applications and fewer vacancies come to the market, the employment rate drops to an average value slightly higher than 0.6. Finally, figure 2 plots the average wage (wage sum divided by employed worker) which is initially also higher due to the higher employment rate.

Figure 3 shows job creation in our labor market as the number of vacancies posted increases holding job searchers fixed at 30. It can be seen that as more vacancies are posted by firms more jobs are created. There is an upper bound to job creation. For relatively large numbers of vacancies labor supply restricts job creation. No more than 30 jobs are created as we fixed job searchers to a number of 30. However, the replications of the experiment show that the market not always hits the labor supply constraint even when

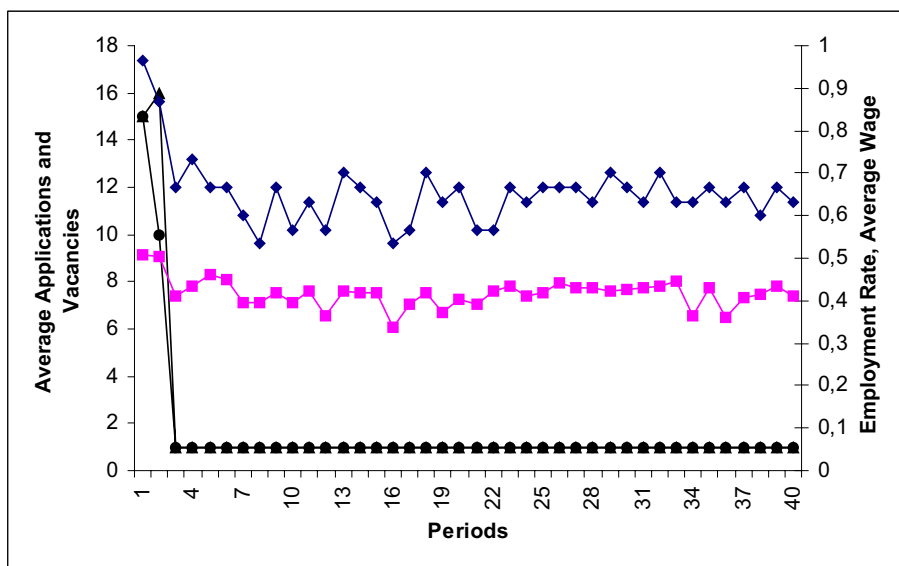


Figure 2: Average applications and vacancies (left hand scale), and employment rate and average wage rate (right hand scale),  $numWorkers = 30$ ,  $numFirms = 30$ ,  $costAppli = 0.1$ ,  $costVac = 0.4$

vacancies exceed labor supply by factor two and more. For the regime where labor supply exceeds labor demand, the latter is the binding constraint for job creation. Job creation cannot be higher than the number of vacancies posted. But also in this regime, market outcomes may not be equal to the labor demand constraint. In fact there is a cloud of points below the labor demand and supply constraints indicating a frictional labor market. Coordination failure by workers when sending applications and by firms when hiring workers generate unemployment.

Job creation is also an increasing function of job searchers holding labor demand fixed (see figure 4). Again labor demand and supply restrictions kick in. For relatively high numbers of job searchers the labor demand constraint is binding which was imposed at 30 vacancies. Job creation becomes a flat function of job searchers. For the case where labor supply falls short of labor demand, the supply constraint is binding indicated by the upward sloping part of the graph. There is also a cloud of points below the restrictions because of coordination failures. Even though there are more job searchers than

vacancies on the labor market in the flat part of the graph not every position is filled. And in the increasing part of the graph not every replication yields full employment even though there are more vacancies than job searchers.

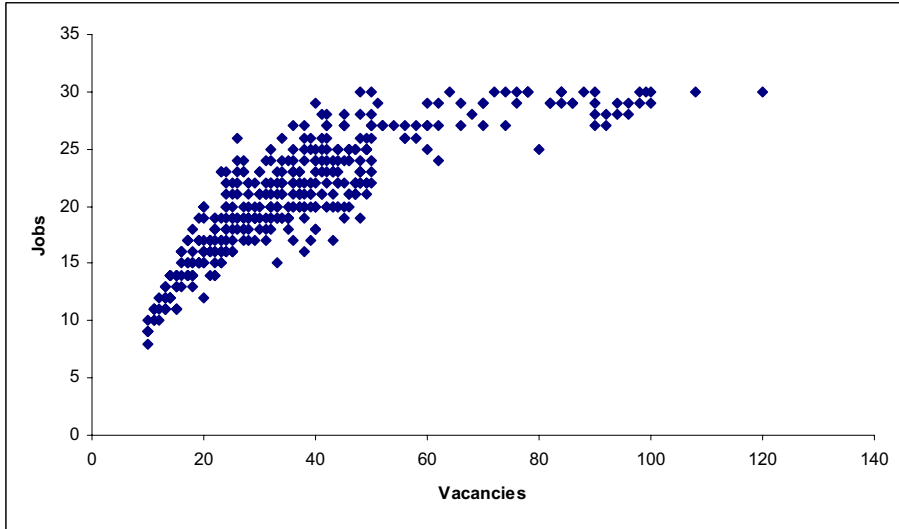


Figure 3: Jobs as a function of vacancies in the market – holding the number of workers fixed at 30

