Stellarator Optimization with Julia, part 2

Benjamin Faber

December 4, 2020

Goal: A New Toolchain for Stellarator Optimization

- Written primarily in the Julia language
  - Utilize the flexibility of Julia and the Julia package ecosystem while leveraging the native performance of Julia
- Flexible and modular: adding/subtracting physics components should not break the implementation
  - Add different equilibrium solvers, GPU vs. CPU physics targets...
- Physics targets should be easily accessible as standalone functions from the Julia REPL to perform detailed analysis on a faster timescale
  - *Interact* with the data, rather than compile & run

Stellarator Optimization Components

- Stellarator optimization requires the following components:
  - Resource manager (MPI processors, distributed processors, threads...)
  - Equilibrium solvers (VMEC, SPEC...)
  - Optimization infrastructure/context (MANGO, Optim.jl...)
  - Geometry functions (Coordinate vectors, ∇B, metric components,...)

`https://gitlab.com/wistell`

- PlasmEquilibriumToolkit.jl (PET.jl)
- VMEC.jl
- lasso.jl

PlasmaEquilibriumToolkit.jl

- Package for defining, manipulating geometric quantities from 3D magnetic equilibria
- Currently provides the following utilities
  - Abstract types for magnetic equilibria and coordinates:
    `MagneticEquilibrium, MagneticCoordinates`
  - Coordinate systems:
    `{PestCoordinates,BoozerCoordinates,...} <: MagneticCoordinates`
  - Interfaces with CoordinateTransformations.jl:
    `{CylindricalFromPest,PestFromFlux,...} <: Transformation`
  - Basis Vectors:
    - Contravariant: $\nabla\psi$, $\nabla\alpha$, $\nabla\theta, \ldots$
    - Covariant: $\mathbf{e}_\psi$, $\mathbf{e}_\alpha, \ldots$
  - Vector Field Transformations:
    `{ContravariantFromCovariant,...} <: Transformation`
  - Abstractions for computing field quantities like ∇B, κ_n, κ_g

- Abstract types provide a convenient way to extend methods to different equilibrium types:

```
function covariant_basis(t::Transformation,x::AbstractArray{T},eq::MagneticEquilibrium)
  where T <: MagneticCoordinates
  res = Array{BasisVectors{typeof(getfield(first(x),1))},ndims(x)}(undef,size(x))
  map!(i->covariant_basis(t,i,eq),res,x)
  return res
end
```

- Immediately implemented by new equilibrium solver (e.g. SPEC) and/or new magnetic coordinates (e.g. Hamada)

VMEC.jl

- Package for interfacing with VMEC equilibria
- Provides different data structures for VMEC data
  `{VmecData,Vmec,VmecSurface} <: MagneticEquilibrium`
  - Uses cubic spline interpolation to compute surface dependence
- Defines VMEC-specific coordinates and coordinate transformations
  `VmecCoordinates <: MagneticCoordinates`
  `PestFromVmec, BoozerFromVmec,... <: Transformation`
- Read NetCDF files, write HDF5 files
- Fortran interface implemented to run PARVMEC from Julia, retrieve results emphin memory

Example: Coordinate Transformation

- Compute the VMEC coordinates for a field line at $s = 0.5$

```
julia> wout = NetCDF.open("wout_aten.nc"); vmec, vmec_data = readVmecWout(wout);
julia> s = 0.5; psi = s*vmec.phi(1.0)/(2*pi);
julia> alpha = 0.0; zeta = -pi:pi/2:pi; #Define PEST coordinates
julia> vmec_surface = VmecSurface(s,vmec); #Extract the VMEC data at s = 0.5
julia> pc = PestCoordinates(psi,alpha,zeta); # Generates a vector of PEST Coordinates
julia> vc = VmecFromPest()(pc,vmec_surface) # Generates the VMEC coordinates
5-element Array{VmecCoordinates{Float64,Float64},1}:
VmecCoordinates(s=0.5, theta=-3.485668445213474, zeta=-3.141592653589793)
VmecCoordinates(s=0.5, theta=-1.6411857501972285, zeta=-1.5707963267948966)
VmecCoordinates(s=0.5, theta=0.0, zeta=0.0)
VmecCoordinates(s=0.5, theta=1.6411857501972282, zeta=1.5707963267948966)
VmecCoordinates(s=0.5, theta=3.485668445213474, zeta=3.141592653589793)
```

Example: Quasisymmetry

- From ATEN equilibrium, compute quasisymmetry deviation on each surface
  - Specify $m_{QS} = 1$, $n_{QS} = 4$ with $m_{Boozer} = n_{Boozer} = 8$
  - Use 32 hardware threads

```
julia> wout = NetCDF.open("wout_aten.nc"); vmec, vmec_data = readVmecWout(wout);
julia> vmec_surfaces = map(s->VmecSurface(s,vmec),0.02:0.02:0.98); # Selects the surfaces
julia> @btime quasisymmetry(1,4,8,8,vmec_surfaces)
779.157 ms (28077361 allocations: 1.80 GiB)
49-element Array{Float64,1}:
0.00847798155710977
0.00814361171219873 6
0.00786245074771378 8
...
```

