

```

1 #This is a simple function that numerically finds a solution to an equation of the form A=fn(x), on the in
2
3 function bisect_root(fn, lower, upper, lhs)
4   x0 = lower
5   x2 = upper
6   x1 = (x0+x2)/2
7
8   y0 = fn(x0)-lhs
9   y1 = fn(x1)-lhs
10  y2 = fn(x2)-lhs
11
12  while x0 < x1 && x1 < x2
13    if sign(y0) == sign(y1)
14      x0, x2 = x1, x2
15      y0, y2 = y1, y2
16    else
17      x0, x2 = x0, x1
18      y0, y2 = y0, y1
19    end
20
21    x1 = (x0+x2)/2
22    y1 = fn(x1)-lhs
23  end
24 x1
25 end
26
27 function bisect_rootalt(fn, lower, upper, lhs)
28   x0 = lower
29   x2 = upper
30   x1 = (x0+x2)/2
31
32   y0 = 0-lhs
33   y1 = 1-lhs
34   y2 = fn(x2)-lhs
35
36  while x0 < x1 && x1 < x2
37    if sign(y0) == sign(y1)
38      x0, x2 = x1, x2
39      y0, y2 = y1, y2
40    else
41      x0, x2 = x0, x1
42      y0, y2 = y0, y1
43    end
44
45    x1 = (x0+x2)/2
46    y1 = fn(x1)-lhs
47  end
48 x1
49 end
50
51 #This function calculates map T, given vc, uc and some function z:e (equivalent of d in the proof of The
52
53 function Tmapf(psi::Array{Float64, 1}, z::Array{Float64, 1}, Cuex::Function, Cvex::Function, piluex::Func
54   step=(1-vc)/(length(psi)-1)
55   test=psi[1]
56   Cevd=Cvex.(psi, z)
57   Ceud=Cuex.(psi, z)
58   Cevd[end]=Cevd[end-1] #This is done because Cvex(dot, v) is not continuous in u. This is not a problem
59   Ceud[end]=Ceud[end-1]
60   M2cum=Array{Float64}(undef, length(psi))
61   M2cum[1]=0
62   M1cum=Array{Float64}(undef, length(psi))
63   M1cum[1]=0
64   for i=2:length(psi)
65     M2cum[i]=M2cum[i-1]+(Cevd[i-1]+Cevd[i])*(step/2) #numerical appoximation of an integral, trapezoi
66     M1cum[i]=M1cum[i-1]+(Ceud[i-1]+Ceud[i])*((psi[i]-psi[i-1])/2)
67   end
68   nom=Array{Float64}(undef, length(z)) #numerator of map T
69   for i=1:length(z)
70     nom[i]=pi2vex(z[i], (R2-M2+M2cum[i])/R2)
71   end
72   denom=Array{Float64}(undef, length(psi)) #denominator of map T

```

```

73   for i=1:length(psi)
74       denom[i]=pi1uex(psi[i], (R1-M1+M1cum[i])/R1) #numerical approximation of an integral, trapezoid me
75   end
76   inside=nom./denom
77   f=Array{Float64}{undef, length(psi)}
78   f[1]=uc
79   for i=2:length(psi)
80       f[i]=f[i-1]+(inside[i-1]+inside[i])*(step/2) #numerical approximation of an integral, trapezoid me
81   end
82   return f, M2cum, M1cum, nom, denom
83 end
84
85 #this is the main function that finds the stable matching, by finding the fixed point of map T (see the p
86 function equilibriumbase(Cex::Function, Cuex::Function, Cvex::Function, pi1uex::Function, pi2vex::Function,
87     r1=1
88     if R1+R2==1
89         adash=vcdash
90     amax=vcmax
91     if vc>0
92         a=vc
93     else
94         a=-uc
95     end
96 end
97
98 #The next line uses the Euler's method to find the fixed point of map T (proof of Theorem 1) for
99 M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Function,
100     Mass=M2cum[end]
101 #The following uses the bisection method to find the equilibrium pair of critical values
102 for i=1:no_step1
103
104     # The next Loop allows for three possibilities. If the mass of manufacturing workers is very clo
105     #If the mass of workers is somewhat close to the mass of firms, we enter an interior function, t
106     #Finally, if we are still far away from the solution, it start iterating over (uc, vc).
