Filtros de duas dimensões

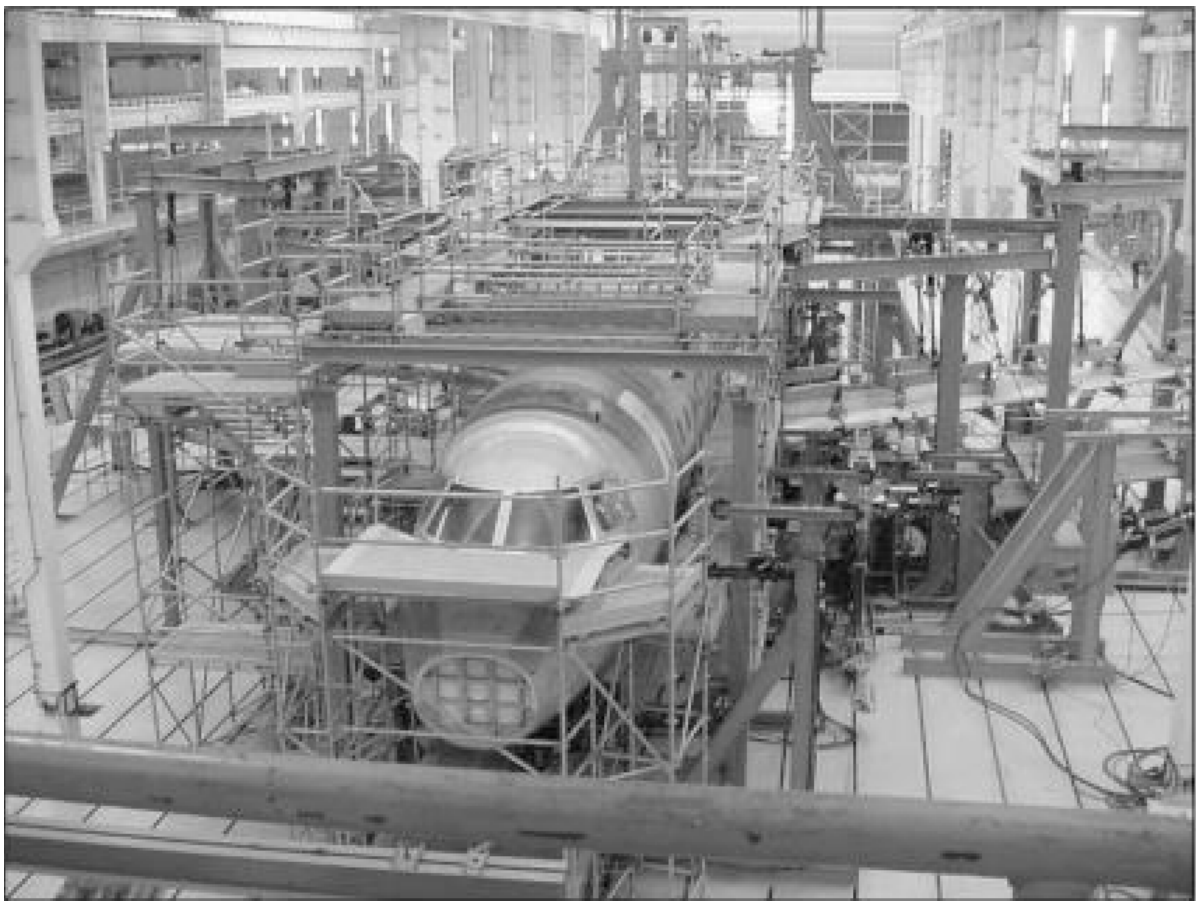
In [54]:
```
using DSP, Images, ImageView, FileIO, MAT, OffsetArrays, FFTViews
include("/Users/vitor/arquivos/docs/cursos/Julia/pfft2.jl")
using PyPlot: surf
```

In [55]:
```
EMB170 = load("Embraer-170-190-fatigue-dt-tests3-n.png")
```

Out[55]:

