SciBmad: A full-featured ecosystem for modern, differentiable accelerator physics simulations

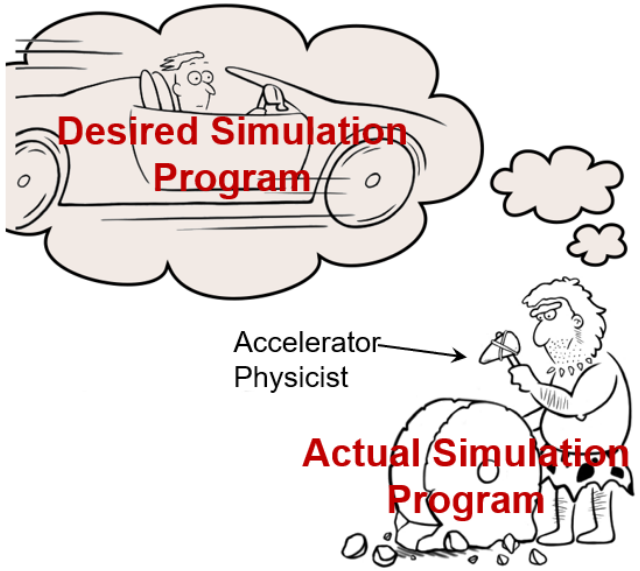
Matt Signorelli, David Sagan

14th International Computational Accelerator Physics Conference
Germany, October 2-5, 2024

Accelerator Software Wish-List

SciBMAD

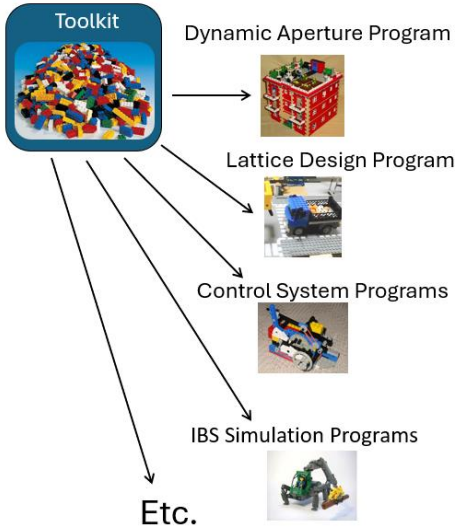
- ❑ **Modularity!!**
  - Maximal code re-use, minimal reinventing the wheel
  - **Plug-and-play** different optimizers, symplectic integrators, tracking methods, etc. with ease



- ❑ **Runs optimally on all architectures, with CPU & GPU parallelization**

- ❑ **Easy to use and integrate with other programs/tools**

- ❑ ***Fully differentiable* using automatic differentiation**
  - Fast, accurate calculation of gradients for optimizations and machine learning using **forward and backward differentiation**



- ❑ **Full featured accelerator software toolkit**
  - Linear and nonlinear tracking, nonlinear parametric normal forms including spin, Bmad's advanced lattice design tools (e.g. superposition, multipass), etc

Can we have *all* of this? Enter: *SciBmad*

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

SciBmad: What it is (and isn't)

- **SciBmad (formerly called Bmad-Julia) is NOT**
  - × **A rewrite of the current Bmad in a different programming language.** No Fortran code in SciBmad
  - × **An interface to the current Bmad.** **Lattice translation between the two will exist though!**
  - × **The end of the current Bmad.** Maintenance development of the current Bmad will continue