107     #The idea is to approach the solution using the fast Euler method, but once sufficiently close,
108     #The first possibility is left so that for the symmetric equilibria, where the solution is reach
109     if (Mass-R2)^2<(0.00001)^2#(0.000001)^2
110         break
111
112     elseif (i>1 && (Mass-R2)^2<(0.4)^2)
113         psivecins, M2cum, M1cum=stableint(C, Cuex, Cvex, pi1uex, pi2vex, psivecins, z, M2cum, M1
114     break
115
116     else
117         if R1+R2<1
118             if Mass>R2 #If not, we change vc and uc. If the mass of agents was too large, we increase vc, ot
119             vcdash, vcmax=vc, vcmax
120         else
121             vcdash, vcmax=vcdash, vc
122         end
123         vc=(vcdash+vcmax)/2
124         Cvc(u)=Cex(u, vc)
125         uc=bisect_rootalt(Cvc, 0, 1, 1-R1-R2)
126     else
127         if Mass>R2
128             adash, amax=a, amax
129         else
130             adash, amax=adash, a
131         end
132         a=(adash+amax)/2
133         if a<0
134             vc=0
135             uc=-a
136         else
137             vc=a
138             uc=0
139         end
140     end
141     r1=i+1
142     #Finally, we calcualte the fixed point of map T (again, using Euler's method) for the new pair u
143     M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Function,
144     Mass=M2cum[end]
145 end
146 end

```

```

147         if r1>no_step1 #This Let's us know if the above loop did not converge
148             print("no convergence")
149         end
150     return    psivecins, z, M2cum, M1cum, uc, vc, r1
151 end
152
153 #This is a function that approximates the solution of a differential equation using Euler's method
154
155 function eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Function, Cuex::Function, R2::Number, R1:
156 #First, define a few arrays that will be used to store the results
157     z=Array{Float64}(undef, grid+1) #the array of argument
158     M1cum=Array{Float64}(undef, grid+1) #the array of values of the cumulative mass function for manufa
159     M2cum=Array{Float64}(undef, grid+1) #the array of values of the cumulative mass function for servic
160     psivecins=Array{Float64}(undef, grid+1) #the array of values of the separation function
161 #The next fixe lines create the vector of arguments
162     step=(1-vc)/grid
163     z[1]=vc
164     for i=2:grid+1
165         z[i]=step*(i-1)+z[1] #vector of arguments
166     end
167     #The next 8 lines set the initial values
168     psivecins[1]=uc
169     vprev=z[1]
170     M1cum[1]=0
171     M2cum[1]=0
172     j=1
173     psiprev=psivecins[1]
174     M2=0
175     M1=0
176     #The next Loop implemets the Euler's method to approximate the fixed point of map T (proof of Theore
177     for i=2:(length(z))
178 #First, we evaluate the derivative of the separation function, using the values of the argument from th
179         up=pi2vex(vprev, M2/R2)
180         down=pi1uex(psiprev, M1/R1)
181         psiprime=pi2vex(vprev, M2/R2)/pi1uex(psiprev, M1/R1)
182         # We use this to update the value of the separation function
183         psi=psiprev+psiprime*step
184         #and then update the balue of the arguemnt
185         v=vprev+step
186         #using those update values, we deploy the trapezoid method to update the values of the mass of worke
187         M2=M2+(Cvex(psiprev, vprev)+Cvex(psi, v))*step/2
188         M1=M1+(Cuex(psiprev, vprev)+Cuex(psi, v))*psiprime*step/2
189         #The new values a stored within the previously defined matrices
190         M1cum[i]=M1
191         M2cum[i]=M2
192         psivecins[i]=psi
193         #and, finally, we update the "old" values of psi and v