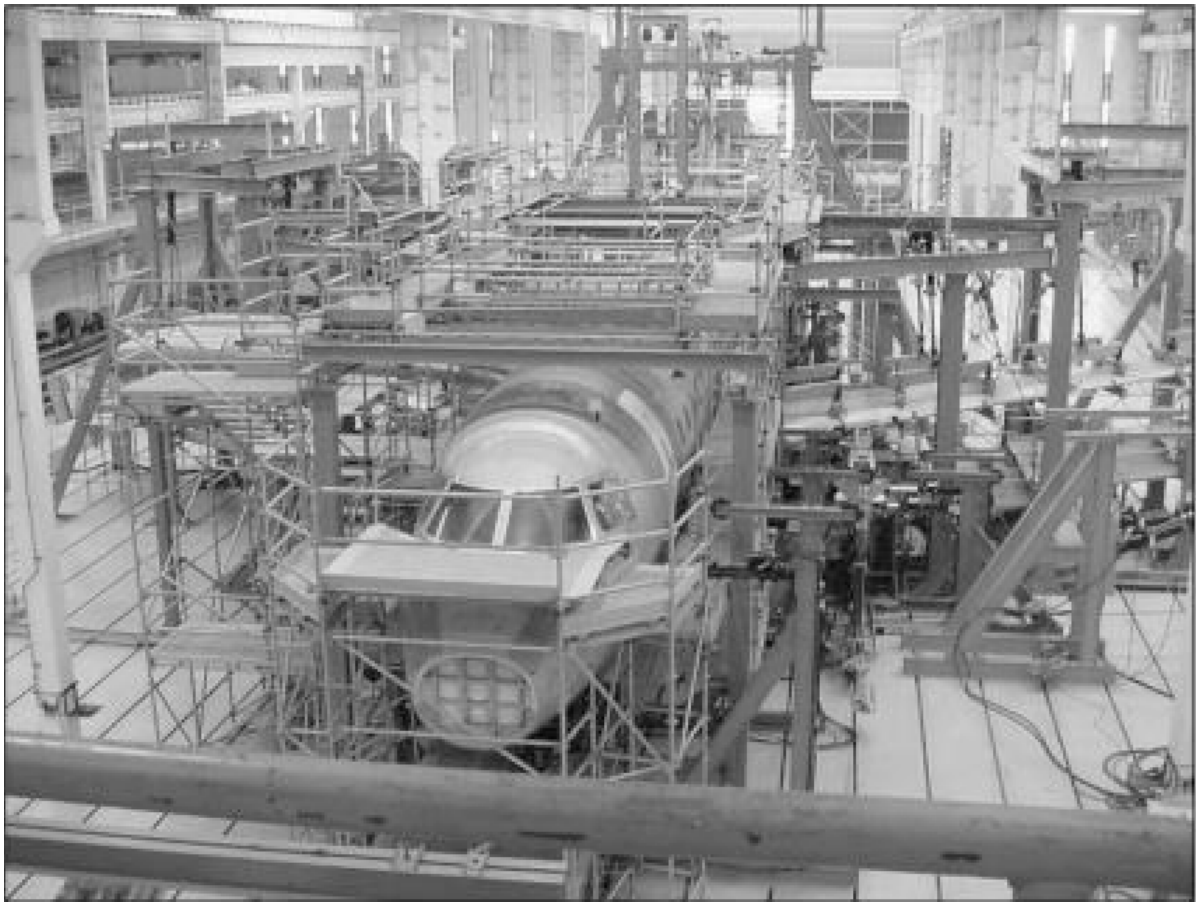
In [56]:
```
EMB170c = [Gray(0.299a.r + 0.587a.g + 0.114a.b) for a in EMB170]
```

Out[56]:



In [57]:
```
EMB170c = Gray.(EMB170)
```

Out[57]:



In [58]:
```
EMB170c1 = 0.299red.(EMB170) + 0.587green.(EMB170) + 0.114blue.(EMB
170)
```

Out[58]:
```
640×852 Array{Float64,2}:
 0.411149  0.321957  0.321957  0.263008  …  0.298039  0.298039  0.
694118
 0.411149  0.321957  0.321957  0.263008     0.298039  0.298039  0.
694118
 0.216353  0.284471  0.284471  0.325863     0.215686  0.215686  0.
584314
 0.216353  0.284471  0.284471  0.325863     0.215686  0.215686  0.
584314
 0.139373  0.356384  0.356384  0.483157     0.294118  0.294118  0.
54902
 0.139373  0.356384  0.356384  0.483157  …  0.294118  0.294118  0.
54902
 0.184686  0.381196  0.381196  0.474294     0.278431  0.278431  0.
564706
 0.184686  0.381196  0.381196  0.474294     0.278431  0.278431  0.
564706
 0.1998    0.390043  0.390043  0.479498     0.309804  0.309804  0.
611765
 0.1998    0.390043  0.390043  0.479498     0.309804  0.309804  0.
611765
 0.195753  0.420118  0.420118  0.513494  …  0.215686  0.215686  0.
54902
 0.195753  0.420118  0.420118  0.513494     0.215686  0.215686  0.
54902
 0.154471  0.40358   0.40358   0.512643     0.247059  0.247059  0.
596078
 ⋮                                       ⋱            ⋮
 0.207365  0.595753  0.595753  0.616686     0.104475  0.104475  0.
578102
 0.207365  0.595753  0.595753  0.616686     0.104475  0.104475  0.
578102
 0.129827  0.506451  0.506451  0.747843  …  0.112275  0.112275  0.
596259
 0.129827  0.506451  0.506451  0.747843     0.112275  0.112275  0.
596259
 0.179357  0.565443  0.565443  0.620455     0.101082  0.101082  0.
592059
 0.179357  0.565443  0.565443  0.620455     0.101082  0.101082  0.
592059
 0.22731   0.548306  0.548306  0.377039     0.105855  0.105855  0.
58511
 0.22731   0.548306  0.548306  0.377039  …  0.105855  0.105855  0.
58511
 0.13869   0.324455  0.324455  0.337949     0.141722  0.141722  0.
599345
 0.13869   0.324455  0.324455  0.337949     0.141722  0.141722  0.
599345
 0.192867  0.13869   0.13869   0.332298     0.142294  0.142294  0.
6152
 0.192867  0.13869   0.13869   0.332298     0.142294  0.142294  0.
6152
```

In [59]:
```
EMB170c == EMB170c1
```

Out[59]:
```
false
```

In [60]:
```
(typeof(EMB170c), typeof(EMB170c1))
```

Out[60]:
```
(Array{Gray{Normed{UInt8,8}},2}, Array{Float64,2})
```

In [61]:
```
gray(EMB170c[1,1])
```

Out[61]:
```
0.412N0f8
```

Filtro passa-baixas

In [62]:
```
hp = [1, 2, 1]
h = hp * hp'
h = h / sum(h)
```

Out[62]:
```
3×3 Array{Float64,2}:
 0.0625  0.125  0.0625
 0.125   0.25   0.125
 0.0625  0.125  0.0625
```

