```
true
```

```
begin
    using ExtendableSparse
    using SparseArrays
    using ForwardDiff
    using SparseDiffTools
    using SparsityDetection
    using Symbolics
    using VoronoiFVM
    using PlutoUI
    using PyPlot
    PyPlot.svg(true);
end
```

Table of Contents


```
TableOfContents()
```

Ways to calculate sparse Jacobians

We investigate different ways to assemble sparse Jacobians for a finite difference operator $A_h(u)$ discretizing the differential operator

$$A(u) = -\Delta u^2 + u^2 - 1$$

in $\Omega = (0,1)$ with homogeneous Neumann boundary conditions on an equidistant grid of n points. We only discuss approaches which can be generalized to higher space dimensions and unstructured grids, so e.g. banded and tridiagonal matrix structures are not discussed.

This is the nonlinear finite difference operator:

```
function A_h!(y,u)
    n=length(u)
    h=1/(n-1)
    y[1]=(u[1]^2-1)*0.5*h
    for i=2:n-1
        y[i]=(u[i]^2-1.0)*h
    end
    y[end]=(u[end]^2-1)*0.5*h

    for i=1:n-1
        du=(u[i+1]^2-u[i]^2)/h
        y[i+1]+=du
        y[i]-=du
    end
end;
```

Number of disceretization points for testing:

```
n_test=5;
```

The methods

Straightforward Jacobian calculation

Just take the operator and calculate the Jacobian as a sparse matrix. Here we use the ExtendableSparse.jl package in order to avoid cumbersome handling of intermediate coordinate format data.

As we will see in the timing test, the complexity of this operation is $O(n^2)$

```
function straightforward(n)
    X=ones(n)
    Y=ones(n)
    jac = ExtendableSparseMatrix(n, n)
    t=@elapsed begin
        ForwardDiff.jacobian!(jac, A_h!, X, Y)
        flush!(jac)
    end
    jac.cscmatrix,t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.992594)
  8.25  -8.0    ⋅     ⋅     ⋅
 -8.0   16.5  -8.0    ⋅     ⋅
   ⋅    -8.0  16.5  -8.0    ⋅
   ⋅      ⋅   -8.0  16.5  -8.0
   ⋅      ⋅     ⋅   -8.0   8.25
```

```
straightforward(n_test)
```

Symbolics.jl for sparsity detection

While in the particular case we the sparsity pattern of the operator – it is tridiagonal – we assume it is not known and we use Symbolics.jl to detect the sparsity pattern and convey the sparsity information to ForwardDiff.

For this purpose we use the multiple dispatch feature of Julia and call `A_h!` with symbolic data, resulting in a symbolic representation of the operator which can be investigated for sparsity in $O(n)$ time.

```
function Symbolics.jacobian_sparsity(op!,output::Array{T},input::Array{T})
where T<:Number
    eqs=similar(output,Num)
    vars=[Symbolics.variable(i) for i in eachindex(input)]
    op!(eqs,vars)
    Symbolics.jacobian_sparsity(eqs,vars)
end
```

```
function symbolics(n)
    input = ones(n)
    output = similar(input)
    tdetect=@elapsed begin
        sparsity_pattern=Symbolics.jacobian_sparsity(A_h!,output,input)
        jac=Float64.(sparsity_pattern)
        colors = matrix_colors(jac)
    end
    tassemble=@elapsed begin
        forwarddiff_color_jacobian!(jac, A_h!, input, colorvec = colors)
    end
    jac,tdetect,tassemble
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.000492987, 4.0342e-
  8.25  -8.0    ⋅     ⋅     ⋅
 -8.0   16.5  -8.0    ⋅     ⋅
   ⋅    -8.0  16.5  -8.0    ⋅
   ⋅      ⋅   -8.0  16.5  -8.0
   ⋅      ⋅     ⋅   -8.0   8.25
```

```
symbolics(n_test)
```

SparsityDetection.jl for sparsity detection

Before the advent of Symbolics.jl, SparsityDetection.jl (now deprecated) provided a method for sparsity detection of nonlinear operators in $O(n)$ time which we test here for comprehensiveness.

```
function sparsitydetect(n)
    input = rand(n)
    output = similar(input)
    tdetect=@elapsed begin
        sparsity_pattern = jacobian_sparsity(A_h!,output,input)
        jac = Float64.(sparsity_pattern)
        colors = matrix_colors(jac)
    end
    u=ones(n)
    tassemble=@elapsed begin
        forwarddiff_color_jacobian!(jac, A_h!, u, colorvec = colors)
    end
    jac,tdetect,tassemble
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.000837247, 4.2508e-
  8.25  -8.0    ⋅     ⋅     ⋅
 -8.0   16.5  -8.0    ⋅     ⋅
   ⋅    -8.0  16.5  -8.0    ⋅
   ⋅      ⋅   -8.0  16.5  -8.0
   ⋅      ⋅     ⋅   -8.0   8.25
```

```
sparsitydetect(n_test)
```