The Beveridge curve (iso-matching curve) consists of combinations of workers and vacancies in the labor market that yield the same number of jobs. Figures 5 and 6 show the simulation results for 15 and 20 jobs, respectively. We plotted the market outcomes for all 10 runs. Looking at the boundaries of the Beveridge curve in figure 5 shows that it is parallel to the vertical axis for relatively high numbers of vacancies, and parallel to the horizontal axis for relatively higher numbers of job searchers. The reason being is that if there are relatively many vacancies on the labor market but only a small number of job searchers, every worker will very likely find a job. Reducing the number of vacancies has not to be compensated by an increase in job searchers to keep matches constant. Considering the other boundary, every vacancy will get filled if it is confronted with relatively many job searchers. Thus, we find a straight line parallel to the horizontal axis at vacancies equal to job creation. Up and right to the boundaries of  $numWorkers = 15$  and  $numVac = 15$  there is a cloud of points. Hence, the market outcome is not

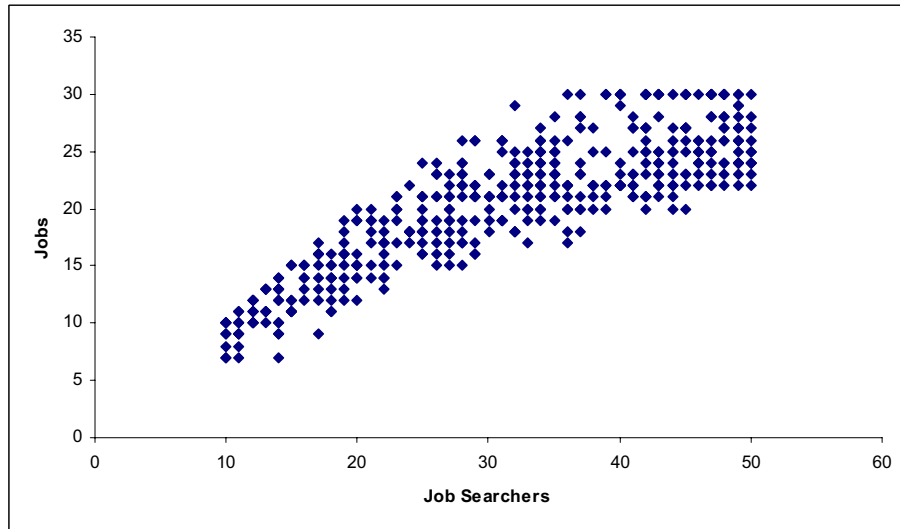


Figure 4: Jobs as a function of workers in the market – holding the number of vacancies fixed at 30

always characterized by its constraints on either the labor demand or labor supply side – another way to illustrate the results that already could be seen in figures 3 and 4. Not all vacancies and job searchers get matched. Also one can see substitutability of vacancies and job searchers for job creation. However, the distribution of market outcomes implies not a strong version of substitutability of inputs. In this example, almost anything can happen. Combinations of vacancies and job searchers that yield the same number of jobs are almost equally distributed in the interval where relatively low numbers of searchers and vacancies are combined. Figure 6 shows an outward shifted Beveridge curve driven by the fact that it shows combinations of job searchers and vacancies that create 25 jobs. Demand and supply restriction are now binding at a labor supply of 25 workers and demand of 25 vacancies. Apart from this the Beveridge curve shows qualitatively the same picture.

Returns to scale are playing a prominent role in the matching function literature. One reason for that is that a labor market with an increasing returns to scale matching technology may have multiple equilibria. In order to determine the returns to scale of our endogenous matching function we plotted jobs over pairs of equal numbers of vacancies and workers in the

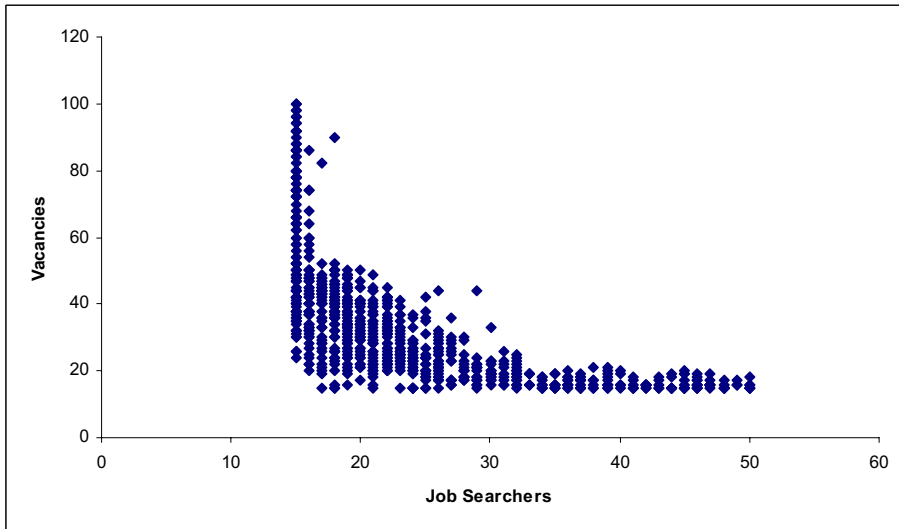


Figure 5: Beveridge Curve, 15 Jobs

market (see figure 7). The regression line with a slope of 0.70 reveals a decreasing returns to scale technology. This is in contrast to the widely accepted result of constant returns to scale or mildly increasing returns from the empirical literature on matching functions. Also note, the heteroscedastic nature of the plotted data.