In [63]:
```
h[2,2]
```

Out[63]:
```
0.25
```

Julia permite que você crie vetores e matrizes com índices que começam em qualquer lugar - isto é muito útil para implementar filtros que não deslocam a imagem. Compare h com hc:

In [64]:
```
hc = centered(h)
```

Out[64]:
```
3×3 OffsetArray(::Array{Float64,2}, -1:1, -1:1) with eltype Float6
4 with indices -1:1×-1:1:
 0.0625  0.125  0.0625
 0.125   0.25   0.125
 0.0625  0.125  0.0625
```

In [65]:
```
hc[0,0]
```

Out[65]:
```
0.25
```

Note que hc tem índices nos intervalos -1:1, -1:1, e portanto não pode ser somado a h:

In [66]:
```
hc + h
```

```
DimensionMismatch("dimensions must match: a has dims (OffsetArrays
.IdOffsetRange(-1:1), OffsetArrays.IdOffsetRange(-1:1)), b has dim
s (Base.OneTo(3), Base.OneTo(3)), mismatch at 1")

Stacktrace:
 [1] promote_shape at ./indices.jl:178 [inlined]
 [2] promote_shape(::OffsetArray{Float64,2,Array{Float64,2}}, ::Ar
ray{Float64,2}) at ./indices.jl:169
 [3] +(::OffsetArray{Float64,2,Array{Float64,2}}, ::Array{Float64,
2}) at ./arraymath.jl:38
 [4] top-level scope at In[66]:1
 [5] include_string(::Function, ::Module, ::String, ::String) at .
/loading.jl:1091
```

Mas na verdade a função imfilter faz a mesma coisa com h ou hc (só que no caso de h, pode aparecer um warning)

In [67]:
```
EMBf = imfilter(EMB170, hc)
```

Out[67]:

In [68]:
```
imshow(EMB170)
```

Out[68]:
```
Dict{String,Any} with 4 entries:
  "gui"         => Dict{String,Any}("window"=>GtkWindowLeaf(name="
", parent, wi…
  "roi"         => Dict{String,Any}("redraw"=>96: "map(f-mapped im
age, input-26…
  "annotations" => 66: "input-26" = Dict{UInt64,Any}() Dict{UInt64
,Any}
  "clim"        => nothing
```

In [69]:
```
imshow(EMBf)
```

Out[69]:
```
Dict{String,Any} with 4 entries:
  "gui"         => Dict{String,Any}("window"=>GtkWindowLeaf(name="
", parent, wi…
  "roi"         => Dict{String,Any}("redraw"=>128: "map(f-mapped i
mage, input-3…
  "annotations" => 98: "input-38" = Dict{UInt64,Any}() Dict{UInt64
,Any}
  "clim"        => nothing
```

In [70]:
```
EMBfd = imfilter(EMB170, h)
```

Out[70]:

In [71]:
```
EMBf == EMBfd
```

Out[71]:
```
true
```

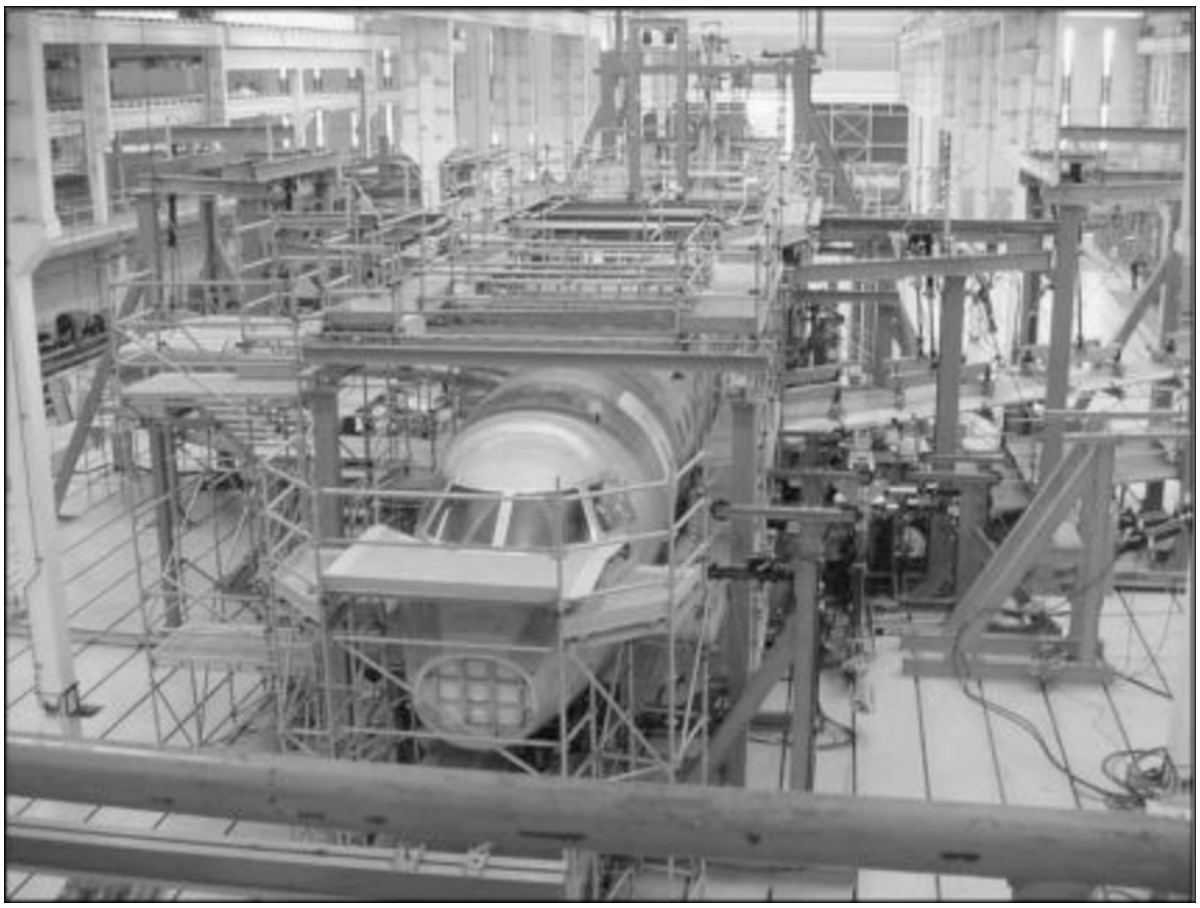
No entanto, o resultado da função de convolução (`conv`) dá um pouco diferente caso você esteja usando `hc` ou `h` :

