Enabling and Embedding Reverse Communication Solvers

Andy Greenwell
Julia Computing, Inc.
andy@juliacomputing.com

June 22nd, 2016
JuliaCon 2016 – Massachusetts Institute of Technology

Solicitation Notice

Julia Computing, Inc. regularly assists clients in developing custom Julia functionality tailored to their specific application and operational needs

This presentation is a synopsis of one such engagement presented with client permission

Notebook and code: https://github.com/AndyGreenwell/JuliaCon2016

Contact us if we can help your organization

Project Requirements

- Provide Julia alternative to existing optimization and root finding solvers
- Wrap client's existing solver with a Julia interface
- Integrate Julia in a large C/C++ batch processing workflow (call Julia from C)
- Add reverse communication to Julia solvers, including wrapped solvers
  - Client has many existing C/C++ based objective and derivative functions:
    - Must not be passed as arguments to Julia functions (do not require wrapping objective/derivative)
    - Must be evaluated directly in original C/C++ application
- Enable code migration between front and back office without full re-coding

Forward vs Reverse Communication Solvers

- Forward communication is utilized in most of Julia's optimization and root finding
  - Examples: Optim.jl, NLopt.jl, Roots.jl, NLsolve.jl
  - Objective/derivative function(s) are input arguments
  - Solver invokes the objective/derivative function(s) internally
  - Technique allows for powerful performance enhancements (e.g. Automatic Differentiation)
  - Requires objective/derivative function(s) to have a particular interface
    - For existing C code, this can involve creation of Julia wrappers
    - Not desired for this particular client's environment

Forward vs Reverse Communication Solvers

- Forward communication is utilized in most of Julia's optimization and root finding
  - Examples: Optim.jl, NLopt.jl, Roots.jl, NLsolve.jl
  - Objective/derivative function(s) are input arguments
  - Solver invokes the objective/derivative function(s) internally
  - Technique allows for powerful performance enhancements (e.g. Automatic Differentiation)
  - Requires objective/derivative function(s) to have a particular interface
    - For C code, this can involve creation of Julia wrappers

- Reverse communication
  - Classic technique where objective/derivative functions are not passed directly to the solver
  - Solver returns to its calling environment when function evaluations are required
  - Solver is then called repeatedly, each time resuming at its prior exit location

Example of Forward Communication

```julia
# Rosenbrock objective and derivative functions as used by Optim.jl
function f(x::Vector)
    (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
end
function g!(x::Vector, storage::Vector)
    storage[1] = -2.0 * (1.0 - x[1]) - 400.0 * (x[2] - x[1]^2) * x[1]
    storage[2] = 200.0 * (x[2] - x[1]^2)
end
function h!(x::Vector, storage::Matrix)
    storage[1, 1] = 2.0 - 400.0 * x[2] + 1200.0 * x[1]^2
    storage[1, 2] = -400.0 * x[1]
    storage[2, 1] = -400.0 * x[1]
    storage[2, 2] = 200.0
end

@show results = Optim.optimize(f, [0.0, 0.0], LBFGS())
@show results = Optim.optimize(f, g!, [0.0, 0.0], LBFGS())
@show results = Optim.optimize(f, g!, h!, [0.0, 0.0], Newton())
```

Mechanical Transformation of FC to RC

Traditional transformation from forward to reverse communication solver:

- Trace execution of the forward solver
- Mark locations of each objective/derivative function evaluations
- Use a stateful object for storing position, objective, derivatives, and location
- Exit and resume solver at function call locations
- Calling the reverse communication solver repeatedly until hitting final condition
  - See notebook for an example mechanical transformation
  - Overall a tedious and error-prone development process