Besides the properties of endogenous matching functions we are also interested to what degree those change if institutions governing the behavior of agents in the labor market are altered. For that purpose we conducted two policy experiments. One of which can be thought of as a subsidy to firms in order to increase labor demand. Firms are paid a lump sum transfer by a third, external party. The transfer makes it less costly to open up a vacancy. The other policy is a transfer to the workers lowering their search costs. For example, this could be a mobility voucher which gives workers an incentive to send applications also to firms that are not in commuting distance. To simulate the impact of the subsidy to the firm we lowered the cost of opening up a vacancy to 0.2 (as compared to 0.4 in the baseline model), leaving everything else constant. We find that the returns to scale parameter increases from 0.70 to 0.81. Clearly this lends support to the idea that one has to be aware that policies may also change the matching technology of labor mar-

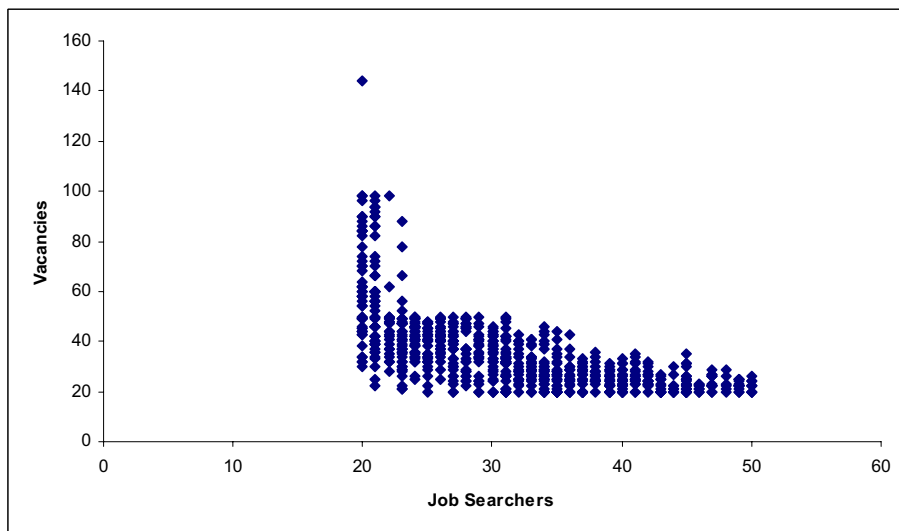


Figure 6: Beveridge Curve, 20 Jobs

kets. Therefore, assuming exogenous matching technologies may generate misleading results in policy experiments conducted within flow models of the labor market. With respect to the mobility vouchers, we only get a minor change in the returns to scale parameter. It drops from 0.70 to 0.68 if the voucher lowers the costs for applying from 0.1 to 0.05.

## 5 Conclusions

We programmed an agent-based computational labor market model with the following features: endogenous vacancy creation, endogenous job search, random applications, firms that pay workers their reservation wages, and a market selection process that eliminates agents that do not have positive payoffs. One purpose of the exercise was to study the properties of the endogenous matching technology and compare them to generally assumed characteristics of exogenous matching functions. We found that job creation is increasing in its arguments, vacancies and job searchers. Contrary to most of the empirical literature our endogenous matching technology has decreasing returns to scale. The simulated Beveridge curve is flat and vertical

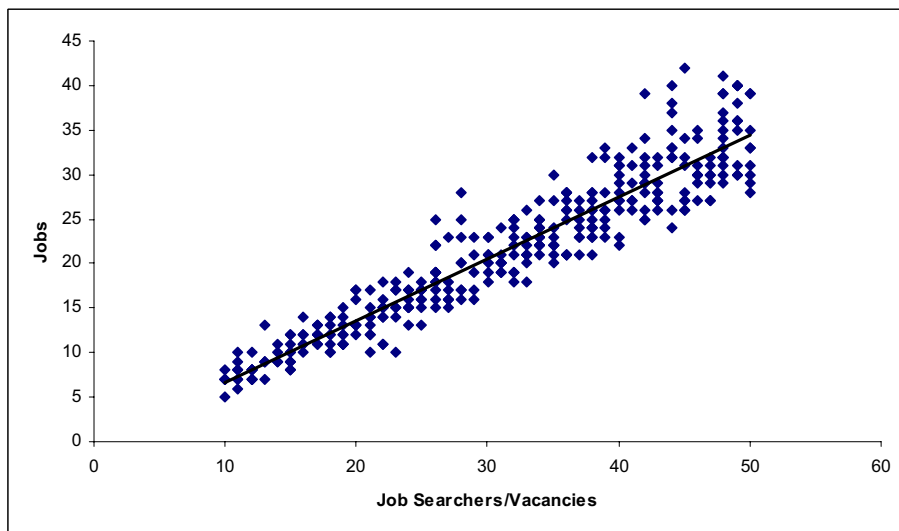


Figure 7: Returns to scale

at the boundaries, respectively. There is substitutability of inputs.

Secondly, we were interested in the endogeneity of the matching function with respect to labor market policies. It occurs that the matching technology is affected by policies which would make an exogenous matching function an inappropriate tool for policy evaluations. Simulating the effects of a subsidy to firms on job creation revealed that the matching technology changed – shown by an increase in the estimated returns to scale parameter. While no such effect is observable when transfers are paid to job searchers, it nevertheless raises an important point with respect to policy evaluations. The results from labor market policy evaluations based on models with exogenous matching functions could be biased if modelers do not take into account that the matching technology itself may also change with the policy. If one takes the interpretation which usually comes with the use of a holistic matching technology seriously, namely that it captures in a neat way the micro-behavior of agents, this is a serious claim.

Biased results may also be a pitfall in estimations of the matching function. Empirical work on the matching function usually starts with the assumption of a Cobb-Douglas technology which can easily be log-linearized for estimating the returns to scale coefficient. It would be interesting to see

whether estimates of the matching function with data generated by agent-based computational models yields the same parameters compared to an analysis that does not impose structure in advance.