Assembly from local Jacobians

Usually, for finite difference and finite volume methods, the sparsity structure is defined by the grid topology, and the operator is essentially a superposition of local contributions translated in space. So we can assemble the global Jacobi matrix from local contributions, very much like in classical finite element asembly. The local Jacobi matrix can be obtained without the need to detect its sparsity (though for larger coupled systems of PDEs this possibility appears to be worth to be investigated)

For this purpose, we define the local contributions as mutating Julia functions, ready to be generalized for systems of PDEs.

```
flux!(f,u) = f[1]=u[1]^2-u[2]^2;
```

```
reaction!(f,u) = f[1]=u[1]^2-1.0;
```

These can be passed to a function which calculates the operator application and assembles the Jacobi matrix at once as it is needed in Newton's method. It is important to ensure that there are no allocations in the inner loop.

```
function A_Jac_h!(y,m,u,flux,reaction)
    n=length(u)
    h=1/(n-1)

    Y=zeros(1)
    UK=zeros(1)
    UKL=zeros(2)

    # Provide space for results
    result_rea  = DiffResults.DiffResult(zeros(1), zeros(1,1))
    result_flx  = DiffResults.DiffResult(zeros(1), zeros(1,2))

    # Fix the Jacobian configuration for ForwardDiff
    cfg_rea     = ForwardDiff.JacobianConfig(reaction, Y, UK,
ForwardDiff.Chunk(UK,1))
    cfg_flx   = ForwardDiff.JacobianConfig(flux, Y,
UKL,ForwardDiff.Chunk(UKL,1))


    for i=1:n
        fac=h
        if i==1||i==n
            fac=0.5*h
        end
        UK[1]=u[i]
        # we could use ForwardDiff.jacobian! here, but that is not allocation
free

        ForwardDiff.vector_mode_jacobian!(result_rea,reaction,Y,UK,cfg_rea)
        y[i]=DiffResults.value(result_rea)[1]*fac
        m[i,i] += DiffResults.jacobian(result_rea)[1,1]*fac
    end

    for i=1:n-1
        UKL[1]=u[i]
        UKL[2]=u[i+1]
        ForwardDiff.vector_mode_jacobian!(result_flx,flux,Y,UKL,cfg_flx)
        du=DiffResults.value(result_flx)
        ddu=DiffResults.jacobian(result_flx)
        y[i]+=du[1]/h
        y[i+1]-=du[1]/h
        m[i,i]+=ddu[1,1]/h
        m[i,i+1]+=ddu[1,2]/h
        m[i+1,i]-=ddu[1,1]/h
        m[i+1,i+1]-=ddu[1,2]/h
    end
end;
```

For the allocations in `ForwardDiff.jacobian!`, see **ForwardDiff.jl#516**.

```
function assemblefromlocal(n)
    u=ones(n)
    jac = ExtendableSparseMatrix(n,n);
    y=ones(n)
    t=@elapsed begin
        A_Jac_h!(y,jac,u,flux!,reaction!)
        flush!(jac)
    end
    jac.cscmatrix,t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.543577)
  8.25  -8.0    ⋅     ⋅     ⋅
 -8.0   16.5  -8.0    ⋅     ⋅
   ⋅    -8.0  16.5  -8.0    ⋅
   ⋅      ⋅   -8.0  16.5  -8.0
   ⋅      ⋅     ⋅   -8.0   8.25
```

```
assemblefromlocal(n_test)
```

Assembly from local: VoronoiFVM.jl

The previous approach is at the core of VoronoiFVM.jl, which works for PDE systems on unstructured grids in 1/2/3D.

```
vfvm_flux! (generic function with 1 method)
```

```
vfvm_flux!(f,u,edge)=f[1]=u[1,1]^2-u[1,2]^2
```

```
vfvm_reaction! (generic function with 1 method)
```

```
vfvm_reaction!(f,u,node)=f[1]=u[1]^2-1.0
```