Using Julia tasks/co-routines for FC to RC

Using Julia tasks, a forward communication solver can behave like a reverse communication solver.

```julia
maintask = current_task()

function f_t(x::Vector)
    yieldto(maintask, x)
end

guess = [0.0; 0.0]

solver = @task Optim.optimize(f_t, guess, NelderMead())

next_x = copy(guess)
while !istaskdone(solver)
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                  # Define the main task

function f_t(x::Vector)
    yieldto(maintask, x)
end

guess = [0.0; 0.0]

solver = @task Optim.optimize(f_t, guess, NelderMead())

next_x = copy(guess)
while !istaskdone(solver)
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                            # Define the main task

function f_t(x::Vector)                              # Define a redirection function that
    yieldto(maintask, x)                             # yields control to the main task
end

guess = [0.0; 0.0]

solver = @task Optim.optimize(f_t, guess, NelderMead())

next_x = copy(guess)
while !istaskdone(solver)
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                  # Define the main task

function f_t(x::Vector)                                    # Define a redirection function that
    yieldto(maintask, x)                                   # yields control to the main task
end

guess = [0.0; 0.0]                                         # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead())


next_x = copy(guess)
while !istaskdone(solver)
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                   # Define the main task

function f_t(x::Vector)                                     # Define a redirection function that
    yieldto(maintask, x)                                    # yields control to the main task
end

guess = [0.0; 0.0]                                          # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver


next_x = copy(guess)
while !istaskdone(solver)
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                    # Define the main task

function f_t(x::Vector)                                      # Define a redirection function that
    yieldto(maintask, x)                                     # yields control to the main task
end

guess = [0.0; 0.0]                                           # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver

next_x = copy(guess)
while !istaskdone(solver)                                    # Driver loop
    fx = f(next_x)
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                              # Define the main task

function f_t(x::Vector)                                # Define a redirection function that
    yieldto(maintask, x)                               # yields control to the main task
end

guess = [0.0; 0.0]                                     # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver

next_x = copy(guess)
while !istaskdone(solver)                              # Driver loop
    fx = f(next_x)                                     # Objective function evaluation
    next_x = yieldto(solver, fx)
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                # Define the main task

function f_t(x::Vector)                                  # Define a redirection function that
    yieldto(maintask, x)                                 # yields control to the main task
end

guess = [0.0; 0.0]                                       # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver

next_x = copy(guess)
while !istaskdone(solver)                                # Driver loop
    fx = f(next_x)                                       # Objective function evaluation
    next_x = yieldto(solver, fx)                         # Yield back to the solver task
end
result = next_x

@show result
;
```

Using Julia tasks/co-routines for FC to RC

```julia
maintask = current_task()                                  # Define the main task

function f_t(x::Vector)                                    # Define a redirection function that
    yieldto(maintask, x)                                   # yields control to the main task
end

guess = [0.0; 0.0]                                         # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver

next_x = copy(guess)
while !istaskdone(solver)                                  # Driver loop
    fx = f(next_x)                                         # Objective function evaluation
    next_x = yieldto(solver, fx)                           # Yield back to the solver task
end
result = next_x                                            # On completion, next_x contains
                                                           # the solver output
@show result
;
```

Using Julia tasks/co-routines for FC to RC

```
maintask = current_task()                              # Define the main task

function f_t(x::Vector)                                # Define a redirection function that
    yieldto(maintask, x)                               # yields control to the main task
end

guess = [0.0; 0.0]                                     # Define an initial guess

solver = @task Optim.optimize(f_t, guess, NelderMead()) # Create a task for the solver

next_x = copy(guess)
while !istaskdone(solver)                              # Driver loop
    fx = f(next_x)                                     # Objective function evaluation
    next_x = yieldto(solver, fx)                       # Yield back to the solver task
end
result = next_x                                        # On completion, next_x contains
                                                       # the solver output
@show result
;
```

Presentation notebook includes an example using both gradient and combined objective/gradient functions

Embedding Julia tasks based RC solver in C

Julia's C API can be used to embed general Julia code within C/C++

- Used for calling Julia from other languages (e.g. Rjulia, pyjulia)

Tasks-based reverse communication solvers can be embedded within a C/C++ application that meets various client requirements:

- Provides Julian alternative to existing optimization and root finding solvers
- Integrates Julia in existing C/C++ batch processing workflow
- Provides reverse communication interface to Julia solvers
- Evaluates objective/derivative functions either as C/C++ functions or Julia functions
  - Enables easy migration of Julia code from front office to back office