In [72]:
```
EMBconvc = conv(hc, EMB170c1)
```

Out[72]:
```
642×854 OffsetArray(::Array{Float64,2}, 0:641, 0:853) with eltype
Float64 with indices 0:641×0:853:
 0.0256968  0.0715159  0.0860637  …  0.0992647  0.105392   0.04338
24
 0.0770904  0.214548   0.258191      0.297794   0.316176   0.13014
7
 0.0906125  0.259371   0.325051      0.374755   0.402696   0.16666
7
 0.066263   0.205987   0.286645      0.330147   0.364951   0.15294
1
 0.049277   0.174166   0.276114      0.320343   0.346569   0.14387
3
 0.0396544  0.16391    0.293459   …  0.345343   0.347549   0.13946
1
 0.0376752  0.165997   0.309616      0.355882   0.34902    0.13823
5
 0.0433395  0.180427   0.324584      0.351961   0.35098    0.14019
6
 0.0471162  0.190084   0.334672      0.358824   0.359804   0.14411
8
 0.0490054  0.194969   0.339879      0.376471   0.37549    0.15
 0.0496971  0.198785   0.347868   …  0.363725   0.369608   0.14902
 0.0491912  0.201532   0.35864       0.320588   0.342157   0.14117
6
 0.0463581  0.196712   0.358346      0.307843   0.336275   0.14019
6
 ⋮                                  ⋱
 0.0469951  0.237347   0.477066   …  0.225479   0.317927   0.14566
 0.0373029  0.2068     0.433885      0.230673   0.323441   0.14793
 0.0355525  0.201405   0.426452      0.23091    0.324974   0.14880
2
 0.0417436  0.221161   0.454765      0.226188   0.322525   0.14827
7
 0.0478363  0.235962   0.468705      0.224287   0.32073    0.14758
 0.0538304  0.245808   0.468273   …  0.225208   0.319589   0.14671
2
 0.0512887  0.225663   0.420546      0.233283   0.32304    0.14716
7
 0.0402113  0.175527   0.325525      0.248513   0.331082   0.14894
7
 0.0380586  0.145621   0.246569      0.257226   0.337121   0.15082
7
 0.0448306  0.135944   0.183679      0.259422   0.341156   0.15280
9
 0.0361625  0.0983294  0.114176   …  0.19539    0.25738    0.11535
 0.0120542  0.0327765  0.0380586     0.0651301  0.0857934  0.03845
```

In [73]:
```
Gray.(EMBconvc)
```

Out[73]:



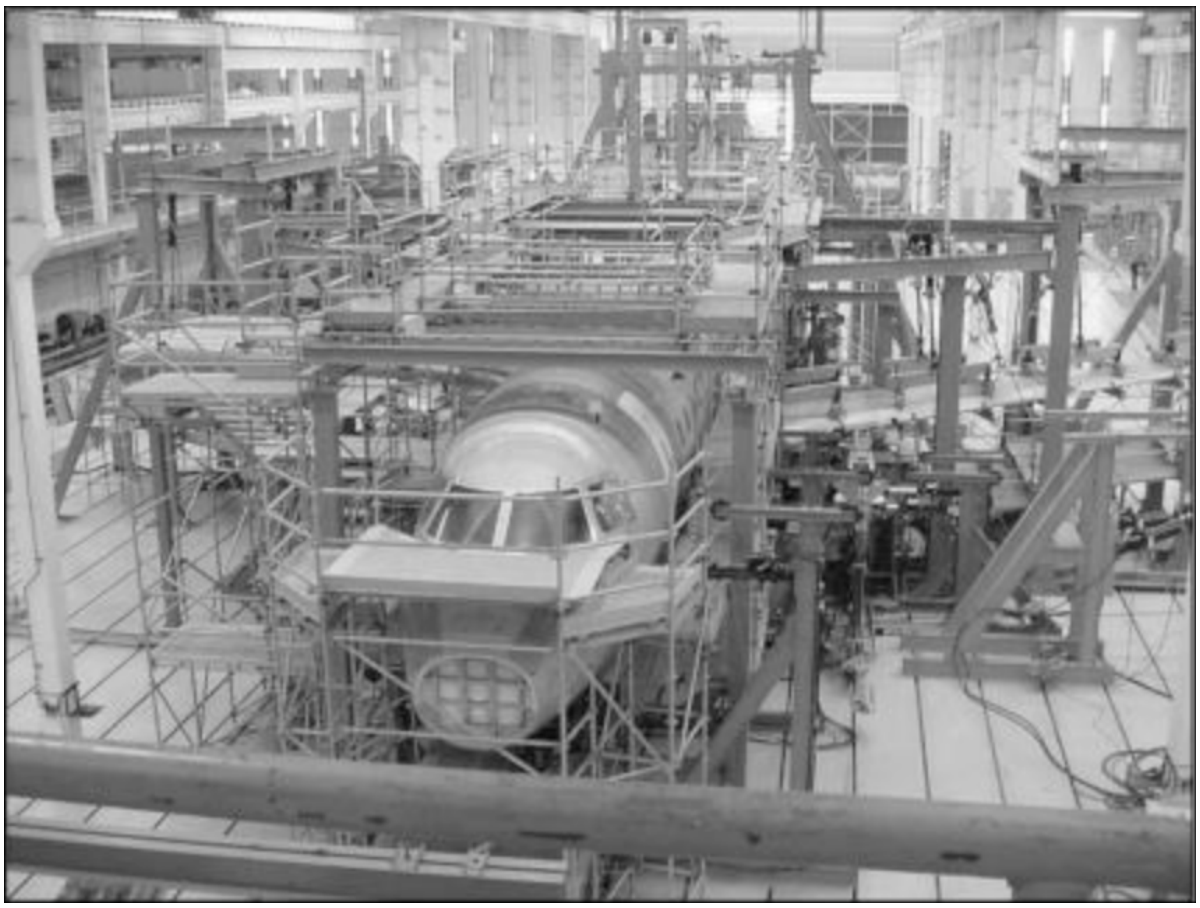
In [74]:
```
EMBconv = conv(h, EMB170c1)
```

