```
#This is a simple function that numerically finds a solution to an equation of the form A=fn(x), on the in

function bisect_root(fn, lower, upper, lhs)
  x0 = lower
  x2 = upper
  x1 = (x0+x2)/2

    y0 = fn(x0)-lhs
    y1 = fn(x1)-lhs
    y2 = fn(x2)-lhs

  while x0 < x1 && x1 < x2
    if sign(y0) == sign(y1)
      x0, x2 = x1, x2
      y0, y2 = y1, y2
    else
      x0, x2 = x0, x1
      y0, y2 = y0, y1
    end

    x1 = (x0+x2)/2
        y1 = fn(x1)-lhs
  end
x1
end

function bisect_rootalt(fn, lower, upper, lhs)
  x0 = lower
  x2 = upper
  x1 = (x0+x2)/2

    y0 = 0-lhs
    y1 = 1-lhs
    y2 = fn(x2)-lhs

  while x0 < x1 && x1 < x2
    if sign(y0) == sign(y1)
      x0, x2 = x1, x2
      y0, y2 = y1, y2
    else
      x0, x2 = x0, x1
      y0, y2 = y0, y1
    end

    x1 = (x0+x2)/2
        y1 = fn(x1)-lhs
  end
x1
end

#This function calculates map T, given vc, uc and some function  z:e (equivalent of d in the proof of The

function Tmapf(psi::Array{Float64, 1}, z::Array{Float64, 1}, Cuex::Function, Cvex::Function, pi1uex::Func
    step=(1-vc)/(length(psi)-1)
    test=psi[1]
    Cevd=Cvex.(psi, z)
    Ceud=Cuex.(psi, z)
    Cevd[end]=Cevd[end-1] #This is done because Cvex(dot, v) is not continuous in u. This is not a proble
    Ceud[end]=Ceud[end-1]
    M2cum=Array{Float64}(undef,  length(psi))
    M2cum[1]=0
    M1cum=Array{Float64}(undef,  length(psi))
    M1cum[1]=0
    for i=2:length(psi)
        M2cum[i]=M2cum[i-1]+(Cevd[i-1]+Cevd[i])*(step/2) #numerical appoximation of an integral, trapezoi
        M1cum[i]=M1cum[i-1]+(Ceud[i-1]+Ceud[i])*((psi[i]-psi[i-1])/2)
end
    nom=Array{Float64}(undef,  length(z)) #numerator of map T
    for i=1:length(z)
        nom[i]=pi2vex(z[i], (R2-M2+M2cum[i])/R2)
end
    denom=Array{Float64}(undef,  length(psi)) #denumerator of map T
```

```
    for i=1:length(psi)
        denom[i]=pi1uex(psi[i], (R1-M1+M1cum[i])/R1) #numerical appoximation of an integral, trapezoid me
end
    inside=nom./denom
    f=Array{Float64}(undef,  length(psi))
    f[1]=uc
    for i=2:length(psi)
        f[i]=f[i-1]+(inside[i-1]+inside[i])*(step/2) #numerical appoximation of an integral, trapezoid me
end
    return f, M2cum, M1cum, nom, denom
end

#this is the main function that finds the stable matching, by finding the fixed point of map T (see the p
function equilibriumbase(Cex::Function, Cuex::Function, Cvex::Function, pi1uex::Function, pi2vex::Functio
        r1=1
        if R1+R2==1
            adash=vcdash
amax=vcmax
            if vc>0
                a=vc
            else
                a=-uc
            end
        end

        #The next line uses the EUler's method to find the fixed point of map T (proof of Theorem 1) for
        M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Function
         Mass=M2cum[end]
         #The following uses the bisection method to find the equilibrium pair of critical values
         for i=1:no_step1

         # The next loop allows for three possibilities. If the mass of manufacturing workers is very clo
         #If the mass of workers is somewhat close to the mass of firms, we enter an interior function, t
         #Finally, if we are still far away from the solution, it start iterating over (uc, vc).
         #The idea is to approach the solution using the fast Euler method, but once sufficiently close,
         #The first possibility is left so that for the symmetric equilibria, where the solution is reach
             if (Mass-R2)^2<(0.00001)^2#(0.0000001)^2
                 break

             elseif (i>1 && (Mass-R2)^2<(0.4)^2)
                 psivecins, M2cum, M1cum=stableint(C, Cuex, Cvex, pi1uex, pi2vex, psivecins, z, M2cum, M1
         break

         else
             if R1+R2<1
         if Mass>R2 #If not, we change vc and uc. If the mass of agents was too large, we increase vc, ot
         vcdash, vcmax=vc, vcmax
         else
         vcdash, vcmax=vcdash, vc
         end
         vc=(vcdash+vcmax)/2
         Cvc(u)=Cex(u, vc)
         uc=bisect_rootalt(Cvc, 0, 1, 1-R1-R2)
     else
         if Mass>R2
             adash, amax=a, amax
         else
             adash, amax=adash, a
         end
     a=(adash+amax)/2
                 if a<0
                     vc=0
                     uc=-a
                 else
                     vc=a
                     uc=0
                 end
             end
         r1=i+1
         #Finally, we calcualte the fixed point of map T (again, using Euler's method) for the new pair u
         M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Functio
         Mass=M2cum[end]
         end
     end
```

```
        if r1>=no_step1 #This let's us know if the above loop did not converge
                    print("no convergence")
                end
        return    psivecins, z, M2cum, M1cum, uc, vc, r1
end

#This is a function that approximates the solution of a differential equation using Euler's method

function eulermethod(pi1uex::Function, pi2vex::Function, Cvex::Function, Cuex::Function, R2::Number, R1:
#First, define a few arrays that will be used to store the results
    z=Array{Float64}(undef,  grid+1) #the array of argument
    M1cum=Array{Float64}(undef,  grid+1) #the array of values of the cumulative mass function for manufa
    M2cum=Array{Float64}(undef,  grid+1) #the array of values of the cumulative mass function for servic
    psivecins=Array{Float64}(undef,  grid+1) #the array of values of the separation function
#The next fixe lines create the vector of arguments
    step=(1-vc)/grid
    z[1]=vc
    for i=2:grid+1
    z[i]=step*(i-1)+z[1] #vector of arguments
    end
    #The next 8 lines set the initial values
    psivecins[1]=uc
    vprev=z[1]
    M1cum[1]=0
    M2cum[1]=0
    j=1
    psiprev=psivecins[1]
    M2=0
    M1=0
    #The next loop implemets the Euler's method to approximate the fixed point of map T (proof of Theore
    for i=2:(length(z))
#First, we evalueate the derivative of the separation function, using the values of the argument from th
    up=pi2vex(vprev, M2/R2)
    down=pi1uex(psiprev, M1/R1)
        psiprime=pi2vex(vprev, M2/R2)/pi1uex(psiprev, M1/R1)
        # We use this to update the value of the separation function
    psi=psiprev+psiprime*step
    #and then update the balue of the arguemnt
    v=vprev+step
    #using those update values, we deploy the trapezoid method to update the values of the mass of worke
    M2=M2+(Cvex(psiprev, vprev)+Cvex(psi, v))*step/2
        M1=M1+(Cuex(psiprev, vprev)+Cuex(psi, v))*psiprime*step/2
        #The new values a stored within the previously defined matrices
    M1cum[i]=M1
    M2cum[i]=M2
        psivecins[i]=psi
        #and, finally, we update the "old" values of psi and v
        vprev=v
        psiprev=psi
    end
    return M2cum, psivecins, z, M1cum, vc, uc
end


#This is an internal function used to improve the precision of the calculations
#The idea is the same as in the external function but here the Euler method's solution is further iterat
function stableint(Cex::Function, Cuex::Function, Cvex::Function, pi1uex::Function, pi2vex::Function, ps
    r=1
    r1=1
    if R1+R2==1
        adash=vcdash
amax=vcmax
        if vc>0
            a=vc
        else
            a=-uc
        end
    end
    #The following loop keeps iterating over function psi, until the former and latter iterations become
    for i=1:no_step
    r=i

         old=psivecins
         psivecins, M2cum, M1cum = Tmapf(psivecins, z, Cuex, Cvex, pi1uex, pi2vex, R2, R1, vc, uc, R1, R
```

```
         dfr=(psivecins-old).^2
         maxdfr=maximum(dfr)
         #print("$maxdfr $r \n")
    if maximum(dfr)<(0.000001)^2
     break
    end
    end
     if r>=no_step #This let's us know if the above loop did not converge
                 print("no convergence")
             end
    Mass=M2cum[end]
    for i=1:no_step1
        #The next loop iterates over (uc, vc) until the mass of manufacturing workers becomes sufficient
    if (Mass-R2)^2<(0.00001)^2
     break
    else
        if R1+R2<1
    if Mass>R2 #If not, we change vc and uc. If the mass of agents was too large, we increase vc, otherw
    vcdash, vcmax=vc, vcmax
    else
    vcdash, vcmax=vcdash, vc
    end
    vc=(vcdash+vcmax)/2
    Cvc(u)=Cex(u, vc)
    uc=bisect_rootalt(Cvc, 0, 1, 1-R1-R2)
else
    if Mass>R2
        adash, amax=a, amax
    else
        adash, amax=adash, a
    end
a=(adash+amax)/2
            if a<0
                vc=0
                uc=-a
            else
                vc=a
                uc=0
            end
        end
        r1=i+1
        #print("$r1 ")
        #print("$vc $uc $adash $amax ")
        #print("$Mass ")
                M2cum, psivecins, z, M1cum, vc, uc=eulermethod(pi1uex::Function, pi2vex::Function, Cvex:
#Next: same as above, iteration over psi.
    for i=1:no_step
                 r=i
         old=psivecins
         psivecins, M2cum, M1cum = Tmapf(psivecins, z, Cuex, Cvex, pi1uex, pi2vex, R2, R1, vc, uc, R1, R
         dfr=(psivecins-old).^2
         maxdfr=maximum(dfr)
    if maximum(dfr)<(0.000001)^2
     break
    end
    end
             if r>=no_step #This let's us know if the above loop did not converge
                 print("no convergence")
             end
    Mass=M2cum[end]
    end
    end
     if r1>=no_step1 #This let's us know if the above loop did not converge
                 print("no convergence")
             end
     return    psivecins, M2cum, M1cum, uc, vc, r1
end


#this is a function that given f: z->d plus the vector of arguments of f's inverse,  returns the values
function inverse(u::Array{Float64, 1}, d::Array{Float64, 1}, z::Array{Float64, 1})
pha=Array{Float64}(undef,  grid+1)
    pha[1]=z[1]
i=2
```

```
for j=2:grid+1
        if u[j]>d[end]
     pha[j]=z[end]
        else
        while u[j]>d[i]
            i=i+1
end
        pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
end

end
    return pha
end

function inverse(u, d, z)
pha=Array{Float64}(undef,  grid+1)
    pha[1]=z[1]
i=2
for j=2:grid+1
        if u[j]>d[end]
     pha[j]=z[end]
        else
        while u[j]>d[i]
            i=i+1
end
        pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
end

end
    return pha
end

#this is a function that, given g: [ucr1, 1]->Cuph, returns (a vector form of) f(x)=int_a^x g(r) dr from
function inttrapvec(Cuph::Array{Float64, 1}, ucr1::Float64)
    mass1r1=Array{Float64}(undef,  size(Cuph)[1])
mass1r1[1]=0
for i=2:grid+1
    mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
end
mass1r1
end
function inttrapvec(Cuph::Array{Float64, 1}, ucr1::Int64)
    mass1r1=Array{Float64}(undef,  size(Cuph)[1])
mass1r1[1]=0
for i=2:grid+1
    mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
end
mass1r1
end
function inttrapvec(Cuph, ucr1)
    mass1r1=Array{Float64}(undef,  size(Cuph)[1])
mass1r1[1]=0
for i=2:grid+1
    mass1r1[i]=mass1r1[i-1]+(Cuph[i-1]+Cuph[i])*(1-ucr1)/(2*grid)
end
mass1r1
end
#The rescale takes as the input a function f: d->z and then returns a vactor pha, such that u->pha also
function rescale(d::Array{Float64, 1}, z::Array{Float64, 1}, grid, start=0)    #first argument is the or
    u=collect(0:(1/grid):1)
    pha=Array{Float64}(undef,  grid+1)
    pha[1]=start
i=1
for j=2:grid+1
        if u[j]<d[1]
            pha[j]=start
        else
        if u[j]>d[end]
     pha[j]=z[end]
        else
        while u[j]>d[i]
            i=i+1
end
        pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
```

```
end
        end


end
    return pha
end

function rescale(d, z, grid, start=0)
u=collect(0:(1/grid):1)
    pha=Array{Float64}(undef,  grid+1)
    pha[1]=start
i=1
for j=2:grid+1
        if u[j]<d[1]
            pha[j]=start
        else
        if u[j]>d[end]
     pha[j]=z[end]
        else
        while u[j]>d[i]
            i=i+1
end
        pha[j]=z[i-1]+(z[i]-z[i-1])*(u[j]-d[i-1])/(d[i]-d[i-1])
end
        end
end
    return pha
end

#the following function takes a function x: f and returns its value for an arbitrary argument a such tha
#This is done by finding the closes highest and lowest values of x than a, and then linearly approximati
function makecont(a::Number, x::Array, f::Array)
    i=1
            if a<x[1]
         print("DOMAIN ERROR")
     elseif a>x[end]
         print("DOMAIN ERROR")
     else
            while a>x[i]
                i=i+1
    end
end
if i==1
    return f[1]
else
            return f[i-1]+(f[i]-f[i-1])*(a-x[i-1])/(x[i]-x[i-1])
        end
            end

function makecont(a::Array, x::Array, f::Array)
                i=1
                z=Array{Float64}(undef,  length(a), 1)
                for j=1:length(a)
                        if a[j]<x[1]
                     print("DOMAIN ERROR")
                 elseif a[j]>x[end]
                     print("DOMAIN ERROR")
                 else
                        while a[j]>x[i]
                            i=i+1
                end
            end
            if i==1
                 z[j]=f[1]
            else
                        z[j]=f[i-1]+(f[i]-f[i-1])*(a[j]-x[i-1])/(x[i]-x[i-1])
                    end
                end
return z
                        end

```