- Fast, but memory intensive calculation due to the Boozer transformation

Lightweight Architecture for Scalable Stellarator Optimization (lasso)

- Glue layer for stellarator optimization specific problems
  - Expose different optimization libraries and methods (MANGO, Optim.jl,…)
  - Provide standardized interface for computing physics targets, defining optimization variables
  - Define submodules for specific implementations, currently:
    - VmecOpt: functions for translating VMEC quantities
    - Mango: interface functions for Mango
  - Records optimzation variables and targets in top-level dictionaries:
    - `lassoOptVariables`, `lassoOptTargets`

Specifying Targets

- Two different types of physics target, subtypes of `AbstractOptTarget`:

```
struct Target{TargetType,WeightType} <: AbstractOptTarget
  target::TargetType  # The target value
  weight::WeightType  # The target weight
  inputArgs::Dict{Symbol,Any} # Inputs for target calculation
  mod::Module # Module containing target code
end
```

```
struct LeastSquaresTarget{TargetType,SigmaType,WeightType} <: AbstractOptTarget
  target::TargetType # The target value
  sigma::SigmaType # The target sigma in a LSQ formulation
  weight::WeightType # The target weight
  inputArgs::Dict{Symbol,Any} # Inputs for target calculation
  mod::Module # Module containing target code
end
```

Specifying Targets

- Macros implemented to define optimization targets
- Arguments for the target can be passed as Julia `Expressions`; the last expression is always the target `Module`

  `@target(value,weight,exprs...)`

  `@leastsquarestarget(value,sigma,weight,exprs...)`
  - Invocation adds an entry to `lassoOptTargets`
- General routine for objective function:

```
function objective(args...)
  objFun = 0.0
  for (index,target) in lassoOptTargets
      objFun += target.weight*(target.mod.computeTarget(target.inputArgs,args...)
                               - target.target)
  end
  return objFun
end
```

- `args...` are implementation-specific arguments (e.g. VMEC/SPEC equilibrium, MPI communicator, threads, etc.)

Specifying Variables

- Macros implemented to define optimization variables
  - Invocation adds an entry to `lassoOptVariables`
- Example - `VmecOpt` input:
  - Specify *any* VMEC input variable
    `@VmecVariable(name[,value,weight=1.0,recompute=true])`
  - Default value taken from input file

```
&VmecOpt
inputFile = /home/bfaber/projects/julia/VMEC.jl/input.aten
vmecLib = /home/bfaber/projects/VMEC2000/lib/libvmec.so
@VmecVariable(rbc(0,0))
@VmecVariable(zbs(0,3))
@VmecVariable(zbs(-3,1))
@VmecVariable(rbc(-2,1))
...
```

Flexiblity at runtime

- `lasso` links to geometry, target modules at dynamically
- Parses input file, records `Module` information as a `Dict`
- '`&Module`' similar to Fortran namelist
- Example:

```
&VMEC.QuasiSymmetry
mQS = 1
nQS = 4
mBoozer = 16
nBoozer = 16
@leastsquarestarget(0.0,1.0,1.0,surfaceLabel=0.3)
@leastsquarestarget(0.0,1.0,1.0,surfaceLabel=0.4)
```

```
julia> lassoOptTargets
Dict{Int64,AbstractOptTarget} with 2 entries:
  1 => LeastSquaresTarget(0.0,1.0,1.0,Dict{Symbol,Any}(:mQS=>1,:nQS=>4,:mBoozer=>16,
                          :nBoozer=>16,:surfaceLabel=>0.3),VMEC.QuasiSymmetry)
  ...
```

Optimization with MPI and MANGO

- Multiple routes for parallelism: fine-grained threads, independent distributed processes, MPI
  - VMEC currently paralleized via MPI, natural to use MPI-based optimization
- Wrap MANGO as a callable library using CxxWrap.jl $\Rightarrow$ MANGO optimization objects can be managed from Julia
  - Immediately exposes number of MPI-based optimization algorithms
- Idea: MANGO controls the optimization context, but compute the residual function in *Julia*
  - Writing target functions in Julia allows us to leverage packages from the Julia ecosystem, for example:
    - Spline libraries, root finding algorithms, adaptive integration (Gauss-Konrad, Double Expoenential…)
  - Leverage advanced features like automatic differentiation, GPU interfaces in future applications

