Usage Examples

July 29, 2019

In [1]:
```
#using Revise
using BDDSolver, LinearAlgebra, Random
Random.seed!(1);
```

```
Info: Recompiling stale cache file /Users/spielman/.julia/compiled/v1.1/BDDSolver/swp1u.ji fo
@ Base loading.jl:1184
```

1 A First Example - a Laplacian

The BDDSolver is for solving systems of linear equations in block matrices. These must be non-singular and block diagonally dominant. To begin testing, let's generate one by casting a SDD (Laplacian plus diagonal) in this form, and then solve the resulting system.

In [2]:
```
n = 10
la = lap(wted_chimera(n,1))
la[1,1] += 1
```

Out[2]:
```
16.03555232314551
```

So, `la` is the matrix in which we want to solve a system. Convert it into a block matrix.

In [3]:
```
B = BlockCSC(la)
```

Out[3]:
```
10×10 BlockCSC{1,Float64,1}:
 [16.0356]     [-0.430149]   [-3.63431]     [0.0]         [-2.65027]
 [-0.430149]   [12.0048]     [-3.20416]      [-1.34011]    [0.0]
 [-3.63431]    [-3.20416]    [16.1088]       [-1.86405]    [0.0]
 [-2.80671]    [-2.37656]    [-0.827597]     [-1.03645]    [0.0]
 [0.0]         [0.0]         [-2.9598]       [-1.09575]    [-1.97577]
 [0.0]         [0.0]         [-3.61885]     [-1.7548]     [-2.63481]
 [-3.4796]     [-3.04945]    [0.0]           [0.0]         [-0.829328]
 [-2.03451]    [-1.60437]    [0.0]           [0.0]         [-0.615758]
 [0.0]         [-1.34011]    [-1.86405]      [7.97118]     [-0.880011]
 [-2.65027]    [0.0]         [0.0]           [-0.880011]   [9.58595]
```

The following code constructs the solver for the linear system.

```
In [4]: sol = approxCholBDD(B, verbose=true)
```

```
Time to construct mat to analyze 5.0067901611328125e-6
Time to compute elimination structure 2.9087066650390625e-5
Time to prepare for elimination 3.0994415283203125e-6
Time to eliminate 9.5367431640625e-7
Total solver build time: 0.10144495964050293
Ratio of operator edges to original edges: 1.3142857142857143
```

```
Out[4]: (::getfield(BDDSolver, Symbol("#f1#12")){BDDSolver.PCG_params,getfield(BDDSolver, Symbo
```

```
In [5]: b = randn(n)
        x = sol(b)
```

```
PCG stopped after: 0.0 seconds and 7 iterations with relative error 3.900346121414374e-8.
```

```
Out[5]: 10-element Array{Float64,1}:
         -3.3359467843576476
         -3.529550419132262
         -3.516582448067082
         -3.526822294753162
         -3.5460464366210562
         -3.5874747784108196
         -3.483910627837018
         -3.322269351611948
         -3.627431834723096
         -3.4216386027745203
```

```
In [6]: norm(la*x - b)
```

```
Out[6]: 9.83622291203584e-8
```

2 An SDD matrix - with both positive and negative off-diagonals

```
In [7]: n = 20
        a_tri = triu(rand_regular(n,3))
        a_tri.nzval .*= rand(-1:2:1, length(a_tri.nzval))
        a = a_tri + a_tri'
```

```
Out[7]: 20Œ20 SparseMatrixCSC{Float64,Int64} with 56 stored entries:
          [5 ,  1]  =  1.0
          [10,  1]  =  -1.0
          [15,  1]  =  -1.0
          [13,  2]  =  1.0
          [14,  2]  =  -1.0
          [20,  2]  =  1.0
          [4 ,  3]  =  1.0
```

```
 [7 ,  3]  =  -1.0
 [19,  3]  =  1.0
 [3 ,  4]  =  1.0
 [7 ,  4]  =  -1.0
 [10,  4]  =  1.0
  
 [12, 16]  =  -1.0
 [6 , 17]  =  -2.0
 [19, 17]  =  1.0
 [5 , 18]  =  -1.0
 [11, 18]  =  -1.0
 [19, 18]  =  1.0
 [3 , 19]  =  1.0
 [17, 19]  =  1.0
 [18, 19]  =  1.0
 [2 , 20]  =  1.0
 [8 , 20]  =  -1.0
 [11, 20]  =  -1.0
```

In [8]:
```
sdd = 3.1*I - a
B = BlockCSC(sdd)
sol = approxCholBDD(B)
```

Out[8]:
```
(::getfield(BDDSolver, Symbol("#f1#12")){BDDSolver.PCG_params,getfield(BDDSolver, Symbo
```