Embedding Julia tasks based RC solver in C

```c
//rc.c

#include <julia.h>
#include <stdio.h>
#include <math.h>

#ifdef _OS_WINDOWS_
__declspec(dllexport) __cdecl
#endif

int r_f(jl_array_t *x, jl_array_t *f)
{
    double* xd = (double*)jl_array_data(x);
    double* fd = (double*)jl_array_data(f);

    fd[0] = pow(1.0 - xd[0], 2.0);
    fd[1] = 100.0 * pow(xd[1] - pow(xd[0], 2.0), 2.0);

    return 0;
}

int r_g(jl_array_t *x, jl_array_t *jac)
{
    double* xd = (double*)jl_array_data(x);
    double* jd = (double*)jl_array_data(jac);

    jd[0] = -2.0 * (1 - xd[0]);                                //1,1 element
    jd[1] = -400.0 * (xd[1] - pow(xd[0], 2.0)) * xd[0]; //2,1 element
    jd[2] = 0.0;                                               //1,2 element
    jd[3] = 200.0 * (xd[1] - pow(xd[0], 2.0));                 //2,2 element
    return 0;
}

void r_fg(jl_array_t *x, jl_array_t *f, jl_array_t *jac)
{
    double* xd = (double*)jl_array_data(x);
    double* fd = (double*)jl_array_data(f);
    double* jd = (double*)jl_array_data(jac);
    fd[0] = pow(1.0 - xd[0], 2.0);
    fd[1] = 100.0 * pow(xd[1] - pow(xd[0], 2.0), 2.0);

    jd[0] = -2.0 * (1 - xd[0]);                                //1,1 element
    jd[1] = -400.0 * (xd[1] - pow(xd[0], 2.0)) * xd[0]; //2,1 element
    jd[2] = 0.0;                                               //1,2 element
    jd[3] = 200.0 * (xd[1] - pow(xd[0], 2.0));                 //2,2 element
}

jl_value_t* yieldto(jl_value_t *solver, jl_value_t *vals)
{
    jl_function_t *yieldto_func = jl_get_function(jl_base_module, "yieldto");
    return jl_call2(yieldto_func, solver, vals);
}

int istaskdone(jl_value_t *solver)
{
    jl_function_t *taskdone = jl_get_function(jl_base_module, "istaskdone");
    return jl_call1(taskdone, solver) == jl_true;
}

#define __noinline __attribute__((noinline))
void __noinline real_main()
{
    int n = 2;

    // Represent arrays that will be created and owned by C
    double *guess    = (double*)malloc(sizeof(double)*n);
    double *fval     = (double*)malloc(sizeof(double)*n);
    double *jacobian = (double*)calloc(n*n, sizeof(double));

    // Assign initial values into the above arrays
    guess[0] = -1.2;
    guess[1] =  1.0;
    fval[0]  =  1.0/0.0;
    fval[1]  =  1.0/0.0;

    // Root objects with the GC
    jl_value_t **args;
    // args[0] - NLsolve.DifferentiableMultivariateFunction(f!, g!, fg!)
    // args[1] - :f
    // args[2] - :g
    // args[3] - :fg
    // args[4] - array_type
    // args[5] - guess::Vector{Float64}
    // args[6] - fval::Vector{Float64}
    // args[7] - jacobian::Matrix{Float64}
    // args[8] - tuple function
    // args[9] - solver_task function

    JL_GC_PUSHARGS(args,10);

    jl_eval_string("using NLsolve");
    jl_eval_string("maintask = current_task()");