## References

- Albrecht, J., Gautier, P. and Vroman, S.: 2003, Matching with multiple applications, *Economics Letters* **78**, 67–70.
- Burdett, K., Shi, S. and Wright, R.: 2001, Pricing and matching with frictions, *Journal of Political Economy* **109**(5), 1060–1085.
- Butters, G.: 1977, Equilibrium distribution of sales and advertising prices, *The Review of Economic Studies* **44**(3), 465–491.
- Cao, M. and Shi, S.: 2000, Coordinatino, matching, and wages, *Canadian Journal of Economics* **33**(4), 1009–1033.
- Cederman, L.-E.: 2001, Agent-based modelling in political sciences, *The Political Methodologist* **10**(1), 16–22.
- Duffy, J.: 2001, Learning to speculate: experiments with artificial and real agents, *Journal of Economic Dynamics and Control* **25**, 295–319.
- Freeman, R.: 1998, War of the models: which labour market institutions for the 21st century?, *Labour Economics* **5**, 1–24.
- Hall, R.: 1979, A theory of the natural unemployment rate and the duration of employment, *Journal of Monetary Economics* **5**, 153–169.
- Julien, B., Kennes, J. and King, I.: 2000, Bidding for labor, *Review of Economic Dynamics* **3**, 619–649.
- Lagos, R.: 2000, An alternative approach to search frictions, *Journal of Political Economy* **108**(5), 851–873.
- Mortensen, D.: 1982, The matching process as a noncooperative bargaining game, in J. McCall (ed.), *The economics of information and uncertainty*, University of Chicago Press.

- Petrongolo, B. and Pissarides, C.: 2001, Looking into the black box: a survey of the matching function, *Journal of Economic Literature* **XXXIX**, 390–431.
- Pingle, M. and Tesfatsion, L.: 2001, Non-employment benefits and the evolution of worker-employer cooperation: experiments with real and computational agents. Economic Report 55, Iowa State University.
- Pissarides, C.: 1990, *Equilibrium unemployment theory*, The MIT Press.
- Roth, A. and Peranson, E.: 1999, The redesign for the matching market for american physicians: some engineering aspects of economic design, *American Economic Review* **89**(4), 748–780.
- Smith, T. and Zenou, Y.: 2003, A discrete-time stochastic model of job matching, *Review of Economic Dynamics* **6**, 54–79.
- Tesfatsion, L.: 2001, Introduction to the special issue on agent-based computational economics, *Journal of Economic Dynamics and Control* **25**, 281–293.
- Tesfatsion, L.: 2002, Agent-based computational economics: growing economies from the bottom up, *Artificial Life* **8**(1), 55–82.

## 6 Appendix

The model is programmed in RePast and consists of six separate files. *Model.java* contains the main code. In *Workers.java* and *Firm.java* the characteristics of the workers and firms are modelled, respectively. *ModelBatch* runs the batch job for which *params.txt* gives the information on the parameter grid. *ModelGui* makes graphs of the simulations.

```
package reservation_wage_matching ;

/*this is the model with endogenous vacancies and endogenous applications
  Version: Turin
*/

import uchicago.src.sim.engine.*;
import java.util.*;
import uchicago.src.sim.util.SimUtilities ;

public class Model extends SimpleModel {

//model variables
  int numWorkers ;
  int numFirms ;
  double totalJobs ;
  double wageSum ;
  int appliSum ;
  int vacSum ;
  double reservationWageSum ;
  double productivity = 1;
  double costAppli = 0.1;
  double costVac = 0.4;

  int maxIter ; //max number of iteration cycles

//creating templates for the list of firms and workers
  ArrayList firmList = new ArrayList ();
  ArrayList workerList = new ArrayList ();

  public void setup () {
    super .setup (); //must be called first

    name = "Matching"; //giving the model a name

//setting the values for the model parameters

    numWorkers = 30;
    numFirms = 30;
    maxIter = 1;
  }

  public void buildModel () {

//cleans workerList and firmList for batch runs before new model is built

    workerList .clear ();
    firmList .clear ();

    totalJobs = 0;
    wageSum = 0;
    appliSum = 0;
    vacSum = 0;
    reservationWageSum = 0;

//creating the firms
    for (int i = 0; i < numFirms; i++){
      Firm aFirm = new Firm(i, productivity, getNextIntFromTo (0,numFirms),0,costVac,0);
      firmList .add(aFirm);
    }

//printing the firms
  /*
    System.out.println("Firms");

    for (int i = 0; i < firmList.size(); i++) {
```

```
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.firmID);
}
System.out.println();

for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.productivity);
}
System.out.println();

for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.numVac);
}
System.out.println();

for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.numJobs);
}
System.out.println();
*/

//creating the workers
for (int i = 0; i < numWorkers; i++){
    double reservationWage = getNextDoubleFromTo (0,productivity);
    Workers aWorker = new Workers (i,reservationWage ,0,0,getNextIntFromTo (0,numWorkers ),costVac);
    workerList .add(aWorker);
}

//printing the workers
/*
System.out.println("Workers");

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.workerID);
}
System.out.println();

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.reservationWage);
}
System.out.println();

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.status);
}
System.out.println();

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.wage);
}
System.out.println();

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
```

```

        System.out.print(" " + aWorker.numAppli);
    }
    System.out.println();
*/
}

//what follows are the activities of the time step

//jobs are dissolved
public void resetWorkers (){
    for (int i = 0; i < numWorkers; i++) {
        Workers aWorker = (Workers)workerList .get(i);
        aWorker.reset();
    }
}

public void resetFirms (){
    for (int i = 0; i < numFirms; i++){
        Firm aFirm = (Firm) firmList .get(i);
        aFirm.reset();
    }
}

//the statistics are reset
public void resetStatistics (){
    totalJobs = 0;
    wageSum = 0;
    vacSum = 0;
    appliSum = 0;
    reservationWageSum = 0;
}

//workers apply for jobs, they send applications randomly to firms

public void applying (){

    for (int i = 0; i < workerList .size(); i++) {
        Workers aWorker = (Workers) workerList .get(i);
        Collections .shuffle (firmList);
        for (int j = 0; j < aWorker.numAppli; j++){
            for (int m = 0; m < firmList .size ();m++){
                Firm aFirm = (Firm) firmList .get(m);
                if (!aFirm.applicationList .contains (aWorker) & aFirm.numVac > 0){
                    aFirm.applicationList .add(aWorker);
                    break ;
                }
            }
        }
    }
}

//prints the application lists of the firms
/*
    System.out.println("Firms' application list:");
    for (int j = 0; j < firmList.size(); j++) {
        Firm aFirm = (Firm) firmList.get(j);
        System.out.print(aFirm.firmID + ": ");
        for (int m = 0; m < aFirm.applicationList.size(); m++) {
            Workers aWorker = (Workers) aFirm.applicationList.get(m);
            System.out.print(aWorker.workerID + " ");
        }
        System.out.println();
    }
*/
}