In [9]:
```
b = randn(n)
x = sol(b)
norm(sdd*x - b)
```

Out[9]:
```
3.687053385990559e-7
```

3 Block Matrices and Vectors

We represent block matrices as sparse matrices of matrices of fixed size, using Static Arrays to represent the latter. We can cast an ordinary vector to a `BlockVec` as follows

In [10]:
```
x = randn(6)
xb = BlockVec(2,x)
```

Out[10]:
```
3-element reinterpret(StaticArrays.SArray{Tuple{2},Float64,1,2}, ::Array{Float64,1}):
 [-0.557151, -0.139962]
 [-0.254902, -1.27133]
 [0.754751, 1.04847]
```

Note that the result is just a view of `x`, which means that changes to `x` also change `xb`

In [11]:
```
x
```

```
Out[11]: 6-element Array{Float64,1}:
          -0.55715149300881
          -0.13996160847232697
          -0.2549015022672395
          -1.271334809102429
           0.7547507621437772
           1.0484665971648446
```

```
In [12]: x[1] = 0
         x
```

```
Out[12]: 6-element Array{Float64,1}:
           0.0
          -0.13996160847232697
          -0.2549015022672395
          -1.271334809102429
           0.7547507621437772
           1.0484665971648446
```

```
In [13]: xb
```

```
Out[13]: 3-element reinterpret(StaticArrays.SArray{Tuple{2},Float64,1,2}, ::Array{Float64,1}):
          [0.0, -0.139962]
          [-0.254902, -1.27133]
          [0.754751, 1.04847]
```

If you want to avoid, this you should copy the vector.

```
In [14]: xb = copy(BlockVec(2, x))
         x[1] = 1
         xb
```

```
Out[14]: 3-element Array{StaticArrays.SArray{Tuple{2},Float64,1,2},1}:
          [0.0, -0.139962]
          [-0.254902, -1.27133]
          [0.754751, 1.04847]
```

We can convert the answer back to an ordinary vector with `FlatVec`. You will note that this is also just a view. So, if you want to avoid the resulting complications, copy it.

```
In [15]: FlatVec(xb)
```

```
Out[15]: 6-element reinterpret(Float64, ::Array{StaticArrays.SArray{Tuple{2},Float64,1,2},1}):
           0.0
          -0.13996160847232697
          -0.2549015022672395
          -1.271334809102429
           0.7547507621437772
           1.0484665971648446
```

In [16]: copy(FlatVec(xb))

```
Out[16]: 6-element Array{Float64,1}:
  0.0
 -0.13996160847232697
 -0.2549015022672395
 -1.271334809102429
  0.7547507621437772
  1.0484665971648446
```

3.1 Constructing a block matrix

The code `randOrthWt` will replace an ordinary sparse matrix with one in which every block has been replaced by a random orthogonal matrix, and fixes the diagonals so the result is a connection Laplacian. Let's do it with 2-by-2 matrices.

```
In [17]: Random.seed!(1)
         a = rand_regular(10,3)
         a[:,1]
```

```
Out[17]: 10-element SparseVector{Float64,Int64} with 3 stored entries:
  [2 ]  =  1.0
  [3 ]  =  1.0
  [9 ]  =  1.0
```

```
In [18]: B = randOrthWt(a,2)
         B[:,1]
```

```
Out[18]: 10-element SparseVector{StaticArrays.SArray{Tuple{2,2},Float64,2,4},Int64} with 4 sto
  [1 ]  =  [3.0 0.0; 0.0 3.0]
  [2 ]  =  [-0.813483 -0.581588; -0.581588 0.813483]
  [3 ]  =  [-0.987632 -0.156788; 0.156788 -0.987632]
  [9 ]  =  [0.770584 0.637338; 0.637338 -0.770584]
```

We can recover the matrix with `sparse`. Let's check its eigenvalues to see if it is singular.

```
In [19]: sB = sparse(B)
```

```
Out[19]: 20E20 SparseMatrixCSC{Float64,Int64} with 160 stored entries:
  [1 ,  1]  =  3.0
  [2 ,  1]  =  0.0
  [3 ,  1]  =  -0.813483
  [4 ,  1]  =  -0.581588
  [5 ,  1]  =  -0.987632
  [6 ,  1]  =  0.156788
  [17,  1]  =  0.770584
  [18,  1]  =  0.637338
  [1 ,  2]  =  0.0
  [2 ,  2]  =  3.0
```

```
  [3 ,  2]  =  -0.581588
  [4 ,  2]  =  0.813483

  [15, 19]  =  0.201878
  [16, 19]  =  -0.979411
  [19, 19]  =  3.0
  [20, 19]  =  0.0
  [3 , 20]  =  -0.355274
  [4 , 20]  =  -0.934762
  [13, 20]  =  -0.984712
  [14, 20]  =  -0.174191
  [15, 20]  =  -0.979411
  [16, 20]  =  -0.201878
  [19, 20]  =  0.0
  [20, 20]  =  3.0
```

```
In [20]: eigvals(Matrix(sB))
```

```
Out[20]: 20-element Array{Float64,1}:
  0.3796506426946875
  0.4356062953226407
  0.9666889031643245
  1.0627240613063844
  1.2247771097913502
  1.3120592060295233
  1.972734487306134
  1.999954012872599
  2.430054323940155
  2.6517500302207546
  3.09818350049866
  3.5961656086619134
  3.9264446348749975
  4.319403051352489
  4.61516933373107
  4.8441603726186555
  4.993075596872082
  5.170596748822682
  5.443432319568885
  5.557369760350015
```