    jl_eval_string("f_t!(x::Vector, fvec::Vector) = yieldto(maintask, (:f, x, fvec))");
    jl_eval_string("g_t!(x::Vector, fjac::Matrix) = yieldto(maintask, (:g, x, fjac))");
    jl_eval_string("fg_t!(x::Vector, fvec::Vector, fjac::Matrix) = yieldto(maintask, (:fg, x, fvec, fjac))");

    args[0] = jl_eval_string("NLsolve.DifferentiableMultivariateFunction(f_t!, g_t!, fg_t!)");
    args[1] = (jl_value_t*) jl_symbol("f");
    args[2] = (jl_value_t*) jl_symbol("g");
    args[3] = (jl_value_t*) jl_symbol("fg");
    args[4] = (jl_value_t*) jl_apply_array_type(jl_float64_type, 1 );

    // Wrap the C arrays as Julia arrays. The final "0" argument means
    // that Julia does not take ownership of the array data for GC purposes.
    args[5] = (jl_value_t*) jl_ptr_to_array_1d(args[4], guess, n, 0);
    args[6] = (jl_value_t*) jl_ptr_to_array_1d(args[4], fval, n, 0);
    args[7] = (jl_value_t*) jl_ptr_to_array_1d(args[4], jacobian, n*n, 0);
    args[8] = (jl_value_t*) jl_get_function(jl_base_module, "tuple");
    args[9] = jl_eval_string("solver_task(df, guess) = @task nlsolve(df, guess)");



    {
        jl_value_t* vals   = jl_call3((jl_function_t*) args[8], args[1], args[5], args[6]); // "vals" tuple
        jl_value_t* solver = jl_call2((jl_function_t*) args[9], args[0], args[5]);          // solver task

        JL_GC_PUSH2(&vals, &solver);

        jl_yield();
        while (!istaskdone(solver)){
            if (jl_get_nth_field(vals, 0) == args[1]) {        // :f case
                args[5] = jl_get_nth_field(vals, 1);
                args[6] = jl_get_nth_field(vals, 2);
                r_f((jl_array_t*) args[5], (jl_array_t*) args[6]);
                vals = yieldto(solver, vals);
            } else if (jl_get_nth_field(vals, 0) == args[2]) { // :g case
                args[5] = jl_get_nth_field(vals, 1);
                args[7] = jl_get_nth_field(vals, 2);
                r_g((jl_array_t*) args[5], (jl_array_t*) args[7]);
                vals = yieldto(solver, vals);
            } else if (jl_get_nth_field(vals, 0) == args[3]) { // :fg case
                args[5] = jl_get_nth_field(vals, 1);
                args[6] = jl_get_nth_field(vals, 2);
                args[7] = jl_get_nth_field(vals, 3);
                r_fg((jl_array_t*) args[5], (jl_array_t*) args[6], (jl_array_t*) args[7]);
                vals = yieldto(solver, vals);
            }
        }


        jl_show(jl_stderr_obj(), vals);
        jl_eval_string("println(\"\n\")");
        JL_GC_POP();
    }
    JL_GC_POP();

    free(guess);
    free(fval);
    free(jacobian);
}

void __noinline real_main_wrapper()
{
    real_main();
}

int main()
{
    jl_init(NULL);

    real_main(); // this should be a separate function from main()

    int ret = 0;
    jl_atexit_hook(ret);

    return ret;
}
```