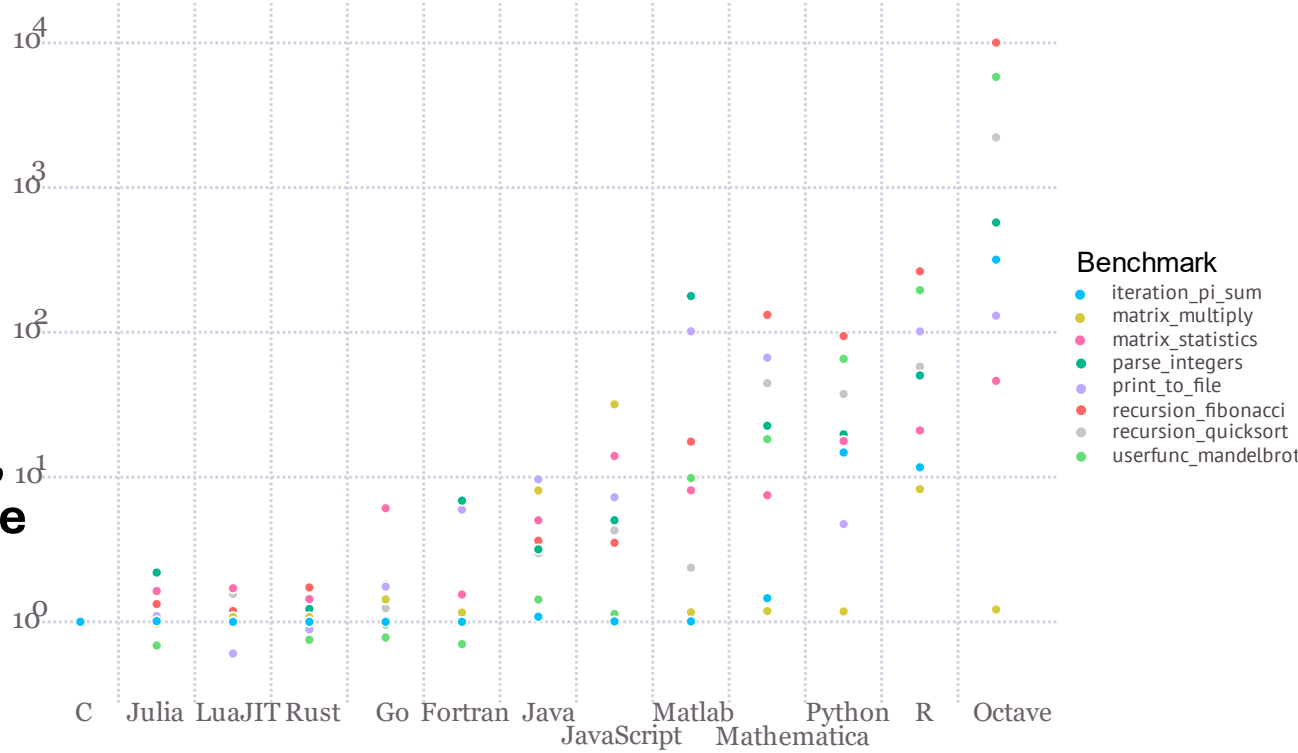
- **SciBmad is**
  - ✓ **Inspired by the experience (both good and bad) with developing the current Bmad**
  - ✓ **A new software ecosystem for modern, differentiable accelerator physics simulations**
  - ✓ **Written fully in the julia programming language**

- By leveraging the julia programming language, SciBmad will achieve **all points on the wish-list!**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

julia ? What's that?

- **Julia is a high-level, HPC language that "walks like Python, runs like C"**
  - As simple as Python, but as fast as C
- Adopts **multiple dispatch** and **just-in-time (JIT)** compilation as central paradigms
  - **Where types are inferable at compile-time, it will be compiled (using LLVM toolkit), else dynamically-dispatched with runtime type**
- **Features a powerful type system for highly-polymorphic code**



Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

The Power of Multiple Dispatch and JIT

- Universal polymorphism in Julia is *easy* and *fast!* **Consider the function:**

```
function track_drift(z0, L)
  zf = similar(z0)

  zf[1] = z0[1]+z0[2]*L/(1.0+z0[6])
  zf[2] = z0[2]
  zf[3] = z0[3]+z0[4]*L/(1.0+z0[6])
  zf[4] = z0[4]
  zf[5] = z0[5]-L*((z0[2]^2)+(z0[4]^2))/(1.0+z0[6])^2/2.0
  zf[6] = z0[6]
  return zf
end
```

- Generally pass:

```
z0 = Vector{Float64}(...)
track_drift(z0, L)
```

- However, say we'd instead like to track a Taylor map of Truncated Power Series (TPS) defined in some other package. Just pass:

```
z0 = Vector{TPS64}(...)
track_drift(z0, L)
```

That's it!
JIT compiled → fast!

- In fact, we can use types from *any* Julia (AD) package
  - *E.g. Dual numbers and "tapes" in* ForwardDiff.jl, ReverseDiff.jl, Enzyme.jl, Zygote.jl, etc etc.
  - For fun, we can even use Symbolics.jl (compiled!):

```
using Symbolics
@variables z0[1:6] L # creates symbolic vars
track_drift(z0, L)
```

- **Multiple dispatch and JIT compilation enable massive *composability* of packages**
  - ***Plug-and-play (and differentiate) to your heart's desire!***

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

SciBmad: Because we are lazy!