It is not singular, so let's try to solve the linear system.

```
In [21]: sol = approxCholBDD(B, verbose=true)
```

```
Time to construct mat to analyze 7.867813110351562e-6
Time to compute elimination structure 1.1920928955078125e-5
Time to prepare for elimination 1.9073486328125e-6
Time to eliminate 3.0994415283203125e-6
Total solver build time: 0.00030684471130371094
```

```
Ratio of operator edges to original edges: 1.65
```

```
Out[21]: (::getfield(BDDSolver, Symbol("#f1#12")){BDDSolver.PCG_params,getfield(BDDSolver, Syml
```

We can solve either with BlockVecs, or ordinary vecs.

```
In [22]: b = randn(20)
         x = sol(b)
```

```
PCG stopped after: 0.0 seconds and 9 iterations with relative error 8.230187662861582e-8.
```

```
Out[22]: 20-element Array{Float64,1}:
          -2.4028579477129344
           0.8224372934294162
          -1.250522521802219
          -1.6314853367104822
          -0.898432169913525
           0.25433062729447226
          -0.08492998940612545
          -0.6934734395229742
           0.19850029055403615
           0.6889350180873219
           1.5258385708676305
          -0.4358129454339404
          -2.3321021748724675
          -0.9016260320518082
          -1.7727272740489426
           0.19698618503819953
           2.227398266725205
           1.7900151429804345
          -0.08540385270388243
          -2.0914512161973726
```

```
In [23]: norm(sB*x - b)
```

```
Out[23]: 3.5513116711894246e-7
```

```
In [24]: bb = BlockVec(2, b)
```

```
Out[24]: 10-element reinterpret(StaticArrays.SArray{Tuple{2},Float64,1,2}, ::Array{Float64,1})
          [-1.458, 1.79734]
          [-0.906487, -0.97022]
          [-0.768929, -0.395993]
          [0.475183, 0.638632]
          [-0.398748, -0.0803466]
          [1.00184, -0.704845]
          [-1.43405, 0.434751]
          [-0.0922058, -0.464083]
          [2.18783, 0.579141]
          [-0.914724, -0.15506]
```

```
In [25]: xb = sol(bb)
```

```
PCG stopped after: 0.0 seconds and 9 iterations with relative error 8.230187662861582e-8.
```

```
Out[25]: 10-element Array{StaticArrays.SArray{Tuple{2},Float64,1,2},1}:
          [-2.40286, 0.822437]
          [-1.25052, -1.63149]
          [-0.898432, 0.254331]
          [-0.08493, -0.693473]
          [0.1985, 0.688935]
          [1.52584, -0.435813]
          [-2.3321, -0.901626]
          [-1.77273, 0.196986]
          [2.2274, 1.79002]
          [-0.0854039, -2.09145]
```

Note that we can multiply BlockCSC matrices by BlockVecs

```
In [26]: B*xb
```

```
Out[26]: 10-element Array{StaticArrays.SArray{Tuple{2},Float64,1,2},1}:
          [-1.458, 1.79734]
          [-0.906487, -0.97022]
          [-0.768929, -0.395993]
          [0.475183, 0.638632]
          [-0.398748, -0.0803466]
          [1.00184, -0.704845]
          [-1.43405, 0.434751]
          [-0.0922056, -0.464083]
          [2.18783, 0.579141]
          [-0.914724, -0.15506]
```

```
In [27]: B*xb - bb
```

```
Out[27]: 10-element Array{StaticArrays.SArray{Tuple{2},Float64,1,2},1}:
          [5.74319e-8, 1.93786e-8]
          [7.27519e-8, -5.61213e-8]
          [-5.40515e-8, -4.54478e-8]
          [7.17976e-8, 2.28316e-8]
          [-2.60333e-8, 3.60279e-8]
          [-1.3279e-8, 9.34246e-9]
          [-1.70287e-7, -1.70453e-7]
          [1.93287e-7, -6.80881e-8]
          [-2.89989e-8, -7.67631e-9]
          [1.21243e-8, 2.05526e-9]
```

```
In [ ]:
```