Out[74]: 642×854 Array{Float64,2}:

```
 0.0256968  0.0715159  0.0860637  …  0.0992647  0.105392   0.04338
24
 0.0770904  0.214548   0.258191      0.297794   0.316176   0.13014
7
 0.0906125  0.259371   0.325051      0.374755   0.402696   0.16666
7
 0.066263   0.205987   0.286645      0.330147   0.364951   0.15294
1
 0.049277   0.174166   0.276114      0.320343   0.346569   0.14387
3
 0.0396544  0.16391    0.293459   …  0.345343   0.347549   0.13946
1
 0.0376752  0.165997   0.309616      0.355882   0.34902    0.13823
5
 0.0433395  0.180427   0.324584      0.351961   0.35098    0.14019
6
 0.0471162  0.190084   0.334672      0.358824   0.359804   0.14411
8
 0.0490054  0.194969   0.339879      0.376471   0.37549    0.15
 0.0496971  0.198785   0.347868   …  0.363725   0.369608   0.14902
 0.0491912  0.201532   0.35864       0.320588   0.342157   0.14117
6
 0.0463581  0.196712   0.358346      0.307843   0.336275   0.14019
6
 ⋮                                  ⋱
 0.0469951  0.237347   0.477066   …  0.225479   0.317927   0.14566
 0.0373029  0.2068     0.433885      0.230673   0.323441   0.14793
 0.0355525  0.201405   0.426452      0.23091    0.324974   0.14880
2
 0.0417436  0.221161   0.454765      0.226188   0.322525   0.14827
7
 0.0478363  0.235962   0.468705      0.224287   0.32073    0.14758
 0.0538304  0.245808   0.468273   …  0.225208   0.319589   0.14671
2
 0.0512887  0.225663   0.420546      0.233283   0.32304    0.14716
7
 0.0402113  0.175527   0.325525      0.248513   0.331082   0.14894
7
 0.0380586  0.145621   0.246569      0.257226   0.337121   0.15082
7
 0.0448306  0.135944   0.183679      0.259422   0.341156   0.15280
9
 0.0361625  0.0983294  0.114176   …  0.19539    0.25738    0.11535
 0.0120542  0.0327765  0.0380586     0.0651301  0.0857934  0.03845
```

In [75]:
```
Gray.(EMBconv)
```

Out[75]:



No caso de um filtro separável, é mais rápido usar um algoritmo que processa cada parcela do filtro de cada vez:

In [76]:
```
hfc = centered([1, 2, 1])
hfc = hfc / sum(hfc)
```

Out[76]:
```
3-element OffsetArray(::Array{Float64,1}, -1:1) with eltype Float6
4 with indices -1:1:
 0.25
 0.5
 0.25
```

Se a resposta ao impulso do filtro for separável, podemos usar esse fato para implementar o filtro de forma mais eficiente:

In [77]:
```
hf = kernelfactors((hfc, hfc))  # Repare que os parênteses devem se
r duplos
```

Out[77]:
```
(ImageFiltering.KernelFactors.ReshapedOneD{Float64,2,0,OffsetArray
{Float64,1,Array{Float64,1}}}([0.25, 0.5, 0.25]), ImageFiltering.K
ernelFactors.ReshapedOneD{Float64,2,1,OffsetArray{Float64,1,Array{
Float64,1}}}([0.25, 0.5, 0.25]))
```

```
In [78]: EMBf1 = imfilter(EMB170, hf)
```

Out[78]:

Transformada de Fourier de duas dimensões

Ao calcular a TDF de um filtro de duas dimensões, mesmo que o filtro seja centrado em zero, a TDF vai resultar em uma fase (porque a função fft sempre considera os índices indo de 0 a M-1):

```
In [79]: fft(hc)
```

```
Out[79]: 3×3 Array{Complex{Float64},2}:
             1.0+0.0im          -0.125+0.216506im        -0.125-0.216506im
          -0.125+0.216506im  -0.03125-0.0541266im        0.0625-6.93889e-18im
          -0.125-0.216506im     0.0625+6.93889e-18im  -0.03125+0.0541266im
```

```
In [80]: fft(h)
```

```
Out[80]: 3×3 Array{Complex{Float64},2}:
             1.0+0.0im          -0.125-0.216506im        -0.125+0.216506im
          -0.125-0.216506im  -0.03125+0.0541266im        0.0625+6.93889e-18im
          -0.125+0.216506im     0.0625-6.93889e-18im  -0.03125-0.0541266im
```

Para obter a transformada da sequência centrada, você pode deslocar circularmente os elementos de h, assim:

In [81]:
```
h[1,2:end]'
```

Out[81]:
```
1×2 LinearAlgebra.Adjoint{Float64,Array{Float64,1}}:
 0.125  0.0625
```

In [82]:
```
hdesl = [h[2:end,2:end] h[2:end,1];
         h[1,2:end]' h[1,1]]
```

Out[82]:
```
3×3 Array{Float64,2}:
 0.25   0.125   0.125
 0.125  0.0625  0.0625
 0.125  0.0625  0.0625
```

In [83]:
```
fft(hdesl)
```

Out[83]:
```
3×3 Array{Complex{Float64},2}:
   1.0+0.0im     0.25+0.0im     0.25+0.0im
 0.25+0.0im   0.0625+0.0im   0.0625+0.0im
 0.25+0.0im   0.0625+0.0im   0.0625+0.0im
```

Em Julia é possível usar os pacotes `FFTViews` e `OffsetArrays` para fazer isso de maneira mais fácil:

In [84]:
```
using FFTViews
```

In [85]:
```
h1 = [1, 2, 1]
```

Out[85]:
```
3-element Array{Int64,1}:
 1
 2
 1
```

In [86]:
```
H1 = fft(h1)
```

Out[86]:
```
3-element Array{Complex{Float64},1}:
  4.0 + 0.0im
 -0.5 - 0.8660254037844386im
 -0.5 + 0.8660254037844386im
```

In [87]:
```
H1[1]
```

Out[87]:
```
4.0 + 0.0im
```

In [88]:
```
H1v = FFTView(H1)
```

Out[88]:
```
3-element FFTView{Complex{Float64},1,Array{Complex{Float64},1}} wi
th indices FFTViews.URange(0,2):
  4.0 + 0.0im
 -0.5 - 0.8660254037844386im
 -0.5 + 0.8660254037844386im
```

```
In [89]: H1v[3]
```

```
Out[89]: 4.0 + 0.0im
```

```
In [90]: h̄c = zeros(size(h))
```

```
Out[90]: 3×3 Array{Float64,2}:
          0.0  0.0  0.0
          0.0  0.0  0.0
          0.0  0.0  0.0
```

```
In [91]: hv = FFTView(h̄c)
```

```
Out[91]: 3×3 FFTView{Float64,2,Array{Float64,2}} with indices FFTViews.URan
         ge(0,2)×FFTViews.URange(0,2):
          0.0  0.0  0.0
          0.0  0.0  0.0
          0.0  0.0  0.0
```

hv é um *view* de $\bar{h}c$, que interpreta os índices de maneira periódica, como a FFT faz. Como é um *view* (essencialmente um ponteiro), mudar os elementos de hv altera também os elementos de $\bar{h}c$. Podemos fazer isso imaginando o deslocamento circular necessário para calcular a FFT com fase nula:

```
In [92]: hv[-1:1,-1:1]=h
```

```
Out[92]: 3×3 Array{Float64,2}:
          0.0625  0.125  0.0625
          0.125   0.25   0.125
          0.0625  0.125  0.0625
```

Repare como $\bar{h}c$ mudou:

```
In [93]: h̄c
```

```
Out[93]: 3×3 Array{Float64,2}:
          0.25   0.125   0.125
          0.125  0.0625  0.0625
          0.125  0.0625  0.0625
```

hv apenas fornece os índices de 0 a 2:

```
In [94]: hv
```

```
Out[94]: 3×3 FFTView{Float64,2,Array{Float64,2}} with indices FFTViews.URan
         ge(0,2)×FFTViews.URange(0,2):
          0.25   0.125   0.125
          0.125  0.0625  0.0625
          0.125  0.0625  0.0625
```

```
In [95]: hv[-3:3,-3:3]
```

```
Out[95]: 7×7 Array{Float64,2}:
 0.25   0.125   0.125   0.25   0.125   0.125   0.25
 0.125  0.0625  0.0625  0.125  0.0625  0.0625  0.125
 0.125  0.0625  0.0625  0.125  0.0625  0.0625  0.125
 0.25   0.125   0.125   0.25   0.125   0.125   0.25
 0.125  0.0625  0.0625  0.125  0.0625  0.0625  0.125
 0.125  0.0625  0.0625  0.125  0.0625  0.0625  0.125
 0.25   0.125   0.125   0.25   0.125   0.125   0.25
```