- We just showed how easy it is in Julia to use *any* 'number' type defined by *any* package
- In fact, we also can use *any optimizer, any (symplectic) integrator, plotting package, architecture-specific parallelization, etc.* written by other people in Julia **with minimal effort!**
  - **SciBmad offloads the work from the accelerator physicists to other experts**

- E.g. suppose you have a tracking function, and you want the closed orbit:

```
function track(ring, z0)
  ...
  return zf  # Final phase space position
end
```

- We can use any optimizer, written by ***other people***, immediately. E.g. Optim:

```
using Optim
optimize(z->norm(z-track(ring, z)), [0,0,0,0,0,0])
```

- Or NLSolve, etc.

```
using NLsolve
nlsolve(z->norm(z-track(ring, z)), [0,0,0,0,0,0])
```

- **This will be JIT compiled + fast too!**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

SciBmad: Because we are lazy!

- **Many forward/backward autodiff packages available**
  - ForwardDiff.jl, ReverseDiff.jl, Enzyme.jl, Zygote.jl
  - Our ecosystem will be compatible with all such

- **We also don't need to write any symplectic integrators DifferentialEquations.jl already has many *differentiable* ones:**

- **All of Julia's plotting packages at one's fingertips**
  - Makie.jl, Plots.jl, PyPlot.jl, etc

- **Powerful scientific ML tools: https://sciml.ai/**

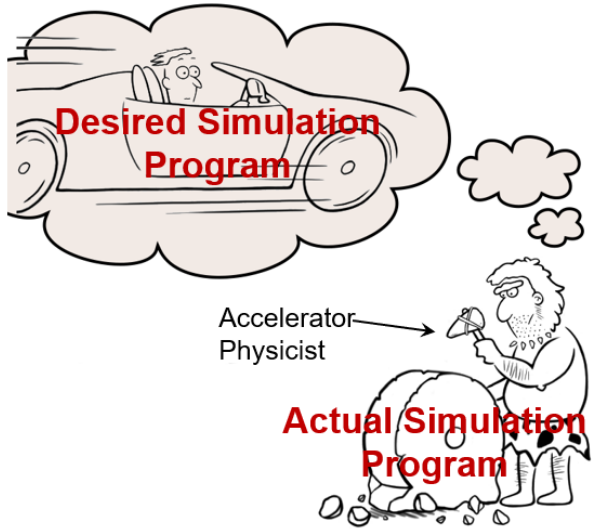
- **GPU parallelization using CUDA.jl type CuArray (so long as structure-of-arrays used on CPU)**
  - Universally polymorphic functions that work on both CPU and GPU!

- ***Lattice definition itself* in the Julia programming language**

Symplectic Integrators

Note that all symplectic integrators are fixed timestep only.

- `SymplecticEuler`: First order explicit symplectic integrator
- `VelocityVerlet`: 2nd order explicit symplectic integrator. Requires `f_2(t,u) = v`, i.e. a second order ODE.
- `VerletLeapfrog`: 2nd order explicit symplectic integrator.
- `PseudoVerletLeapfrog`: 2nd order explicit symplectic integrator.
- `McAte2`: Optimized efficiency 2nd order explicit symplectic integrator.
- `Ruth3`: 3rd order explicit symplectic integrator.
- `McAte3`: Optimized efficiency 3rd order explicit symplectic integrator.
- `CandyRoz4`: 4th order explicit symplectic integrator.
- `McAte4`: 4th order explicit symplectic integrator. Requires quadratic kinetic energy.
- `CalvoSanz4`: Optimized efficiency 4th order explicit symplectic integrator.
- `McAte42`: 4th order explicit symplectic integrator. (Broken)
- `McAte5`: Optimized efficiency 5th order explicit symplectic integrator. Requires quadratic kinetic energy.
- `Yoshida6`: 6th order explicit symplectic integrator.
- `KahanLi6`: Optimized efficiency 6th order explicit symplectic integrator.



CLASSE Cornell Laboratory for Accelerator-based ScienceS &Education | Brookhaven National Laboratory
14th ICAP Conference
Germany, October 2-5, 2024
Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

But I love Python and refuse to learn Julia!

- That's ok: we will have a full-featured Python interface to the (fast) SciBmad Julia ecosystem
- Gradients output for use with **PyTorch** and **Xopt** will be simple
- All other features will also be available, except the lattice definition will always be in Julia (with translators from other formats available)
- **For Julia users, the experience will be fantastic. For Python users, it will be as good as the usual two-language experience: fast underlying library, with a full-featured Python wrapper for ease of use**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Current SciBmad Ecosystem Status

Stable, officially-registered, ready for use:

- **GTPSA.jl**: Julia interface to L. Deniau's Generalised Truncated Power Series Algebra library

In development:

- **AcceleratorLattice.jl**: Accelerator lattice definition/manipulation
- **NonlinearNormalForm.jl**: Parametric nonlinear normal forms and analysis of DA maps including spin using Lie algebraic methods
- **AtomicAndPhysicalConstants.jl**: Atomic/subatomic particle properties and other physical constants for simulations

Starting development:

- **BeamTracking.jl**: CPU/GPU particle tracking methods/interfaces

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

AcceleratorLattice.jl

Accelerator lattice construction and manipulation is done using the Julia language itself:

```
using AcceleratorLattice

# Returns a FODO cell with specified quad strength
function FODO(k1)
  @eles begin
    qf = Quadrupole(L = 0.6, Kn1 = k1)
    d  = Drift(L = 0.4)
    qd = Quadrupole(L = -0.6, Kn1 = -k1)
  end
  return BeamLine([qf, d, qd, d])
end

@ele begin0 = BeginningEle(pc_ref = 1e7, species_ref = Species("electron"))

# Construct a BeamLine using Julia functions!
my_beamline = BeamLine([begin0, FODO(0.36), FODO(0.30)])
my_lat = Lat([my_beamline])
```


Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

AcceleratorLattice.jl

Fully-featured **lattice elements:**

```
julia> show(lat["qf"][1])
Ele: "qf" (b1>>2)   Quadrupole
  AlignmentGroup:
    offset                [0.0, 0.0, 0.0] m          offset_tot           [0.0, 0.0, 0.0] m
    x_rot                 0 rad                      x_rot_tot            0 rad
    y_rot                 0 rad                      y_rot_tot            0 rad
    tilt                  0 rad                      tilt_tot             0 rad
  BMultipoleGroup:
    Order Integrated              Tilt (rad)
        1      false                     0.0                      0.34  Kn1                      0.0  Ks1 (1/m^2)
                                                -0.011341179236737171  Bn1                     -0.0  Bs1 (T/m^1)
  FloorPositionGroup:
    r (r_floor)           [0.0, 0.0, 0.0] m
    q (q_floor)           1.0 + 0.0·i + 0.0·j + 0.0·k
    theta (theta_floor)   0.0 rad
    phi (phi_floor)       0.0 rad                    psi (psi_floor)      0.0 rad
  LengthGroup:
    L                     0.6 m                      orientation          1
    s                     0.0 m                      s_downstream         0.6 m
  ReferenceGroup:
    species_ref           Species("electron")        species_ref_exit     Species("electron")
    pc_ref                1.0e7 eV                   pc_ref_exit          1.0e7 eV
    E_tot_ref             1.000000005e7 eV           E_tot_ref_exit       1.000000005e7 eV
    ... Etc ...
```


Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

NonlinearNormalForm.jl

Real and complex parametric DAMaps including spin and coasting beam, for example:

```
julia> m   # Variable m is a map which has 6 variables and 2 parameters in this case
DAMap{Vector{ComplexF64}, Vector{ComplexTPS64}, Quaternion{ComplexTPS64}, Nothing, Bool}:
-----------------------
# Variables:        6
Maximum order:      3
# Parameters:       2
Parameter order:    3
-----------------------
Reference Orbit Vector{ComplexF64}:
1:    0.0 + 0.0im
2:    0.0 + 0.0im
3:    0.0 + 0.0im
4:    0.0 + 0.0im
5:    0.0 + 0.0im
6:    0.0 + 0.0im

Last plane is coasting: variable #6 is constant
Orbital Ray Vector{ComplexTPS64}:
  Out  Real                        Imag                          Order   Exponent
-------------------------------------------------------------------------------------------
    1:   1.0016106150325099e+00   0.0000000000000000e+00       1       1   0   0   0   0   0   |   0   0
    1:   7.9725998724008134e-03   0.0000000000000000e+00       1       0   1   0   0   0   0   |   0   0
⋮              ⋮                             ⋮              ⋮  ⋮   ⋮  ⋮   ⋮   ⋮   ⋮   ⋮   ⋮   ⋮   ⋮   ⋮   ⋮
                                                                                  596 rows omitted
```

Full customizable # variables and # parameters

Spin!