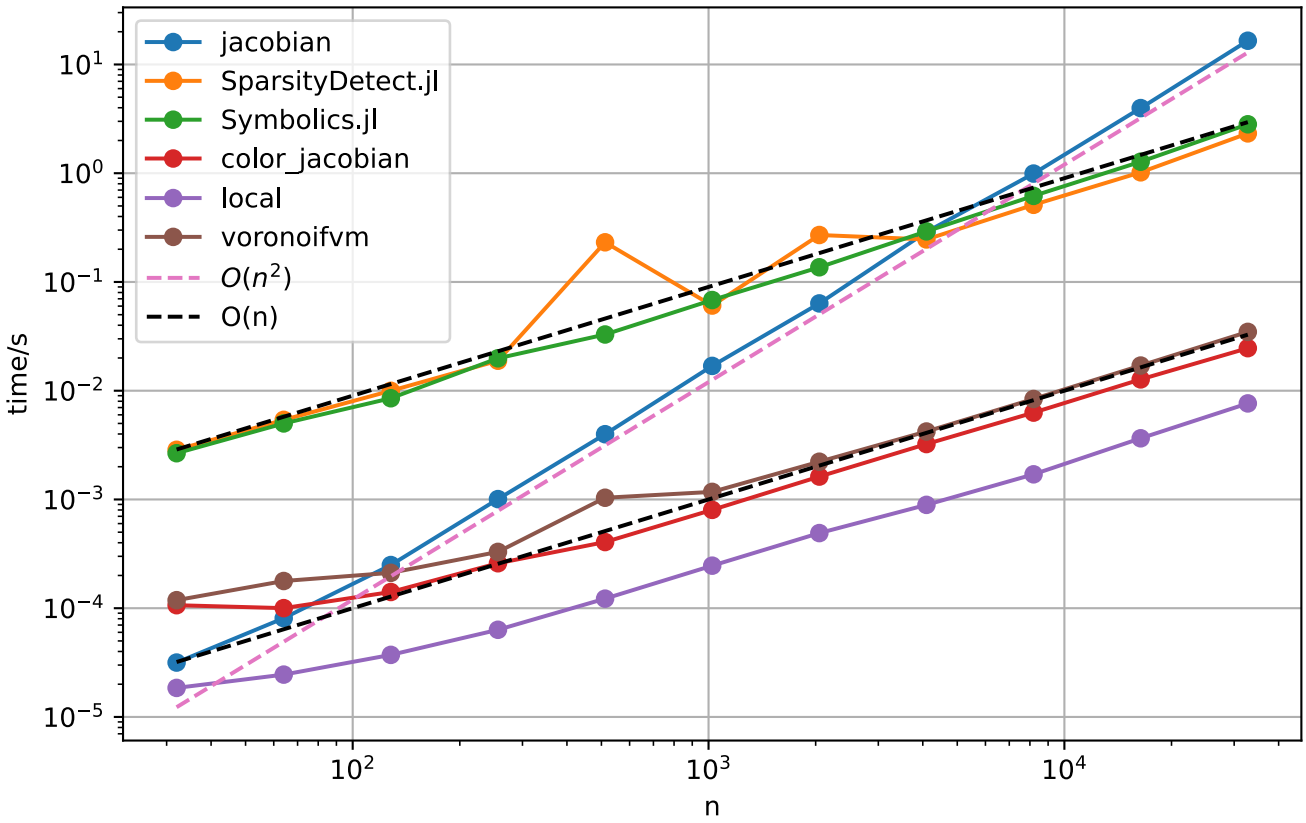
```
function voronoifvm(n)
    h=1.0/convert(Float64,n-1)
    X=collect(0:h:1)
    grid=VoronoiFVM.Grid(X)
    physics=VoronoiFVM.Physics(flux=vfvm_flux!,reaction=vfvm_reaction!)
    sys=VoronoiFVM.System(grid,physics,unknown_storage=:dense)
    enable_species!(sys,1,[1])
    solution=unknowns(sys,inival=1.0)
    oldsol=unknowns(sys,inival=1.0)
    residual=unknowns(sys,inival=1.0)
    tstep=1.0e100

    t=@elapsed begin
        VoronoiFVM.eval_and_assemble(sys,solution,oldsol,residual,tstep)
        flush!(sys.matrix)
    end
sys.matrix.cscmatrix,t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 3.10021)
  8.25  -8.0    ⋅     ⋅     ⋅
 -8.0   16.5  -8.0    ⋅     ⋅
   ⋅    -8.0  16.5  -8.0    ⋅
   ⋅      ⋅   -8.0  16.5  -8.0
   ⋅      ⋅     ⋅   -8.0   8.25
```

```
voronoifvm(5)
```

Scaling comparison




```
plot_scaling(powmax=15)
```

Discussion

- As expected, the straightforward method shows $O(n^2)$ complexity.
- Julia's sparsity detection has $O(n)$ complexity but is around two orders of magnitudes slower than `color_jacobian` used to calcualate the actual Jacobian based on the sparsity detected.
- Assembly from local Jacobians as implemented here is $O(n)$ and the fastest option. It does not need the sparsity detection step.
- Assembly from local Jacobians in VoronoiFVM.jl is $O(n)$ but performs worse than the sample implementation provided and sligtly worse than `color_jacobian`. This is due to the bookeeping for the discretization grid and other overheads which come form the generic nature of that package. It does not need the sparsity detection step as well.