Optimization with MPI and MANGO

```
function Mango.optimize()
  ...
  #Setup optimization problem
  ...
  prob = MangoWrapper.Least_squares_problem(nPar,stateVec,nTerms,targets,sigmas,best_func)
  residualFunction_cxx = @safe_cfunction(residualFunction,Cvoid,
    (CxxPtr{Cint},ConstCxxPtr{Cdouble},CxxPtr{Cint},
    CxxPtr{Cdouble},CxxPtr{Cint},
    CxxPtr{MangoWrapper.Problem},Ptr{Cvoid}))
  MangoWrapper.set_residual_function(prob,residualFunction_cxx)
  if head
    best_residual = MangoWrapper.optimize(prob)
    MangoWrapper.stop_workers(prob)
  else
    worker(prob)
    best_residual = nothing
  end
  best_residual = MPI.bcast(best_residual,0,comm)
  return best_residual
end
```

Optimization with MPI and MANGO: computing residual

- Group leader:

```
function residualFunction(nPar,stateVecPtr,nTerms,resPtr,failed,probPtr,userData)
  comm = MangoWrapper.get_comm_worker_groups(probPtr) # Get the MPI group comm
  stateVec = unsafe_load(stateVecPtr) # Get state vector from mango
  MPI.Bcast!(stateVec,0,comm) # Broadcast the state vector to worker group
  translate!(stateVec) #Translates between state vector entries and VMEC inputs
  vmec = VMEC.run_vmec(comm,VmecOpt.vmecInput,VmecOpt.vmecLib) # Run VMEC
  residual = compute_residual(vmec,comm) # Compute residual using Julia functions
  unsafe_store!(residualPtr,residual) # Pass residual to mango
end
```

- Group worker:

```
function worker(prob::MangoWrapper.Problem)
  comm =  MangoWrapper.get_comm_worker_groups(prob)
  stateVector = Vector{Float64}(undef,nParameters)
  while MangoWrapper.continue_worker_loop(prob)
    MPI.Bcast!(stateVector,0,comm) # Wait for the next state vector
    translate!(stateVector)
    vmec = VMEC.runVmec(comm,vmecInput,vmecLib) # Run VMEC on group comm
    compute_residual(vmec,comm) # Compute residual using Julia functions
  end
end
```

Optimization with MPI and MANGO: computing residual

- Computing residual function particularly simple. In Mango:

```
function compute_residual(vmec::Vmec)
  residual = Vector{Float64}(undef,length(lassoOptTargets))
  if !isnan(vmec.iota(0.99)) && !isapprox(vmec.iota(0.99),0.0) && !isinf(vmec.iota(0.9
    residual = lasso.computeTarget(vmec)
  else
    residual = fill(1e10,length(lassoOptTargets))
  end
end
```

- In lasso:

```
function computeTarget(vmec)
  targetValues = Vector{Float64}(undef,length(lassoOptTargets))
  for (key, target) in lassoOptTargets
    # Use the computeTarget function defined in each target module
    targetValues[key] = target.mod.computeTarget(target.inputArgs,vmec)
  end
  return targetValues
end
```

Writing Physics Targets

- Physics targets are wrapped in a `Module`
  - `Module` must provide the `computeTarget` function:
    `function computeTarget(inputArgs::Dict{Symbol,Any},args...)`
  - The first argument is always a dictionary with the necessary input parameters
  - Optionally provide `parseInputLines` function to read from input file

```
module VMEC.QuasiSymmetry
using VMEC
#Specialized function for computing QS on a VmecSurface
function computeTarget(args::Dict{Symbol,Any},vmecSurface::VmecSurface{N,T}) where {N,T}
  if args[:surfaceLabel] != vmecSurface.s
    throw(error("Input surface label and VmecSurface label do not match!"))
  end
  return quasisymmetry(args[:mQS],args[:nQS],args[:mBoozer],args[:nBoozer],vmecSurface)
end
# More complicated function for computing quasisymmetry target
function quasisymmetry(m,n,mBoozer,nBoozer,vmecSurface)
  ...
end
end
```

Writing Physics Targets

- Simple targets, like $\iota(s)$, need to also be wrapped:

```
module VMEC.RotationalTransform
using VMEC
function computeTarget(args::Dict{Symbol,Any},vmec::Vmec)
  return vmec.iota(args[:surfaceLabel])
end
end
```

- Slightly more overhead needed for simple targets, but provides a common way of computing physics targets regardless of form (simple VMEC quantity vs. complex functional dependence)

Current Status + Future Directions

- VMEC-based optimization calculations can now be performed with Julia-defined physics targets
  - Tested with quasisymmetry, rotational transform, Γ_c targets
  - More extensive testing needed: parallelized target evaluation, running on a cluster...
- Next steps:
  - Adding MPI-based turbulence saturation metric
  - Writing *a lot* of documentation and examples
  - Implementing unit tests and code coverage metrics
  - Getting feedback from **you**