Coasting beam supported if wanted

Variables

Parameters

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

NonlinearNormalForm.jl

All the following tools are implemented already:

```
julia> m_lin = cutord(m, 2); # extract the linear part in orbital

julia> m_nonlinear = inv(m_lin) ∘ m; # remove the linear part

julia> F = log(m_nonlinear); # Get the Lie operator (including quaternion) generating nonlinear part

julia> m = m_lin ∘ exp(F);  # Reconstruct same map using Lie exponent and linear part separately

julia> a = normal(m);  # Calculate the nonlinear (parametric) normalizing canonical transformation

julia> R_z = inv(a) ∘ m ∘ a; # Nonlinear amplitude-dependent rotation in regular phase space (x, px, …)

julia> c = to_phasor(m);  # Get the transform to phasors basis √(J)*exp(±im*φ)

julia> R_J = inv(c) ∘ R_z ∘ c; # Nonlinear amplitude-dependent rotation in phasors basis

julia> a_spin, a0, a1, a2 = factorize(a); # Spin part, nonlinear parameter-dependent fixed point, a1, a2

julia> Σ = equilibrium_moments(m, a); # Calculate equilibrium sigma matrix when fluctuation-dissipation

julia> a = normal(m, m=[0; 1], m_spin=[-1]);  # Leaving in a Q_y - Q_spin resonance
```


Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Conclusions

SciBmad is a modern accelerator physics ecosystem which will provide:

☑ **Modularity!!**
- Maximal code re-use, minimal reinventing the wheel
- **Plug-and-play** different optimizers, symplectic integrators, tracking methods, etc. with ease

☑ **Runs optimally on all architectures, with CPU & GPU parallelization**

☑ **Easy to use and integrate with other programs/tools**

☑ ***Fully differentiable* using automatic differentiation**
- Fast, accurate calculation of gradients for optimizations and machine learning **using forward and backward differentiation**

☑ **Full featured accelerator software toolkit**
- Linear and nonlinear tracking, nonlinear parametric normal forms including spin, Bmad's advanced lattice design tools (e.g. superposition, multipass), etc

- **Goal: first accelerator simulations by end of year**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Contributors and Thank You!

- Development of SciBmad is currently in full-gear, and would not be this far along without the help of many

  - **Dan Abell (BeamTracking.jl and AtomicAndPhysicalConstants.jl)**
  - J. Scott Berg
  - **Oleksii Beznozov (BeamTracking.jl - GPU)**
  - **Alex Coxe (AtomicAndPhysicalConstants.jl)**
  - Laurent Deniau
  - Auralee Edelen
  - **Etienne Forest (significant help with NonlinearNormalForm.jl)**
  - Juan-Pablo Gonzalez
  - Georg Hoffstaetter de Torquat
  - Gavin Hunsche (BeamTracking.jl)
  - Lixing Li (AtomicAndPhysicalConstants.jl)
  - Chris Mayes
  - Ryan Roussel
  - **David Sagan (AcceleratorLattice.jl)**
  - **Matt Signorelli (NonlinearNormalForm.jl and GTPSA.jl)**
  - Sophia Yang (BeamTracking.jl)

- Open to more collaborators!

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Thank you!
Questions?

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Backup Slides

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

C++: Single Dispatch

```
class A { };
class B : public A { };
class C : public A { };

class Foo {
  virtual void my_fun(A* arg1, A* arg2) { std::println("A,A"); }
  virtual void my_fun(B* arg1, B* arg2) { std::println("B,B"); }
  virtual void my_fun(C* arg1, B* arg2) { std::println("C,B"); }
  virtual void my_fun(B* arg1, C* arg2) { std::println("B,C"); }
  virtual void my_fun(C* arg1, C* arg2) { std::println("C,C"); }
};

void call_my_fun(A* arg1, A* arg2) {
  Foo *pFoo = new Foo;
  pFoo->my_fun(arg1, arg2);  // SINGLE DISPATCH: prints “A,A”
}

int main() {
  A* arg1 = new B();
  A* arg2 = new C();

  call_my_fun(arg1, arg2);  // prints “A,A”
  return 0;
}
```

- **C++ only uses *single dispatch*, so this will print “A,A” even though both are B,C**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)

Julia: Multiple Dispatch

```julia
abstract type A end
struct B <: A end
struct C <: A end

my_fun(arg1::A, arg2::A) = println("A,A")
my_fun(arg1::B, arg2::B) = println("B,B")
my_fun(arg1::C, arg2::B) = println("C,B")
my_fun(arg1::B, arg2::C) = println("B,C")
my_fun(arg1::C, arg2::C) = println("C,C")


function call_my_fun(arg1::A, arg2::A)
  return my_fun(arg1, arg2)
end

arg1 = B()
arg2 = C()

call_my_fun(arg1, arg2)   # prints B,C
```

- **Julia will *dynamically-dispatch* based on the *runtime types***
- **Call to *my_fun* done by going to method lookup table using *runtime types*, and JIT-compiling *my_fun* if not already compiled**

Matt Signorelli (mgs255@cornell.edu)
David Sagan (dcs16@cornell.edu)