194         vprev=v
195         psiprev=psi
196     end
197     return M2cum, psivecins, z, M1cum, vc, uc
198 end
199
200
201 #This is an internal function used to improve the precision of the calculations
202 #The idea is the same as in the external function but here the Euler method's solution is further iterat
203 function stableint(Cex::Function, Cuex::Function, Cvex::Function, pi1uex::Function, pi2vex::Function, ps
204     r=1
205     r1=1
206     if R1+R2==1
207         adash=vcdash
208     amax=vcmax
209     if vc>0
210         a=vc
211     else
212         a=-uc
213     end
214 end
215 #The following loop keeps iterating over function psi, until the former and latter iterations become
216 for i=1:no_step
217     r=i
218
219     old=psivecins
220     psivecins, M2cum, M1cum = Tmapf(psivecins, z, Cuex, Cvex, pi1uex, pi2vex, R2, R1, vc, uc, R1, R

```

```

221         dfr=(psivecins-old).^2
222         maxdfr=maximum(dfr)
223         #print("$maxdfr $r \n")
224     if maximum(dfr)<(0.000001)^2
225         break
226     end
227 end
228     if r>=no_step #This Let's us know if the above loop did not converge
229         print("no convergence")
230     end
231     Mass=M2cum[end]
232     for i=1:no_step1
233         #The next loop iterates over (uc, vc) until the mass of manufacturing workers becomes sufficient
234         if (Mass-R2)^2<(0.000001)^2
235             break
236         else
237             if R1+R2<1
238                 if Mass>R2 #If not, we change vc and uc. If the mass of agents was too large, we increase vc, otherw
239                     vcdash, vcmax=vc, vcmax
240                 else
241                     vcdash, vcmax=vcdash, vc
242                 end
243                 vc=(vcdash+vcmax)/2
244                 Cvc(u)=Cex(u, vc)
245                 uc=bisect_rootalt(Cvc, 0, 1, 1-R1-R2)
246     else
247         if Mass>R2
248             adash, amax=a, amax
249         else
250             adash, amax=adash, a
251         end
252         a=(adash+amax)/2
253         if a<0
254             vc=0
255             uc=-a
256         else
257             vc=a
258             uc=0
259         end
260     end
261     r1=i+1
262     #print("$r1 ")
263     #print("$vc $uc $adash $amax ")
264     #print("$Mass ")
265     M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex:
266 #Next: same as above, iteration over psi.
267     for i=1:no_step
268         r=i
269         old=psivecins
270         psivecins, M2cum, M1cum = Tmapf(psivecins, z, Cuex, Cvex, pi1uex, pi2vex, R2, R1, vc, uc, R1, R
271         dfr=(psivecins-old).^2
272         maxdfr=maximum(dfr)
273         if maximum(dfr)<(0.000001)^2
274             break
275         end
276     end
277         if r>=no_step #This Let's us know if the above loop did not converge
278             print("no convergence")
279         end
280         Mass=M2cum[end]
281     end
282 end
283     if r1>=no_step1 #This Let's us know if the above loop did not converge
284         print("no convergence")
285     end
286     return psivecins, M2cum, M1cum, uc, vc, r1
287 end
288
289
290 #this is a function that given f: z->d plus the vector of arguments of f's inverse, returns the values
291 function inverse(u::Array{Float64, 1}, d::Array{Float64, 1}, z::Array{Float64, 1})
292     pha=Array{Float64}(undef, grid+1)
293     pha[1]=z[1]
294     i=2

```

```

295 for j=2:grid+1
296     if u[j]>d[end]
297         pha[j]=z[end]
298     else
299         while u[j]>d[i]
300             i=i+1
301         end
302         pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
303     end
304
305 end
306 return pha
307 end
308