```

```
//firms pick workers with the lowest reservation wages

public void hiring(){
    //creates a counter for the while loop by counting the number of vacancies
    int counter = 0;
    for (int i = 0; i < firmList.size(); i++){
        Firm aFirm = (Firm) firmList.get(i);
        counter = counter + aFirm.numVac;
    }

    vacSum = counter; //vacSum is needed for statistics

    for (int i = 0; i < workerList.size(); i++){//counts the number of applications
        Workers aWorker = (Workers) workerList.get(i);
        appliSum = appliSum + aWorker.numAppli;
    }

//
    System.out.println("numVac " + counter);//prints the number of vacancies

    while (counter != 0){
        Collections.shuffle(firmList);
        for (int j = 0; j < firmList.size(); j++){
            Firm bFirm = (Firm) firmList.get(j);
            if (!bFirm.applicationList.isEmpty() && bFirm.numVac > bFirm.numJobs){//checks whether
                //bFirm still has applicants and free vacancies

                //stores those applicants that do not yet have jobs
                ArrayList tempApplicationList = new ArrayList();
                Iterator it = bFirm.applicationList.iterator();
                while (it.hasNext()) {
                    Workers aWorker = (Workers) it.next();
                    if (aWorker.status == 0) {
                        tempApplicationList.add(aWorker);
                    }
                }

                if (!tempApplicationList.isEmpty()) {

                    //sorts the tempApplicationList in ascending order of reservation wages
                    ArrayList temp = new ArrayList();
                    while (!tempApplicationList.isEmpty()) {
                        Workers lowest = (Workers) tempApplicationList.get(0);
                        for (int i = 0; i < tempApplicationList.size(); i++) {
                            Workers aWorker = (Workers) tempApplicationList.get(i);
                            if (aWorker.reservationWage < lowest.reservationWage)
                                lowest = aWorker;
                        }
                        tempApplicationList.remove(lowest);
                        temp.add(lowest);
                    }
                    tempApplicationList = temp; //the sorted temp list is handed over
                    //to become the new tempApplicationList

/*
                    System.out.println("tempApplicationList"); //prints the content of the tempApplicationList
                    System.out.print(bFirm.firmID + ": ");
                    for (int k = 0; k < tempApplicationList.size(); k++) {
                        Workers aWorker = (Workers) tempApplicationList.get(k);
                        System.out.print(aWorker.reservationWage + " ");
                    }
                    System.out.println();
*/
                }
            }
        }
    }
}
```

```

        //firm picks worker with lowest reservation wage
        Workers aWorker = (Workers) tempApplicationList .get(0);
        aWorker.status = 1; //worker becomes employed
        aWorker.wage = aWorker.reservationWage; //worker accepts wage offer
        bFirm.numJobs = bFirm.numJobs + 1; //if a jobs gets filled the job counter is increased by one
        bFirm.jobList.add(aWorker);
        bFirm.applicationList.remove(aWorker); //worker is removed from application list
        //of the firm where he found a job
    }
}
}
counter = counter - 1;
}

//printing the wages paid by firms
/*
    System.out.println();
    System.out.println("Wages paid by firms");
    for (int j = 0; j < firmList.size(); j++) {
        Firm aFirm = (Firm) firmList.get(j);
        System.out.print(aFirm.firmID + " ");
        for (int m = 0; m < aFirm.jobList.size(); m++) {
            Workers bWorker = (Workers) aFirm.jobList.get(m);
            System.out.print(bWorker.wage + " ");
        }
        System.out.println();
    }
    System.out.println();
*/

//calculates the firms' wage sum
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    if (!aFirm.jobList.isEmpty()) {
        for (int k = 0; k < aFirm.jobList.size(); k++) {
            Workers aWorker = (Workers) aFirm.jobList.get(k);
            aFirm.firmWageSum = aFirm.firmWageSum + aWorker.wage;
        }
    }
}

}

//reports the results

public void reportResults () {
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        totalJobs = totalJobs + aWorker.status;
        wageSum = wageSum + aWorker.wage; //as the wage of an unemployed worker is
        //zero it does not add to the wage sum
    }

//prints the time series
//    System.out.println("Tick " + "Employment Rate " + "Average Wages " + "Vacancy Rate " + "Application
Rate");
/*
    System.out.print(getTickCount() + " " +
        (totalJobs / numWorkers) + " " +
        (wageSum / totalJobs) + " " +
        (vacSum / numFirms) + " " +
        (appliSum / numWorkers));
    System.out.println();
*/