Embedding Julia tasks based RC solver in C

Julia computing

```c
//rc.c

#include <julia.h>
#include <stdio.h>
#include <math.h>

#ifdef _OS_WINDOWS_
__declspec(dllexport) __cdecl
#endif
```

Include headers and definitions -

```c
void __noinline real_main_wrapper()
{
    real_main();
}

int main()
{
    jl_init(NULL);

    real_main_wrapper();   // This needs to be a separate function from main()

    int ret = 0;
    jl_atexit_hook(ret);

    return ret;
}
```

Wrap the real_main function (PR#13099) -

Define a main function -

Initialize Julia -

Call the tasks based function -

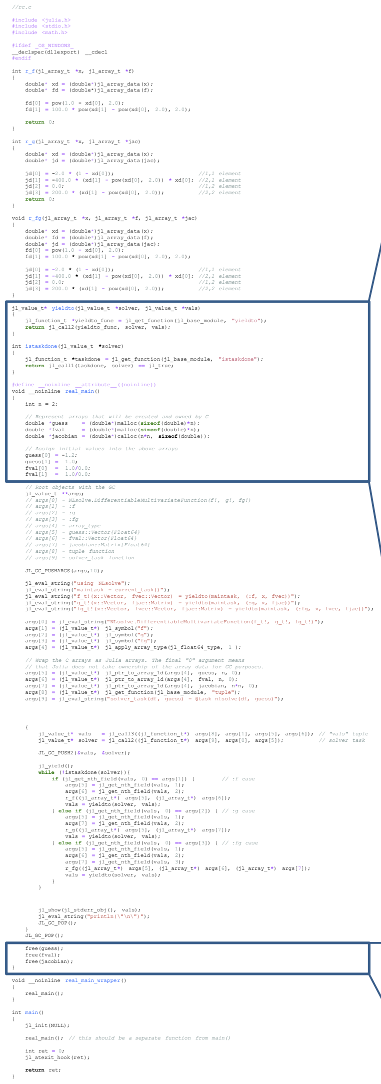
Shutdown and exit -

Embedding Julia tasks based RC solver in C

Define Objective and Gradient Functions -

```c
int r_f(jl_array_t *x, jl_array_t *f)
{
    double* xd = (double*)jl_array_data(x);
    double* fd = (double*)jl_array_data(f);

    fd[0] = pow(1.0 - xd[0], 2.0);
    fd[1] = 100.0 * pow(xd[1] - pow(xd[0], 2.0), 2.0);

    return 0;
}
int r_g(jl_array_t *x, jl_array_t *jac)
{
    double* xd = (double*)jl_array_data(x);
    double* jd = (double*)jl_array_data(jac);

    jd[0] = -2.0 * (1 - xd[0]);                            //1,1 element
    jd[1] = -400.0 * (xd[1] - pow(xd[0], 2.0)) * xd[0];  //2,1 element
    jd[2] = 0.0;                                           //1,2 element
    jd[3] = 200.0 * (xd[1] - pow(xd[0], 2.0));            //2,2 element
    return 0;
}
void r_fg(jl_array_t *x, jl_array_t *f, jl_array_t *jac)
{
    double* xd = (double*)jl_array_data(x);
    double* fd = (double*)jl_array_data(f);
    double* jd = (double*)jl_array_data(jac);
    fd[0] = pow(1.0 - xd[0], 2.0);
    fd[1] = 100.0 * pow(xd[1] - pow(xd[0], 2.0), 2.0);

    jd[0] = -2.0 * (1 - xd[0]);                            //1,1 element
    jd[1] = -400.0 * (xd[1] - pow(xd[0], 2.0)) * xd[0];  //2,1 element
    jd[2] = 0.0;                                           //1,2 element
    jd[3] = 200.0 * (xd[1] - pow(xd[0], 2.0));            //2,2 element
}
```

Embedding Julia tasks based RC solver in C



```
jl_value_t* yieldto(jl_value_t *solver, jl_value_t *vals)
{
    jl_function_t *yieldto_func = jl_get_function(jl_base_module, "yieldto");
    return jl_call2(yieldto_func, solver, vals);
}

int istaskdone(jl_value_t *solver)
{
    jl_function_t *taskdone = jl_get_function(jl_base_module, "istaskdone");
    return jl_call1(taskdone, solver) == jl_true;
}
```

Access functions from base Julia -

```
#define __noinline __attribute__((noinline))
void __noinline real_main()
{
    int n = 2;

    // Represent arrays that will be created and owned by C
    double *guess    = (double*)malloc(sizeof(double)*n);
    double *fval     = (double*)malloc(sizeof(double)*n);
    double *jacobian = (double*)calloc(n*n, sizeof(double));

    // Assign initial values into the above arrays
    guess[0] = -1.2;
    guess[1] =  1.0;
    fval[0]  =  1.0/0.0;
    fval[1]  =  1.0/0.0;

    // Body of real_main here
```

Define "real_main" -

Allocate C Arrays -

```
    free(guess);
    free(fval);
    free(jacobian);
}
```