309 function inverse(u, d, z)
310 pha=Array{Float64}(undef, grid+1)
311 pha[1]=z[1]
312 i=2
313 for j=2:grid+1
314     if u[j]>d[end]
315         pha[j]=z[end]
316     else
317         while u[j]>d[i]
318             i=i+1
319         end
320         pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
321     end
322
323 end
324 return pha
325 end
326
327 #this is a function that, given g: [ucr1, 1]->Cuph, returns (a vector form of) f(x)=int_a^x g(r) dr from
328 function inttrapvec(Cuph::Array{Float64, 1}, ucr1::Float64)
329     mass1r1=Array{Float64}(undef, size(Cuph)[1])
330     mass1r1[1]=0
331     for i=2:grid+1
332         mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
333     end
334     mass1r1
335 end
336 function inttrapvec(Cuph::Array{Float64, 1}, ucr1::Int64)
337     mass1r1=Array{Float64}(undef, size(Cuph)[1])
338     mass1r1[1]=0
339     for i=2:grid+1
340         mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
341     end
342     mass1r1
343 end
344 function inttrapvec(Cuph, ucr1)
345     mass1r1=Array{Float64}(undef, size(Cuph)[1])
346     mass1r1[1]=0
347     for i=2:grid+1
348         mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
349     end
350     mass1r1
351 end
352 #The rescale takes as the input a function f: d->z and then returns a vector pha, such that u->pha also
353 function rescale(d::Array{Float64, 1}, z::Array{Float64, 1}, grid, start=0) #first argument is the or
354     u=collect(0:(1/grid):1)
355     pha=Array{Float64}(undef, grid+1)
356     pha[1]=start
357     i=1
358     for j=2:grid+1
359         if u[j]<d[1]
360             pha[j]=start
361         else
362             if u[j]>d[end]
363                 pha[j]=z[end]
364             else
365                 while u[j]>d[i]
366                     i=i+1
367                 end
368                 pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])

```

```

369 end
370     end
371
372
373 end
374     return pha
375 end
376
377 function rescale(d, z, grid, start=0)
378 u=collect(0:(1/grid):1)
379     pha=Array{Float64}(undef, grid+1)
380     pha[1]=start
381 i=1
382 for j=2:grid+1
383     if u[j]<d[1]
384         pha[j]=start
385     else
386         if u[j]>d[end]
387             pha[j]=z[end]
388         else
389             while u[j]>d[i]
390                 i=i+1
391             end
392             pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
393         end
394     end
395 end
396     return pha
397 end
398
399 #the following function takes a function x: f and returns its value for an arbitrary argument a such tha
400 #This is done by finding the closes highest and lowest values of x than a, and then linearly approximati
401 function makecont(a::Number, x::Array, f::Array)
402     i=1
403     if a<x[1]
404         print("DOMAIN ERROR")
405     elseif a>x[end]
406         print("DOMAIN ERROR")
407     else
408         while a>x[i]
409             i=i+1
410         end
411     end
412 if i==1
413     return f[1]
414 else
415     return f[i-1]+(f[i]-f[i-1])*(a-x[i-1])/(x[i]-x[i-1])
416 end
417 end
418
419 function makecont(a::Array, x::Array, f::Array)
420     i=1
421     z=Array{Float64}(undef, length(a), 1)
422     for j=1:length(a)
423         if a[j]<x[1]
424             print("DOMAIN ERROR")
425         elseif a[j]>x[end]
426             print("DOMAIN ERROR")
427         else
428             while a[j]>x[i]
429                 i=i+1
430             end
431         end
432         if i==1
433             z[j]=f[1]
434         else
435             z[j]=f[i-1]+(f[i]-f[i-1])*(a[j]-x[i-1])/(x[i]-x[i-1])
436         end
437     end
438     return z
439 end
440

```