```

```
//prints the contents of the firm list

/*      System.out.println("Contents of firmList before adaptation");
System.out.print("firmID: ");
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.firmID);

}
System.out.println();

System.out.print("productivity: ");
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.productivity);

}
System.out.println();

System.out.print("numVacs: ");
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.numVac);

}
System.out.println();

System.out.print("numJobs: ");
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.numJobs);

}
System.out.println();

System.out.print("payOff: ");
for (int i = 0; i < firmList.size(); i++) {
    Firm aFirm = (Firm) firmList.get(i);
    System.out.print(" " + aFirm.payOff());

}
System.out.println();
System.out.println();

//prints the contents of the workerList

System.out.println("Contents of workerList before adaptation");
System.out.print("workerID: ");

for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.workerID);

}
System.out.println();
System.out.print("status: ");
for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
    System.out.print(" " + aWorker.status);

}
System.out.println();

System.out.print("reservationWage: ");
for (int i = 0; i < workerList.size(); i++) {
    Workers aWorker = (Workers) workerList.get(i);
```

```
        System.out.print(" " + aWorker.reservationWage);

    }
    System.out.println();

    System.out.print("wage: ");
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        System.out.print(" " + aWorker.wage);

    }
    System.out.println();

    System.out.print("numAppli: ");
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        System.out.print(" " + aWorker.numAppli);

    }
    System.out.println();

    System.out.print("payOff: ");
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        System.out.print(" " + aWorker.payOff());

    }
    System.out.println();
    System.out.println();
*/
}

//adaptation mechanism: firms with negative or zero payOffs die and are replaced with new ones which are
//assigned random strategies

    public void adaptationFirms () {

//creates the survivorList
        ArrayList survivorList = new ArrayList ();
        ArrayList outOfMarketList = new ArrayList ();
        for (int i = 0; i < firmList.size(); i++) {
            Firm aFirm = (Firm) firmList.get(i);
            if (aFirm.payOff() > 0) {
                survivorList.add(aFirm);
            }
            else {
                outOfMarketList.add(aFirm);
            }
        }

//prints the survivorList
/*
        System.out.println("survivorList firmID");
        for (int i = 0; i < survivorList.size(); i++) {
            Firm aFirm = (Firm) survivorList.get(i);
            System.out.print(aFirm.firmID + " ");
        }
        System.out.println();

        System.out.println("outOfMarketList firmID");
        for (int i = 0; i < outOfMarketList.size(); i++) {
            Firm aFirm = (Firm) outOfMarketList.get(i);
            System.out.print(aFirm.firmID + " ");
        }
    }
}
```

```

    }
    System.out.println();
*/

//replaces firms that have negative or zero payOffs by randomly assigning a strategy of a
//surviving firm
    if (!outOfMarketList.isEmpty()){
        for (int i = 0; i < outOfMarketList.size(); i++) {
            Firm aFirm = (Firm)outOfMarketList.get(i);
            if (!survivorList.isEmpty()){
                int randomIndex = getNextIntFromTo (0, survivorList.size() - 1);
                Firm bFirm = (Firm) survivorList.get(randomIndex);
                aFirm.numVac = bFirm.numVac;
            }
            else {
                int randomIndex = getNextIntFromTo (0, numFirms);
                aFirm.numVac = randomIndex;
            }
        }
    }

    survivorList.clear();
    outOfMarketList.clear();

//prints the contents of the firm list after the adaptation
/*
    System.out.println("Contents of firmList after adaptation");
    System.out.print("firmID: ");
    for (int i = 0; i < firmList.size(); i++) {
        Firm aFirm = (Firm) firmList.get(i);
        System.out.print(" " + aFirm.firmID);

    }
    System.out.println();

    System.out.print("numVac: ");
    for (int i = 0; i < firmList.size(); i++) {
        Firm aFirm = (Firm) firmList.get(i);
        System.out.print(" " + aFirm.numVac);

    }

    System.out.println();
    System.out.println();
*/
}

//workers with negative or zero payOffs die and are replaced by new ones that are randomly
//assigned strategies of the survivors

    public void adaptationWorkers (){

//sorts workers in survivor and outOfMarket lists
        ArrayList survivorList = new ArrayList ();
        ArrayList outOfMarketList = new ArrayList ();
        for (int i = 0; i < workerList.size(); i++) {
            Workers aWorker = (Workers) workerList.get(i);
            if (aWorker.payOff() > 0) {
                survivorList.add(aWorker);
            }
            else {
                outOfMarketList.add(aWorker);
            }
        }

        if (!outOfMarketList.isEmpty()) { //assigns strategies
            for (int i = 0; i < outOfMarketList.size(); i++) {
                Workers aWorker = (Workers) outOfMarketList.get(i);
                if (!survivorList.isEmpty()) {
                    int randomIndex = getNextIntFromTo (0, survivorList.size() - 1);