Agora $\bar{hc}$ está preparada para a FFT de fase nula:

```
In [96]: Hc = fft(h̄c)
```

```
Out[96]: 3×3 Array{Complex{Float64},2}:
   1.0+0.0im    0.25+0.0im    0.25+0.0im
 0.25+0.0im  0.0625+0.0im  0.0625+0.0im
 0.25+0.0im  0.0625+0.0im  0.0625+0.0im
```

```
In [97]: Hc = real.(Hc)
```

```
Out[97]: 3×3 Array{Float64,2}:
 1.0   0.25    0.25
 0.25  0.0625  0.0625
 0.25  0.0625  0.0625
```

```
In [98]: (Mx, My)=size(h)
```

```
Out[98]: (3, 3)
```

Uma outra opção é usar a função `fftshift` e `ifftshift` (que existem também no Matlab):

```
In [109]: hc2 = ifftshift(h)
```

```
Out[109]: 3×3 Array{Float64,2}:
 0.25   0.125   0.125
 0.125  0.0625  0.0625
 0.125  0.0625  0.0625
```

A transformada virá com todos os valores reais, como queríamos, mas com o frequência DC (`[0,0]`) na posição `[1,1]` :

```
In [110]: Hc2 = fft(hc2)
```

```
Out[110]: 3×3 Array{Complex{Float64},2}:
   1.0+0.0im    0.25+0.0im    0.25+0.0im
 0.25+0.0im  0.0625+0.0im  0.0625+0.0im
 0.25+0.0im  0.0625+0.0im  0.0625+0.0im
```

Para colocar o DC no centro, use `fftshift` :

In [112]:
```
fftshift(Hc2)
```

Out[112]:
```
3×3 Array{Complex{Float64},2}:
 0.0625+0.0im  0.25+0.0im  0.0625+0.0im
   0.25+0.0im   1.0+0.0im    0.25+0.0im
 0.0625+0.0im  0.25+0.0im  0.0625+0.0im
```

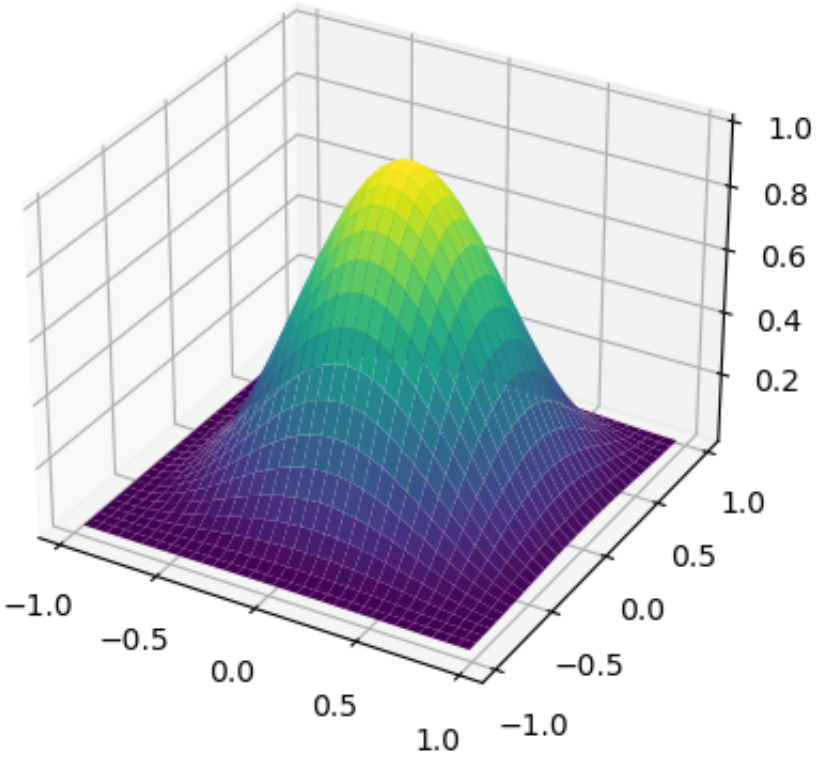
A função `pfft2` faz *zero-padding* (calcula a transformada com mais pontos, completando com zeros), e já centra as transformadas

