Table of Contents

Ways to calculate sparse Jacobians

by Jürgen Fuhrmann, 2025-04-15

```
begin
    using ExtendableSparse
    using SparseArrays
    using ForwardDiff
    using SparseDiffTools
    using SparseConnectivityTracer
    using DifferentiationInterface
    using SparseMatrixColorings
    using Symbolics
    using VoronoiFVM
end
```

	package	version
1	VoronoiFVM	"2.9.1"
2	Symbolics	"6.38.0"
3	ForwardDiff	"0.10.38"
4	SparseDiffTools	"2.24.0"
5	DifferentiationInterface	"0.6.42"
6	SparseConnectivityTracer	"0.6.13"
7	SparseMatrixColorings	"0.4.16"

```
pkgversions([VoronoiFVM, Symbolics, ForwardDiff, SparseDiffTools,
DifferentiationInterface, SparseConnectivityTracer, SparseMatrixColorings])
```

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 corresponding 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 in 2:(n - 1)
        y[i] = (u[i]^2 - 1.0) * h
    end
    y[end] = (u[end]^2 - 1) * 0.5 * h

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

Number of discretization points for initial testing:

```
n_test = 5;
```

The methods

Straightforward Jacobian calculation

Just take the operator and calculate the jacobian using ForwardDiff.jl and use a sparse matrix to collect the derivatives. 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 $O(n^2)$. Without further structural information, automatic differencing needs to investigate the partial derivatives with respect to all unknowns.

```
function straightforward(n)
    X=ones(n)
    Y=ones(n)
    jac = ExtendableSparseMatrix(n, n)

    t=@belapsed begin
        ForwardDiff.jacobian!($jac, $A_h!, $X, $Y)
        flush!($jac)
    end
    jac.cscmatrix,t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 5.5139e-7)
  8.25  -8.0    .     .     .
```

```
straightforward(n_test)
```

Symbolics.jl for sparsity detection

While in the particular case the sparsity pattern of the operator is known – 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.jl.

This approach uses the multiple dispatch feature of Julia and calls `A_h!` with symbolic data, resulting in a symbolic representation of the operator which can be investigated for sparsity in $O(n)$ time. For a description of this approach see this paper.

In addition, for calculating the Jacobian efficiently it is useful to provide a matrix coloring obtained using SparseDiffTools.jl.

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

    end
    return jac, t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.000348425)
  8.25  -8.0    .     .     .
```

```
symbolics(n_test)
```

DifferentiationInterface

DifferentiationInterface.jl provides a common interface to automatic differentiation and sparsity detection. The later can use different backends.

Sparsity detection backend from SparseConnectivityTracer.jl:

```
SCTBackend =
AutoSparse(dense_ad=AutoForwardDiff(), sparsity_detector=TracerSparsityDetector(), color
```

```
SCTBackend = AutoSparse(
    AutoForwardDiff();  # any object from ADTypes
    sparsity_detector = TracerSparsityDetector(),
    coloring_algorithm = GreedyColoringAlgorithm(IncidenceDegree(), decompression =
:direct, postprocessing = true)
)
```

Sparsity detection via Symbolics.jl:

```
SymbolicsBackend =
AutoSparse(dense_ad=AutoForwardDiff(), sparsity_detector=SymbolicsSparsityDetector(), co
```

```
SymbolicsBackend = AutoSparse(
    AutoForwardDiff();  # any object from ADTypes
    sparsity_detector = SymbolicsSparsityDetector(),
    coloring_algorithm = GreedyColoringAlgorithm(IncidenceDegree(), decompression =
:direct, postprocessing = true),
)
```

```
di (generic function with 1 method)
```

```
function di(n, backend)
    input = ones(n)
    output = similar(input)
    tdetect = @elapsed begin
        jac_prep = prepare_jacobian(A_h!, output, backend, input)
        jac = similar(sparsity_pattern(jac_prep), Float64)
    end
    tassemble = @elapsed begin
        jacobian!(A_h!, output, jac, jac_prep, backend, input)
    end
    return jac, tdetect + tassemble
end
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.724217)
  8.25  -8.0    .     .     .
```

```
di(n_test, SCTBackend)
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.0603978)
  8.25  -8.0    .     .     .
```

```
di(n_test, SymbolicsBackend)
```

Assembly from local jacobians

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

For this purpose, we define the local contributions to the nonlinear difference operator 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, and so we need to prepare the memory for local Jacobian calculation beforehand using the tools provided by DiffResults.jl.

Note that if one wants to check the follwing function for allocations in the assembly loop, one needs to be aware that the first run with a given matrix whill show the allocations needed to set up the matrix. A second run with the same sparsity pattern shows the allocations created by the automatic differentiation process.

```
function A_Jac_h!(y, m, u, flux::F, reaction::R; nrun = 1) where {F, R}
    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, 2))

    for irun in 1:nrun
        flush!(m)
        m.cscmatrix.nzval .= 0

        nallocs = @allocated for i in 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

        irun > 1 && @info "allocations in reaction assembly: $nallocs"

        nallocs = @allocated for i in 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
        irun > 1 && @info "allocations in flux assembly: $nallocs"
    end
    return
end;
```

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

```
function assemblefromlocal(n; nrun = 1)
    u = ones(n)
    jac = ExtendableSparseMatrix(n, n)
    y = ones(n)
    t = @elapsed begin
        A_Jac_h!(y, jac, u, flux!, reaction!; nrun)
        flush!(jac)
    end
    return jac.cscmatrix, t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 0.512129)
  8.25  -8.0    .     .     .
```

```
assemblefromlocal(n_test; nrun = 2)
```

```
allocations in reaction assembly: 0

allocations in flux assembly: 0
```

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, data) = flux!(f, u)
```

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

```
vfvm_reaction!(f, u, node, data) = reaction!(f, u)
```

```
function voronoifvm(n)
    # "standard" setup of VoronoiFVM system
    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])

    # Setup test data
    solution = unknowns(sys, inival = 1.0)

    t = @elapsed begin
        residual, matrix = VoronoiFVM.evaluate_residual_and_jacobian(sys, solution)
    end
    return matrix.cscmatrix, t
end;
```

```
(5×5 SparseMatrixCSC{Float64, Int64} with 13 stored entries:, 5.79962)
  8.25  -8.0    .     .     .
```

```
voronoifvm(5)
```

Scaling comparison

Define the largest problems size 2^{powmax}:

```
powmax = 20
```

```
1 powmax = 20
```

```
N =
 [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288
```

```
1 N = (2) .^ collect(5:powmax)
```

```
t_straight =
 [1.2762e-5, 4.6247e-5, 0.000184248, 0.000729422, 0.00290372, 0.0116399, 0.0465045, 0.1858
```

```
1 t_straight = [straightforward(n)[2] for n in N[1:11]]
```

```
t_symbolic =
 [0.00116905, 0.00200886, 0.003743, 0.00790997, 0.0155273, 0.0336033, 0.069378, 0.159929,
```

```
1 t_symbolic = [symbolics(n)[2] for n in N]
```

```
t_vfvm =
 [0.000204833, 0.000194104, 0.00031337, 0.000589318, 0.00114653, 0.00222522, 0.00435482, 0
```

```
1 t_vfvm = [voronoifvm(n)[2] for n in N]
```

```
t_local =
 [3.3841e-5, 4.1826e-5, 5.1695e-5, 9.8547e-5, 0.000185446, 0.000361937, 0.000717264, 0.001
```

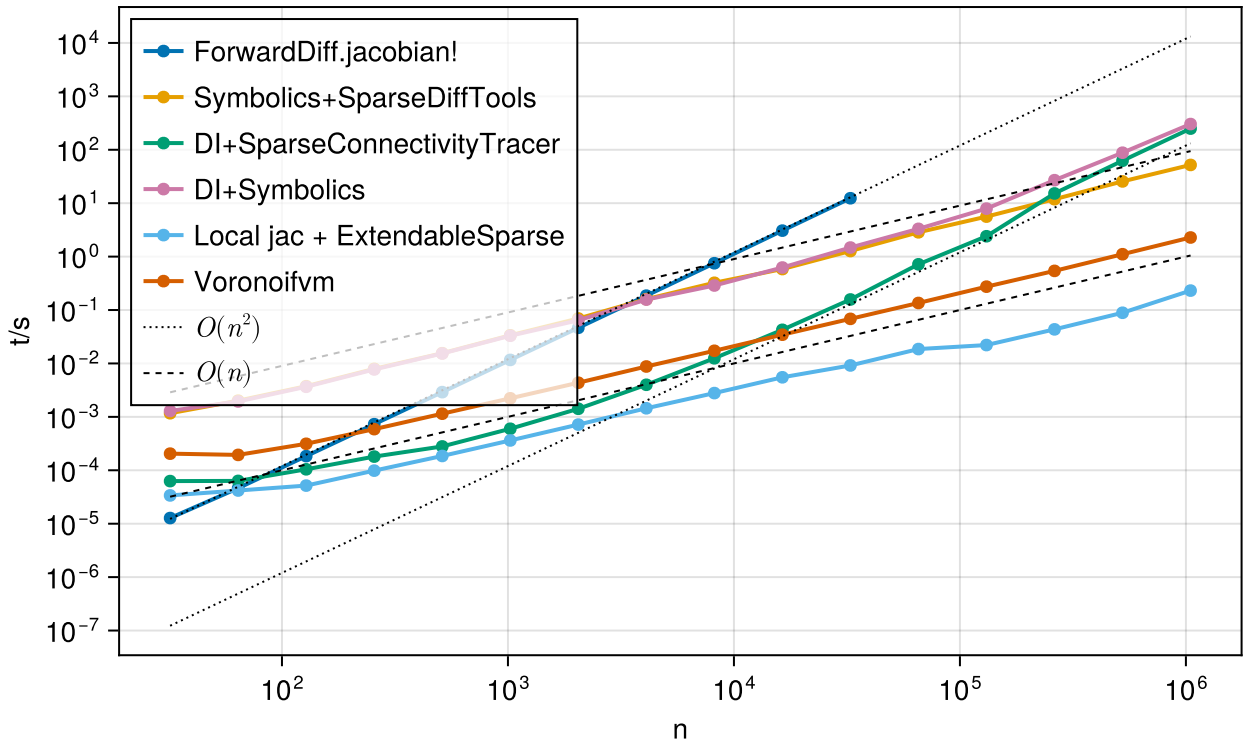
```
1 t_local = [assemblefromlocal(n)[2] for n in N]
```

```
t_di_sct =
 [6.2773e-5, 6.3258e-5, 0.000104324, 0.000180373, 0.000279569, 0.000598793, 0.00141697, 0.
```

```
1 t_di_sct = [di(n, SCTBackend)[2] for n in N]
```

```
t_di_symbolics =
 [0.00129265, 0.00194257, 0.00366137, 0.00772732, 0.015363, 0.0330247, 0.0642345, 0.156757
```

```
1 t_di_symbolics = [di(n, SymbolicsBackend)[2] for n in N]
```




```
let
    fig = CairoMakie.Figure(linewidth=2,size=(650,400))
    ax=Axis(fig[1,1], xscale=log10,yscale=log10, xlabel="n", ylabel="t/s",
           yticks=LogTicks(LinearTicks(13)))
    scatterlines!(ax,N[1:length(t_straight)], t_straight, label =
"ForwardDiff.jacobian!")
    scatterlines!(ax,N, t_symbolic, label = "Symbolics+SparseDiffTools")
    scatterlines!(ax,N, t_di_sct, label = "DI+SparseConnectivityTracer")
    scatterlines!(ax,N, t_di_symbolics, label = "DI+Symbolics")
    scatterlines!(ax,N, t_local, label = "Local jac + ExtendableSparse")
    scatterlines!(ax,N, t_vfvm, label = "Voronoifvm")
    lines!(ax,N, 1.2e-8 * N .^ 2, label = L"O(n^2)", color=:black, linestyle=:dot,
linewidth=1)
    lines!(ax,N, 1.2e-10 * N .^ 2, color=:black, linestyle=:dot, linewidth=1)
    lines!(ax,N, 1.0e-6 * N, label = L"O(n)", color=:black, linestyle=:dash,
linewidth=1)
    lines!(ax,N, 0.9e-4 * N, color=:black, linestyle=:dash,linewidth=1)
    axislegend(ax, merge=true, position=:lt,backgroundcolor = (:white, 0.75))
    save("sparsejac2.png",fig)
    fig
end
```

Preliminary Discussion

- As expected, the straightforward method shows $O(n^2)$ complexity.
- Symbolics.jl + SparseDiffTools.jl sparsity detection has $O(n)$ complexity.
- DifferentiationInterface.jl with Symbolics.jl backend deviates from $O(n)$ behavior.
- DifferentiationInterface.jl with SparseConnectivityTracer.jl backend is very fast for small problems, but appears to have an asymptotic scaling like $O(n^2)$
- Assembly from local Jacobians using the linked-list based sparse matrix assembly from ExtendableSparse.jl as implemented here is $O(n)$ and the fastest option.
- Assembly from local Jacobians using VoronoiFVM.jl is $O(n)$. It uses a similar approach but performs worse than the sample implementation provided here. This is due to the bookkeeping for the discretization grid and other overheads which come from the generic nature of that package. It does not need the sparsity detection step as well.

```
statement"""
- As expected, the straightforward method shows  ``O(n^2)`` complexity.
- Symbolics.jl + SparseDiffTools.jl sparsity detection has ``O(n)`` complexity.
- DifferentiationInterface.jl with Symbolics.jl backend deviates from ``O(n)`` behavior.
- DifferentiationInterface.jl with SparseConnectivityTracer.jl backend is very fast for small problems, but appears to have an asymptotic scaling like ``O(n^2)``
- Assembly from local Jacobians using the linked-list based sparse matrix assembly from ExtendableSparse.jl  as implemented here is ``O(n)`` and the fastest option.
- Assembly from local Jacobians using  VoronoiFVM.jl is ``O(n)``. It uses a similar approach but performs worse than the sample implementation provided here. This is due to the bookkeeping for the discretization grid and other overheads which come from the generic nature of that package. It does not need the sparsity detection step as well.
"""
```

Appendix: supporting tools

```
begin
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
    using PkgVersion
    using BenchmarkTools
    using Tables
    using Dates
    using CairoMakie
    CairoMakie.activate!(type="svg")
end;
```

```
pkgversions (generic function with 1 method)
```

```
function pkgversions(pkgs)
    return Tables.table(reduce(hcat, [[pkg, PkgVersion.Version(pkg) |> string] for pkg in pkgs]) |> permutedims; header = [:package, :version])
end
```

```
begin
    hrule() = html"""<hr>"""
    highlight(mdstring, color) = htl"""<blockquote style="padding: 10px; background-color: $(color);">$(mdstring)</blockquote>"""

    macro important_str(s)
        return :(highlight(Markdown.parse($s), "#ffcccc"))
    end
    macro definition_str(s)
        return :(highlight(Markdown.parse($s), "#ccccff"))
    end
    macro statement_str(s)
        return :(highlight(Markdown.parse($s), "#ccffcc"))
    end


    html"""
            <style>
             h1{background-color:#dddddd;  padding: 10px;}
             h2{background-color:#e7e7e7;  padding: 10px;}
             h3{background-color:#eeeeee;  padding: 10px;}
             h4{background-color:#f7f7f7;  padding: 10px;}

             pluto-log-dot-sizer  { max-width: 655px;}
             pluto-log-dot.Stdout { background: #002000;
                                    color: #10f080;
                                    border: 6px solid #b7b7b7;
                                    min-width: 18em;
                                    max-height: 300px;
                                    width: 675px;
                                        overflow: auto;
                                   }

        </style>
    """
end
```