```

```
        Workers bWorker = (Workers) survivorList .get (randomIndex );
        aWorker.numAppli = bWorker.numAppli ;
    }
    else {
        int randomIndex = getNextIntFromTo (0, numWorkers );
        aWorker.numAppli = randomIndex ;
    }
}
}
survivorList .clear ();
outOfMarketList .clear ();

//prints the contents of the workerList after the adaptation
/*
    System.out.println("Contents of workerList after adaptation:");

    System.out.print("workerID: ");
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        System.out.print(" " + aWorker.workerID);
    }
    System.out.println();

    System.out.print("numAppli: ");
    for (int i = 0; i < workerList.size(); i++) {
        Workers aWorker = (Workers) workerList.get(i);
        System.out.print(" " + aWorker.numAppli);
    }
    System.out.println();
    System.out.println();
*/
}

//the main activity of the time step
public void step(){
    //these are the subactivities carried out
    resetWorkers ();
    resetFirms ();
    resetStatistics ();
    applying ();
    hiring ();
    reportResults ();
    adaptationFirms ();
    adaptationWorkers ();
}

public static void main(String[] args) {
    SimInit init = new SimInit ();
    Model m = new Model ();
    init.loadModel (m, null , false );
}
}
```

```
package reservation_wage_matching ;

/*this is the model with endogenous vacancies and endogenous applications
  Version: Turin
*/

import java.util.*;

public class Firm {
    double productivity; //job productivity
    double firmWageSum; //this is the average wage a firm pays its workers

    int firmID; //this is the ID of the firm

    int numVac; //this is the number of vacancies that a firm posts
    int numJobs; //this is the number of filled vacancies at a firm
    double costVac; //cost of posting one vacancy

    ArrayList applicationList = new ArrayList (); //stores applications to firms
    ArrayList jobList = new ArrayList (); //stores wages of productive jobs

    //creating the firm
    public Firm(int i, double p, int v, int j, double c, double s) {
        firmID = i;
        productivity = p;
        numVac = v;
        numJobs = j;
        costVac = c;
        firmWageSum = s;
    }

    //payoff for the firm
    public double payOff () {
        return ( productivity * numJobs - firmWageSum - costVac * numVac);
    }

    //initializing the firm's variables - all jobs of a firm are dissolved
    public void reset () {
        numJobs = 0;
        firmWageSum = 0;
        applicationList .clear ();
        jobList .clear ();
    }
}
```

```
package reservation_wage_matching ;

/*this is the model with endogenous vacancies and endogenous applications
Version: Turin
*/

public class Workers {
    double reservationWage ; //this is the worker's reservation wage
    int workerID ; //this is the ID of a worker
    int status ; //status is 0 if unemployed and 1 if employed
    double wage ; //the worker's wage
    int numAppli ; //number of applications a worker sends
    double costAppli ; //cost for sending one application

//creating the worker
    public Workers (int i, double r, int sta, double w, int n, double c){
        workerID = i;
        reservationWage = r;
        status = sta;
        wage = w;
        numAppli = n;
        costAppli = c;
    }

//payoff for the worker
    public double payOff (){
        return (wage * status - costAppli * numAppli);
    }

//resetting the worker
    public void reset (){
        status = 0;
        wage = 0;
    }
}
```

```
package reservation_wage_matching ;

/*This class does the batch runs. The parameter file params.txt will
be read upon starting the simulations and a data file data.txt will be
generated for output.

    Version: Turin
*/

import uchicago.src.sim.engine.*;
import uchicago.src.sim.analysis.*;
import java.util.*;

public class ModelBatch extends Model {

//Batch variables

int numOfTimeSteps ; //defines the length of each batch replication run

DataRecorder recorder ; //Repast's mechanism for data recording

//the batch model's default settings have to be defined
//all the other default settings are defined in Model.setup()
//all settings here can be overridden by the parameter files

public void setup(){
    //the original model has to be initialized first
    super.setup();

    //this specifies the parameters to be manipulated by Repast's parameter
    //mechanism (i.e. set from the parameter file).

    params = new String[] {"numOfTimeSteps", "numWorkers", "numFirms", "costAppli", "costVac", "productivity"};

    //this initializes the batch part
    numOfTimeSteps = 20; //this sets the number of iterations, also set in params.txt
}

//this method builds the model's internals
//the model instrumentation is build after the model itself has been built
//the data recorder has to be initialized

public void buildModel(){
    //build the original model first

    super.buildModel();

    //set the number of simulation steps
    setStoppingTime (numOfTimeSteps);

    //builds the batch part
    //creates the DataRecorder object and specifies the name of the output file.
    recorder = new DataRecorder ("./data.txt",this );

    //columns are added to the output file
    recorder.createNumericDataSource ("jobs", this , "computeJobs");
    recorder.createNumericDataSource ("aveWage", this , "computeAveWage" );
    recorder.createNumericDataSource ("aveVac", this , "computeAveVacancies" );
    recorder.createNumericDataSource ("aveAppli", this , "computeAveApplications" );
    recorder.createNumericDataSource ("vacSum", this , "computeSumVacancies" );

}

//iterated methods

//in order to prevent excessive output on the screen the original's model
//reportResults() is overridden
public void reportResults(){
```

```
}

//the SimpleModel's atEnd() method is overridden to add functionality that
//writes the simulation results to a file at the end of each replication run

public void atEnd(){
    //reportResults() of the original model is used in order to calculate the results
    //by explicitly calling super.reportResultss() the 'suppression' from above is
    //circumvented

    super.reportResults ();

    //record the results into the DataRecorder
    recorder.record ();