In [99]:
```
Hresp,ωx,ωy = pfft2(h, 32, 32)
maximum(abs.(imag.(Hresp)))
```

Out[99]:
```
7.954818481105237e-17
```

In [100]:
```
Hresp = real.(Hresp);
```

In [101]:
```
surf(ωx/π, ωy/π, Hresp, cmap = "viridis");
```



Filtro passa-altas

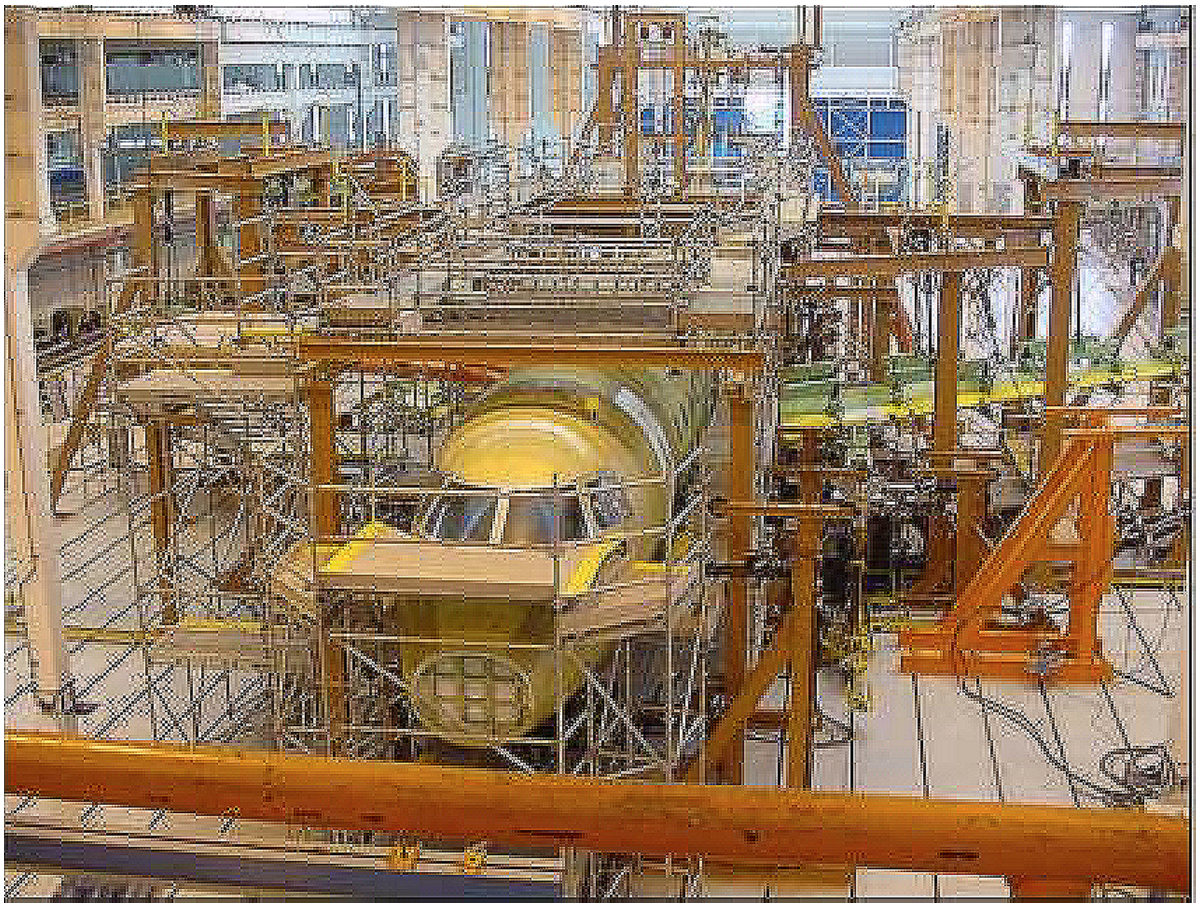
In [ ]:
```

```

In [102]:
```
hpa = [-0.75, 2, -0.75]
hpa = hpa / sum(hpa)
hpaf = kernelfactors((hpa, hpa))
```

Out[102]:
```
(ImageFiltering.KernelFactors.ReshapedOneD{Float64,2,0,Array{Float
64,1}}([-1.5, 4.0, -1.5]), ImageFiltering.KernelFactors.ReshapedOn
eD{Float64,2,1,Array{Float64,1}}([-1.5, 4.0, -1.5]))
```

In [103]:
```
EMBpa = imfilter(EMB170, hpaf)
```

Out[103]:

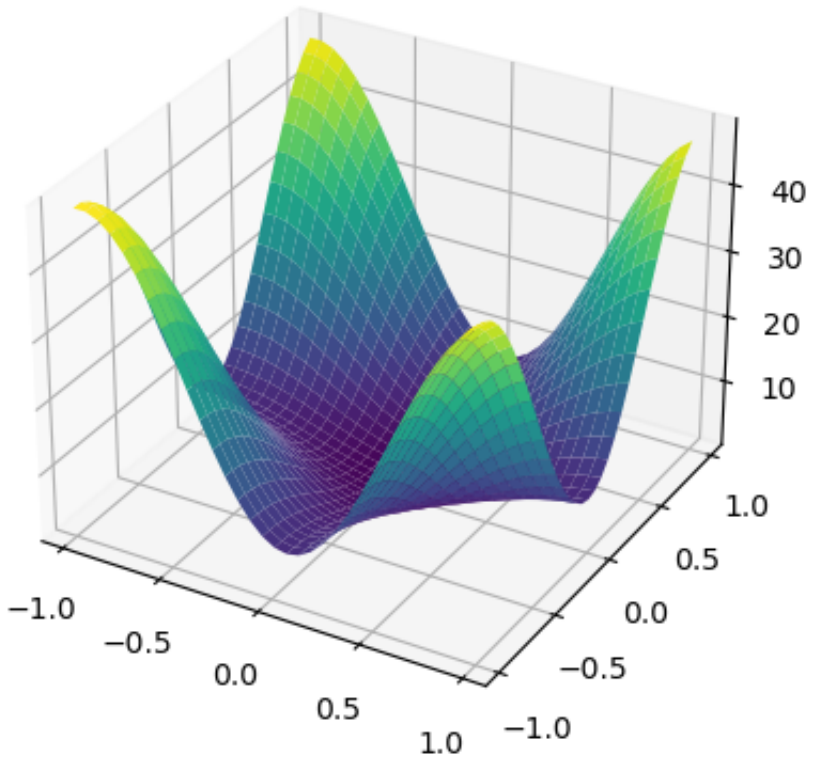


In [104]:
```
Hpa,ωx,ωy = pfft2(hpa*hpa', 32, 32)
maximum(abs.(imag.(Hpa)))
```

Out[104]:
```
3.195733643360402e-15
```

In [105]:
```
Hpa = real.(Hpa);
```

In [106]:
```
surf(ωx/π, ωy/π, Hpa, cmap = "viridis");
```



In [ ]: