```
using PlutoUI, Viznet, Compose, Plots
```

present

Cool automatic differentiation applications

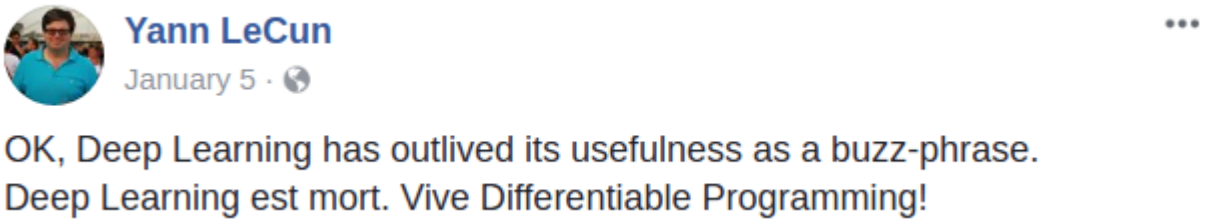
– Jinguo Liu

- What is automatic differentiation (AD)?
  - A true history of AD
  - Forward mode AD
  - Reverse mode AD
    - primitves on tensors (including tensorflow, pytorch et al.)
    - primitves on elementary instructions (usually source code transformation based)
    - defined on a reversible program
- Some applications in **scientific computing**
  - solving the graph embedding problem
  - inverse engineering a hamiltonian
  - obtaining maximum independent set (MIS) configurations
  - towards differentiating `expmv` ◀ will be used in our emulator

The true history of automatic differentiation

- 1964 ~ Robert Edwin Wengert, A simple automatic derivative evaluation program.
  ◀ first forward mode AD
- 1970 ~ Seppo Linnainmaa, Taylor expansion of the accumulated rounding error.
  ◀ first backward mode AD

- 1986 ~ Rumelhart, D. E., Hinton, G. E., and Williams, R. J., Learning representations by back-propagating errors.
- 1992 ~ Andreas Griewank, Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation.
  ◀ foundation of source code transformation based AD.
- 2000s ~ The boom of tensor based AD frameworks for machine learning.
- 2018 ~ People re-invented AD as differential programming (**wiki** and this **quora answer**.)



- 2020 ~ Me, Differentiate everything with a reversible embeded domain-specific language
  ◀ AD based on reversible programming.

Forward mode automatic differentiation

Forward mode AD attaches a infitesimal number ϵ to a variable, when applying a function f, it does the following transformation

$$f(x + g\epsilon) = f(x) + f'(x)g\epsilon + \mathcal{O}(\epsilon^2)$$

The higher order infinitesimal is ignored.

In the program, we can define a *dual number* with two fields, just like a complex number

```
f((x, g)) = (f(x), f'(x)*g)
```

```
using ForwardDiff: Dual
```

```
res = Dual{Nothing}(0.7071067811865475,1.4142135623730951)
```

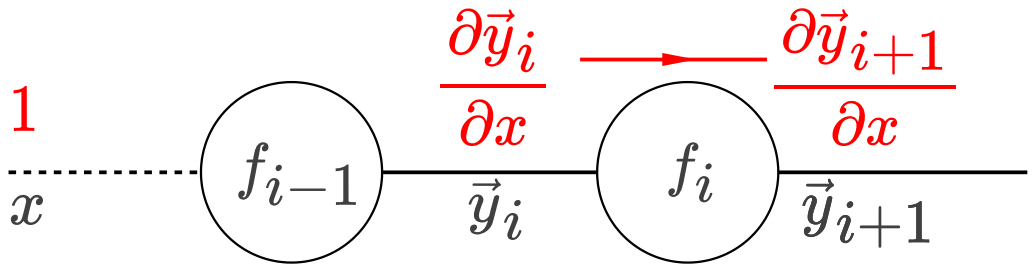
```
res = sin(Dual(π/4, 2.0))
```

```
true
```

```
res === Dual(sin(π/4), cos(π/4)*2.0)
```

We can apply this transformation consecutively, it reflects the chain rule.

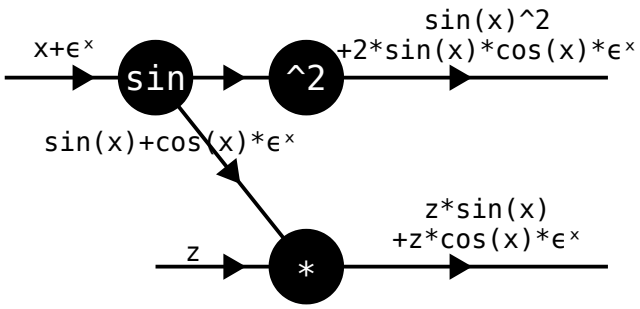
$$\frac{\partial \vec{y}_{i+1}}{\partial x} = \underbrace{\boxed{\frac{\partial \vec{y}_{i+1}}{\partial \vec{y}_i}}}_{\text{local Jacobian}} \frac{\partial \vec{y}_i}{\partial x}$$




```
Dual{Nothing}(0.43854142638025273,0.15675010696766714)
```

```
let
    x = Dual(π/4, 1.0)
    for i=1:10
        x = sin(x)
    end
    x
end
```

Example: Computing two gradients $\frac{\partial z \sin x}{\partial x}$ and $\frac{\partial \sin^2 x}{\partial x}$ at one sweep



so the gradients are $z \cos x$ and $2 \sin x \cos x$

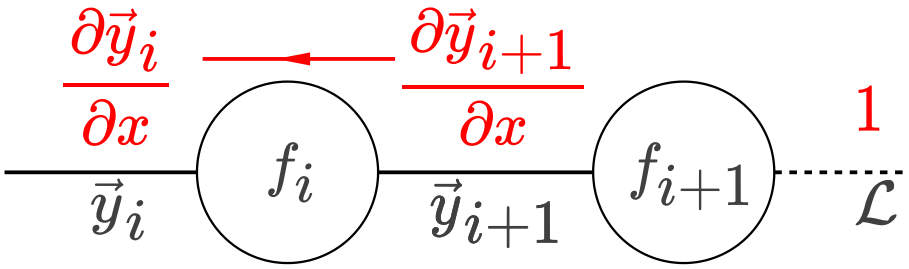
What if we want to compute gradients for multiple inputs?

The computing time grows **linearly** as the number of variables that we want to differentiate. But does not grow significantly with the number of outputs.

Reverse mode automatic differentiation

On the other side, the back-propagation can differentiate **many inputs** with respect to a **single output** efficiently

$$\frac{\partial \mathcal{L}}{\partial \vec{y}_i} = \frac{\partial \mathcal{L}}{\partial \vec{y}_{i+1}} \boxed{\frac{\partial \vec{y}_{i+1}}{\partial \vec{y}_i}}$$
local jacobian?

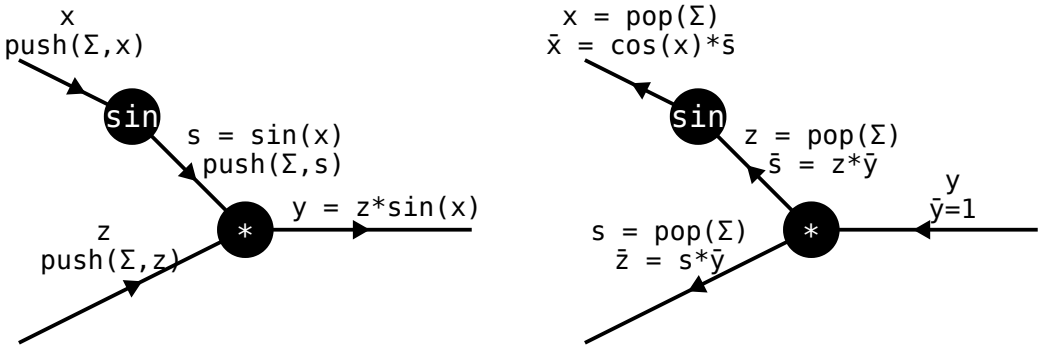


How to visite local Jacobians in the reversed order?

Design Decision

1. Compute forward pass and caching inetermediate results into a global stack Σ （packages except NiLang），
2. reversible programming.

Example: Computing the gradient $\frac{\partial z \sin x}{\partial x}$ and $\frac{\partial z \sin x}{\partial z}$ by back propagating cached local information.



Here, we use $\overline{y}$ for $\frac{\partial \mathcal{L}}{\partial y}$, which is also called the adjoint.

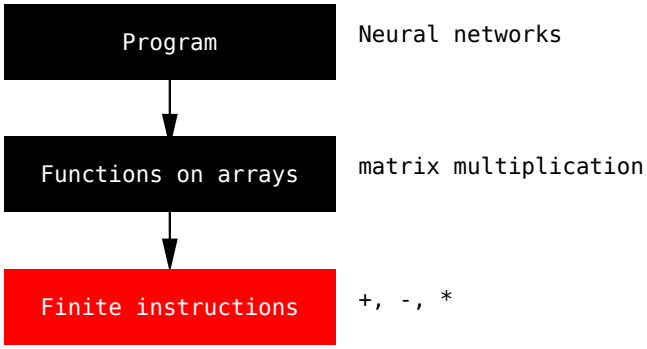
Primitives on different scales

We call the leaf nodes defining AD rules "**primitives**"

Design Decision

- A: If we define primitives on **arrays**, we need tons of manually defined backward rules. (Jax, Pytorch, Zygote.jl, ReverseDiff.jl et al.)
- B: If we define primitives on **scalar instructions**, we will have worse tensor performance. (Tapenade, Adept, NiLang et al.)

Note: Here, implementing AD on scalars means specifically the **optimal checkpointing** approach, rather than a package like Jax, Zygote and ReverseDiff that having scalar support.



	on tensors	on finite instructions
meaning	defining backward rules manully for functions on tensors	defining backward rules on a limited set of basic scalar operations, and generate gradient code using source code transformation
pros and cons	1. Good tensor performance 2. Mature machine learning ecosystem 3. Need to define backward rules manually	1. Reasonalbe scalar performance 2. hard to utilize GPU kernels (except NiLang.jl) and BLAS
packages	Jax PyTorch	**Tapenade** **Adept** **NiLang.jl**

The AD ecosystem in Julia

Please check JuliaDiff: **https://juliadiff.org/**

A short list:

- Forward mode AD: ForwardDiff.jl
- Reverse mode AD (tensor): ReverseDiff.jl/Zygote.jl
- Reverse mode AD (scalar): NiLang.jl

Warnings

- The main authors of `Tracker`, `ReverseDiff` and `Zygote` are not maintaining them anymore.

Quick summary

1. The history of AD is longer than many people have thought. People are most familar with *reverse mode AD with primitives implemented on tensors* that brings the boom of machine learning. There are also AD frameworks that can differentiate a general program directly, which does not require users defining AD rules manually.
2. **Forward mode AD** propagate gradients forward, it has a computational overhead propotional to the number of input parameters.
3. **Backward mode AD** propagate gradients backward, it has a computational overhead propotional to the number of output parameters.
    - primitives on **tensors** v.s. **scalars**
    - reverse the program tape by **caching/checkpointing** v.s. **reversible programming**
4. Julia has one of the most active AD community!

Forward v.s. Backward

when is forward mode AD more useful?

- It is often combined with backward mode AD for obtaining Hessians (forward over backward).
- Having <20 input parameters.

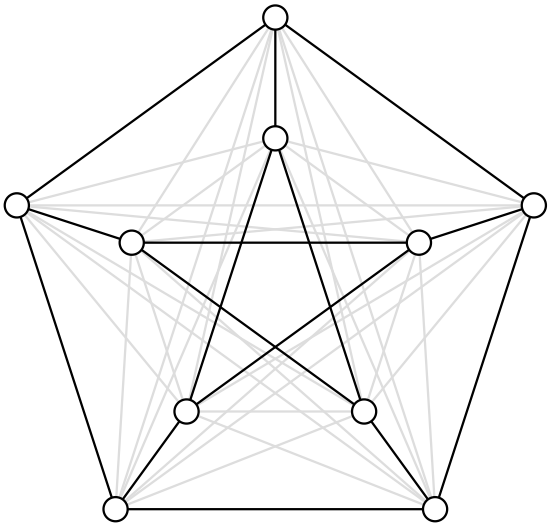
when is backward mode AD more useful?

- In most variational optimizations, especially when we are training a neural network with ~ 100M parameters.

1. Embedding a peterson Graph

One day, A postdoc of Anders Sandvik Jun Takahashi went to me, said "Hey, Jinguo, can you help me figure out what is the minimum embedding dimension of a Peterson graph?"

A Peterson graph is a famous 3-regular graph with very high symmetry. It is well know to graph theory people. It looks like



It has 10 vertices, 15 edges, while these vertices are all equivalent to each other. By embedding a graph into a k-dimensional space, it requires

1. assigning a k-dimensional vector to each node as the Euclidean coordinate,
2. the distance between each pair of connected nodes are the same, meanwhile, the distance between each pair of disconnected nodes are same too.
3. the distance between disconnected vertices are larger than connect vertices

```
using NiLang, Random
```

```
# connected vertex-pairs in a petersen graph
const L1 = [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10),
    (1, 2), (2, 3), (3, 4), (4, 5), (1, 5), (6, 8),
    (8, 10), (7, 10), (7, 9), (6, 9)];
```

```
# disconnected vertex-pairs in a petersen graph
const L2 = [(1, 3), (1, 4), (1, 7), (1, 8), (1, 9),
    (1, 10), (2, 4), (2, 5), (2, 6), (2, 8), (2, 9),
    (2, 10), (3, 5), (3, 6), (3, 7), (3, 9), (3, 10),
    (4, 6), (4, 7), (4, 8), (4, 10), (5, 6), (5, 7),
    (5, 8), (5, 9), (6, 7), (6, 10), (7, 8), (8, 9),
    (9, 10)];
```

For dimension $k \in 1, 2, \ldots, 10$, we assign a coordinate to each vertex. Then we define the loss as

$$
\begin{aligned}
D_1 &= \{d_{(i,j)} | (i,j) \in L_1\} \\
D_2 &= \{d_{(i,j)} | (i,j) \in L_2\} \\
\mathcal{L} &= \text{var}(D_1) + \text{var}(D_2) \\
&\quad + \exp(\text{relu}(\text{mean}(D_1) - \text{mean}(D_2) + 0.1)) - 1 \quad \text{◀ if } d_2 < d_1\text{, punish}
\end{aligned}
$$

`relu` is defined as `x > 0 ? x : 0`

```
"""The loss of graph embedding problem."""
@i function embedding_loss(out!::T, x) where T
    @routine @invcheckoff begin
        @zeros T v1 varsum1 varsum2 s1 s2 m1 v2 m2 diff
        d1 ← zeros(T, length(L1))
        d2 ← zeros(T, length(L2))
        # 1. compute distances
        for i=1:length(L1)
            sqdistance(d1[i], x[:,L1[i][1]],x[:,L1[i][2]])
        end
        for i=1:length(L2)
            sqdistance(d2[i], x[:,L2[i][1]],x[:,L2[i][2]])
        end
        # 2. compute variances
        NiLang.i_var_mean_sum(v1, varsum1, m1, s1, d1)
        NiLang.i_var_mean_sum(v2, varsum2, m2, s2, d2)
        m1 -= m2 - 0.1
    end
    out! += v1 + v2
    if m1 > 0
        # to ensure mean(v2) > mean(v1)
        # if mean(v1)+0.1 - mean(v2) > 0, punish it.
        out! += exp(m1)
        out! -= 1
    end
    ~@routine
end
```

```
@i function sqdistance(dist!, x1::AbstractVector{T}, x2::AbstractVector) where T
    for i=1:length(x1)
        @routine begin
            diff ← zero(T)
            diff += x1[i] - x2[i]
        end
        dist! += diff ^ 2
        ~@routine
    end
end
```

```
using Optim
```

Seed =

dimension 5

```
x_minimizer, x_minimum = let
    Random.seed!(seed)
    x = randn(dimension,10)
```

```
    # `NiLang.AD.gradient` to obtain the gradients
    res = Optim.optimize(x->embedding_loss(0.0, x)[1], x->NiLang.AD.gradient(embedding_loss, (0.0, x); iloss=1)[2], x, LBFGS(), Optim.Options(f_abstol=1e-12, f_reltol=1e-12, g_abstol=1e-12, g_reltol=1e-12), inplace=false)
    res.minimizer, res.minimum
end;
```

```
5.013885330240668e-14
```

```
x_minimum
```

```
d1s =
  Float64[0.32641, 0.326411, 0.32641, 0.326411, 0.326411, 0.326411, 0.326411, 0.326411
```

```
d1s = [norm(x_minimizer[:,a] .- x_minimizer[:,b]) for (a, b) in L1]
```

```
d2s =
  Float64[0.461614, 0.461614, 0.461614, 0.461614, 0.461614, 0.461614, 0.461614, 0.4616
```

```
d2s = [norm(x_minimizer[:,a] .- x_minimizer[:,b]) for (a, b) in L2]
```

```
using Statistics: mean
```

```
1.414213562371709
```

```
mean(d2s)/mean(d1s)
```

His work of finding the SO(5) symmetric tensor order representation is later published as

"Valence-bond solids, vestigial order, and emergent SO(5) symmetry in a two-dimensional quantum magnet." (Phys. Rev. Research 2, 033459, Jun Takahashi, Anders W. Sandvik)

2. Inverse engineering a Hamiltonian

This problem is from "Notes on Adjoint Methods for 18.335", Steven G. Johnson

Consider a 1D Shrodinger equation

$$\left[-\frac{d^2}{dx^2} + V(x)\right]\Psi(x) = E\Psi(x), x \in [-1, 1]$$

We can solve its gound state numerically by discretizing the space and diagonalize the Hamiltonian matrix. The Hamiltonian matrix is

$$A = \frac{1}{\Delta x^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 & -1 \\ -1 & 2 & -1 & 0 & \dots & \\ 0 & -1 & 2 & -1 & 0 & \dots \\ \vdots & & & \ddots & & \\ & & & -1 & 2 & -1 \\ -1 & 0 & \dots & 0 & -1 & 2 \end{pmatrix} + \mathrm{diag}(V)$$

where the matrix size is equal the descretized lattice size

```
dx = 0.02;
```

```
xgrid = -1.0:dx:1.0;
```

```
@i function hamiltonian!(a, x, V::AbstractVector{T}) where T
    @routine begin
        @zeros T dx2 invdx2
        n ← length(x)
        dx2 += (@const Float64(x.step))^2
        invdx2 += 1/dx2
    end
    @safe @assert size(a) == (n, n)
    for i=1:n
        a[i, i] += 2 * invdx2
        a[i, i] += V[i]
        a[i, mod1(i+1, n)] -= invdx2
        a[mod1(i+1, n), i] -= invdx2
    end
    ~@routine
end
```

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

```
hamiltonian(x, V) = hamiltonian!(zeros(length(x), length(x)), x, V)[1]
```

```
101×101 Matrix{Float64}:
  5000.15  -2500.0       0.0       0.0   …      0.0       0.0       0.0   -2500.0
 -2500.0    4999.31  -2500.0       0.0          0.0       0.0       0.0       0.0
     0.0   -2500.0    5000.63  -2500.0          0.0       0.0       0.0       0.0
     0.0       0.0   -2500.0    5000.07         0.0       0.0       0.0       0.0
     0.0       0.0       0.0   -2500.0          0.0       0.0       0.0       0.0
     0.0       0.0       0.0       0.0   …      0.0       0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0       0.0
     ⋮                                    ⋱                                  ⋮
     0.0       0.0       0.0       0.0   …      0.0       0.0       0.0       0.0
     0.0       0.0       0.0       0.0      -2500.0       0.0       0.0       0.0
     0.0       0.0       0.0       0.0       4998.58  -2500.0       0.0       0.0
     0.0       0.0       0.0       0.0      -2500.0    5000.79  -2500.0       0.0
     0.0       0.0       0.0       0.0          0.0   -2500.0    5000.34  -2500.0
 -2500.0       0.0       0.0       0.0   …      0.0       0.0   -2500.0    4999.77
```

```
hamiltonian(xgrid, randn(length(xgrid)))
```

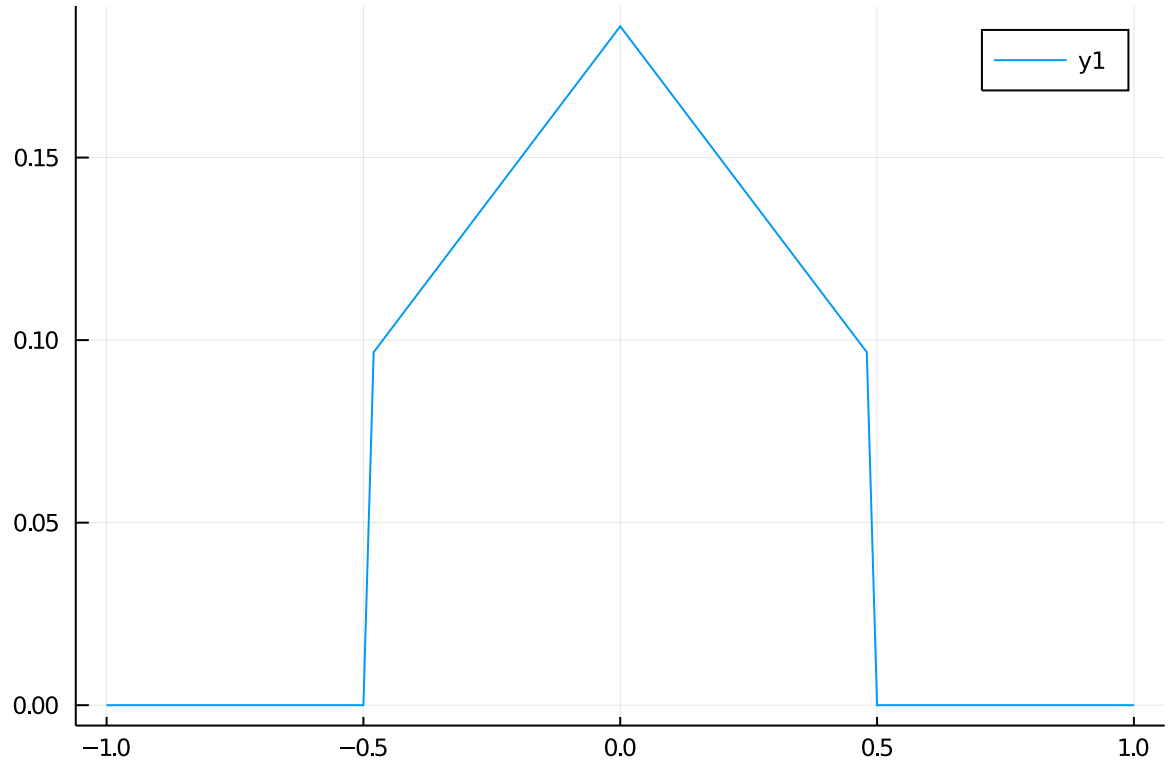
Because we are going to use Zygote (with rules set defined in ChainRules)

```
using ChainRules
```

```
function ChainRules.rrule(::typeof(hamiltonian), x, V)
    y = hamiltonian(x, V)
    function hamiltonian_pullback(Δy)
        gV = NiLang.AD.grad((~hamiltonian!)(GVar.(y, Δy), x, GVar.(V))[3])
        return (ChainRules.NO_FIELDS, ChainRules.DoesNotExist(), gV)
    end
    return y, hamiltonian_pullback
end
```

We want the ground state be a house.

```
ψ0 = [abs(xi)<0.5 ? 1 - abs(xi) : 0 for xi in xgrid]; normalize!(ψ0);
```




```
plot(xgrid, ψ0)
```

So we define a loss function

$$E, \psi = \text{eigensolve}(A)$$
$$\mathcal{L} = \sum_i |(|(\psi_0)_i| - |(\psi_G)_i|)|$$

where ψ_G is state vector in ψ that corresponds to the minimum energy.

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

```
function solve_wave(x, V)
    a = hamiltonian(x, V)
```

```
    ψ = LinearAlgebra.eigen(LinearAlgebra.Hermitian(a)).vectors[:,1]
end
```

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

```
function loss(x, V, ψ0)
    ψ = solve_wave(x, V)
    sum(map(abs, map(abs, ψ) - map(abs, ψ0))) * dx
end
```

```
using LinearAlgebra
```

```
0.14403694685265442
```

```
loss(xgrid, randn(length(xgrid)), ψ0)
```

```
1.0
```

```
solve_wave(xgrid, randn(length(xgrid))) |> norm
```

```
0.1437166117076478
```

```
loss(xgrid, randn(length(xgrid)), ψ0)
```

```
using Zygote
```

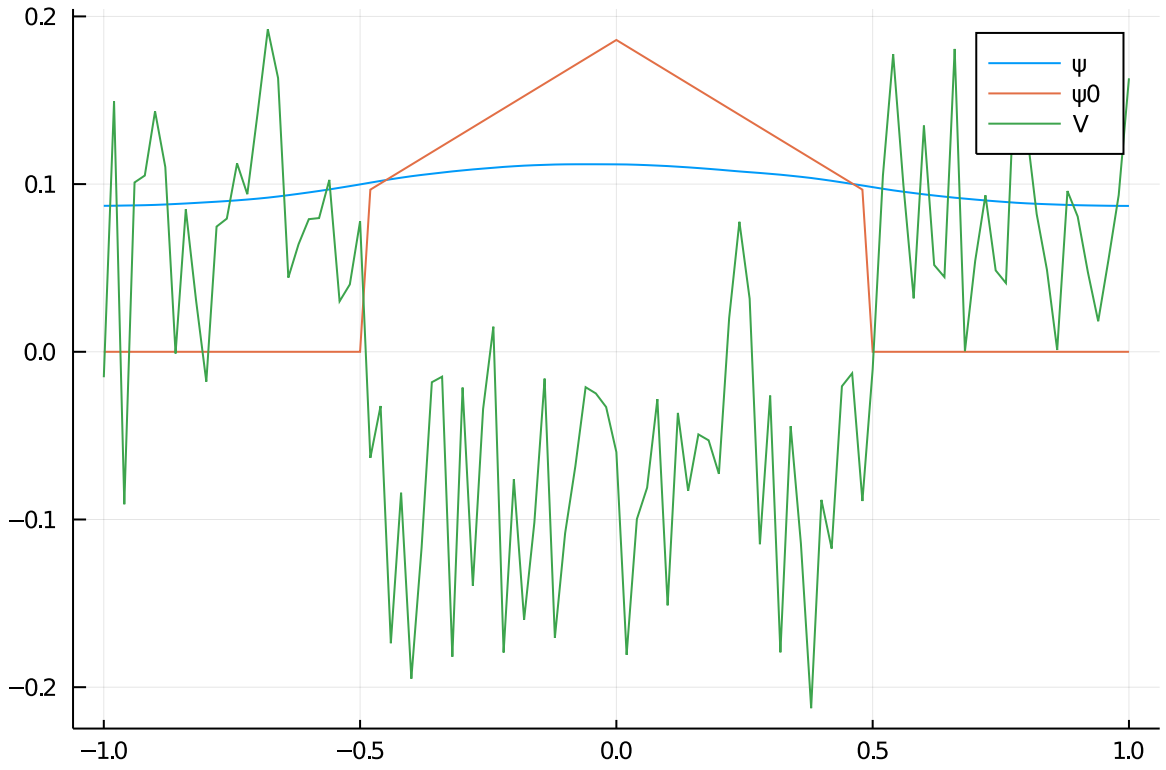
```
(Float64[-0.000242508, -0.00024219, -0.000241064, -0.000239543, -0.000237412, -0.0002
```

```
Zygote.gradient(v->loss(xgrid, v, ψ0), randn(length(xgrid)))
```

Start speed: 0.1 *secs / tick*

```
using StochasticOptimizers
```

```
it = adam(v->loss(xgrid, v, ψ0), x->Zygote.gradient(v->loss(xgrid, v, ψ0), x)[1],
randn(length(xgrid)); η=1.0);
```




```
let
    clock
    state = step!(it)
    v = minimizer(state)
    ψ = solve_wave(xgrid, v)
    @show loss(xgrid, v, ψ0)
    plot(xgrid, abs.(ψ); label="ψ")
    plot!(xgrid, abs.(ψ0); label="ψ0")
    plot!(xgrid, normalize(v); label="V")
end |> PlutoUI.as_svg
```

3. Obtaining MIS configurations

We are able to get the weighted maximum independent set (MIS) size of the following graph

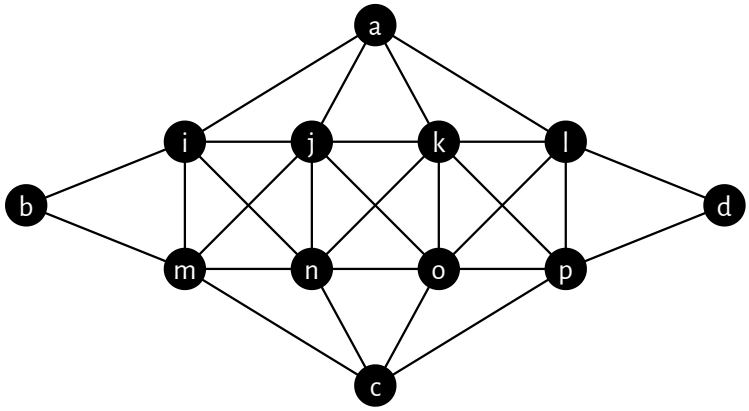
$$S = \max_{\vec{s}} \left(\sum_i w_i s_i - \infty \sum_{ij \in E} s_i s_j \right), s_i \in \{0, 1\}$$

where s_i and w_i are the configuration (in MIS: 1, not in MIS: 0) and weight of node i.

Question: how to get the configuration with MIS?

The optimal configuration is a gradient!

$$\frac{\partial S}{\partial w_i} = \begin{cases} 1 & s_i \in \vec{s}_{\max} \\ 0 & otherwise \end{cases}$$



The actual problem is harder, if we fix the boundary configurations `a`, `b`, `c`, `d`, what is the optimal configurations for interior?

```
nodes_simple = let
    a = 0.12
    ymid = xmid = 0.5
    X = 0.33
    Y = 0.17
    D = 0.15
    y = [ymid-Y, ymid-Y+D, ymid-a/2, ymid+a/2, ymid+Y-D, ymid+Y]
    x = [xmid-X, xmid-X+D, xmid-1.5a, xmid-a/2, xmid+a/2, xmid+1.5a, xmid+X-D,
xmid+X]
    xmin, xmax, ymin, ymax = x[1], x[end], y[1], y[end]
    ["a"=>(xmid, y[1]), "b"=>(xmin, ymid), "c"=>(xmid, ymax), "d"=>(xmax, ymid),
        "i"=>(x[3], y[3]), "j"=>(x[4], y[3]),
        "k"=>(x[5], y[3]), "l"=>(x[6], y[3]), "m"=>(x[3], y[4]),
        "n"=>(x[4], y[4]), "o"=>(x[5], y[4]), "p"=>(x[6], y[4])]
end;
```

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

```
function find_edges(nodes, distance)
    edges = Tuple{Int,Int}[]
    for (i, p) in enumerate(nodes)
        for (j,p2) in enumerate(nodes)
            if i<j && sqrt(sum(abs2, p2 .- p)) < distance
                push!(edges, (i,j))
            end
        end
    end
    edges
end
```

```
edges_simple = find_edges(getindex.(nodes_simple, 2), 0.23);
```

```
"""
* `optsize` stores the MIS size the configuration specified by the 4th argument.
```

```
* `out` stores the contraction results, which is a 2^4 tensor. The entries represents the MIS size for a given boundary configuration (e.g. MIS size is 2 for a = b = c = d = 0)
* `x` is the node weights.
* `config` is the boundary configurations.
"""
@i function compute_mis(optsize, out::Array{T,4}, x::Vector{T}, config) where T
    @routine begin
        # defining contraction patterns
        ixs ← (('a',), ('b',), ('c',), ('d',), ('i',), ('j',), ('k',), ('l',), ('m',), ('n',),('o',), ('p',),('a', 'i'), ('a', 'j'), ('a', 'k'), ('a', 'l'), ('b', 'i'), ('b', 'm'), ('c', 'm'), ('c', 'n'), ('c', 'o'), ('c', 'p'), ('d', 'l'), ('d', 'p'), ('i', 'j'), ('i', 'm'), ('i', 'n'), ('j', 'k'), ('j', 'm'), ('j', 'n'), ('j', 'o'), ('k', 'l'), ('k', 'n'), ('k', 'o'), ('k', 'p'), ('l', 'o'), ('l', 'p'), ('m', 'n'), ('n', 'o'), ('o', 'p'))
        iy ← ('a', 'b', 'c', 'd')
        # construct tropical tensors
        xs ← ([ones(T,2) for i=1:length(x)]..., [ones(T, 2, 2) for j=1:28]...)
        for i=1:length(x)
            vertex_tensor(xs |> tget(i), 1, x[i])
        end
        for j=length(x)+1:length(x)+28
            bond_tensor(xs |> tget(j))
        end
    end
    # contract tropical tensors
    i_einsum!(ixs, xs, iy, out)
    # store the entry with specific boundary configuration to `optsize`
    optsize += out[config...].n
    ~@routine
end
```

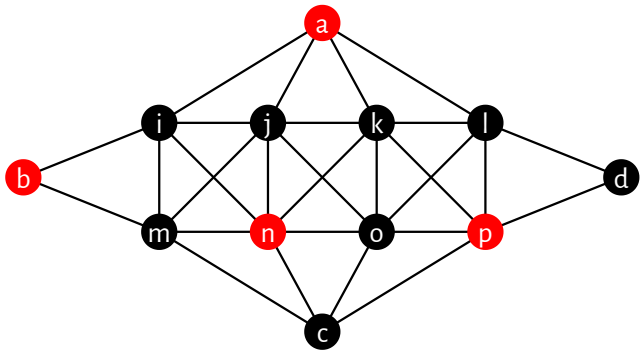
```
(4, 2×2×2×2 Array{Tropical{Int64}, 4}:, TropicalNumbers.Tropical{Int64}[1ₜ, 1ₜ, 1ₜ, 1
[:, :, 1, 1] =
 2ₜ  3ₜ
 3ₜ  4ₜ

[:, :, 2, 1] =
 3ₜ  4ₜ
 2ₜ  3ₜ

[:, :, 1, 2] =
 3ₜ  3ₜ
 4ₜ  4ₜ

[:, :, 2, 2] =
 4ₜ  4ₜ
 3ₜ  4ₜ
```

```
compute_mis(0, ones(Tropical{Int},2,2,2,2), Tropical.(ones(Int,12)), [1,1,2,2])
```

We want to differentiate the weights (3rd argument) with respect to the loss (1st argument).

☑ a ☑ b ☐ c ☐ d

```
configs = NiLang.AD.gradient(compute_mis, (0, ones(Tropical{Int64},2,2,2,2), Tropical.(ones(Int,12)), 1 .+ [ca,cb,cc,cd]); iloss=1)[3];
```




```
vizconfig(nodes_simple, edges_simple, content.(configs))
```

Here is a quick reference of the function definitions of Tropical algebra and Tropical einsum.

Note: For regular tensors, we can use existing backward rules defined in `ChainRules`.

```
begin
    using TupleTools, TropicalNumbers
    using NiLang.AD: GVar

    @i function bond_tensor(res::Matrix{T}) where T
        x ← zero(T)
        SWAP(res[2, 2], x)
        x → one(T)
    end

    @i function vertex_tensor(res::Array{T}, n::Int, val::T) where T
        for i=2:length(res)-1
            x ← zero(T)
            SWAP(res[i], x)
            x → one(T)
        end
        x ← one(T)
        res[1] *= x
        res[end] *= val
    end

    @i @inline function :(*=)(+)(z::Tropical, x::Tropical, y::Tropical)
        if x.n > y.n
            z.n += x.n
        else
```

```
            z.n += y.n
        end
    end

    @i @inline function (:*=(identity))(x::Tropical, y::Tropical)
        x.n += y.n
    end

    @i @inline function (:*=(*))(out!::Tropical, x::Tropical, y::Tropical)
        out!.n += x.n + y.n
    end

    """
        i_einsum!(ixs, xs, iy, y::AbstractArray{T})

    A naive reversible implementation of `i_einsum` function for tropical numbers.
        * `ixs`: input tensor indices,
        * `xs`: input tensors,
        * `iy`: output tensor indices,
        * `y`: accumulated tensor, notice it is initialized to 0 as output!

    # NOTE: this function is general purposed and slow!
    """
    @i function i_einsum!(ixs, xs, iy, y::AbstractArray{T}) where {T<:Tropical}
        @routine begin
            # outer legs and inner legs
            outer_indices ← unique(iy)
            inner_indices ← setdiff(TupleTools.vcat(ixs...), outer_indices)

            # find size for each leg
            all_indices ← TupleTools.vcat(ixs..., iy)
            all_sizes ← TupleTools.vcat(size.(xs)..., size(y))
            outer_sizes ← [map(i->all_sizes[i], indexin(outer_indices,
[all_indices...]))...]
            inner_sizes ← [map(i->all_sizes[i], indexin(inner_indices,
[all_indices...]))...]

            # cartesian indices for outer and inner legs
            outer_ci ← CartesianIndices((outer_sizes...,))
            inner_ci ← CartesianIndices((inner_sizes...,))

            # for indexing tensors (leg binding)
            indices ← (outer_indices..., inner_indices...)
            locs_xs ← map(ix->map(i->findfirst(isequal(i), indices), ix), ixs)
            locs_y ← map(i->findfirst(isequal(i), outer_indices), iy)
        end
        i_loop!(locs_xs, xs, locs_y, y, outer_ci, inner_ci)
        ~@routine
    end

    """take an index subset from `ind`"""
    index_map(ind::CartesianIndex, locs::Tuple) =
CartesianIndex(TupleTools.getindices(Tuple(ind), locs))

    """
    loop and accumulate products to y, the GPU version, the CPU version.
    """
    @i function i_loop!(locs_xs::NTuple{N,Any}, xs::NTuple{N, AbstractArray}, locs_y,
y::AbstractArray{T}, outer_ci::CartesianIndices, inner_ci::CartesianIndices) where
{N, T<:Tropical}
        @invcheckoff @inbounds for i in outer_ci
            @routine begin
                el ← zero(T)
                ind_y ← outer_ci[i]
                iy ← index_map(ind_y, locs_y)
```

```
                branch_keeper ← zeros(Bool, size(inner_ci)...)
                pl ← ones(T, size(inner_ci)...)
                for ind_x in inner_ci
                    pli ← one(T)
                    ind_xy ← CartesianIndex(TupleTools.vcat(ind_y.I, ind_x.I))
                    for I=1:N
                        pli *= xs[I][index_map(ind_xy, locs_xs[I])]
                    end
                    if (el.n < pli.n, branch_keeper[ind_x])
                        FLIP(branch_keeper[ind_x])
                        SWAP(el, pli)
                    end
                    SWAP(pl[ind_x], pli)
                    pli → one(T)
                end
            end
            @inbounds y[iy] *= el
            ~@routine
        end
    end

    # patches
    Base.zero(x::Tropical{GVar{T,GT}}) where {T,GT} =zero(Tropical{GVar{T,GT}})
    Base.zero(::Type{Tropical{GVar{T,T}}}) where T =
Tropical(GVar(zero(Tropical{T}).n, zero(T)))

    NiLang.AD.GVar(x::Tropical{T}) where T = Tropical(GVar{T,T}(x.n, zero(T)))

    function NiLangCore.deanc(x::T, v::T) where T<:Tropical
        x === v || NiLangCore.deanc(content(x), content(v))
    end
end
```

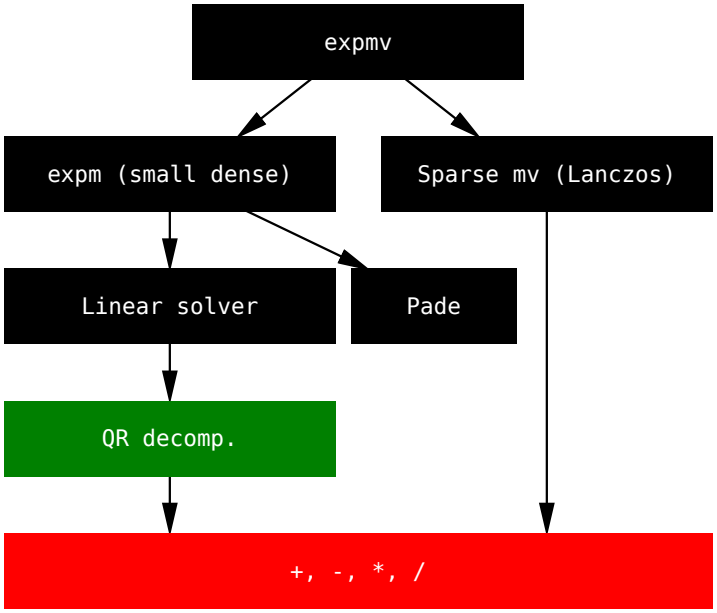
```
TropicalNumbers.TropicalF64[0.0ₜ, 1.0ₜ]
```

```
vertex_tensor(ones(TropicalF64,2), 1, Tropical(1.0))[1]
```

```
2×2 Matrix{TropicalF64}:
 0.0ₜ   0.0ₜ
 0.0ₜ  -Infₜ
```

```
bond_tensor(ones(TropicalF64,2, 2))
```

4. Towards differentiating `expmv`



Resouces

- **Differentiating sparse operations**
- **How to compute expm**

More interesting AD examples

- **Gate based quantum simulation**, Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design, Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, Lei Wang **arxiv: 1912.10877**
- **Reverse time migration**, Reverse time migration with optimal checkpointing, William W. Symes **DOI**
- **Gaussian mixture models, Bundle adjustment and hand tracking**, A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning, Filip Srajer,Zuzana Kukelova &Andrew Fitzgibbon **DOI**

Videos

- **Transformations & AutoDiff | MIT Computational Thinking Spring 2021 | Lecture 3**
  ◄ AD in image processing

Quick Summary

- Every program is differentiable
- For packages implementing AD rules on tensors, they have problems handling effective codes.