    //write it into the file
    recorder.writeToFile ();
}

//in the following the get/set methods for all the parameters are defined

public int getNumOfTimeSteps (){
    return numOfTimeSteps ;
}

public void setNumOfTimeSteps (int t){
    numOfTimeSteps = t;
}

public int getNumWorkers (){
    return numWorkers ;
}

public void setNumWorkers (int w){
    numWorkers = w;
}

public int getNumFirms (){
    return numFirms ;
}

public void setNumFirms (int f){
    numFirms = f;
}

public double getProductivity (){
    return productivity ;
}

public void setProductivity (double r){
    productivity = r;
}

public double getCostVac (){
    return costVac ;
}

public void setCostVac (double v){
    costVac = v;
}

public double getCostAppli (){
    return costAppli ;
}
}
```

```
public void setCostAppli (double a){
    costAppli = a;
}

public double computeJobs (){
    return (double) totalJobs;
}

public double computeNumWorkers (){
    return (double) numWorkers;
}

public double computeNumFirms (){
    return (double) numFirms;
}

public double computeSumVacancies (){
    return (double) vacSum;
}

public double computeAveWage (){
    if (totalJobs !=0){
        return (double) (wageSum / totalJobs);
    }
    else {
        return -1;
    }
}

public double computeAveVacancies (){
    return (double) (vacSum / numFirms);
}

public double computeAveApplications (){
    return (double) (appliSum / numFirms);
}

//this creates and starts the model
public static void main (String[] args){
    SimInit init = new SimInit();
    //one must create a Model Batch object instead of an instance of Model
    //or that of ModelGUI, in order to get the Batch functionality

    Model m = new ModelBatch();

    //the second parameter specifies the name of the parameter file, while
    //the third one declares that the model is to be run in batch-mode

    init.loadModel (m,"params.txt" ,true );
}
}
```

```
runs: 10

numWorkers {
  start: 10
  end: 50
  incr: 1
  {
    runs: 1
    numFirms {
      start: 10
      end: 50
      incr: 1
    }
  }
}

costAppli {
  set: 0.1
}

costVac {
  set: 0.4
}

numOfTimeSteps {
  set: 20
}

productivity {
  set: 1.0
}
```

```
package reservation_wage_matching ;

/* this is the model with endogenous vacancies and endogenous applications
   Version: Turin
*/

import uchicago.src.sim.gui.*;
import uchicago.src.sim.engine.*;
import uchicago.src.sim.analysis.*;
import java.util.*;

public class ModelGUI extends Model{

    public OpenSequenceGraph graph; //the dynamic graph chart provided by repast
    public OpenSequenceGraph graph2;

// the constructor of the GUI
    public ModelGUI (){
        //setting the order of appearance of parameters to non-alphabetic
        Controller.ALPHA_ORDER = false ;
    }

// what follows is the definition of the sequence(s) to be shown on the graph

    class SeqEmpRate implements Sequence {
        //this is the method to be defined for sequence
        public double getSValue () {
            //returns the appropriate value
            return (double ) (totalJobs / numWorkers);
        }
    }

    class SeqAveWage implements Sequence {
        //this is the method to be defined for sequence
        public double getSValue () {
            //returns the appropriate value
            return (double ) (wageSum / totalJobs);
        }
    }

    class SeqAveVac implements Sequence {
        //this is the method to be defined for sequence
        public double getSValue () {
            //returns the appropriate value
            return (double ) (vacSum / numFirms);
        }
    }

    class SeqAveAppli implements Sequence {
        //this is the method to be defined for sequence
        public double getSValue () {
            //returns the appropriate value
            return (double ) (appliSum / numWorkers);
        }
    }

//initializing the GUI model
    public void setup () {
        //initializing the original model first
        super .setup ();

        //specify the parameters to be displayed for setting by the user

        params = new String [] { "numWorkers", "numFirms", "costAppli", "costVac" };

        //initializing the GUI part, the graphics objects from the previous run have to
        //be deleted
    }
}
```

```
if (graph != null )
    graph.dispose ();

if (graph2 != null )
    graph2.dispose ();

}

public void buildModel (){
    //this builds the original model first
    super .buildModel ();

    //this builds the GUI part, creates a sequence chart and sets it up
    graph = new OpenSequenceGraph ("Time Series" , this );
    graph2 = new OpenSequenceGraph ("Time Series" , this );

    //this adds the defined sequences
    graph.addSequence ("EmpRate" , new SeqEmpRate ());
    graph.addSequence ("AveWage" , new SeqAveWage ());
    graph2.addSequence ("AveReservationWage" , new SeqAveAppli ());
    graph2.addSequence ("AveVacancies" , new SeqAveVac ());

    //displays the graph
    graph.display ();
    graph2.display ();

    //does the first update
    graph.step ();
    graph2.step ();

}

//the activity of the time step
//in each time step first execute the model then update the graph

public void step(){
    //do the original model's step() method first
    super .step ();
    //do the GUI part by updating the graph
    graph.step ();
    graph2.step ();

}

//in the following the get,set methods for all parameter(s) listed in params
//are provided

public int getNumWorkers (){
    return numWorkers ;
}

public void setNumWorkers (int n){
    numWorkers = n;
}

public int getNumFirms (){
    return numFirms ;
}

public void setNumFirms (int p){
```

```
    numFirms = p;
}

public double getCostVac (){
    return costVac;
}

public void setCostVac (int v){
    costVac = v;
}

public double getCostAppli (){
    return costAppli;
}

public void setCostAppli (int a){
    costAppli = a;
}

//creating and starting the model

public static void main(String[] args){
    SimInit init = new SimInit();
    //there has to be created a ModelGui object instead of an instance of Model
    //as in the parent class, in order to get the GUI functionality.
    Model m = new ModelGUI ();
    init.loadModel (m, null , false );
}
}
```