Free C Arrays -

Embedding Julia tasks based RC solver in C

Root variables with Julia's GC -

Load required Julia package (NLsolve.jl) -

Define main and solver tasks -

Define redirection functions -

```c
// Root objects with the GC
jl_value_t **args;
// args[0] - NLsolve.DifferentiableMultivariateFunction(f!, g!, fg!)
// args[1] - :f
// args[2] - :g
// args[3] - :fg
// args[4] - array_type
// args[5] - guess::Vector{Float64}
// args[6] - fval::Vector{Float64}
// args[7] - jacobian::Matrix{Float64}
// args[8] - tuple function
// args[9] - solver_task function

JL_GC_PUSHARGS(args,10);

jl_eval_string("using NLsolve");
jl_eval_string("maintask = current_task()");
jl_eval_string("f_t!(x::Vector, fvec::Vector) = yieldto(maintask, (:f, x, fvec))");
jl_eval_string("g_t!(x::Vector, fjac::Matrix) = yieldto(maintask, (:g, x, fjac))");
jl_eval_string("fg_t!(x::Vector, fvec::Vector, fjac::Matrix) = yieldto(maintask, (:fg, x, fvec, fjac))");

args[0] = jl_eval_string("NLsolve.DifferentiableMultivariateFunction(f_t!, g_t!, fg_t!)");
args[1] = (jl_value_t*) jl_symbol("f");
args[2] = (jl_value_t*) jl_symbol("g");
args[3] = (jl_value_t*) jl_symbol("fg");
args[4] = (jl_value_t*) jl_apply_array_type(jl_float64_type, 1 );

// Wrap the C arrays as Julia arrays. The final "0" argument means
// that Julia does not take ownership of the array data for GC purposes.
args[5] = (jl_value_t*) jl_ptr_to_array_1d(args[4], guess, n, 0);
args[6] = (jl_value_t*) jl_ptr_to_array_1d(args[4], fval, n, 0);
args[7] = (jl_value_t*) jl_ptr_to_array_1d(args[4], jacobian, n*n, 0);
args[8] = (jl_value_t*) jl_get_function(jl_base_module, "tuple");
args[9] = jl_eval_string("solver_task(df, guess) = @task nlsolve(df, guess)");
```

Embedding Julia tasks based RC solver in C

```
{
    jl_value_t* vals   = jl_call3((jl_function_t*) args[8], args[1], args[5], args[6]); // "vals" tuple
    jl_value_t* solver = jl_call2((jl_function_t*) args[9], args[0], args[5]);          // solver task

    JL_GC_PUSH2(&vals, &solver);

    jl_yield();
    while (!istaskdone(solver)){
        if (jl_get_nth_field(vals, 0) == args[1]) {         // :f case
            args[5] = jl_get_nth_field(vals, 1);
            args[6] = jl_get_nth_field(vals, 2);
            r_f((jl_array_t*) args[5], (jl_array_t*) args[6]);
            vals = yieldto(solver, vals);
        } else if (jl_get_nth_field(vals, 0) == args[2]) { // :g case
            args[5] = jl_get_nth_field(vals, 1);
            args[7] = jl_get_nth_field(vals, 2);
            r_g((jl_array_t*) args[5], (jl_array_t*) args[7]);
            vals = yieldto(solver, vals);
        } else if (jl_get_nth_field(vals, 0) == args[3]) { // :fg case
            args[5] = jl_get_nth_field(vals, 1);
            args[6] = jl_get_nth_field(vals, 2);
            args[7] = jl_get_nth_field(vals, 3);
            r_fg((jl_array_t*) args[5], (jl_array_t*) args[6], (jl_array_t*) args[7]);
            vals = yieldto(solver, vals);
        }
    }

    jl_show(jl_stderr_obj(), vals);
    jl_eval_string("println(\"\n\")");
    JL_GC_POP();
}
JL_GC_POP();
```

Implement the driver while loop -

Print the results -

Pop variables from the GC -

Summary

- Most current Julia packages for root finding and optimization implement forward communication strategies (as they should!)
- Julia tasks can be utilized to transform forward communication solvers into reverse communication solvers without modifying the original forward solvers
- The reverse communication wrapper strategy can be embedded within a C/C++ application allowing for unmodified use of existing C/C++ objective/derivative functions